Scheduling Deterministic Parallel Programs

Daniel John Spoonhower

CMU-CS-09-126
May 18, 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy E. Blelloch (co-chair)
Phillip B. Gibbons
Robert Harper (co-chair)
Simon L. Peyton Jones (Microsoft Research)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored by Microsoft, Intel, and IBM. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: parallelism, scheduling, cost semantics

Abstract

Deterministic parallel programs yield the same results regardless of how parallel tasks are interleaved or assigned to processors. This drastically simplifies reasoning about the correctness of these programs. However, the performance of parallel programs still depends upon this assignment of tasks, as determined by a part of the language implementation called the scheduling policy.

In this thesis, I define a novel cost semantics for a parallel language that enables programmers to reason formally about different scheduling policies. This cost semantics forms a basis for a suite of prototype profiling tools. These tools allow programmers to simulate and visualize program execution under different scheduling policies and understand how the choice of policy affects application memory use.

My cost semantics also provides a specification for implementations of the language. As an example of such an implementation, I have extended MLton, a compiler and runtime system for Standard ML, with support for parallelism and implemented several different scheduling policies. Using my cost semantics and profiler, I found a memory leak caused by a bug in one of the existing optimizations in MLton.

In addition to memory use, this cost semantics precisely captures when a parallel schedule deviates from the uni-processor execution of a given program. The number of deviations acts as a important conceptual tool in reasoning about concrete measures of performance. I use this metric to derive new bounds on the overhead of existing work stealing implementations of parallel futures and to suggest more efficient alternatives.

This work represents an important step in understanding how to develop programs and programming languages for multi-core and multi-processor architectures. In particular, a cost semantics provides an abstraction that helps to ensure adequate performance without sacrificing the ability to reason about programs at a high level.

to Copic

Acknowledgments

When I arrived at Carnegie Mellon, I expected to find bright, motivated people working on exciting problems. I found not only that but also a vibrant community of students, especially among the students of the Principles of Programming group. Many of these students became not only my colleagues but also my friends. Their passion and enthusiasm for programming languages, logic, and computer science escapes from campus and into homes, restaurants, and bars. In particular, I would like to thank those senior students who helped to establish this community within our group, especially Kevin Watkins, Leaf Peterson, Derek Dreyer, and Tom Murphy VII, as well as those who continue to maintain it, especially Jason Reed, Dan Licata, William Lovas, Rob Simmons, and the other members of the ConCert Reading Group. It have been extremely gratifying to have participated in this culture and to see that culture persist beyond my time at CMU.

I would like to thank the MLton developers, especially Matthew Fluet, for providing the foundation upon which I did much of my implementation and for providing technical support.

Pittsburgh has been a great place to live while pursuing my graduate studies. This is due in a large part to the great people I have met here. This includes many fantastic people at CMU, including Rose Hoberman, Katrina Ligett, Adam Goode, Chris Twigg, Tiankai Tu, and those mentioned above; the D's regulars, especially Heather Hendrickson, Stephen Hancock, and Laura Marinelli; and the Alder Street crowd, especially Ryan Kelly and Jessica Nelson who provided a place to stay and a home while I was traveling frequently.

My advisers, Bob Harper and Guy Blelloch, and Phil Gibbons, who became a third adviser, were (of course) instrumental in my success as a graduate student. I am deeply grateful for their unflinching support and confidence in me. I would also like to acknowledge the role of Dan Huttenlocher, who has been a friend and mentor since before I started graduate school and who has helped me to make many important decisions along the way.

Finally, I would also like to thank my parents, Tom and Sue, and the rest of my family for always encouraging me to do my best and Kathy Copic for never accepting less than that.

Preface

Like all endeavors of human understanding, the goals of computer science are to generalize our experience and to build models of the world around us. In the case of computer science, these models describe computers, programs, algorithms, and other means of processing information. The purpose of these models is to enable us to understand more about existing phenomena and artifacts, including both those that are purely conceptual and those that are physical in nature. These models also enable us to discover and develop new ways of processing information. As in other areas of study, these models are justified by their capacity to make predictions about existing phenomena or to connect ideas that were previously considered unrelated or disparate.

As a study of information, computer science is also concerned with these models themselves and with the means used to justify them. In a sense, abstraction, one of the most important conceptual tools of computer science, is the practice of dividing information about a system into that which will help us to learn more about its behavior and that which impedes our understanding.

Despite the unifying goal of computer science to better understand how information is processed, work in our field is often divided into two broad but disjoint categories: theory and systems. However, these need not be competing sub-fields but instead, complementary approaches for evaluating models about information and information processing systems. In this thesis, I develop new models about languages for parallel programming. In addition, I use approaches based on both theory and practice to evaluate these models. My hope is that these models will serve to help future language implementers understand how to efficiently exploit parallel architectures.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallelism abounds! The use of parallel architectures is no longer limited to applications in high-performance computing: multi-core and multi-processor architectures are now commonplace in desktop and laptop computers. The costs of increasing processor frequency (in terms of power consumption) are continuing to increase, and hardware designers are turning to chip-level parallelism as a means of providing more computational resources to programmers. It then falls to programmers to exploit this parallelism to make programs run faster.

In this thesis, I take the view that we already have algorithms with adequate parallelism as well as programming languages to describe these algorithms. Despite this, there remains a significant obstacle to programming for parallel architectures effectively: a lack of models that enable programmers to reason about parallel performance and to understand when they can expect programs to run faster.

I will address this obstacle by giving an abstract specification for performance. I will consider how the sequential performance of programs can be used as the basis for a model of parallel performance and how such a model can be used both as a guide for language implementers and for programmers. Finally, I will show implementations of a parallel programming language that meet this specification and, at the same time, efficiently map parallel tasks onto multi-core architectures and reduce the execution time of programs.

1.1 Parallelism

Among the many existing programming techniques and languages designed to support programming for parallel architectures, I would like to make a distinction between two broad classes of approaches: parallel programming and concurrent programming.

By **parallel programming**, I mean a style of programming where the programmer makes some symmetric structure of the program explicit. For example, divide-and-conquer algorithms exhibit this sort of symmetry. A less rigid form of this structure is also present in dynamic programming algorithms (where each sub-problem is solved using the same method) and applications that use a producer-consumer model (where there is a correspondence between actions of the producer and those of the consumer). I use the word "parallel" in the sense of its Greek origins to describe two or more parts of a program that may be considered "beside one another."

Parallelism in this sense is a useful conceptual tool for understanding and reasoning about different parts of the program independently.

Apropos to this thesis, the implementation of a parallel language may also take advantage of this structure to evaluate several parts of the program simultaneously. We do not, however, require the programmer to consider this aspect of the implementation. Instead we fix the meaning of parallel programs independently of the language implementation, hardware, and runtime environment. Stated another way, the evaluation of parallel programs is *deterministic*. Deterministic parallelism has a long history, including work on data-flow languages (*e.g.,* Arvind et al. [1989]) and collection-oriented languages (*e.g.,* Sabot [1988]).

This style of programming stands in contrast to **concurrent programming**, where programmers manage the simultaneous execution of tasks explicitly. For example, this style of programming may be used to respond simultaneously to multiple incoming requests from the environment (*e.g.* from the network or from the user). I use the word "concurrent" to describe this way of thinking about programs, since for two tasks to concur, they must first *occur*. That is, "concurrent" puts an emphasis on action and implies a notion of time: we must consider the state of the program execution before, during and after a task is evaluated. Examples of concurrent programming include languages such as Concurrent ML [Reppy, 1999], Erlang [Virding et al., 1996], and Jocaml [Conchon and Fessant, 1999], as well as programming explicitly with threads and synchronization in many other languages.

Concurrent programming is well-suited to reactive environments, where programmers are already forced to consider the passage of time, or applications which consist of a set of distinct and largely unrelated tasks. Programming at this level of abstraction means that the result of evaluating a program depends on how and when tasks are evaluated simultaneously. The meaning of a concurrent program is defined by all possible interleavings of tasks that respect the constraints given by the programmer: concurrent programming is inherently non-deterministic. While we might consider a parallel program simply as a sequential program where the programmer has specified opportunities for tasks to be evaluated simultaneously (with the guarantee that the result will be the same regardless), concurrent programming requires programmers to start from the set of all possible interleavings and restrict these possibilities so that all remaining interleavings yield a correct result.

In contrast, concurrent programming is *not* well-suited to using parallel architectures to improve the performance of programs. First, before programmers consider performance, they must ensure that their concurrent programs are correct. This is significantly more difficult than for sequential programs and their parallel counterparts. Second, programmers must either find a large number of independent tasks to utilize processor cores (a problem that becomes increasingly difficult as the number of cores scales from 16 to 32, 64 and more) or they must exploit hardware-level parallelism by processing data in parallel. In the latter case, they must re-implement many aspects of a parallel language implementation (*e.g.*, task communication, load balancing) duplicating development effort and also time spent in understanding the performance consequences of these implementation decisions.

Parallelism abounds not only in the profusion of parallel architectures but also in existing programs: if we think at a sufficiently high level, there is ample parallelism in many existing algorithms. Furthermore, this parallelism is already manifest in many **functional programs** through uses of recursion and higher-order functions on collections (*e.g.*, `map`). Functional

programming also emphasizes building *persistent* data structures, which can be safely shared among parallel tasks without introducing non-determinism. In this thesis, I use a functional language as a basis for building parallel programs.

Previous work on parallel implementations of functional languages (*e.g.*, pH [Aditya et al., 1995], the thesis work of Roe [1991], NESL [Blelloch et al., 1994], Manticore [Fluet et al., 2008b]) have demonstrated that deterministic parallelism can adequately express many parallel algorithms. In this thesis, I will focus on the implementation of a parallel functional language and how implementation choices affect performance.

1.2 Scheduling

Many parts of the implementation of a parallel language affect application performance, including communication among processors, the granularity of tasks, and the scheduling policy. In this thesis, I will focus on scheduling policies. A **scheduling policy** determines the **schedule**, the order which parallel tasks are evaluated and assignment of parallel tasks to processors or processor cores. This assignment may be determined as part of the compilation process, by the runtime system, or by a combination of the two.

Parallel programming encourages programmers to express a large number of parallel tasks, often many more tasks than processors or processor cores. This ensures that parallel programs are portable: programmers can develop programs independently of the number of processors and be confident that there will be enough parallel tasks to take advantage of whatever parallel hardware is available. The length of any given task may be difficult to predict, and dividing the program into many small tasks also ensures that load can be evenly balanced among these processors.

An online or dynamic scheduling policy determines the schedule as the application runs. Examples of online schedulers include those with a centralized task queue, such as round-robin and parallel depth-first scheduling [Blelloch et al., 1999], as well those with one queue per processor, as in implementations of work stealing [Halstead, 1984].

Though online scheduling policies help ensure that load is balanced across processors, understanding how this policy and available resources affect performance can be difficult. For example, Graham [1966] showed examples of scheduling "anomalies," instances where relaxing a resource constraint (*e.g.*, increasing the number of processors) can lead to *longer* execution times. Small changes in the program input or compiler can also affect performance in ways that are difficult to predict.

As it is not feasible to test programs on all possible inputs and hardware configurations, one approach to understanding performance is to give bounds on the time or memory required by an application. For example, no scheduling policy can run faster than the optimal schedule with an unbounded number of processors. Graham also showed that any greedy scheduling policy (one which will never allow a processor to remain idle if there is a task ready to execute) will never require more than twice the time of the optimal greedy policy for a fixed number of processors.

Another useful baseline for performance is the single-processor execution as determined by the sequential semantics. Programmers are already familiar with the performance of the sequential implementation. We expect a parallel execution on multiple processors to run at least as fast

as the sequential execution, and (assuming the total amount of computation is fixed and ignoring effects of the memory hierarchy) a parallel execution with P processors will generally run no more than P times faster than it would on a single processor. One measure of the performance of a scheduling policy is how close it comes to achieving this ideal.

The choice of scheduling policy can also affect how an application uses memory. At minimum, a parallel program run on P processors should require no more than P times as much memory as it would on a single processor. For unrestricted deterministic parallel programs, no scheduling policy can both achieve linear speedup and use no more than linear space [Blumofe and Leiserson, 1993]. However, for certain classes of parallel programs and certain scheduling policies, both time and memory use can be bounded in terms of the performance of the sequential execution [Blumofe, 1994, Blelloch et al., 1999].

In this thesis, I take a broad view of scheduling and include not only decisions made by the language implementation at runtime, but also those where the compiler affects the assignment of parallel tasks to processors. This includes program transformations common to both sequential and parallel programs, which can have asymptomatic effects on memory use [Gustavsson and Sands, 1999], as well compiler techniques specific to parallel programs. As an example of the latter, flattening [Blelloch and Sabot, 1990] is a transformation that converts nested parallelism into a flat form of parallelism implemented by vector machines and graphics processors. Though this enables programmers to write high-level parallel programs and still take advantage of these platforms, it may constrain what sort of scheduling decisions can be made during program execution.

In previous work [Blumofe and Leiserson, 1993, 1999, Blelloch et al., 1999], scheduling policies are often applied to abstract computations, usually structured as directed, acyclic graphs. Each such graph represents a particular execution of a given program with vertices representing computation (with some nodes possibly representing larger units of computation than others) and edges representing sequential dependencies. Properties of these graphs can then be used as a basis for reasoning about the performance of different schedulers. For example, the number of vertices is proportional to the time required by a sequential implementation, and the length of the longest path determines the amount of time required by an implementation with an unbounded number of processors. I will also make use of such graphs and connect them formally with programs, as outlined in the next section.

1.3 Cost Semantics

To understand the effects of scheduling on performance, I introduce a cost semantics that explicitly accounts for both parallelism and memory use. In general, a **cost semantics** assigns a measure of cost to programs [Sands, 1990, Sansom and Peyton Jones, 1993, Blelloch and Greiner, 1996]. This cost may describe some abstract measure of space or time as a scalar value, or it may capture the behavior of the program more precisely using a program trace. A cost semantics is *not* a static analysis but a dynamic semantics: it assigns costs to closed programs (*i.e.*, to a program together with one input).

A cost semantics serves as a specification for a language implementation. For example, many functional programmers assume that tail recursion is implemented efficiently. Whether

this assumption is omitted, as in more formal definitions that address only extensional behavior [Milner et al., 1997], or described informally [Abelson et al., 1998], programmers nonetheless rely on it to achieve good performance. Such an assumption can be made explicit through a cost semantics, and moreover, can be described without resorting to an explicit representation of activation records, closures, or other aspects of the implementation. As a specification, a cost semantics provides a guide for language implementers, but does not dictate the precise details of such an implementation.

A cost semantics also enables programmers to reason about the performance of programs independently of a particular implementation: because such a semantics acts as a specification, that specification can be used in place of an implementation. By "reason about programs," I mean that programmers can establish some expectations about program performance independently of any implementation. Programmers use a cost semantics, at least implicitly, whenever they perform an analytic analysis of programs. For most sequential programs, programmers use a semantics that is simple enough that they do not state it explicitly: it assigns unit cost to each arithmetic operation, memory access, or function call. Like most cost semantics, this is an approximation of actual program performance, but it is sufficient to model the asymptotic behavior of programs.

Though I will consider many different ways of evaluating a single parallel program, the cost semantics used in this thesis assigns one cost, in the form of a pair of directed graphs, to each closed program. This cost is independent of the number of processors and the scheduling policy, and it may be used to reason about the performance of a program under an arbitrary scheduling policy. By fixing the number of processors and a scheduling policy, the programmer may also draw more precise conclusions of performance. This method thus factors reasoning about the performance of a program into two stages. First, programmers use the semantics to derive a cost for the program and abstract away many of the details of the program. Second, they analyze that cost, possibly including information about the scheduling policy. Even when considering scheduling, however, such an analysis remains abstract as it does not explicitly consider the implementation of the scheduling policy nor any of the compilation techniques required to transform a source program into an executable binary.

As an example of an area where abstract reasoning about performance has been successful, consider the memory hierarchies found in modern computer systems. Processor caches and virtual memory are sophisticated implementations of a simple interface: a uniform address space. Programmers develop applications using this interface even though some memory accesses will incur costs that are orders of magnitude larger than others. This enables programmers to achieve correctness in a straightforward way. When performance becomes an issue, programmers generally rely on locality, an abstract measure of performance, instead of concrete implementation details such as cache size, structure, associativity, or replacement policy. Accounting for all of these possibilities on all the different hardware configurations on which such a program will eventually be run would be difficult. Locality provides a more tractable means to reason about performance. Various abstractions of locality have been developed including the notion of cache-oblivious algorithms [Frigo et al., 1999].

The goal of this thesis is to work toward a similar methodology for writing parallel programs. Like a uniform address space, deterministic parallelism enables programmers to separately consider correctness and performance, and like locality, the cost graphs used in this thesis provide a basis for understanding performance at an abstract level.

1.4 Overview

With this background as a foundation, I can now state a summary of this thesis.

> **Thesis Statement.** A cost semantics provides a useful abstraction for understanding how scheduling policies affect the performance of deterministic parallel programs.

I will consider how the scheduling policy affects memory use among other aspects of performance. I shall also consider programs that use both nested parallelism as well as parallel futures. As evidence for this claim, I will consider several implementations of a parallel functional language, outlined below. I will show how a cost semantics can be used as a specification for these implementations and also consider how such a specification can be used by programmers to reason about the memory use of parallel programs.

Cost Semantics. I will begin by introducing a cost semantics that will serve as the foundation for this thesis (Chapter 2). This semantics defines the cost of a parallel execution as a pair of directed, acyclic graphs. I then define schedules as partial orders over the nodes of the graphs. Using partial orders allows me to easily characterize the differences between different executions, including between the standard sequential execution and parallel executions. These differences, called *deviations*, can be used to measure the overhead of parallel scheduling.

Online Scheduling. I will give an implementation of this language as an operational semantics that embodies all possible parallel executions (Chapter 3). Though this semantics is non-deterministic, I will show that all executions that lead to an answer will yield the same answer and that my cost semantics models the memory use of each of these executions. I then give implementations of three scheduling policies as refinements of the non-deterministic semantics and show how results about memory use are easily extended to these implementations.

Semantic Profiling. Profiling, and in particular space profiling, has been established as an important tool in improving the performance of both sequential [Runciman and Wakeling, 1993a] and parallel [Hammond et al., 1995] functional programs. In previous work, profilers were built by instrumenting an existing compiler and runtime system. This approach, however, makes it difficult to untangle performance problems in the program from those in the language implementation. In Chapter 4, I will describe a *semantic* space profiler, a profiler based on my cost semantics, that yields results independent of any compiler or runtime system. I will address some challenges in summarizing the fine-grained results of the semantics and presenting them visually to programmers.

Vector Parallelism. As an example of offline or static scheduling, I consider an implementation of vector parallelism called flattening [Blelloch and Sabot, 1990] (Chapter 5). Flattening is a compiler technique that transforms high-level *nested* parallelism into the flat parallelism that is

implemented by vector processors, graphics processors, and multimedia extensions of general-purpose processors. While not always considered as a scheduling policy, this implementation technique is similar to a scheduling policy, in that it determines the order in which parallel tasks are evaluated. To model the behavior of flattened code, I use a constrained schedule based on a labeling [Lévy, 1978] of source programs.

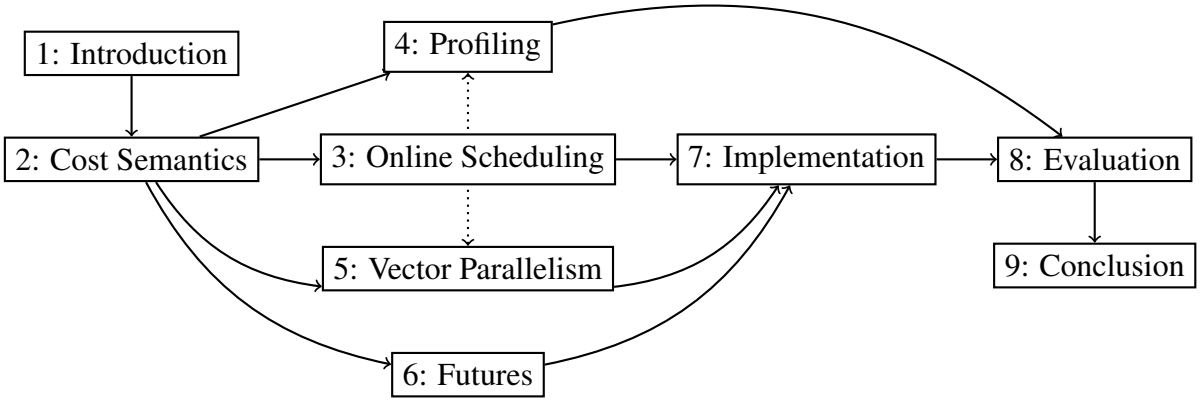
Futures. The forms of parallelism discussed so far have been adequate to describe the parallel structure of divide-and-conquer algorithms and those that use parallel map, but not algorithms that use parallel pipelining. Parallel futures [Halstead, 1985] are a more powerful form of parallelism that enable programmers to build pipelines by expressing data dependencies between parallel tasks. Though more expressive than nested parallelism, using futures in a functional language still results in a deterministic form of parallelism. Though parallel futures have appeared in a number of language implementations (*e.g.*, Mohr et al. [1990]), there has been less work in understanding the performance of programs that use them. In Chapter 6, I consider implementations of parallel futures that use *work stealing* to balance load across processors. A key feature of a work-stealing scheduling policy is that it puts the burden of load balancing and communication on idle processors rather those with unfinished work to perform. I use deviations to compare the performance of different implementations of futures and give tight bounds on the number of deviations that will be incurred by a broad class of work-stealing scheduling policies.

Implementation in MLton. MLton [Weeks, 2006] is a whole-program, optimizing compiler for Standard ML [Milner et al., 1997]. As part of my work, I have extended MLton to support multi-processor execution and built a framework for implementing different parallel schedulers (Chapter 7). This implementation serves as a validation of my profiling methods and as an example of how a cost semantics constrains an implementation. In the course of developing this extension, I used my cost semantics to discover a bug in one of the optimizations in MLton that would cause some applications to use asymptotically more memory than would be required without the optimization. This and other empirical evaluations of my implementation are described in Chapter 8. This evaluation serves to validate the theories and methods described in earlier chapters.

Organization of Dissertation. The material found in Chapter 2 is critical in understanding all of the subsequent chapters. However, many of later chapters can be read independently. Only a cursory reading of Chapter 3 (Online Scheduling) is necessary for later chapters. Chapters 5 (Vector Parallelism) and 6 (Futures) both describe orthogonal extensions of the core language and can be understood independently of each other or Chapter 4 (Profiling). In summary, any schedule[1] of the graph in Figure 1.1 should allow the reader to form a complete picture of this dissertation.

[1] See Chapter 2.

Figure 1.1 Organization.



1.5 Contributions

This dissertation contains three major contributions. First, my cost semantics is the first to account for both application memory use and different scheduling policies. It captures memory use precisely but without committing to many implementation choices (*e.g.*, closure representation). It also allows scheduling policies to be expressed abstractly as refinements of the order in which nodes in cost graphs are visited during a parallel traversal. This cost semantics forms the basis of a implementation-independent profiler.

Second, I present new bounds on the overhead of parallel implementations based on work stealing using deviations and show tight bounds on the number of deviations for programs that use parallel futures. This work suggests that alternative scheduling policies to conventional work-stealing approaches may yield better performance for programs that use parallel futures.

Third, I have extended MLton with support for multiple processors. This is the first multi-processor version of Standard ML. Though there are still opportunities for optimization, this implementation enables programmers to use multiple processors or cores to make functional programs run faster. This implementation also validates my semantic approach to profiling by showing that the my profiler makes accurate predictions about memory use.

Chapter 2

Cost Semantics

In this chapter, I will lay the foundations of my framework for analyzing the performance of parallel programs and scheduling policies. A key part of this analysis is understanding when and how a scheduling policy *deviates* from the sequential schedule. To establish when deviations occur and understand the performance effects of these deviations, we must first fix a sequential schedule. We must also define the range of possible parallel schedules. These are both achieved using the cost semantics presented in this chapter.

The language studied in this thesis is a call-by-value functional language. Although this choice is not essential, it enables us to establish that parallel evaluation is deterministic and to focus on the performance effects of parallelism independently of other aspects of the language: similar analyses could also be performed for a pure fragment of an imperative language or a call-by-need language. This language uses a left-to-right evaluation strategy.

Examples in this thesis use the syntax of Standard ML [Milner et al., 1997] though the language studied formally is a subset of the core ML. Language features such as data types, exceptions, pattern matching, and modules pose no technical challenges and are omitted only for the sake of clarity.

2.1 Example: Matrix Multiplication

Before presenting my cost semantics, I first give an example of a parallel functional program along with some intuition as to how the choice of scheduling policy can affect performance.

Figure 2.1 shows the source code for one implementation of parallel matrix multiplication. This example defines two top-level functions. The first is a parallel version of tabulate and the second, mm_mult, multiplies two matrices of appropriate sizes using persistent vectors. Within the definition of mm_mult, two additional functions are defined. The first, vv_mult, computes the dot product of two vectors. The second, mv_mult, computes a matrix-vector product. In this example, variables A and B stand for matrices, a and b for vectors, and x for scalars. Persistent vectors are created with one of two primitives. vector (s, x) creates an vector of size s where every element is given by the value of x. The other primitive, append, creates a new vector that is the result of appending its arguments. (For simplicity, this implementation of multiplication assumes that one of its arguments is in row-major form while the other is in column-major form.)

Figure 2.1 Parallel Matrix Multiplication. This is an example of an application that uses nested parallelism.

```
fun tabulate f n =
  (* offset i, length j *)
  let fun loop (i, j) =
        if j = 0 then vector (0, f 0)
        else if j = 1 then vector (1, f i)
        else let
               val k = j div 2
               val lr = { loop (i,     k),
                          loop (i + k, j − k) }
             in
               append (#1 lr, #2 lr)
             end
  in
    loop (0, n)
  end

fun mm_mult (A, B) =
  let
    fun vv_mult (b, a) = reducei op+ (fn (i, x) ⇒ sub (b, i) ∗ x) 0.0 a
    fun mv_mult (B, a) = tabulate (fn i ⇒ vv_mult (sub (B, i), a)) (length B)
  in
    tabulate (fn i ⇒ mv_mult (B, sub (A, i))) (length A)
  end
```

The parallelism in this example is derived in part from the implementation of tabulate. As in an ordinary implementation of tabulate, this function returns a vector where each element is initialized using the function supplied as an argument. In this implementation, whenever the result will have two or more elements, each half of the result may be built in parallel. In this example and throughout the rest of the thesis, I use curly braces "{}" to denote expressions where parallel evaluation may occur. This syntax is further explained in Section 2.2.

This form of extremely fine-grained parallelism is difficult to implement efficiently, but examples such as this still serve to model the performance of more coarsely-grained implementations. In this example, multiplying two $n \times n$ matrices may result in n^3 scalar multiplications occurring parallel. While few (if any) languages or platforms support parallel tasks in this size and number, we might consider this as a model for *block* matrix multiplication: we can use this program to reason about the performance of a block algorithm by assigning the cost of multiplying two blocks to each scalar multiplication.

Not shown here are the implementations of several other functions on vectors, including length, which returns the length of its argument, and reducei, which acts like a form of fold but may reduce two halves of a vector in parallel before combining the results.

Figure 2.2 Language Syntax and Semantic Objects. This figure defines the syntax of the core language studied in this thesis. It also defines values and locations, which are used in the cost semantics below.

(expressions)	e	::=	$x \mid () \mid \mathsf{fun}\ f(x) = e \mid e_1\ e_2 \mid \{e_1, e_2\} \mid \#1\ e \mid \#2\ e \mid v$
(values)	v	::=	$\langle\rangle \mid \langle f.x.e\rangle^{\ell} \mid \langle v_1, v_2\rangle^{\ell}$
(locations)	ℓ	$\in$	L

With this example in hand, I will now briefly consider this example in a broader context of parallel programs, how these programs can be implemented, and the performance consequences of these implementation choices.

Fork-Join Parallelism. For this and the following three chapters, we will consider a form of parallelism called fork-join parallelism. Fork-join (or fully strict) parallelism is a form of parallelism where the point in the source code at which a new task is created (the *fork*) always coincides with the point at which the result of that task is required (the *join*). If we think of a parent-child relationship between a task and each new task that it creates, then this form of parallelism stipulates that each task joins with its parent (as opposed to another ancestor or peer task). In the matrix multiplication example, fork-join parallelism is expressed as a pair of expressions $\{e_1, e_2\}$ where e_1 and e_2 may be evaluated in parallel. That is, the two recursive calls to loop may occur in parallel, but evaluation cannot continue until both calls have returned.

While restrictive, this model allows programmers to express parallelism in many common programming paradigms, including divide-and-conquer algorithms, and is sufficient to demonstrate important differences between scheduling policies.

Scheduling Policies. As noted above, the matrix multiplication example produces many parallel tasks. It is up to the scheduling policy to determine the order in which they are evaluated. One method of managing these tasks is to maintain one or more work queues. For example, a "fair" scheduling policy maintains a single global queue. Under this policy, idle processors remove tasks from the beginning of the queue, and whenever new tasks become available, they are added to the end of the queue. In addition, after a processor works on some task for a specified period of time, it returns that task to the end of the queue and takes another task from the beginning. This method exposes a tremendous amount of the available parallelism.

Performance. In this thesis, I will use the performance of sequential programs as a basis for understanding the performance of parallel programs, including the effects of different scheduling policies. For the moment, we take the semantics of Standard ML (including order of evaluation) as the definition of the sequential schedule. The sequential implementation of $\{e_1, e_2\}$ is the same as for ordinary pairs (e_1,e_2). That is, e_1 is evaluated before e_2. A deviation (defined formally below) occurs in a parallel implementation whenever a processor begins to evaluate e_2 before e_1 has finished evaluation. The number of deviations can be used to understand the relationship between parallel and sequential performance.

In the matrix multiplication example, a sequential implementation of the language computes each element of the result matrix in turn. Specifically, each such element is evaluated completely (using reducei) before work on the next element begins. Any parallel implementation will give rise to *some* deviations, but some implementations incur more deviations than others. In a fully parallel implementation, such as a fair scheduling policy, all floating-point multiplications are performed before any floating-point additions. This means that $O(n^3)$ deviations will occur: under the sequential semantics these multiplications and additions will be interleaved. While the number of deviations is only an abstract measure of performance, it gives us a clue as to how this implementation will perform relative to a sequential implementation. The cost semantics described below will also enables programmers to make more concrete comparisons of time, memory use, and cache behavior.

2.2 Syntax

Figure 2.2 gives the syntax for the language and semantic objects studied in this thesis. This is a simple functional language with recursive functions and pairs loosely based on the syntax of Standard ML.

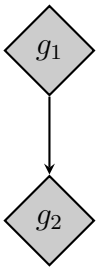
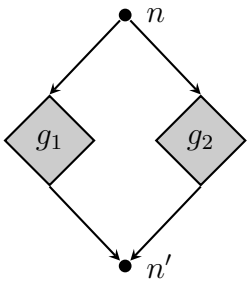
Program variables are denoted by x (and variants); f stands for program variables that are bound to function values. The unit expression is written (). Recursive function expressions fun $f(x)$ = e bind f and x in e. I use recursive functions to emphasize that the potential for non-termination does not significantly complicate the analyses in this thesis. Function application is denoted by juxtaposition: $e_1\ e_2$. Initially, all pairs will be constructed in parallel as $\{e_1, e_2\}$ though I will later add syntax for sequentially constructed pairs. I use SML syntax #1 and #2 to denote the projection of the first and second components of a pair. As the extensional result of parallel pairs is identical to that of sequentially constructed pairs, the typing rule follows the same pattern.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \{e_1, e_2\} : \tau_1 \times \tau_2}$$

The remainder of the static semantics of this language is completely standard and therefore omitted.

The syntax of function and pair values is distinguished from the syntax of source expressions for functions and pairs: programmers never write values in their code. Values are delimited with angle brackets "$\langle\rangle$". Associated with each value is a location ℓ drawn from an infinite set of unique locations L. Locations are used to express sharing and will be important in reasoning about the memory use of programs. For example, the value $\langle v^{\ell_1}, v^{\ell_1}\rangle^{\ell}$ denotes a pair where both components occupy the same memory cells. The value $\langle v^{\ell_1}, v^{\ell_2}\rangle^{\ell}$ denotes a pair where the two components are extensionally identical but may require as much as twice the memory. Since unit values occupy no space, they are not annotated with locations. We refer to the subset of expressions that does not contain any values as **source expressions**.

Figure 2.3 Graph Operations for Pairs. The graphs associated with the core language (a language with parallel pairs) are built from single nodes as well as the serial and parallel composition of smaller graphs.

Single Node	Serial Composition	Parallel Composition
$[n]$	$g_1 \oplus g_2$ $(n_s; n_e; E) \oplus (n'_s; n'_e; E')$	$g_1 \otimes g_2$ $(n_s; n_e; E) \otimes (n'_s; n'_e; E')$
		
$(n; n; \emptyset)$	$(n_s; n'_e; E \cup E' \cup (n_e, n'_s))$	$(n; n'; E \cup E' \cup (n, n_s) \cup$ $(n, n'_s) \cup (n_e, n') \cup (n'_e, n'))$ n, n' fresh

2.3 Graphs

To reason about parallel evaluations and memory use, I associate a pair of acyclic graphs with each closed program. These graphs, the computation graph and the heap graph, form an execution trace of each program, as part of the cost semantics described below.

Computation Graphs. A computation graph is a directed, acyclic graph. Each node in the computation graph represents a single reduction of some sub-expression, or more specifically an event associated with that reduction. Edges represent constraints on the order in which these events can occur. Edges in the computation graph point forward in time: an edge from node n_1 to node n_2 indicates that the expression corresponding n_1 must be evaluated before that corresponding to n_2. Computation graphs are similar to the graphs introduced by Blumofe and Leiserson [1993]. In the figures in this thesis, nodes will be arranged so that time passes from top to bottom.

If there is an edge from n_1 to n_2 then we say that n_1 is the parent of n_2, and n_2 is the child of n_1. If there is a non-empty path from n_1 to n_2 then we write $n_1 \prec_g n_2$ and say that n_1 is an ancestor of n_2, and that n_2 is a descendant of n_1.

For fork-join programs, each computation graph is a series-parallel graph. Each such graph consists of a single-node, or a sequential or parallel composition of smaller graphs. The primitive operations required to build these graphs are shown in Figure 2.3. Each such graph has one **start node** (with in-degree zero) and one **end node** (with out-degree zero), *i.e.*, computation graphs are directed series-parallel graphs. Nodes are denoted ℓ and n (and variants). Graph are written as tuples such as $(n_s; n_e; E)$ where n_s is the start node, n_e is the end node, and E is a list of edges. The remaining nodes of the graph are implicitly defined by the edge list. In figures, nodes are drawn as circles while arbitrary graphs are represented as diamonds.

Some nodes in the computation graph will represent the allocation of memory. These nodes will be given names such as ℓ (and variants) to draw a tight connection with the locations that appear in values: the node ℓ will correspond to the allocation of a value with location ℓ. Nodes that do not correspond to allocation (or nodes in general) will be given names such as n (and variants).

I will identify graphs up to reordering of nodes and edges: binary operations $\oplus$ and $\otimes$ on computation graphs are commutative and associative. As suggested by the notation, $\otimes$ is assigned higher precedence than $\oplus$.

Heap Graphs. I extend previous work on parallel cost semantics with heap graphs [Spoonhower et al., 2008]. Heap graphs are also directed, acyclic graphs but, unlike computation graphs, do not have distinguished start or end nodes. Each node again represents the evaluation of a subexpression. For source expressions, each heap graph shares nodes with the computation graph arising from the same execution. In a sense, computation and heap graphs may be considered as two sets of edges on a shared set of nodes. While edges in the computation graph point forward in time, edges in the heap graph point backward in time.

Edges in the heap graph represent a dependency on a value: if there is an edge from n to ℓ then n depends on the value with location ℓ. The sink of a heap edge will always be a node ℓ corresponding to the value allocated at that point in time. The source of a heap edge may be either a node ℓ' (indicating a dependency among values) or node n (indicating a dependency on a value by an arbitrary computation). In the first case, the memory associated with ℓ cannot be reclaimed while the value represented by ℓ' is still in use. As will be discussed in more detail below, in the second case (where n represents a point in the evaluation of an expression) the memory associated with ℓ cannot be reclaimed until after the expression corresponding to n has been evaluated. Heap graphs will be written as a set of edges. As above, the nodes are left implicit.

2.4 Semantics

A cost semantics is an evaluation semantics that computes both the result of the computation and an abstract cost reflecting *how* the result was obtained. That is, it allows us to distinguish two programs which compute the same result but do so in different ways.

While the semantics is inherently sequential, the cost will allow us to reconstruct different parallel schedules and reason about the memory use of programs evaluated using these schedules. The judgment

$$e \Downarrow v; g; h$$

is read, *expression e evaluates to value v with computation graph g and heap graph h.* The extensional portions of this judgment are completely standard in the way they relate expressions to values. As discussed below, edges in a computation graph represent control dependencies in the execution of a program, while edges in a heap graph represent dependencies on and between values. Figure 2.4 gives the inference rules for this judgment.

The cost associated with evaluating a pair (C-FORK) makes use of the parallel composition of computation graphs $\otimes$: each component of the pair may be evaluated in parallel. Two edges

Figure 2.4 Cost Semantics. This semantics assigns abstract costs to expressions. The substitution $[v/x]$ of a value v for a variable x is defined in Figure 2.6. The outermost locations of values $\text{loc}(v)$ and expressions $\text{locs}(e)$ are defined in Figure 2.7. Parallel composition takes precedence over serial composition so $g_1 \otimes g_2 \oplus g_3$ should be read as $(g_1 \otimes g_2) \oplus g_3$.

$\boxed{e \Downarrow v; g; h}$

$$\frac{e_1 \Downarrow v_1; g_1; h_1 \qquad e_2 \Downarrow v_2; g_2; h_2 \qquad (\ell \text{ fresh})}{\{e_1, e_2\} \Downarrow \langle v_1, v_2 \rangle^{\ell}; g_1 \otimes g_2 \oplus [\ell]; h_1 \cup h_2 \cup \{(\ell, \text{loc}(v_1)), (\ell, \text{loc}(v_2))\}} \text{ C-FORK}$$

$$\frac{e \Downarrow \langle v_1, v_2 \rangle^{\ell}; g; h \qquad (n \text{ fresh})}{\#i\ e \Downarrow v_i; g \oplus [n]; h \cup \{(n, \ell)\}} \text{ C-PROJ}_i \qquad \frac{(n \text{ fresh})}{v \Downarrow v; [n]; \emptyset} \text{ C-VAL} \qquad \frac{(n \text{ fresh})}{() \Downarrow \langle\rangle; [n]; \emptyset} \text{ C-UNIT}$$

$$\frac{(\ell \text{ fresh})}{\text{fun } f(x) = e \Downarrow \langle f.x.e \rangle^{\ell}; [\ell]; \bigcup_{\ell' \in \text{locs}(e)} (\ell, \ell')} \text{ C-FUN}$$

$$\frac{\begin{array}{c} e_1 \Downarrow \langle f.x.e_3 \rangle^{\ell_1}; g_1; h_1 \\ e_2 \Downarrow v_2; g_2; h_2 \qquad [\langle f.x.e_3 \rangle^{\ell_1}/f][v_2/x]e_3 \Downarrow v_3; g_3; h_3 \qquad (n \text{ fresh}) \end{array}}{e_1\ e_2 \Downarrow v_3; g_1 \oplus g_2 \oplus [n] \oplus g_3; h_1 \cup h_2 \cup h_3 \cup \{(n, \ell_1), (n, \text{loc}(v_2))\}} \text{ C-APP}$$

are added to the heap graph to represent the dependencies of the pair on each of its components: if the pair is reachable, so is each component. This is an example where edges in the heap graph record dependencies among values. An example of the cost graphs for this rule are displayed in Figure 2.5.

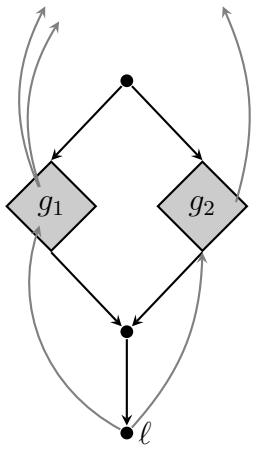
Edges in the computation graph (black) point downward. This graph consists of two sub-graphs g_1 and g_2 (one for each component) and three additional nodes. The first node, at the top, represents the cost of forking a new parallel computation, and the second node, in the middle, represents the cost of joining these parallel threads. The final node ℓ represents the cost of the allocation of the pair. Heap edges (gray) point toward the top of the page. Those two heap edges originating from the node ℓ represent the dependency of the pair on each of its components. Note that these components need not have been allocated as the final step in the sub-graph associated with either component.

Projections from a pair are written as a single rule (C-PROJ$_i$) where $i \in \{1, 2\}$. A heap edge is added to represent the use of a value: memory associated with the pair itself cannot be reclaimed until after the selected component is projected.

Evaluating a value (C-VAL) requires a unit of computation (corresponding to the cost of looking up a variable in the environment) but has no (direct) impact on memory use. The unit expression () is evaluated similarly (C-UNIT).

The evaluation of a function (C-FUN) requires one unit of computation to allocate the value representing the function (*e.g.*, the closure); that value depends on each of values that have already been substituted into the body of the function. As this semantics uses substitution, closures

Figure 2.5 Example Cost Graphs. These graphs arise from the evaluation of a pair where each component of that pair may be evaluated in parallel.



will not include an explicit environment but merely serve to account for the memory required in such an environment.

The application of recursive functions is evaluated as usual, by first evaluating the function and its argument, and then substituting the function and the argument into the body of the function, using standard capture-avoiding substitution [Barendregt, 1984]. (Alternatively, I could define a semantics that would allow the function and its argument to be evaluated in parallel, but I choose to isolate parallel evaluation in a single construct.)

The costs associated with function application reflect this order of evaluation. Before evaluating the body of the function, one additional node is added to the computation graph, along with two heap edges, to represent this use of the function and its argument. In this case, no new values are allocated; instead these heap edges represent dependencies on values by other parts of the program state. That is, these heap edges represent *uses* of values. The first such edge marks a use of the function. The second edge denotes a possible last use of the argument. For strict functions, this second edge is redundant: there will be another edge leading to the argument when it is used. However, for non-strict functions, this is the first point at which the garbage collector might reclaim the memory associated with the argument. The ways in which heap edges constrain the memory use of a language implementation will be discussed in more detail in Section 2.6 below.

It is important to note that this cost semantics is *deterministic*. That is, it assigns one pair of graphs to each (terminating) closed expression. This factors reasoning about parallel programs into two phases. First, it enables programmers to reason about the performance programs abstractly and independently of the scheduling policy. For example, the number of nodes and the longest path in the computation graph can be used to bound the performance of arbitrary scheduling policies. Second, programmers can also use cost graphs to reason about the performance of particular scheduling policies as will be described in the next section.

Figure 2.6 Substitution. These equations give the definition of substitution for the core language. This is a standard, capture-avoiding substitution.

$$
\begin{array}{lcll}
[v/x]x & = & v & \\
[v/x]y & = & y & (\text{if } x \neq y) \\
[v/x]() & = & () & \\
[v/x]\mathsf{fun}\ f(y) = e & = & \mathsf{fun}\ f(y) = [v/x]e & (\text{if } x \neq y \text{ and } x \neq f) \\
[v/x]e_1\ e_2 & = & [v/x]e_1\ [v/x]e_2 & \\
[v/x]\{e_1, e_2\} & = & \{[v/x]e_1, [v/x]e_2\} & \\
[v/x](\#i\ e) & = & \#i\ ([v/x]e) & \\
[v/x]v' & = & v' & (\text{where } v' \text{ is a value})
\end{array}
$$

2.5 Schedules and Deviations

Together, the computation and heap graphs allow programmers to analyze the behavior of programs under a variety of hardware and scheduling configurations. Key components of this analysis are the notions of schedules and deviations.

Recall that nodes in a computation graph represent events that occur during the evaluation of a program. In our analysis, each such event is single reduction of a sub-expression. In some implementations of our language, these might also correspond to the execution of an instruction or the reading or writing of a value from a memory location. Each schedule describes one possible parallel execution of the program by recording the order in which these events occur. Two or more events may occur simultaneously if they occur on different processors, so a schedule is defined by a partial order of these events. We assume that any given schedule is defined for a fixed set of processors. A schedule is defined formally as follows.

Definition (Schedule)**.** A **schedule** $\mathcal{S}$ of a graph g on processors P is defined by a partial order $\leq_{\mathcal{S}}$ on pairs of processors and nodes of g such that,

- $\forall n \in \text{nodes}(g).\ \exists p \in P.\ (p, n) \leq_{\mathcal{S}} (p, n)$,
- $\forall n_1, n_2 \in \text{nodes}(g).\ n_1 \prec_g n_2 \Rightarrow \exists p, q \in P.\ (p, n_1) <_{\mathcal{S}} (q, n_2)$,
- $\forall p \in P.\ \forall n_1, n_2 \in \text{nodes}(g).\ (p, n_1) \leq_{\mathcal{S}} (p, n_2) \wedge (p, n_2) \leq_{\mathcal{S}} (p, n_1) \Rightarrow n_1 = n_2$, and
- $\forall n \in \text{nodes}(g).\ \forall p, q \in P.(p, n) \leq_{\mathcal{S}} (q, n) \Rightarrow p = q$.

Where $<_{\mathcal{S}}$ is the corresponding strict partial order given by the reflexive reduction of $\leq_{\mathcal{S}}$.

We say that a node n is **visited** by p if $(p, n) \leq_{\mathcal{S}} (p, n)$. An instance $(p, n_1) \leq_{\mathcal{S}} (q, n_2)$ of this partial order can be read as "p visits n_1 no later than q visits n_2." I will sometimes write $\text{visits}_{\mathcal{S}}(p, n)$ as shorthand if processor p visits node n in the schedule $\mathcal{S}$. Informally, a schedule determines the order in which nodes are visited and which processor visits each node. The definition above requires that every node in g is visited. It also requires that, if we ignore the processor component, the corresponding strict partial order is a sub-order of the precedence relation given by the edges of g. It follows that the start node of a graph is the first node visited in any schedule, and the end node is the last node visited. Finally, a processor visits at most one node at a time, and each node is visited by exactly one processor.

One might also consider a schedule as a mapping from nodes to the pairs of processors and

steps of time. Such a map defines which processor visits each node and when that node is visited, and can be derived from the definition above. In Section 2.6, I use the order $<_{\mathcal{S}}$ defining a schedule to define a more time-centric view of a parallel execution. I have chosen to define schedules in terms of an order rather than a map because it allows a natural definition deviations (as given in the next section) and because it also allows concise descriptions of different kinds of scheduling policies (as shown in the next chapter).

We extend the partial order $\leq_{\mathcal{S}}$ defining a schedule to sets of nodes in a straightforward way. Given two sets of nodes N_1 and N_2, N_1 occurs before N_2 if every node in N_1 is visited before every node in N_2.

$$N_1 <_{\mathcal{S}} N_2 \text{ iff } \forall n_1 \in N_1, n_2 \in N_2.\ \exists p, q.\ (p, n_1) <_{\mathcal{S}} (q, n_2)$$

The partial order defining a schedule gives a global view of all the events that occur during the evaluation of a program. In some cases, it will be useful to consider more narrow views of the events that occur in a schedule. First, we can view a schedule from a particular moment in time and disregard any information about which processor visits each node. Two nodes n_1, n_2 are visited **simultaneously** (written $n_1 \diamond_{\mathcal{S}} n_2$) in a schedule $\mathcal{S}$ if neither n_1 nor n_2 is visited before the other.

$$n_1 \diamond_{\mathcal{S}} n_2 \text{ iff } \forall p, q \in P.\ (p, n_1) \not<_{\mathcal{S}} (q, n_2) \wedge (q, n_2) \not<_{\mathcal{S}} (p, n_1)$$

We shall use this notion of simultaneity to reason about memory use in the next section. Second, we can also consider the events in a schedule from the perspective of a single processor, ignoring nodes visited by other processors. For every schedule, each processor p gives rise to a total order $<_p$ of the nodes visited by that processor. It is defined as follows,

$$n_1 <_p n_2 \text{ iff } (p, n_1) <_{\mathcal{S}} (p, n_2)$$

Note that a single processor need not relate all nodes in the computation graph but just the nodes visited by that processor. In cases where I write $n_1 <_p n_2$ the schedule $\mathcal{S}$ will be clear from context.

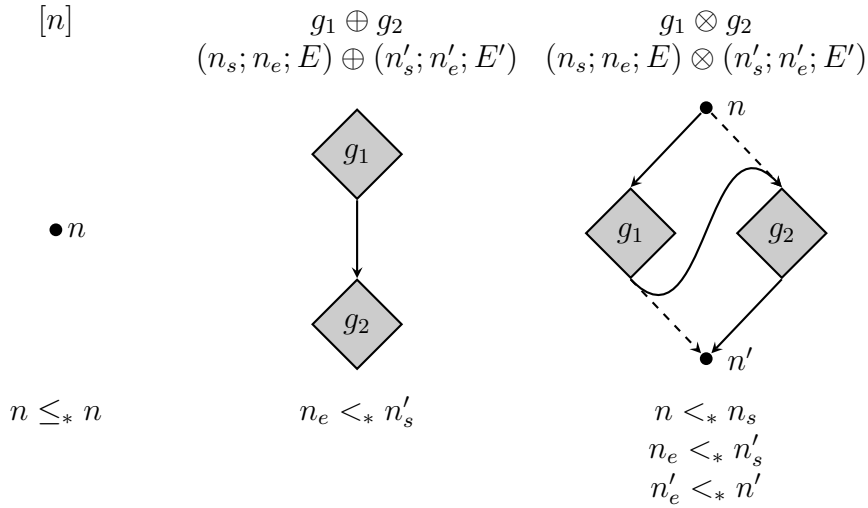
Schedules enable us to capture the essential properties of scheduling policies, without considering how they might be implemented: we can omit details such the number of queues, synchronization mechanisms, and communication protocols. Instead, schedules define the behaviors of these policies abstractly. For example, the fair scheduler mentioned in the example above can be described simply as a breadth-first traversal of the computation graph. In addition to online scheduling policies, schedules of computation graphs can also capture the behavior of compilation techniques, such as flattening [Blelloch and Sabot, 1990], that determine the order of evaluation of parallel tasks. These connections will be explored in the subsequent chapters.

2.5.1 Sequential Schedules

One schedule will play an important role in assessing the performance of different implementations. Given a graph g, the **sequential schedule** is a schedule $\mathcal{S}^*$ that assigns tasks to exactly one processor, *i.e.*, $P = \{*\}$. Though there may be many schedules for a given computation

graph that assign nodes to only a single processor, only one of these corresponds to the standard left-to-right, call-by-value semantics. Here, we define the sequential schedule $\mathcal{S}^*$ based on the structure of the graph g. The order $\leq_{\mathcal{S}^*}$ defining this schedule is the same as the order $\leq_*$ in which nodes are visited by the sole processor in that schedule, and I will use $<_*$ to denote both orders.

The graphs below show how $\leq_*$ is defined for single nodes and for graphs built from the serial or parallel composition of smaller graphs.[1] In these graphs, dashed edges are part of g but not $\leq_*$. For single nodes and graphs formed by serial composition, the $\leq_* = \prec_g$. For parallel compositions $g_1 \otimes g_2$, the end node n_e of g_1 is visited before the start node n'_s of g_2. Given the edges in g_1 and g_2 this implies that every node in g_1 is also visited before any node in g_2.



It is straightforward to show that this defines a unique total order on the nodes of g.

2.5.2 Deviations

The total orders given by each processor give rise to a notion of adjacency between nodes (or alternatively a covering relation on nodes). Given a total order on nodes $<_p$ we say that two nodes n_1 and n_2 are **adjacent** (written $n_1 \prec_p n_2$) if $n_1 <_p n_2$ and there exists is no node n_3 such that $n_1 <_p n_3$ and $n_3 <_p n_2$. With this, we now have a framework in place to formally define when a parallel evaluation of a program deviates from the sequential schedule. Following Acar et al. [2000], we define deviations from the sequential schedule as follows.[2]

Definition (Deviation)**.** A **deviation** occurs whenever a node n_1 is visited by a processor p and there exists a node n_2 such that $n_2 \prec_* n_1$ but $n_2 \not\prec_p n_1$.

In other words, if n_1 is visited immediately after n_2 in the sequential schedule, a deviation occurs in a parallel schedule whenever (i) the processor that visits n_1 does not visit n_2, (ii) that processor visits one or more additional nodes between n_2 and n_1, or (iii) that processor visits n_2

[1] In this section, I view descriptions of graphs such as $g_1 \otimes g_2$ as syntax and define $\leq_*$ as a function of this syntax.
[2] Acar et al. use the term "drifted node" instead of deviation.

after n_1. We define the total deviations of a schedule as the sum of the number of deviations of each processor.

This definition can be motivated in a general sense as follows. If our goal is to understand the performance of parallel schedules in terms of the performance of the sequential schedule, then we first need to understand when and how these parallel schedules differ from the sequential schedule. Deviations measure exactly these differences.

Though deviations are not necessarily bad, they do imply an interaction with the scheduler and therefore some additional overhead. It is possible that some schedules that deviate from the sequential schedule may perform better than the sequential implementation (achieving a super-linear speed-up). However, as will be shown in this dissertation, the number of deviations can be used to bound the worst-case overhead of a parallel implementation. In addition, there are many applications where the number of deviations together with the sequential performance can be used to determine a lower bound on parallel overhead.

2.6 Memory Use

In this section, I shall show how a heap graph can be used to reconstruct how memory is used by different schedules and how the number of deviations can be used to bound memory use. Just as the computation graphs reflect how an implementation of a scheduling policy would evaluate a program, the heap graphs reflect how a garbage collector (or another memory management technique) would reclaim memory. To understand the memory use of a schedule $\mathcal{S}$, we first partition the set of nodes in the computation graph into a sequence of sets of simultaneously visited nodes. We call such a partition the **steps** of the schedule.

$$\text{steps}(\mathcal{S}) = N_1, \ldots, N_k \text{ such that } \begin{cases} \forall n \in \text{nodes}(g).\ \exists N_i . n \in N_i, \\ N_1 <_{\mathcal{S}} \ldots <_{\mathcal{S}} N_k, \text{ and} \\ \forall 1 \leq i \leq k, \forall n_1, n_2 \in N_i.\ n_1 \diamond_{\mathcal{S}} n_2 \end{cases}$$

The first two conditions ensure that these sets define a partition: no node is related to itself by the strict partial order $<_{\mathcal{S}}$. This partition is unique because every node in N_i is visited simultaneously and is therefore incomparable with every other node in N_i. Thus no node in N_i can be moved to any other set that is visited before or after N_i.

For any step N_i we define the closure $\widehat{N_i}$ with respect to the schedule $\mathcal{S}$ as the set of all the nodes that have been visited up to and including that step.

$$\widehat{N_i} = \bigcup_{j=1}^{j \leq i} N_j$$

While in general, nodes in $\widehat{N_i}$ are not visited simultaneously, it is still the case that $\widehat{N_i} <_{\mathcal{S}} N_{i+1}$. With these sets in mind, we can think of a schedule as defining a wavefront that advances across the computation graph, visiting some number of nodes at each step. We say that a node n is **ready** at step $i+1$ if all of its parents appear in $\widehat{N_i}$ but $n \notin \widehat{N_i}$.

Now consider computation and heap graphs g and h as well as one step N_i of a schedule $\mathcal{S}$. Note that for each edge (n, ℓ) in h, we have that $\ell \prec_g n$ and therefore, $\ell <_{\mathcal{S}} n$. It follows that each such edge (n, ℓ) in h will fall into one of the following three categories.

Figure 2.7 Locations of Values and Expressions. Locations are used in measuring the memory required to represent a value and can appear in expressions after a substitution has occurred.

$$
\begin{array}{lcl}
\text{loc}(\langle\rangle) & = & \emptyset \\
\text{loc}(\langle f.x.e\rangle^{\ell}) & = & \ell \\
\text{loc}(\langle v_1, v_2\rangle^{\ell}) & = & \ell \\
 & & \\
\text{locs}(()) & = & \emptyset \\
\text{locs}(\mathsf{fun}\ f(x) = e) & = & \text{locs}(e) \\
\text{locs}(e_1\ e_2) & = & \text{locs}(e_1) \cup \text{locs}(e_2) \\
\text{locs}(\{e_1, e_2\}) & = & \text{locs}(e_1) \cup \text{locs}(e_2) \\
\text{locs}(\#i\ e) & = & \text{locs}(e) \\
\text{locs}(v) & = & \text{loc}(v)
\end{array}
$$

- Both $n, \ell \notin \widehat{N_i}$. As the value associated with ℓ has not yet been allocated, the edge (n, ℓ) does not contribute to memory use at step $\widehat{N_i}$.
- Both $n, \ell \in \widehat{N_i}$. While the value associated with ℓ has been allocated, the use of this value represented by this edge is also in the past. Again, the edge (n, ℓ) does not contribute to memory use at step $\widehat{N_i}$.
- $\ell \in \widehat{N_i}$, but $n \notin \widehat{N_i}$. The value associated with ℓ has been allocated, and n represents a possible use in the future. The edge (n, ℓ) *does* contribute to memory use at step $\widehat{N_i}$.

In the definition below, we must also explicitly account for the location of the final value resulting from evaluation. Though this value may never be used in the program itself, we must include it when computing memory use.

Definition (Roots)**.** Given $e \Downarrow v; g; h$ and a schedule $\mathcal{S}$ of g, the **roots** after visiting the nodes in N and with respect to location ℓ, written $\text{roots}_{h,\ell}(N)$, is the set of nodes $\ell' \in N$ where $\ell' = \ell$ or h contains an edge leading from outside N to ℓ'. Symbolically,

$$\text{roots}_{h,\ell}(N) = \{\ell' \in N \mid \ell' = \ell \vee (\exists n.(n, \ell') \in h \wedge n \notin N)\}$$

The location of a value $\text{loc}(v)$ is the outermost location of a value, as defined in Figure 2.7, and serves to uniquely identify that value. The locations of an expression $\text{locs}(e)$ are the locations of any values that appear in that expression as a result of substitution. As we will often be interested in the roots with respect to a *value*, I will write $\text{roots}_{h,v}(N)$ as an abbreviation for $\text{roots}_{h,\text{loc}(v)}(N)$.

I use the term *roots* to evoke a related concept from work in garbage collection. For the reader that is most comfortable thinking in terms of an implementation, the roots might correspond to those memory locations that are reachable directly from the processor registers or the call stack. This is just one possible implementation, however: the computation and heap graphs stipulate only the *behavior* of an implementation.

The following lemma serves as a sanity check for our definition of roots and of the heap graphs defined in the cost semantics. (It will also be useful in proofs in the next chapter.) Note

that the cost semantics is defined over expressions that include values and thus $\text{locs}(e)$ (as it appears below) is not necessarily empty. Informally, the lemma states that if we start from the locations appearing in an expression and use the definition of roots and the heap graph to determine the amount of memory initially required, we get the same result as if we had considered the term itself.

Lemma 2.1 (Initial Roots)**.** If $e \Downarrow v; g; h$ then $\text{locs}(e) = \text{roots}_{h,v}(\text{locs}(e))$.

Proof. By induction on the given derivation.
Case C-FORK**:** Here, we have $\text{locs}(e) = \text{locs}(e_1) \cup \text{locs}(e_2)$. For $i \in \{1, 2\}$, the induction hypothesis states that $\text{locs}(e_i) = \text{roots}_{h_i,v_i}(\text{locs}(e_i))$. It suffices to show that for each $\ell \in \text{locs}(e)$ either $\ell = \text{loc}(v)$ or that there exist an edge $(n, \ell) \in h$ such that $n \notin \text{locs}(e)$. Consider one location ℓ' such that $\ell' \neq \text{loc}(v)$. Without loss of generality, assume that $\ell' \in \text{locs}(e_1)$, and therefore $\ell' \in \text{roots}_{h_1,v_1}(\text{locs}(e_1))$. It follows that either $\ell' = \text{loc}(v_1)$ or there exists an edge $(n, \ell') \in h_1$ such that $n \notin \text{locs}(e_1)$. Take the first case; by construction, there is an edge $(\ell, \text{loc}(v_1))$ and $\ell \notin \text{locs}(e)$ since ℓ was chosen to be fresh. In the second case, the edge (n, ℓ') is also in h and it remains to show that $n \notin \text{locs}(e_2)$. Since $(n, \ell') \in h_1$ it follows that $n \in \text{nodes}(g_1)$ and since nodes in the computation graph are chosen to be fresh, $n \notin \text{locs}(e_2)$.
Case C-PROJ$_i$**:** In this case, $e = \#i\ e'$ and $\text{locs}(e) = \text{locs}(e')$. As a sub-derivation, we have $e' \Downarrow \langle v_1, v_2 \rangle^{\ell}; g'; h'$ with $h = h' \cup (n, \ell)$. Inductively, $\text{locs}(e') = \text{roots}_{h',\ell}(\text{locs}(e'))$. It is sufficient to show that for each $\ell' \in \text{locs}(e')$ either $\ell' = \text{loc}(v)$ or there exists an edge $(n', \ell') \in h$ such that $n' \notin \text{locs}(e')$. Assume that $\ell' \neq \text{loc}(v)$. Since $\ell' \in \text{locs}(e')$ we have $\ell' \in \text{roots}_{h',\ell}(\text{locs}(e'))$. Then either $\ell' = \ell$ or there exists an edge $(n', \ell') \in h'$ such that $n' \notin \text{locs}(e')$. In the first case, we have an edge $(n, \ell') \in h$ and since n was chosen to be fresh $n \notin \text{locs}(e')$. In the second case, the required edge is also in h since $h' \subseteq h$.
Case C-VAL**:** In this case, $\text{locs}(e) = \text{loc}(v)$. From the definition, $\text{roots}_{h,v}(\{\text{loc}(v)\}) = \{\text{loc}(v)\}$.
Case C-UNIT**:** Since $\text{locs}(e) = \emptyset$, the result is immediate.
Case C-FUN**:** In this case, $e = \mathsf{fun}\ f(x) = e'$ and $\text{locs}(e) = \text{locs}(e')$. Because $\text{loc}(v) = \ell$ and there are edges in h from ℓ to each $\ell' \in \text{locs}(e')$, it follows that $\text{roots}_{h,v}(\text{locs}(e')) = \text{locs}(e')$.
Case C-APP**:** Here, $e = e_1\ e_2$. Each of the locations in $\text{locs}(e)$ comes from e_1 or e_2. As in the case for C-FORK, we apply the inductive hypothesis to each sub-expression and then show that each location in $\text{locs}(e_i)$ is also in $\text{roots}_{h,v}(e)$, either because of an edge appearing in one of the heap sub-graphs or because of one of the edges added in this rule. □

Roots describe only those values which are *immediately* reachable from the current program state. To understand the total memory required to represent a value, we must consider all reachable nodes in the heap graph. We write $\ell_1 \preccurlyeq_h \ell_2$ if there is a (possibly empty) path from ℓ_1 to ℓ_2 in h. We can now define the memory required by values and during the execution of schedules. Though these definitions do not precisely account for the memory required to represent values in some implementations of this language, they will still enable us to account for the asymptotic memory use of programs.

Definition (Memory Required by a Value)**.** The **memory required by a value** v is the set of nodes in a heap graph h reachable from the location of v.

$$\text{memory}_h(v) = \{\ell \in h \mid \text{loc}(v) \preccurlyeq_h \ell\}$$

Definition (Memory Required at a Step)**.** The **memory required at a step** N of a schedule which eventually computes a value v is the set of nodes in the heap graph h that are reachable starting from the nodes in $\text{roots}_{h,v}(\widehat{N})$.

$$\text{memory}_{v,h}(\widehat{N}) = \{\ell \in h \mid \exists \ell' \in \text{roots}_{h,v}(\widehat{N}).\ell' \preccurlyeq_h \ell\}$$

I define $\overline{\max}(\{A_1, \ldots, A_k\})$ for sets $A_1, \ldots, A_k$ to be the largest such set A_i for $1 \leq i \leq k$ (or if there is more than one largest set, arbitrary one of these largest sets).

Definition (Memory Required by a Schedule)**.** The **memory required by a schedule** $\mathcal{S}$ that computes a value v on a heap graph h is the largest of the sets of nodes defined as the memory required at each step in that schedule.

$$\text{memory}_{v,h}(\mathcal{S}) = \overline{\max}(\{\text{memory}_{v,h}(\widehat{N}) \mid N \in \text{steps}(\mathcal{S})\})$$

Bounding Memory Use. Given these definitions, we can now use the number of deviations to bound the memory required by a parallel schedule $\mathcal{S}$ in terms of the memory required by the sequential schedule $\mathcal{S}^*$. The key insight is that between two deviations, a single processor can require no more memory than would be required by the sequential schedule. We can therefore quantify the total memory use in terms of the number of deviations and the memory required by the sequential schedule. Though this bound is quite large, there do exist programs and scheduling policies for which it is tight, as demonstrated by the example below.

Theorem 2.1 (Memory Upper Bound)**.** Given $e \Downarrow v; g; h$ and a schedule $\mathcal{S}$ of g, if $\mathcal{S}$ deviates from the sequential schedule d times, then the amount memory required to execute $\mathcal{S}$ is given as,

$$|\text{memory}_{v,h}(\mathcal{S})| \leq (d+1)|\text{memory}_{v,h}(\mathcal{S}^*)|$$

where $\mathcal{S}^*$ denotes the sequential schedule of g.

Proof. By induction on the derivation of $e \Downarrow v; g; h$. The case for pairs is shown here. As the other cases do not introduce any potential parallelism, there are no additional opportunities to deviate from the sequential schedule.

Case C-FORK**:** Let the parallel schedules for each subgraph be denoted $\mathcal{S}_1$ and $\mathcal{S}_2$, and the sequential schedules $\mathcal{S}_1^*$ and $\mathcal{S}_2^*$. The amount of memory required for the sequential schedule is,

$$\text{memory}_{v,h}(\mathcal{S}^*) = \overline{\max}(\text{memory}_{v_1,h}(\mathcal{S}_1^*), \text{memory}_{v_2,h}(\mathcal{S}_2^*) \cup \text{memory}_h(v_1))$$

since it will require at most the memory required to visit each node the first subgraph, or to visit each node in the second subgraph while holding on to the result of the first. (Recall that the sequential schedule corresponds to a left-to-right evaluation strategy.) Note that some of the reachable values in the first half of the schedule may also be reachable in the second half of the schedule. In other words, there may be sharing between the computations described by g_1 and g_2. This is accounted for by taking the larger of the amount of memory required for either half.

Inductively, we also know that memory required for each parallel schedule is bounded by the memory required for each corresponding sequential schedule.

$$|\text{memory}_{v_1,h}(\mathcal{S}_1)| \leq (d_1+1)|\text{memory}_{v_1,h}(\mathcal{S}_1^*)|$$
$$|\text{memory}_{v_2,h}(\mathcal{S}_2)| \leq (d_2+1)|\text{memory}_{v_2,h}(\mathcal{S}_2^*)|$$

Let n_e be the end node of g_1 and n_s be the start node of g_2. Note that $n_e \prec_* n_s$. We now consider whether or not any additional deviations occur in $\mathcal{S}$ that do not occur in either $\mathcal{S}_1$ or $\mathcal{S}_2$. If not, then $d = d_1 + d_2$. It follows that no deviation occurs at n_s and that there is some processor p for which $n_e \prec_p n_s$. Given the structure of g_1 and g_2, we know that $\mathcal{S}$ must therefore visit every node in g_1 before any node in g_2. As in the case of the sequential schedule, it follows that the memory required by $\mathcal{S}$ can be written as follows.

$$|\text{memory}_{v,h}(\mathcal{S})| = |\overline{\max}(\text{memory}_{v_1,h}(\mathcal{S}_1), \text{memory}_{v_2,h}(\mathcal{S}_2) \cup \text{memory}_h(v_1))|$$

Or equivalently,

$$|\text{memory}_{v,h}(\mathcal{S})| = \max(|\text{memory}_{v_1,h}(\mathcal{S}_1)|, |\text{memory}_{v_2,h}(\mathcal{S}_2) \cup \text{memory}_h(v_1)|)$$

Then inductively, we know that,

$$\leq \max\left((d_1+1)|\text{memory}_{v_1,h}(\mathcal{S}_1^*)|, (d_2+1)|\text{memory}_{v_2,h}(\mathcal{S}_2^*) \cup \text{memory}_h(v_1)|\right)$$

Because both d_1 and d_2 are less than or equal to d, we have,

$$\begin{aligned}
&\leq \max\left((d+1)|\text{memory}_{v_1,h}(\mathcal{S}_1^*)|, (d+1)|\text{memory}_{v_2,h}(\mathcal{S}_2^*) \cup \text{memory}_h(v_1)|\right) \\
&= (d+1)\max\left(|\text{memory}_{v_1,h}(\mathcal{S}_1^*)|, |\text{memory}_{v_2,h}(\mathcal{S}_2^*) \cup \text{memory}_h(v_1)|\right) \\
&= (d+1)|\text{memory}_{v,h}(\mathcal{S}^*)|
\end{aligned}$$

Alternatively, at least one deviation occurs in $\mathcal{S}$ that does not occur in either $\mathcal{S}_1$ or $\mathcal{S}_2$. Thus $d \geq d_1 + d_2 + 1$. Note that this occurs when a single processor that visits g_2 before g_1 or when two or more processors visit the nodes of g_1 and g_2 in parallel. In either case, we can bound the amount of memory required as the sum of the amounts of memory required for each half.

$$|\text{memory}_{v,h}(\mathcal{S})| \leq |\text{memory}_{v_1,h}(\mathcal{S}_1)| + |\text{memory}_{v_2,h}(\mathcal{S}_2)|$$

Note that this is no less than $\max(|\text{memory}_{v_2,h}(\mathcal{S}_2^*)|, |\text{memory}_{v_1,h}(\mathcal{S}_1^*) \cup \text{memory}_h(v_2)|)$, the amount of memory required in the case were no parallel evaluation occurs, but g_2 is visited before g_1. Inductively, we have,

$$\leq (d_1+1)|\text{memory}_{v_1,h}(\mathcal{S}_1^*)| + (d_2+1)|\text{memory}_{v_2,h}(\mathcal{S}_2^*)|$$

The memory use of each half is no greater than the maximum of the two. (We also are free to include locations from the result of the first subgraph.)

$$\begin{aligned}
&\leq (d_1 + 1)\left(\max\left(|\text{memory}_{v_1,h}(\mathcal{S}_1^*)|, |\text{memory}_{v_2,h}(\mathcal{S}_2^*) \cup \text{memory}_h(v_1)|\right)\right) \\
&\quad + (d_2 + 1)\left(\max\left(|\text{memory}_{v_1,h}(\mathcal{S}_1^*)|, |\text{memory}_{v_2,h}(\mathcal{S}_2^*) \cup \text{memory}_h(v_1)|\right)\right)
\end{aligned}$$

By redistributing and substituting d, we have the desired result:

$$\begin{aligned}
&= (d_1 + 1 + d_2 + 1) \max\left(|\text{memory}_{v_1,h}(\mathcal{S}_1^*)|, |\text{memory}_{v_2,h}(\mathcal{S}_2^*) \cup \text{memory}_h(v_1)|\right) \\
&\leq (d + 1) \max\left(|\text{memory}_{v_1,h}(\mathcal{S}_1^*)|, |\text{memory}_{v_2,h}(\mathcal{S}_2^*) \cup \text{memory}_h(v_1)|\right) \\
&= (d + 1)|\text{memory}_{v,h}(\mathcal{S}^*)|
\end{aligned}$$

□

This bound is tight in the sense that we can construct programs where a parallel schedule with d deviations requires nearly $(d + 1)$ times the memory required by the sequential schedule. (The $\log d$ term accounts for the different between the state required for a single processor and multiple processors.)

Theorem 2.2 (Memory Lower Bound)**.** Given d such that $d = 2^n$ for some $n \geq 1$, there exists a family of expressions e with $e \Downarrow v; g; h$ and schedules $\mathcal{S}$ of g such that

$$|\text{memory}_{v,h}(\mathcal{S})| = \Omega((d + 1)(|\text{memory}_{v,h}(\mathcal{S}^*)| - \log d))$$

where $\mathcal{S}^*$ denotes the sequential schedule of g.

Proof. The proof is given by constructing such a family of expressions. We assume the existence of an expression e_{big} that requires m units of memory under the sequential schedule but that evaluates to the unit value (which itself requires no space). Given d, we define e as the expression that computes nested pairs of elements (in parallel) where each leaf is e_{big} and there are d leaves. For example, if $d = 4$ then e is defined as,

$$\{\{e_{\text{big}}, e_{\text{big}}\}, \{e_{\text{big}}, e_{\text{big}}\}\}$$

Using the sequential schedule, this expression will require at most $m + 3$ units of memory. The three additional units correspond to the maximum size of the set of roots. In general, the sequential execution will require $m + \log d + 1$ units of space.

There is a parallel schedule for this example where each of the four instances of that expression are evaluated in parallel, resulting in nearly perfect speed-up. This can be achieved with three deviations. Note that each of these four expressions will require m units of memory. Furthermore, at any point at which these expressions are being evaluated in parallel, there are at least 4 nodes in the root set (one for each processor). Thus the total memory required by the parallel schedule for this example is $4m + 4$ units. In general, there is a schedule that uses d processors and requires $dm + d = (d + 1)m$ units of memory. Plugging in the value of m, we achieve the desired bound. □

Note that this bound is more widely applicable than that given by Blumofe and Leiserson [1993]. First, it applies to programs that use memory in an arbitrary fashion as, for example, in a garbage collected language such as ML or Java. (In the work of Blumofe and Leiserson, memory must be managed using a stack discipline.) Second, as my analysis uses deviations and not steals as the basis for deriving memory use, it can be applied to arbitrary scheduling policies, not just work-stealing policies.

2.7 Sequential Extensions

To better understand the cost semantics described above, I extend the language and the cost semantics with several *sequential* constructs (Figure 2.8). The proofs of both Theorem 2.1 and Lemma 2.1 can be extended to include these additional rules. As these constructs introduce no additional opportunities for parallel evaluation, the extension of the proof of Theorem 2.1 is straightforward. Cases for the extended proof of Lemma 2.1 are discussed briefly below.

First, I add syntax for sequentially constructed pairs. The difference from the parallel rule shown above (C-FORK) is the use of sequential graph composition $\oplus$ instead of parallel composition $\otimes$. Though deterministic parallelism encourages programmers to express many parallel tasks, in some cases it is still necessary for programmers to control the granularity of these tasks. Using sequentially constructed pairs, programmers can write purely sequential code.

Second, I add booleans and conditional expressions to the language. Though these offer no possibilities for parallel evaluation, they shed some light on how this cost semantics reflects the way programs use memory. The rules to evaluate constant booleans (C-TRUE, C-FALSE) are straightforward, but the rules for evaluating conditional expressions require some explanation.[3] Take the case of C-IFTRUE. As expected, after evaluating e_1 to true (or more precisely, a value true^{ℓ_1}), we evaluate e_2. We add an additional node n to the computation graph that corresponds to the cost of testing the boolean value and taking the branch. In addition, we add heap graph edges from n to the values of e_3, the branch that is *not* taken. These edges correspond to possible last uses of the values that have been substituted into e_3. By adding these edges, I allow implementations to maintain these values until the branch is taken. As discussed below, these edges are critical in ensuring that the specification given by the cost semantics can be implemented. The rule C-IFFALSE is analogous.

These edges play a role when extending the proof of Lemma 2.1 (Initial Roots) when e is $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$. As the locations of a conditional expression may come from either branch, we must ensure that they are also in the initial roots. In C-IFTRUE, the locations of e_2 are included in the set of roots as in previous cases. For each of the locations ℓ in e_3, there is an edge from n to each ℓ and, because n does not appear in the locations of e, each of these locations is also in the roots of e.

It is possible to give a cost semantics that does *not* include these edges that point to the values of the branch not taken. However, giving an implementation faithful to such a cost semantics would be impossible: it would require the language implementation (*e.g.*, the garbage collector) to predict which branch will be taken for each conditional expression in the program. (Further-

[3]Note that one might also consider a cost semantics where boolean values do carry locations.

more, it must make such predictions without consuming any additional resources.) Thus, my cost semantics is designed, in part, with an eye for the implementation. These additional edges distinguish what might be called "true" garbage from "provable" garbage: though we would like to measure the memory required by only those values that will actually be used in the remainder of the computation, we cannot expect an implementation to efficiently determine that set of values. Instead, we settle for an approximation of those values based on reachability.

As I shall make explicit in the next chapter, this cost semantics maintains an invariant that at each point in a schedule, the roots of the heap graph correspond to values that must be preserved given information available *at that point in time*. Thus the heap graph encodes not only information about values in the heap, but also information about *uses* of those values over time.

2.8 Alternative Rules

There are a number of design choices for in the rules given in Figures 2.4 and 2.8. In addition to the examples where an alternatives rule would be too strict (as in the case of conditionals), it is also possible for a cost semantics to yield constraints that are too lax.

Consider as an example the following alternative rule for the evaluation of function application. The premises remain the same, and the only difference from the conclusion in Figure 2.4 is highlighted with a rectangle.

$$\frac{e_1 \Downarrow \langle f.x.e_3 \rangle^{\ell_1}; g_1; h_1 \qquad e_2 \Downarrow v_2; g_2; h_2 \qquad \cdots}{e_1\ e_2 \Downarrow v_3; g_1 \oplus g_2 \oplus \boxed{g_3 \oplus [n]}; h_1 \cup h_2 \cup h_3 \cup \{(n, \ell_1), (n, \mathrm{loc}(v_2))\}}$$

This rule yields the same extensional result as the version given in Figure 2.4, but it admits more implementations. Recall that the heap edges (n, ℓ_1) and $(n, \mathrm{loc}(v_2))$ represent possible last uses of the function and its argument. This variation of the rule moves those dependencies until *after* the evaluation of the function body.

This rule allows implementations to preserve these values, even in the case where they are not used during the function application or in subsequent evaluation. This includes the case where these values are bound to variables that do not appear in the program text of the function body or after the application. This is precisely the constraint described by Shao and Appel [1994] as "safe-for-space."

In contrast, the original version of this rule requires that these values be considered for reclamation by a garbage collector as soon as the function is applied. Note, that the semantics does not specify *how* the implementation makes these values available for reclamation: it is left to the implementer to determine whether this is achieved through closure conversion, by clearing slots in the stack frame, or by some other means.

As this example suggests, there is some leeway in the choice of the semantics itself. My goal was to find a semantics that describes a set of common implementation techniques. When they fail to align, my experience suggests that either the semantics or the implementation can be adjusted to allow them to fit together.

2.9 Summary

In this chapter, I have defined the language of study used in this thesis and given a novel cost semantics that describes *how* programs in this language may be evaluated, and in particular, constrains how implementations of this language may use memory. These costs can also be used by programmers to analyze the performance of parallel programs. This analysis is factored into two parts: it first assigns a pair of graphs to each closed program which can be used to reason about program performance independently of the implementation; programmers can also fix a scheduling policies to derive more precise performance bounds. A key part of this analysis is the identification of *deviations*, or points where a parallel schedule diverges from the sequential schedule. Using deviations, I give a novel bound on the memory use of arbitrary scheduling policies. I have further explored this cost semantics in how it constrains implementations of both parallel and sequential expressions, and I have given an example of how this cost semantics can formalize a notion of "safe-for-space."

2.10 Related Work

Directed graphs are a natural representation of parallel programs. Graphs introduced by Blumofe and Leiserson [1993] are similar to my computation graphs, though my computation graphs are simpler and do not expose the structure of threads. Furthermore, Blumofe and Leiserson's work did not allow for general heap structures such as those modeled by my heap graphs.

Extended language semantics were used by Rosendahl [1989], Sands [1990], and Roe [1991] to automatically derive bounds on the time required to execute programs. In all cases, the semantics yields a *cost* that approximates the intensional behavior of programs. Ennals [2004] uses a cost semantics to compare the work performed by a range of sequential evaluation strategies, ranging from lazy to eager. As in this thesis, he uses cost graphs with distinguished types of edges, though his edges serve different purposes. He does not formalize the use of memory by these different strategies. Gustavsson and Sands [1999] also use a cost semantics to compare the performance of sequential, call-by-need programs. They give a semantic definition of what it means for a program transformation to be safe-for-space [Shao and Appel, 1994] and provide several laws to help prove that a given transformation does not asymptotically increase the space use of programs. To prove the soundness of these laws, they use a semantics that yields the maximum heap and stack space required for execution. In the context of a call-by-value language, Minamide [1999] showed that a CPS transformation is space-efficient using a cost semantics. Sansom and Peyton Jones [1995] used a cost semantics in the design of their profiler.

Cost semantics are also used to describe provably efficient implementations of speculative parallelism [Greiner and Blelloch, 1999] and explicit data-parallelism [Blelloch and Greiner, 1996]. These cost semantics yield directed, acyclic graphs that describe the parallel dependencies, just as the computation graphs in this thesis. Execution time on a bounded parallel machine is given in terms of the (parallel) depth and (sequential) work of these graphs. In one case [Blelloch and Greiner, 1996], an upper bound on memory use is also given in terms of depth and work. This work was later extended to a language with recursive product and sum types [Lechtchinsky et al., 2002]. Acar et al. [2000] defined drifted nodes and used that definition to bound the

number of cache misses for a work-stealing scheduling policy. These works, however, have all assumed a fixed scheduling policy.

Sized types [Hughes et al., 1996, Hughes and Pareto, 1999] are a static counterpart to cost semantics and provide a decidable approximation of the resource use of a program. Loidl and Hammond [1996] extend this work to a parallel functional language. Jay et al. [1997] describe a static framework for reasoning about the costs of parallel execution using a monadic language. Static cost models have also been used to choose automatically a parallel implementation during compilation based on hardware performance parameters [Hammond et al., 2003] and to inform the granularity of scheduling [Portillo et al., 2002]. Unlike this thesis, the latter work focuses on how the size of program data structures affect parallel execution (*e.g.* through communication costs), rather than how different parallel schedules affect the use of memory at a given point in time.

Figure 2.8 Sequential Language Extensions. This figure extends the core language by adding booleans, conditional expressions, and pairs whose components are always evaluated in series.

$$\begin{array}{llll} \text{(expressions)} & e & ::= & \ldots \mid (e_1,e_2) \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\ \text{(values)} & v & ::= & \ldots \mid \mathsf{true}^\ell \mid \mathsf{false}^\ell \end{array}$$

(a) Syntax.

$$\frac{e_1 \Downarrow v_1; g_1; h_1 \qquad e_2 \Downarrow v_2; g_2; h_2 \qquad (\ell \text{ fresh})}{(e_1,e_2) \Downarrow \langle v_1, v_2\rangle^\ell; g_1 \oplus g_2 \oplus [\ell]; h_1 \cup h_2 \cup \{(\ell, \mathrm{loc}(v_1)), (\ell, \mathrm{loc}(v_2))\}}\ \textsc{c-Pair}$$

$$\frac{(\ell \text{ fresh})}{\mathsf{true} \Downarrow \mathsf{true}^\ell; [\ell]; \emptyset}\ \textsc{c-True} \qquad \frac{(\ell \text{ fresh})}{\mathsf{false} \Downarrow \mathsf{false}^\ell; [\ell]; \emptyset}\ \textsc{c-False}$$

$$\frac{e_1 \Downarrow \mathsf{true}^{\ell_1}; g_1; h_1 \qquad e_2 \Downarrow v_2; g_2; h_2 \qquad (n \text{ fresh})}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v_2; g_1 \oplus [n] \oplus g_2; h_1 \cup h_2 \cup \{(n,\ell_1)\} \cup \bigcup_{\ell \in \mathrm{locs}(e_3)} (n,\ell)}\ \textsc{c-IfTrue}$$

$$\frac{e_1 \Downarrow \mathsf{false}^{\ell_1}; g_1; h_1 \qquad e_3 \Downarrow v_3; g_3; h_3 \qquad (n \text{ fresh})}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v_3; g_1 \oplus [n] \oplus g_3; h_1 \cup h_3 \cup \{(n,\ell_1)\} \cup \bigcup_{\ell \in \mathrm{locs}(e_2)} (n,\ell)}\ \textsc{c-IfFalse}$$

(b) Cost Semantics.

$$\begin{array}{lcl} [v/x](e_1,e_2) & = & ([v/x]e_1,[v/x]e_2) \\ [v/x]\mathsf{true} & = & \mathsf{true} \\ [v/x]\mathsf{false} & = & \mathsf{false} \\ [v/x]\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & = & \mathsf{if}\ [v/x]e_1\ \mathsf{then}\ [v/x]e_2\ \mathsf{else}\ [v/x]e_3 \end{array}$$

(c) Substitution.

$$\begin{array}{lcl} \mathrm{loc}(\mathsf{true}^\ell) & = & \ell \\ \mathrm{loc}(\mathsf{false}^\ell) & = & \ell \\ \\ \mathrm{locs}((e_1,e_2)) & = & \mathrm{locs}(e_1) \cup \mathrm{locs}(e_2) \\ \mathrm{locs}(\mathsf{true}) & = & \emptyset \\ \mathrm{locs}(\mathsf{false}) & = & \emptyset \\ \mathrm{locs}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) & = & \mathrm{locs}(e_1) \cup \mathrm{locs}(e_2) \cup \mathrm{locs}(e_3) \end{array}$$

(d) Locations.

Chapter 3

Online Scheduling

While the cost semantics given in the previous section allows a programmer to draw conclusions about the performance of programs, these conclusions would be meaningless if the implementation of the language did not reflect the costs given by that semantics. In this section, I define several *provable implementations* [Blelloch and Greiner, 1996] of this language, each as a transition (or small-step) semantics. At the same time, I will show how the behavior of each semantics can be described as a refinement of a schedule as defined in the previous chapter.

The first semantics is a non-deterministic semantics that defines all possible parallel executions. Each subsequent semantics will define a more constrained behavior and in many cases serve as a guide to a lower-level implementation of a particular scheduling policy. The following table gives a brief overview of the semantics that appear in Chapters 2 and 3.

Semantics (Figure)	Judgment(s)	Notes
Cost (2.4)	$e \Downarrow v; g; h$	big-step, sequential, profiling semantics
Primitive (3.2)	$e \longrightarrow e'$	axioms shared among parallel implementations
Non-Deterministic (3.4)	$e \overset{\text{nd}}{\longmapsto} e'$ $d \overset{\text{nd}}{\longmapsto} d'$	defines all possible parallel executions
Depth-First (3.6)	$e \overset{\text{df}}{\longmapsto} e'$ $d \overset{\text{df}}{\longmapsto} d'$	algorithmic implementation favoring left-most sub-expressions
Breadth-First (3.8)	$e \overset{\text{bf}}{\longmapsto} e'$ $d \overset{\text{bf}}{\longmapsto} d'$ $e \equiv^{\text{bf}} e'$ $d \equiv^{\text{bf}} d'$	implementation favoring expressions at least depth
Work-Stealing (3.11)	$e \overset{\text{df-ws}}{\longmapsto} e'$ $d \overset{\text{df-ws}}{\longmapsto} d'$ $e \equiv^{\text{ws}} e'$ $d \equiv^{\text{ws}} d'$	order of execution derived from one task queue per processor

As part of the implementation of this language, I extend the syntax to include a parallel let construct. Parallel lets and underlined expressions $\underline{e}$ are used to denote expressions that have

Figure 3.1 Syntax Extensions for Scheduling Semantics. Declarations as well as **let** and underlined expressions are used to represent the intermediate steps of parallel evaluation.

(declarations)	d	$::=$	$x = e \mid d_1$ and d_2
(value declarations)	δ	$::=$	$x = v \mid \delta_1$ and δ_2
(expressions)	e	$::=$	$\ldots \mid$ let d in $e \mid \underline{e}$

been partially evaluated. Note that this new syntax is only meant to reflect the intermediate state of the implementations described in this chapter. In particular, it is *not* meant to provide the programmer with any additional expressive power. Like values, these new forms of syntax are never written by programmers.

Declarations within a let may step in parallel, depending on the constraints enforced by one of the transition semantics below. Declarations and let expressions reify a stack of expression contexts such as those that appear in many abstract machines (*e.g.* [Landin, 1964]). Unlike a stack, which has exactly one topmost element, there are many "leaves" in our syntax that may take transitions in parallel; thus declarations form a cactus stack [Moses, 1970]. These extensions are shown in Figure 3.1. Declarations either bind a single variable as in $x = e$ or are parallel composition of two sub-declarations as in d_1 and d_2. In the case of d_1 and d_2, the variables bound are the union of those bound in d_1 and d_2. I maintain an invariant that each variable bound in a declaration is distinct from all other variables in the expression. Furthermore, each variable bound in a declaration will appear in the body of a let expression of exactly once.

Underlined expressions are used to mark expressions that have already been expanded as part of a let declaration. Thus the different between an expression such as (v_1,v_2) and $\underline{(v_1,v_2)}$ is that the former will step to a let expression while the latter will step directly to a value representing the pair. Each underlined expression represents a frame on the cactus stack that indicates what operation should be performed once all sub-expressions are evaluated to values.

To simplify the presentation of different scheduling policies, I will factor out those steps that are common to all of the scheduling semantics in this chapter. This will allow us to compare these different semantics more directly. These steps are called **primitive transitions**. Figure 3.2 shows the inference rules defining these transitions, and these rules are explained in detail below.

Values such as functions and booleans that require no further evaluation may step to the resulting value immediately (P-FUN, P-TRUE, P-FALSE). Sequential pairs are expanded into a pair of nested declarations (P-PAIR). This will ensure that the first component of such a pair is always evaluated before the second component. The underlined expression $\underline{(x_1,x_2)}$ denotes an expression which has already be expanded.

Once all declarations have been evaluated to values, the primitive rule P-JOIN can be applied to eliminate bindings in a declaration by substitution. Substitution of value declarations is defined in Figure 3.3.

In the case of pairs, once both components are evaluated to values and substituted using P-JOIN, the expression will be of a form that P-PAIRALLOC can be applied to create a new pair value. Note that there is no expression to which both P-PAIR and P-PAIRALLOC can both be applied. Again, underlined expressions are used to denote those expressions that are playing a second role as frames.

Parallel pairs (P-FORK) bind two fresh variables (one for each component) like the rule for sequential pairs, but these variables are bound using and instead of nested let expressions. This will allow these two sub-expressions to transition in parallel.

Function application (P-APP), projection (P-PROJ$_i$), and conditional statements (P-IF) are all expanded into let expressions in such a way to force sequential execution. Once the appropriate sub-expression(s) have been evaluated, they will be substituted using P-JOIN, as above, and reduced using one of P-APPBETA, P-PROJ$_i$BETA, P-IFTRUE, or P-IFFALSE.

Note that each rule is an axiom and that the rules are syntax-directed: the applicability of a rule can be determined solely by considering the structure of the expression and at most one rule applies to any expression. Also note that primitive transitions do not allow any evaluation of the expressions embedded within declarations (*e.g.*, $x = e$): primitive transitions will only be used as the leaves of a transition derivation. Transitions within a declaration will be handled by the scheduling semantics found in subsequent sections.

3.1 Non-Deterministic Scheduling

The non-deterministic schedule semantics does not serve as a guide for an actual implementation, but as an inclusive definition of all possible parallel implementations. Though this semantics is non-deterministic in *how* it evaluations an expression, it always yields the same result. A proof of this fact is given below.

Figure 3.4 shows the inferences rules for the non-deterministic transition semantics. The non-determinism of this semantics is concentrated in the application of the ND-IDLE rule: it can be applied to *any* expression, and thus for every expression, a choice between two rules must be made. The idle rule also embodies another unorthodox aspect of this transition semantics: it is reflexive. This means that there is an infinite sequence of steps that can be taken by any expression in the language. In contrast, the scheduling semantics found later in this chapter shall always be forced to make some progress. I have found it to be simpler to prove this property about each such semantics individually, rather than the non-deterministic semantics, since each semantics below enforces this property in a different manner. Proving termination for the non-deterministic semantics would require either some global notion of progress or setting some artificial limit on the number of steps that some (sub-)expression could remain unchanged.

The remainder of the rules are relatively straightforward. The primitive rule (ND-PRIM) embeds primitive transitions within the non-deterministic semantics. Parallel declarations may step in parallel (ND-BRANCH). While one can read the ND-BRANCH rule as stating that both branches of a parallel declaration must step in parallel, when taken together with the ND-IDLE, this means that either branch may take a step forward or both (or neither). The remaining rules, ND-LET and ND-LEAF, allow transitions to be make on sub-expressions.

3.1.1 Confluence

As noted above, one important property of the non-deterministic semantics (and therefore every semantics that is consistent with it) is that despite that fact that there are many ways of evaluating an expression, all of these sequences of transitions yield the same result. To prove this fact, we

Figure 3.2 Primitive Transitions. These transitions will be shared among the scheduling semantics defined in the chapter. Substitution is defined in Figures 2.6 and 3.3.

$\boxed{e \longrightarrow e'}$

$$\frac{(\ell \text{ fresh})}{\mathsf{fun}\ f(x) = e \longrightarrow \langle f.x.e \rangle^{\ell}}\ \textsc{p-Fun} \qquad \frac{(\ell \text{ fresh})}{\mathsf{true} \longrightarrow \mathsf{true}^{\ell}}\ \textsc{p-True} \qquad \frac{(\ell \text{ fresh})}{\mathsf{false} \longrightarrow \mathsf{false}^{\ell}}\ \textsc{p-False}$$

$$\frac{(\ell \text{ fresh})}{(e_1,e_2) \longrightarrow \mathsf{let}\ x_1 = e_2\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ \underline{(x_1,x_2)}}\ \textsc{p-Pair} \qquad \frac{}{\mathsf{let}\ \delta\ \mathsf{in}\ e \longrightarrow [\delta]e}\ \textsc{p-Join}$$

$$\frac{(\ell \text{ fresh})}{\underline{(v_1,v_2)} \longrightarrow \langle v_1, v_2 \rangle^{\ell}}\ \textsc{p-PairAlloc} \qquad \frac{(x_1, x_2 \text{ fresh})}{e_1\ e_2 \longrightarrow \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ \underline{x_1\ x_2}}\ \textsc{p-App}$$

$$\frac{(x_1, x_2 \text{ fresh})}{\{e_1, e_2\} \longrightarrow \mathsf{let}\ x_1 = e_1\ \mathsf{and}\ x_2 = e_2\ \mathsf{in}\ \underline{(x_1,x_2)}}\ \textsc{p-Fork}$$

$$\frac{}{\underline{\langle f.x.e \rangle^{\ell}\ v_2} \longrightarrow [\langle f.x.e \rangle^{\ell}/f, v_2/x]e}\ \textsc{p-AppBeta} \qquad \frac{(x \text{ fresh})}{\#i\ e \longrightarrow \mathsf{let}\ x = e\ \mathsf{in}\ \underline{\#i\ x}}\ \textsc{p-Proj}_i$$

$$\frac{}{\underline{\#i\ \langle v_1, v_2 \rangle^{\ell}} \longrightarrow v_i}\ \textsc{p-Proj}_i\textsc{Beta}$$

$$\frac{(x_1 \text{ fresh})}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \longrightarrow \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \underline{\mathsf{if}\ x_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3}}\ \textsc{p-If}$$

$$\frac{}{\underline{\mathsf{if}\ \mathsf{true}^{\ell}\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3} \longrightarrow e_2}\ \textsc{p-IfTrue} \qquad \frac{}{\underline{\mathsf{if}\ \mathsf{false}^{\ell}\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3} \longrightarrow e_3}\ \textsc{p-IfFalse}$$

will use a method based on that given by Huet [1980]. We first prove a lemma that will be used in the proof of confluence below.

Lemma 3.1 (Strong Confluence)**.** If $e \overset{\text{nd}}{\longmapsto} e'$ and $e \overset{\text{nd}}{\longmapsto} e''$ then there exists an expression e''' such that $e' \overset{\text{nd}}{\longmapsto}{}^{*} e'''$ and $e'' \overset{\text{nd}}{\longmapsto} e'''$. Similarly, $d \overset{\text{nd}}{\longmapsto} d'$ and $d \overset{\text{nd}}{\longmapsto} d''$ then there exists a declaration d''' such that $d' \overset{\text{nd}}{\longmapsto}{}^{*} d'''$ and $d'' \overset{\text{nd}}{\longmapsto} d'''$.

Pictorially, given the solid arrows, we would like to show the existence of e''', d''', and the dashed arrows.

Figure 3.3 Substitution for Declarations. This figure defines substitution for declarations and **let** expressions. Substitution of declarations is a critical part of the P-JOIN primitive transition.

$$
\begin{array}{lcl}
[v/x](\mathsf{let}\ d\ \mathsf{in}\ e) & = & \mathsf{let}\ [v/x]d\ \mathsf{in}\ [v/x]e \\
[v/x]\underline{e} & = & \underline{[v/x]e} \\
\\
[v/x](x = e) & = & x = [v/x]e \\
[v/x](d_1\ \mathsf{and}\ d_2) & = & [v/x]d_1\ \mathsf{and}\ [v/x]d_2 \\
\\
[x = v]e & = & [v/x]e \\
[\delta_1\ \mathsf{and}\ \delta_2]e & = & [\delta_1, \delta_2]e
\end{array}
$$

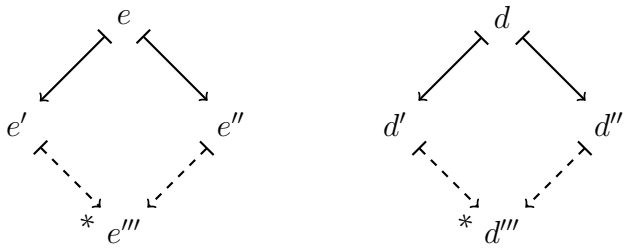
Figure 3.4 Non-Deterministic Transition Semantics. These rules define all possible parallel executions of an expression.

$\boxed{e \overset{\text{nd}}{\longmapsto} e'}$

$$\frac{}{e \overset{\text{nd}}{\longmapsto} e}\ \text{ND-IDLE} \qquad \frac{e \longrightarrow e'}{e \overset{\text{nd}}{\longmapsto} e'}\ \text{ND-PRIM} \qquad \frac{d \overset{\text{nd}}{\longmapsto} d'}{\mathsf{let}\ d\ \mathsf{in}\ e \overset{\text{nd}}{\longmapsto} \mathsf{let}\ d'\ \mathsf{in}\ e}\ \text{ND-LET}$$

$\boxed{d \overset{\text{nd}}{\longmapsto} d'}$

$$\frac{e \overset{\text{nd}}{\longmapsto} e'}{x = e \overset{\text{nd}}{\longmapsto} x = e'}\ \text{ND-LEAF} \qquad \frac{d_1 \overset{\text{nd}}{\longmapsto} d_1' \qquad d_2 \overset{\text{nd}}{\longmapsto} d_2'}{d_1\ \mathsf{and}\ d_2 \overset{\text{nd}}{\longmapsto} d_1'\ \mathsf{and}\ d_2'}\ \text{ND-BRANCH}$$



Proof. By simultaneous induction on the derivations of $e \overset{\text{nd}}{\longmapsto} e'$ and $d \overset{\text{nd}}{\longmapsto} d'$.

Case ND-IDLE**:** In this case $e' = e$. Assume that $e \overset{\text{nd}}{\longmapsto} e''$ was derived using rule R. Let $e''' = e''$. Then we have $e \overset{\text{nd}}{\longmapsto} e''$ (by applying R) and $e'' \overset{\text{nd}}{\longmapsto} e''$ (by ND-IDLE), as required.

As all of the non-determinism in this semantics is focused in the use of the ND-IDLE rule, the remaining cases follow from the immediate application of the induction hypothesis or similar use of the ND-IDLE rule. One example is shown here.

Case ND-LET**:** In this case $e = \mathsf{let}\ d\ \mathsf{in}\ e_1$ and $e' = \mathsf{let}\ d'\ \mathsf{in}\ e_1$ with $d \overset{\text{nd}}{\longmapsto} d'$. Assume that

$e \overset{\text{nd}}{\longmapsto} e''$ by rule R. There are two sub-cases:

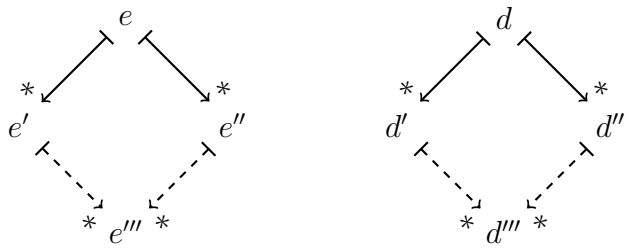
Sub-case R **=** ND-LET**:** Then we have $e'' = \mathsf{let}\ d''\ \mathsf{in}\ e_1$ and $d \overset{\text{nd}}{\longmapsto} d''$. Applying the induction hypothesis, there exists d''' with $d' \overset{\text{nd}}{\longmapsto}{}^* d'''$ and $d'' \overset{\text{nd}}{\longmapsto} d'''$. Applying ND-LET (iteratively) to e' and (once) to e'', we obtain the desired result.

Sub-case R **=** ND-IDLE**:** Here we have $e'' = e$. Let $e''' = e''$. The required transitions are achieved by applying ND-IDLE to e' on the left, and ND-LET to e'' on the right. □

One way of reading the diagrams above is to say that whenever $e' \overset{\text{nd}}{\longmapsto}{}^* e'''$ then e'' can "catch up" in one step. As these diagrams can be flipped left-for-right, we also know that whenever $e'' \overset{\text{nd}}{\longmapsto}{}^* e'''$ (though possibly a different e''') then it is also the case that e' can catch up in one step. Thus the intuition behind strong confluence is that while two transitions may each take an expression in two different directions, neither one of these transitions gets very far "ahead" of the other.

Given strong confluence, we can now prove confluence: when starting from a single expression (or declaration), no matter how far apart two sequences of transitions get, they can always find their way back to a common expression.

Theorem 3.1 (Confluence)**.** If $e \overset{\text{nd}}{\longmapsto}{}^* e'$ and $e \overset{\text{nd}}{\longmapsto}{}^* e''$ then there exists an expression e''' such that $e' \overset{\text{nd}}{\longmapsto}{}^* e'''$ and $e'' \overset{\text{nd}}{\longmapsto}{}^* e'''$. Similarly, $d \overset{\text{nd}}{\longmapsto}{}^* d'$ and $d \overset{\text{nd}}{\longmapsto}{}^* d''$ then there exists a declaration d''' such that $d' \overset{\text{nd}}{\longmapsto}{}^* d'''$ and $d'' \overset{\text{nd}}{\longmapsto}{}^* d'''$. Pictorially,



Proof. Following Huet, let $e \overset{\text{nd}}{\longmapsto}{}^n e''$ (*i.e.*, e steps to e'' in n steps non-deterministically). We first prove that if $e \overset{\text{nd}}{\longmapsto} e'$ and $e \overset{\text{nd}}{\longmapsto}{}^n e''$ then there exists an e''' such that $e' \overset{\text{nd}}{\longmapsto}{}^* e'''$ and $e'' \overset{\text{nd}}{\longmapsto} e'''$. This follows by induction on n, using ND-IDLE (or $e' \overset{\text{nd}}{\longmapsto}{}^0 e''$) when $n = 0$ and strong confluence in the inductive step. The inductive step can be shown as follows,

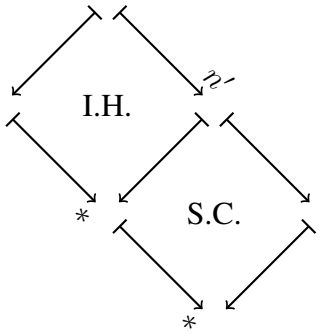


Figure 3.5 Embedding of Declarations into Source Syntax. To relate the behavior of the non-deterministic semantics to the cost semantics, this function maps the extended syntax used in this chapter back to the core syntax found in the previous chapter.

$$
\begin{array}{lcll}
\ulcorner \mathsf{let}\ d\ \mathsf{in}\ e \urcorner & = & & [\ulcorner d \urcorner]\ulcorner e \urcorner \\
\ulcorner \underline{e} \urcorner & = & & e \\
\ulcorner e \urcorner & = & e & \text{(otherwise)} \\
 & & & \\
\ulcorner x = e \urcorner & = & & \ulcorner e \urcorner / x \\
\ulcorner d_1\ \mathsf{and}\ d_2 \urcorner & = & & \ulcorner d_1 \urcorner, \ulcorner d_2 \urcorner
\end{array}
$$

We can then use Huet's lemma 2.3, which states that $\overset{\text{nd}}{\longmapsto}$ is confluent whenever

$$e \overset{\text{nd}}{\longmapsto} e' \wedge e \overset{\text{nd}}{\longmapsto}{}^* e'' \Rightarrow \exists e'''.e' \overset{\text{nd}}{\longmapsto}{}^* e''' \wedge e'' \overset{\text{nd}}{\longmapsto}{}^* e'''$$

This follows immediately from the statement above. □

3.1.2 Soundness

We now turn back to the semantics given in the previous chapter and show that whenever the non-deterministic semantics gives an answer, the same answer can also be derived using the cost semantics. Given that every sequence of transitions of the non-deterministic semantics that leads to a value will yield the *same* value, it suffices to take a single, arbitrarily chosen sequence of transitions.

The transition semantics is defined on a larger set of terms than the evaluation semantics, so we shall define an embedding of declarations and parallel let expressions into the syntax used in the previous chapter. As shown in Figure 3.5, declarations are mapped to substitutions, which are then applied at let expressions. This requires an extension of substitution to arbitrary expressions, not just values. Extending the definition of substitution to such expressions is not difficult, especially as the variables bound by let expressions are chosen so that they never appear anywhere else in the expression. In addition, each variable bound in a declaration appears exactly once in the body of the let expression. (This can be verified by an inspection of the primitive transition rules.) As a consequence, the substitution performed by the embedding will neither duplicate nor eliminate any sub-expressions. Finally, underlined expressions are replaced with their ordinary counterparts.

Theorem 3.2 (ND Soundness)**.** For any closed expression e, if $e \overset{\text{nd}}{\longmapsto}{}^* v$ then there exist some g and h such that $\ulcorner e \urcorner \Downarrow v; g; h$. Similarly, if $d \overset{\text{nd}}{\longmapsto}{}^* \delta$, then there exist g_i and h_i such that $\ulcorner e_i \urcorner \Downarrow v_i; g_i; h_i$ for each $x_i = e_i$ in d and corresponding $x_i = v_i$ in δ.

Proof. By induction on the length of the sequence of non-deterministic transitions. Assume that $e \overset{\text{nd}}{\longmapsto}{}^n v$. (An analogous method applies for $d \overset{\text{nd}}{\longmapsto}{}^* \delta$.)
Case $n = 0$**:** It follows that $e = v$. We have the desired result using the rule C-VAL.

Case $n = n' + 1$**:** In this case we have $e' \overset{\text{nd}}{\longmapsto}^{n'} v$ and $e \overset{\text{nd}}{\longmapsto} e'$. Inductively, $\ulcorner e' \urcorner \Downarrow v; g'; h'$ for some g' and h'. The remainder of this case follows from the following lemma. □

Lemma 3.2. If $e \overset{\text{nd}}{\longmapsto} e'$, $e' \overset{\text{nd}}{\longmapsto}^* v$, and $\ulcorner e' \urcorner \Downarrow v; g'; h'$ then there exist computation and heap graphs g and h such that $\ulcorner e \urcorner \Downarrow v; g; h$. Similarly, if $d \overset{\text{nd}}{\longmapsto} d'$, $d' \overset{\text{nd}}{\longmapsto}^* \delta$, and $\ulcorner e'_i \urcorner \Downarrow v_i; g'_i; h'_i$ then $\ulcorner e_i \urcorner \Downarrow v_i; g_i; h_i$ for each x_i, e'_i, and v_i in d' and δ.

Proof. By simultaneous induction on the derivations of $e \overset{\text{nd}}{\longmapsto} e'$ and $d \overset{\text{nd}}{\longmapsto} d'$.
Case ND-IDLE**:** In this case $e' = e$, and the required result is one of our assumptions.
Case ND-PRIM**:** We must consider each primitive transition. P-PAIR, P-APP, P-PROJ$_i$, P-FORK and P-IF follow immediately since $\ulcorner e' \urcorner = \ulcorner e \urcorner$. In the case of P-JOIN, $\ulcorner e' \urcorner = \ulcorner e \urcorner$ as well. P-FUN follows by application of C-FUN and similarly for P-TRUE and P-FALSE. For the cases of P-PAIRALLOC, P-PROJ$_i$BETA, P-APPBETA, P-IFTRUE, and P-IFFALSE, we apply rules C-PAIR, C-PROJ$_i$, C-APP, C-IFTRUE, and C-IFFALSE (respectively), using the fact that every value is related to itself in the cost semantics.
Case ND-LET**:** Here, e is $\mathsf{let}\ d\ \mathsf{in}\ e_1$ and e' is $\mathsf{let}\ d'\ \mathsf{in}\ e_1$ with $d \overset{\text{nd}}{\longmapsto} d'$ and $\ulcorner e' \urcorner \Downarrow v; g'; h'$. From Lemma 3.3, for each x_i bound to e'_i in d', we have $\ulcorner e'_i \urcorner \Downarrow v_i; g'_i; h'_i$. Each of these x_i also appears in d, and thus we have $e_i \overset{\text{nd}}{\longmapsto} e'_i$. By induction, we have $\ulcorner e_i \urcorner \Downarrow v_i; g_i; h_i$ for each e_i. We then construct a derivation for $\ulcorner e \urcorner \Downarrow v; g; h$ using the result of applying the induction hypothesis as well as sub-derivations of the assumption $\ulcorner e' \urcorner \Downarrow v; g'; h'$ as necessary.
Case ND-LEAF**:** In this case, d' is $x = e'$ and d is $x = e$ with $d \overset{\text{nd}}{\longmapsto} d'$ and $\ulcorner e' \urcorner \Downarrow v; g'; h'$. By a sub-derivation, $e \overset{\text{nd}}{\longmapsto} e'$ and the required result follows immediately from the induction hypothesis.
Case ND-BRANCH**:** Finally, for this case, we have d' is d'_1 and d'_2 and for each expression e'_i that appears in d'_1 or d'_2, we have $\ulcorner e'_i \urcorner \Downarrow v_i; g'_i; h'_i$. As $d_1 \overset{\text{nd}}{\longmapsto} d'_1$ and $d_2 \overset{\text{nd}}{\longmapsto} d'_2$ are sub-derivations, we have the required cost derivations for each expression e that appears in d. □

Lemma 3.3. If $\ulcorner \mathsf{let}\ d\ \mathsf{in}\ e \urcorner \Downarrow v; g; h$ then for all expressions e_i bound in d, there exist values v_i, and graphs g_i and h_i such that $\ulcorner e_i \urcorner \Downarrow v_i; g_i; h_i$.

The proof follows by straightforward induction on the derivation of the cost judgment.

3.1.3 Completeness

The non-deterministic semantics is complete in the sense that for every program that terminates under the evaluation semantics, there is a sequence of transitions to a value in the non-deterministic semantics. In this section, we will prove an even stronger form of completeness that states that for every schedule of the graphs derived from the cost semantics, there is a sequence of transitions to a value.

Recall that I use the term **source expression** to distinguish those forms of expressions written by programmers (and presented in the previous chapter) from those used to represent values and the intermediate state used in the implementation semantics described in this chapter. In particular, let declarations and underlined expressions are *not* source expressions.

Theorem 3.3 (ND Completeness)**.** For any source expression e, if $e \Downarrow v; g; h$ and $\mathcal{S}$ is a schedule of g, then there exists a sequence of expressions $e_0, \ldots, e_k$ such that

- $e_0 = e$ and $e_k = v$,
- $\forall i \in [0, k).\ e_i \overset{\text{nd}}{\longmapsto}{}^* e_{i+1}$, and
- $\forall i \in [0, k].\ \text{locs}(e_i) = \text{roots}_{h,v}(\widehat{N_i})$

where $N_0 = \emptyset$ and $N_1, \ldots, N_k = \text{steps}(\mathcal{S})$.

Recall that each set of nodes N_i is a set of simultaneously visited nodes called a step of the schedule. The theorem states that for each schedule there is a sequence of transitions in the non-deterministic semantics and that the space use as measured by the locations of each expression in that sequence is the same as the space use read from the cost graphs.

Each step of the schedule may require several transitions in the non-deterministic semantics (as indicated by the * in the second item in the theorem). This is due to the fact that there is some overhead in the implementation that is not accounted for in the cost graphs. In this implementation, the overhead is bounded by the size of the expression. Other implementations may have more or less overhead.

To prove this theorem, it must be strengthened to account for partially evaluated expressions e that already contain locations. The theorem follows immediately from the lemma below since for any source expression $\text{locs}(e) = \emptyset$. As the locations of non-source expressions e correspond to evaluation that has already occurred, we include them when computing the roots of a step of the schedule.

Lemma 3.4 (ND Completeness, Strengthened)**.** For any expression e, if $e \Downarrow v; g; h$ and $\mathcal{S}$ is a schedule of g, then there exists a sequence of expressions $e_0, \ldots, e_k$ such that

- $e_0 = e$ and $e_k = v$,
- $\forall i \in [0, k).\ e_i \overset{\text{nd}}{\longmapsto}{}^* e_{i+1}$, and
- $\forall i \in [0, k].\ \text{locs}(e_i) = \text{roots}_{h,v}(\widehat{N_i} \cup \text{locs}(e))$

where $N_0 = \emptyset$ and $N_1, \ldots, N_k = \text{steps}(\mathcal{S})$.

Proof. By induction on the derivation of $e \Downarrow v; g; h$.

Case C-FUN**:** We have e as fun $f(x) = e'$, v as $\langle f.x.e' \rangle^{\ell}$, $g = [\ell]$. Then N_1 must be $\{\ell\}$. Define e_1 as v with a transition defined by ND-PRIM along with P-FUN. Note that $\text{roots}_{h,v}(\widehat{N_0} \cup \text{locs}(e)) = \text{locs}(e_0)$ by Lemma 2.1, and $\text{roots}_{h,v}(\widehat{N_1} \cup \text{locs}(e)) = \text{loc}(v) = \{\ell\}$ because v is the result of evaluation.

Case C-FORK**:** In this case e is $\{e^L, e^R\}$ and $g = (g_1 \otimes g_2) \oplus [n]$. We first construct a sequence of expressions related by the non-deterministic semantics. Define e_1 to be let $x_1 = e^L$ and $x_2 = e^R$ in (x_1,x_2). We have e_0 transitions to e_1 by ND-PRIM and P-FORK.

We have sub-derivations $e^L \Downarrow v_1; g_1; h_1$ and $e^R \Downarrow v_2; g_2; h_2$. If we restrict $\mathcal{S}$ to the nodes of g_1 and g_2, we obtain schedules $\mathcal{S}^L$ and $\mathcal{S}^R$ for these sub-graphs. Inductively, we have that $e^L \overset{\text{nd}}{\longmapsto}{}^{k_1} v_1$ and $e^R \overset{\text{nd}}{\longmapsto}{}^{k_2} v_2$. Let $k' = \max(k_1, k_2)$ and define two new sequences of expressions of length k' and indexed by $[1, k' + 1]$ as follows. Define e_1^L as e^L. If $N_2 \cap \text{nodes}(g^L) = \emptyset$ then let $e_2^L = e_1^L$, otherwise let e_2^L be the next expression in the sequence given by the induction hypothesis. In the first case, we have $e_1^L \overset{\text{nd}}{\longmapsto} e_2^L$ by ND-IDLE, and in the second case by the rule

given inductively. Define each subsequent expression e_i and transition using $N_i \cap \text{nodes}(g^L)$. Define a sequence of expressions and transitions for e^R analogously.

For each $i \in [2, k'+1]$, define e_i using these sequences as $\mathsf{let}\ x_1 = e_i^L\ \mathsf{and}\ x_2 = e_i^R\ \mathsf{in}\ \underline{(x_1,x_2)}$. (Note that e_1 defined in this manner would match the definition we used above.) We use the transitions above and apply rules ND-LET, ND-BRANCH, and ND-LEAF to the let expression at each step. By construction $e_{k'+1}$ is $\mathsf{let}\ x_1 = v_1\ \mathsf{and}\ x_2 = v_2\ \mathsf{in}\ \underline{(x_1,x_2)}$. Let $e_{k'+2}$ be $\underline{(v_1,v_2)}$ and note that $e_{k'+1}$ transitions to $e_{k'+2}$ by P-JOIN and ND-PRIM. Let $k = k' + 3$ and define e_k as $\langle v_1, v_2\rangle^\ell$. It follows that $e_{k'+2} \overset{\text{nd}}{\longmapsto} e_k$ by rules P-PAIRALLOC and ND-PRIM.

Now must show that the locations of each expression in this sequence match the roots of each step of the schedule. By Lemma 2.1, $\text{roots}_{h,v}(\widehat{N_0} \cup \text{locs}(e)) = \text{locs}(e_0)$. Let n_s and n_e be the start and end nodes of g. Then $N_1 = \{n_s\}$ since $n_s \prec_g n$ for all other nodes $n \in \text{nodes}(g)$. Note that $\text{locs}(e_1) = \text{locs}(e_0)$. As n_s does not appear in any edge $(n_s, \ell) \in h$, it follows that $\text{roots}_{h,v}(\widehat{N_1} \cup \text{locs}(e)) = \text{roots}_{h,v}(\widehat{N_0} \cup \text{locs}(e))$.

For $i \in [2, k'+1]$ the locations of e_i are exactly the locations $\text{locs}(e_i^L) \cup \text{locs}(e_i^R)$. From the induction hypothesis, the locations of the left-hand expression $\text{locs}(e_i^L) = \text{roots}_{h_1,v_1}((\widehat{N_i} \cap \text{nodes}(g^L)) \cup \text{locs}(e^L))$, and similarly for the right-hand expression. To show that these locations ℓ' are also in the roots of the parallel let expression, we must consider the cases where $\ell' = \text{loc}(v_1)$ or not. If $\ell' = \text{loc}(v_1)$ then, from the rule, there is an edge $(n, \ell') \in h$ with $n \notin N_i$. If not, then there is an edge $(n', \ell') \in h_1 \subseteq h$ with $n' \notin (N_i \cap \text{nodes}(g^L))$. However, $n' \in \text{nodes}(g^L)$ and therefore $n' \notin N_i$. A similar argument follows for locations in the right-hand sequence of expressions.

Finally, since $n \prec_g n_e \prec_g \ell$ for all other nodes $n \in \text{nodes}(g)$, it must be the case that $N_{k-1} = \{n_e\}$ and $N_k = \{\ell\}$. For the penultimate step, it is the case that $\text{locs}(\underline{(v_1,v_2)}) = \{\text{loc}(v_1), \text{loc}(v_2)\}$ which, given the edges of h, is exactly the roots of N_{k-1} since $\ell \notin \widehat{N_{k-1}}$. For the final result, $\ell = \text{loc}(v)$ is in the roots of N_i by the first clause of the definition of roots().

Case C-VAL**:** In this case $e = v$ so $\text{locs}(e) = \text{loc}(v)$. There is only one step in $\mathcal{S}$, namely $N_1 = \{n\}$. We have $e \overset{\text{nd}}{\longmapsto} v$ by ND-IDLE. By Lemma 2.1, $\text{roots}_{h,v}(\widehat{N_o} \cup \text{loc}(v)) = \text{loc}(v)$. From the definition of roots, $\text{roots}_{h,v}(\widehat{N_1} \cup \text{loc}(v)) = \text{loc}(v)$.

Case C-APP**:** In this case e is $e^F\ e^A$ and $g = g_1 \oplus g_2 \oplus [n] \oplus g_3$. We have sub-derivations $e^F \Downarrow \langle f.x.e\rangle^\ell; g_1; h_1$ and $e^A \Downarrow v_2; g_2; h_2$ and thus, inductively, $e^F \overset{\text{nd}}{\longmapsto}{}^{k_1} \langle f.x.e\rangle^\ell$ and $e^A \overset{\text{nd}}{\longmapsto}{}^{k_2} v_2$. Note that e steps to $\mathsf{let}\ x_1 = e^F\ \mathsf{in}\ \mathsf{let}\ x_2 = e^A\ \mathsf{in}\ \underline{x_1\ x_2}$ by ND-PRIM along with P-APP. We then apply rules ND-LET and ND-LEAF, along with the transitions given inductively from the first sub-derivation. Let e_{k_1} be the **let** expression derived from the last expression in that sequence $\mathsf{let}\ x_1 = \langle f.x.e\rangle^\ell\ \mathsf{in}\ \mathsf{let}\ x_2 = e^A\ \mathsf{in}\ \underline{x_1\ x_2}$. From here, we apply the transitions derived inductively from the second sub-derivation. This yields $e_{k_1+k_2}$ as $\mathsf{let}\ x_1 = \langle f.x.e\rangle^\ell\ \mathsf{in}\ \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ \underline{x_1\ x_2}$. Let $e_{k'}$ be $[\langle f.x.e\rangle^\ell/f, v_2/x]e$. We have $e_{k_1+k_2}$ steps to $e_{k'}$ by two applications of P-JOIN and one application of P-APPBETA (along with ND-PRIM). We also have $[\langle f.x.e\rangle^\ell/f, v_2/x]e \Downarrow v; g_3; h_3$ as a sub-derivation and therefore, $[\langle f.x.e\rangle^\ell/f, v_2/x]e \overset{\text{nd}}{\longmapsto}{}^{k_3} v$, inductively. We use these transitions directly to achieve the desired result.

We now show that memory use is accurately modeled by the roots of each step of the schedule. Note that if we restrict $\mathcal{S}$ to the nodes of g_1, g_2, or g_3 we obtain schedules $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ for these sub-graphs, respectively. For the first k_1 steps, the locations of the expression are those

that appear in e^F, one of its reductions, or e^A. Inductively, each of the locations in e^F (or one of the expressions to which it transitions) is in the roots of $\mathcal{S}_1$ and so there must be an edge in g_1 corresponding to that root or that location must be the location ℓ of the resulting function value. In the later case, the location is also in the roots of the composite heap graph because of the edge (n, ℓ). Similarly for $\mathcal{S}_2$ and g_2.

Note that the locations of $\mathsf{let}\ x_1 = \langle f.x.e\rangle^\ell\ \mathsf{in}\ \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ \underline{x_1\ x_2}$ are exactly $\{\ell, \mathrm{loc}(v_2)\}$ and that these correspond to the roots at the step before the node n is visited. Finally, we know that the locations of the remaining expressions match the roots of the remaining steps by applying the induction hypothesis to the third sub-derivation.

Case C-PROJ$_i$**:** In this case e is $\#i\ e^P$ and $g = g_1 \oplus [n]$. We have a sub-derivation $e^P \Downarrow \langle v_1, v_2\rangle^\ell; g; h$ and therefore, inductively, a sequence of transitions $e^P \overset{\mathrm{nd}}{\longmapsto}{}^{k'} \langle v_1, v_2\rangle^\ell$. e steps to $\mathsf{let}\ x = e\ \mathsf{in}\ \underline{\#i\ x}$ by ND-PRIM and P-PROJ$_i$. The sequence of transitions derived inductively gives rise to a sequence of transitions on this **let** expression, defined by applying ND-LET and ND-LEAF to each transition. Let e_{k-1} be the last expression in this sequence, or more specifically, $\mathsf{let}\ x = \langle v_1, v_2\rangle^\ell\ \mathsf{in}\ \underline{\#i\ x}$. Using rules P-JOIN and P-PROJ$_i$BETA (along with ND-PRIM in both cases), this steps to v_i, as required.

We can derive a schedule $\mathcal{S}_1$ by restricting $\mathcal{S}$ to the nodes of g_1. Inductively, the roots of the steps of this schedule match the locations of each expression in the transition sequence. Each such root corresponds to an edge in g_1 or is the location ℓ of the pair itself. It follows that each such location is also a root in the corresponding step of $\mathcal{S}$ because g contains every edge in g_1 plus an edge from n to ℓ. After the final step of the schedule, we have the roots equal to $\mathrm{loc}(v_i)$ since v_i is the result of the computation.

Case C-UNIT**:** This case follows trivially since $\mathrm{locs}(e) = \mathrm{loc}(v) = \emptyset$.

Case C-PAIR**:** This case follows as the case for C-FORK with the only exception that since no parallel evaluation occurs between the two sub-expressions, the construction of the sequence of expressions does not require any use of the ND-IDLE rule.

Case C-TRUE**:** In this case $e = \mathsf{true}$ and $v = \mathsf{true}^\ell$. Note that e transitions to v by P-TRUE and ND-PRIM. As $\mathrm{locs}(\mathsf{true}) = \emptyset$, this is trivially equal to $\mathrm{roots}_{h,v}(\widehat{N_0})$. There is only one node in g and so $N_1 = \{\ell\}$. This implies $\mathrm{roots}_{h,v}(\widehat{N_1}) = \mathrm{loc}(v)$.

Case C-FALSE**:** As for C-TRUE.

Case C-IFTRUE**:** In this case, e is $\mathsf{if}\ e^B\ \mathsf{then}\ e^T\ \mathsf{else}\ e^F$ and $g = g_1 \oplus [n] \oplus g_2$ with sub-derivations $e^B \Downarrow \mathsf{true}^{\ell_1}; g_1; h_1$ and $e^T \Downarrow v_2; g_2; h_2$. Inductively, this yields transition sequences $e^B \overset{\mathrm{nd}}{\longmapsto}{}^{k_1} \mathsf{true}^{\ell_1}$ and $e^T \overset{\mathrm{nd}}{\longmapsto}{}^{k_2} v_2$. We have e steps to $\mathsf{let}\ x_1 = e^B\ \mathsf{in}\ \underline{\mathsf{if}\ x_1\ \mathsf{then}\ e^T\ \mathsf{else}\ e^F}$ by P-IF and ND-PRIM. We construct a sequence of transitions using the inductively derived sequence along with ND-LET and ND-LEAF. Let e_{k_1} be $\mathsf{let}\ x_1 = \mathsf{true}^{\ell_1}\ \mathsf{in}\ \underline{\mathsf{if}\ x_1\ \mathsf{then}\ e^T\ \mathsf{else}\ e^F}$. This expression transitions to e^T in two steps (using P-JOIN and P-IFTRUE). The remaining transitions follow from the inductively derived sequence directly.

We now show that every location in any one of these expressions is also a root of the corresponding step of the schedule $\mathcal{S}$. For the first half of the transition sequence, expressions are of the form $\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \underline{\mathsf{if}\ x_1\ \mathsf{then}\ e^T\ \mathsf{else}\ e^F}$ where e_1 is one of the expressions in the sequence derived from the first sub-derivation. Let $\mathcal{S}_1$ be the schedule defined by restricting $\mathcal{S}$ to the nodes in g_1. Each location ℓ of such an expression is either ℓ_1, some other location in e_1, or a location of either e^T or e^F. If $\ell = \ell_1$ then it is also in the roots because of the edge (n, ℓ_1). A similar

argument holds for any location found in e^F. If ℓ is another location in some expression e_1 in the sequence, then it must be in the roots with respect to $\mathcal{S}_1$ and there must be an edge (n', ℓ) in h_1. That edge is also in the composite graph h and so ℓ is also in the roots with respect to $\mathcal{S}$. If $\ell \in \text{locs}(e^T)$ then by the induction hypothesis, we have $\ell \in \text{roots}_{h_2,v_2}(\text{locs}(e^T))$ (*i.e.*, ℓ is in the initial set of roots for the schedule of g_2). It follows that $\ell = \text{loc}(v_2)$ or there exists an edge $(n, \ell) \in h_2$ with $n \notin \text{locs}(e^T)$. In the former case, the result is immediate since v_2 is the result of evaluating the **if** expression. In the latter case, such an edge (n, ℓ) is also in h by construction and, since any such node n does not appear in g_1, ℓ must also be in the roots with respect to $\mathcal{S}$. The connection between locations and roots in the second half of the transition sequence follows inductively from the second sub-derivation.
Case C-IFFALSE**:** As for C-IFTRUE. □

The final cases of this proof (those for **if** expressions) provide a more concrete motivation for the edges added to heap graphs corresponding to those locations that appear in the branch that is *not* taken. (These edges were discussed in Section 2.7.) Without these edges, the theorem would not hold.

3.2 Greedy Scheduling

Given the primitive transitions that are possible in a given step, a **greedy** semantics will always use as many of those transitions as possible. Greedy schedulers are an important class of schedulers and have been studied in some detail (*e.g.*, going back to Graham [1966]). In terms of graphs, greedy schedules visit as many nodes as possible in each step (without violating the other conditions of a schedule). Greediness can be defined as follows.

Definition (Greedy)**.** A schedule $\mathcal{S}$ of a graph g for processors P is **greedy** if

$$\begin{aligned}\forall p, q \in P.\, \forall n_1, n_2 \in \text{nodes}(g).\, (p, n_1) <_{\mathcal{S}} (q, n_2) \\ \Rightarrow \exists n_3 \in \text{nodes}(g).\, (n_3 \prec_g n_2 \wedge n_1 \diamond_{\mathcal{S}} n_3) \\ \vee\, |\{n \mid n \diamond_{\mathcal{S}} n_1\}| = |P|\end{aligned}$$

I say that a schedule is P-greedy if it is greedy for the processors in P. The definition states that a schedule is P-greedy if, whenever it visits n_2 after n_1 then either there was a sequential dependency to n_2 from some node visited simultaneously with n_1, or the schedule visited $|P| - 1$ other nodes simultaneously with n_1.

In many cases, scheduling policies can be defined in terms of a total order on the nodes of a computation graph. Such a policy adheres as closely as possible that order when visiting nodes.

Definition (Order-Based)**.** A schedule $\mathcal{S}$ of a graph g for processors P is **based on the order** $<_1$ if $<_1$ is a total order consistent with $\prec_g$ and

$$\begin{aligned}\forall p, q \in P.\, \forall n_1, n_2 \in \text{nodes}(g).\, (p, n_1) <_{\mathcal{S}} (q, n_2) \\ \Rightarrow n_1 <_1 n_2 \vee \exists n_3 \in \text{nodes}(g).\, (n_3 \prec_g n_2 \wedge n_1 \diamond_{\mathcal{S}} n_3)\end{aligned}$$

By consistent, I mean that $\forall n_1, n_2 \in \text{nodes}(g).\ n_1 \prec_g n_2 \Rightarrow n_1 <_1 n_2$. Any total order that is consistent with $\prec_g$ is also a schedule for a single processor. This definition of an order-based schedule coincides with the definition of a schedule "based on a 1-schedule" as given by [Blelloch et al., 1999].

For any set of processors P and total order $<_1$ on the nodes of a graph g, there is a unique (up to the assignment of processors) P-greedy schedule based on $<_1$. To prove this, we first show that each greedy schedule of g based on $<_1$ gives rise to greedy schedules of the sub-graphs of g also based on $<_1$.

Lemma 3.5 (Sequential, Greedy, Order-Based Decomposition)**.** If $\mathcal{S}$ is a greedy schedule of $g_1 \oplus g_2$ based on a total order $<_1$ then $\mathcal{S}_1 = \mathcal{S} \cap \text{nodes}(g_1)$ is a greedy schedule for g_1 based on $<_1$ and $\mathcal{S}_2 = \mathcal{S} \cap \text{nodes}(g_2)$ is a greedy schedule for g_2 based on $<_1$.

Proof. Assume that $\mathcal{S}$ is a schedule for processors P. First, take the case of g_1. To show that it is greedy, take any processors in P and nodes such that $(p, n_1) <_{\mathcal{S}} (q, n_2)$ with $n_1, n_2 \in \text{nodes}(g_1)$. $\mathcal{S}_1$ is greedy whenever $\exists n_3 \in \text{nodes}(g_1).\ (n_3 \prec_{g_1} n_2 \wedge n_1 \diamond_{\mathcal{S}_1} n_3)$ or $|\{n \mid n \diamond_{\mathcal{S}_1} n_1\}| = |P|$. Because $\mathcal{S}$ is greedy, either $\exists n_3 \in \text{nodes}(g).\ (n_3 \prec_g n_2 \wedge n_1 \diamond_{\mathcal{S}} n_3)$ or $|\{n \mid n \diamond_{\mathcal{S}_1} n_1\}| = |P|$. In either case, such a node n must also appear in g_1 since if n is visited simultaneously with n_1 then n is not in g_2 (since $n_1 \prec_g n$ for every node n in g_2).

To show that g_1 is based on $<_1$, again take any processors and nodes such that $(p, n_1) <_{\mathcal{S}} (q, n_2)$ with $n_1, n_2 \in \text{nodes}(g_1)$. If $\mathcal{S}$ is based on $<_1$ then either $n_1 <_1 n_2$ or there exists a node n_3 such that $(n_3 \prec_g n_2 \wedge n_1 \diamond_{\mathcal{S}} n_3)$. In the first case, we have the desired result immediately. In the second case, it must be that n_3 is also in g_1 since there is no node $n \in \text{nodes}(g_2)$ such that $n \prec_g n_2$.

The proof for $\mathcal{S}_2$ and g_2 follows similarly. When showing that $\mathcal{S}_2$ is based on $<_1$ and there is some node $n_3 \in \text{nodes}(g)$ such that $n_3 \prec_g n_2 \wedge n_1 \diamond_{\mathcal{S}} n_3$, note that $n_3 \notin \text{nodes}(g_1)$ since it is visited simultaneously with a node from g_2 and every node in g_1 is a predecessor of every node in g_2. □

When considering graphs formed by parallel composition, we must account for the fact that processors visiting nodes in one sub-graph cannot simultaneously visit nodes in the other sub-graph. We say that a schedule $\mathcal{S}$ is **restricted** to processors $P_1, \ldots, P_k$ if $\text{steps}(\mathcal{S}) = N_1, \ldots, N_k$ and for all $i \in [1, k]$, the processors used to visit nodes in N_i are a subset of P_i. We also refine the definition of greedy for restricted schedules.

Definition (Restricted Greedy)**.** A schedule $\mathcal{S}$ of a graph g **restricted** to processors $P_1, \ldots, P_k$ is **greedy** if $\text{steps}(\mathcal{S}) = N_1, \ldots, N_k$

$$\begin{aligned}\forall p, q \in P.\ \forall n_1, n_2 \in \text{nodes}(g).\ (p, n_1) <_{\mathcal{S}} (q, n_2) \\ \Rightarrow \exists n_3 \in \text{nodes}(g).\ (n_3 \prec_g n_2 \wedge n_1 \diamond_{\mathcal{S}} n_3) \\ \vee\ (n_1 \in N_i \wedge |N_i| = |P_i|)\end{aligned}$$

With this definition, we can show that any greedy schedule based on a total order for a graph defined by parallel composition can be decomposed into greedy schedules of each sub-graph based on the same order.

Lemma 3.6 (Parallel, Greedy, Order-Based Decomposition)**.** If $\mathcal{S}$ is a greedy schedule of $g_1 \otimes g_2$ based on a total order $<_1$, then

- $\mathcal{S}_1 = \mathcal{S} \cap \text{nodes}(g_1)$ is a greedy schedule of g_1 based on $<_1$ and restricted to processors P_i in step i where P_i is defined as the set of processors p such that there is a node $n \in \text{nodes}(g_1)$ and $\text{visits}_{\mathcal{S}}(p, n)$ in step i, and
- $\mathcal{S}_2 = \mathcal{S} \cap \text{nodes}(g_2)$ is a greedy schedule of g_2 based on $<_1$ and restricted to processors P'_i in step i, defined analogously.

Proof. Let n_s be the start node and n_e the end node of $g_1 \otimes g_2$. Consider $\mathcal{S}_1$ and any two nodes such that $(p, n_1) <_{\mathcal{S}_1} (q, n_2)$. This is the case when both $n_1, n_2 \in \text{nodes}(g_1)$ and $(p, n_1) <_{\mathcal{S}} (q, n_2)$. First, we show that $\mathcal{S}_1$ is based on $<_1$. We know that $\mathcal{S}$ is based on $<_1$ so either $n_1 <_1 n_2$ or $\exists n_3 \in \text{nodes}(g).\ (n_3 \prec_g n_2 \wedge n_1 \diamond_{\mathcal{S}} n_3)$. If $n_1 <_1 n_2$ or (in the second case) $n_3 \in \text{nodes}(g_1)$ then we have immediately that $\mathcal{S}_1$ is based on $<_1$. Assume that $n_3 \notin \text{nodes}(g_1)$. Then n_3 is either n_s, n_e, or in $\text{nodes}(g_2)$. Since $n_3 \prec_g n_2$ and $n_2 \in \text{nodes}(g_1)$, then $n_3 = n_s$. This cannot be the case, however, because n_s cannot be visited simultaneously with n_1 as $n_s \prec_g n_1$.

We now show that $\mathcal{S}_1$ is greedy restricted to the processors used by $\mathcal{S}$ in g_1. Again, we consider two nodes such that $(p, n_1) <_{\mathcal{S}_1} (q, n_2)$. Because $\mathcal{S}$ is greedy then either $\exists n_3 \in \text{nodes}(g).\ (n_3 \prec_g n_2 \wedge n_1 \diamond_{\mathcal{S}} n_3)$ or $|\{n \mid n \diamond_{\mathcal{S}} n_1\}| = |P|$. In the first case, any such n_3 must either be n_s or in g_1. It cannot be the case that $n_3 = n_s$ since $n_3 \diamond_{\mathcal{S}} n_1$. If $n_3 \in \text{nodes}(g_1)$ then it satisfies the required condition. In the second case, assume that $\mathcal{S}_1$ visits nodes N_i simultaneously with n_1. By definition, $N_i = \text{nodes}(g_1) \cap \{n \mid n \diamond_{\mathcal{S}} n_1\}$. Let P_i be the set of processors that visit nodes in g_1 under $\mathcal{S}$ and simultaneously with n_1. It follows immediately that $|N_i| = |P_i|$.

Finally, we show that $\mathcal{S}_2$ is based on $<_1$ and greedy. If $(p, n_1) <_{\mathcal{S}_2} (q, n_2)$ then either $n_1 <_1 n_2$ or there is a node visited simultaneously with n_1 that is a predecessor of n_2. As above, such a node must also be in g_2. Similarly, as $\mathcal{S}$ is greedy, for any pair of nodes with $(p, n_1) <_{\mathcal{S}_2} (q, n_2)$, there must be a predecessor of n_2 in g_2 that is visited simultaneously with n_1 or no additional processors remain. □

With these lemmata, we can now show that every greedy schedule based on a total order $<_1$ is unique up to the assignment of processors.

Theorem 3.4 (Uniqueness of Greedy Schedules Based on $<_1$)**.** If $\mathcal{S}$ and $\mathcal{S}'$ are greedy schedules of g for processors P based on a total order $<_1$, then

$$\begin{aligned} \forall p, q \in P.\ \forall n_1, n_2 \in \text{nodes}(g).\ (p, n_1) \leq_{\mathcal{S}} (q, n_2) \\ \Rightarrow \exists p', q' \in P.\ (p', n_1) \leq_{\mathcal{S}'} (q', n_2) \end{aligned}$$

Proof. By induction on the structure of the graph g.
Case $g = [n]$**:** There is only one node in g so since $\mathcal{S}$ is a schedule, $(p, n_1) \leq_{\mathcal{S}} (q, n_2)$ implies that $n_1 = n_2 = n$ and $p = q$. Likewise for $\leq_{\mathcal{S}'}$ along with p' and q'. The necessary result is immediate.
Case $g = g_1 \oplus g_2$**:** We consider only the case where $n_1 \neq n_2$ as otherwise the result is trivial. By Lemma 3.5, there are greedy schedules for g_1 and g_2 based on $<_1$ and defined as $\mathcal{S} \cap \text{nodes}(g_1)$ and $\mathcal{S} \cap \text{nodes}(g_2)$, respectively. By the induction hypothesis these schedules are unique up to

the assignment of processors, so we must only consider the case where $(p, n_1) \leq_{\mathcal{S}} (q, n_2)$ with one of $\{n_1, n_2\}$ in g_1 and the other in g_2. First, it cannot be the case that $n_2 \in \text{nodes}(g_1)$ and $n_1 \in \text{nodes}(g_2)$ as in that case $n_2 \prec_g n_1$ and $\mathcal{S}$ would not be a valid schedule. Thus we have $n_1 \in \text{nodes}(g_1)$ and $n_2 \in \text{nodes}(g_2)$ and we must show that there exist processors p', q' such that $(p', n_1) \leq_{\mathcal{S}'} (q', n_2)$. This follows immediately since $n_1 \prec_g n_2$ and $\mathcal{S}'$ is a schedule.

Case $g = g_1 \otimes g_2$**:** We consider only the case where $n_1 \neq n_2$ as otherwise the result is again trivial. Let n_s and n_e be the start and end nodes of g (respectively). By Lemma 3.6, there are greedy schedules for g_1 and g_2 based on $<_1$ and defined as $\mathcal{S} \cap \text{nodes}(g_1)$ and $\mathcal{S} \cap \text{nodes}(g_2)$, respectively. By the induction hypothesis, these are unique up to the assignment of processors, so again we must only consider cases where $(p, n_1) \leq_{\mathcal{S}} (q, n_2)$ and n_1 and n_2 do not come from the same sub-graph. The first case is that $n_1 = n_s$ in which case there must exist p', q' such that $(p', n_1) \leq_{\mathcal{S}'} (q', n_2)$ since $n_s \prec_g n_2$ for all n_2. Analogously if $n_2 = n_e$. Now take the case where $n_1 \in \text{nodes}(g_1)$ and $n_2 \in \text{nodes}(g_2)$. (The case where $n_2 \in \text{nodes}(g_1)$ and $n_1 \in \text{nodes}(g_2)$ is symmetric.) Here, we have $n_1 \not\prec_g n_2$ and $n_2 \not\prec_g n_1$. Assume that it is not the case that $(p', n_1) \leq_{\mathcal{S}'} (q', n_2)$, for a contradiction. Equivalently, we assume $(q', n_2) <_{\mathcal{S}'} (p', n_1)$. Recall that $\mathcal{S}'$ is based on $<_1$. We have either $n_1 <_1 n_2$ or $n_2 <_1 n_1$. Take the case of $n_1 <_1 n_2$ first. There exists a node n_3 such that $n_3 \prec_g n_1$ and $n_3 \diamond_{\mathcal{S}'} n_2$. Let N be the set of nodes $\{n \in g_1 \mid n \diamond_{\mathcal{S}'} n_3\}$. Note that $|N| < |P|$ since there is at least one node in g_2, namely n_2, that is visited simultaneously with n_3. Since $\mathcal{S} \cap \text{nodes}(g_1)$ is identical to $\mathcal{S}' \cap \text{nodes}(g_1)$ up to the assignment of processors, the set of nodes in g_1 visited by $\mathcal{S}$ simultaneously with n_3 is also N. However, $\mathcal{S}$ does not visit n_2 simultaneously with n_3 since it visits n_2 no earlier than n_1. From the induction hypothesis, $\mathcal{S} \cap \text{nodes}(g_2)$ is equivalent to $\mathcal{S}' \cap \text{nodes}(g_2)$, but we have shown that they are not equivalent, a contradiction. Thus, it is the case that $(p', n_1) \leq_{\mathcal{S}'} (q', n_2)$. Finally, if $n_2 <_1 n_1$ then we apply a similar argument, but instead select a node n_3 that $\mathcal{S}$ (rather than $\mathcal{S}'$) visits simultaneously with n_2. □

3.3 Depth-First Scheduling

I now present a deterministic semantics that implements a particular scheduling policy. This policy performs a *depth-first* traversal of the computation graph. Depth-first scheduling [Blelloch et al., 1999, Narlikar and Blelloch, 1999] has been shown to be an effective method of scheduling parallel programs, especially with respect to memory use overhead. In this section, I present an implementation of a parallel language that uses a depth-first scheduling policy but does so abstractly, without representing state as queues, pointers, or other extra-lingual data structures. Instead, it will use the structure of a transition derivation to enforce a scheduling policy.

The depth-first semantics will be our first example of a greedy semantics. The semantics described in this section implements a *left-to-right* depth-first traversal of the computation graph. More precisely, a **depth-first** schedule is a greedy schedule based the total order $<_*$ given by the sequential schedule. The depth-first schedule might be considered to be "as sequential as possible" since it only goes ahead of the sequential schedule when it is constrained by the dependencies in g. Note that the sequential schedule for a graph g is trivially a depth-first schedule and is the only one-processor depth-first schedule.

As a depth-first schedule is a greedy schedule based on a total order, the lemmata in the

previous section yield the following corollaries.

Corollary 3.1 (Sequential Depth-First Decomposition)**.** If $\mathcal{S}$ is a depth-first schedule of $g_1 \oplus g_2$ then $\mathcal{S}_1 = \mathcal{S} \cap \mathrm{nodes}(g_1)$ is a depth-first schedule for g_1 and $\mathcal{S}_2 = \mathcal{S} \cap \mathrm{nodes}(g_2)$ is a depth-first schedule for g_2.

Corollary 3.2 (Parallel Depth-First Decomposition)**.** If $\mathcal{S}$ is a depth-first schedule of $g_1 \otimes g_2$ then

- $\mathcal{S}_1 = \mathcal{S} \cap \mathrm{nodes}(g_1)$ is a depth-first schedule of g_1, and
- $\mathcal{S}_2 = \mathcal{S} \cap \mathrm{nodes}(g_2)$ is a depth-first schedule of g_2 restricted to processors P_i in step i where P_i is defined as the set of processors p such that there is a node $n \in \mathrm{nodes}(g_2)$ and $\mathrm{visits}_{\mathcal{S}}(p, n)$ in step i.

I state the above corollary in a way that is specific to depth-first schedules: when considering a parallel composition, the schedule for the left sub-graph is also a (unrestricted) depth-first schedule.

Corollary 3.3 (Uniqueness of Depth-First Schedules)**.** If $\mathcal{S}$ and $\mathcal{S}'$ are depth-first schedules of g for processors P, then

$$\begin{aligned}\forall p, q \in P.\, \forall n_1, n_2 \in \mathrm{nodes}(g).\, (p, n_1) \leq_{\mathcal{S}} (q, n_2) \\ \Rightarrow \exists p', q' \in P.\, (p', n_1) \leq_{\mathcal{S}'} (q', n_2)\end{aligned}$$

3.3.1 Semantics

Unlike the non-deterministic semantics, the semantics given in this section constrains the number of parallel transitions that may be taken in each step. This is meant to model executions on a hardware platform with a fixed number of processors or processor cores. The depth-first semantics is implemented using the following judgment,

$$P; e \overset{\text{df}}{\longmapsto} P'; e'$$

which is read *with processors P available, expression e steps to e' with processors P' remaining unused*. In this judgment, P is a set of processors and as such represents resources that are to be or have been consumed. The (set) difference between P' and P tells us how many processors are used in a given transition. Though I will write this set of processors as a list, the order in which processors appears in that list has no significance. I write ϵ for the empty set of processors.

Like the non-deterministic semantics, the depth-first semantics relies on the primitive transitions to expand the constructs found in the source language into let expressions. The inference rules shown in Figure 3.6 define when primitive transitions can be applied and how let expressions are evaluated.

It is easiest to understand the rules defining depth-first semantics by starting with the final rule shown in the figure, DF-BRANCH. This rule states that, when evaluating a parallel declaration, any computation resources should first be used to evaluate the left-hand expression. If any resources remain, they should be used to evaluate the right-hand expression. This is made explicit by threading P' from one sub-derivation to the other: it appears as a result of the judgment on the left and as an input to the judgment on the right.

Figure 3.6 Depth-First Parallel Transition Semantics. When combined with the primitive transitions, these rules define an implementation based on a depth-first scheduling policy.

$\boxed{P; e \overset{\text{df}}{\longmapsto} P'; e'}$

$$\frac{P \neq \epsilon}{P; v \overset{\text{df}}{\longmapsto} P; v}\ \textsc{df-Val} \qquad \frac{}{\epsilon; e \overset{\text{df}}{\longmapsto} \epsilon; e}\ \textsc{df-None} \qquad \frac{e \longrightarrow e'}{P, p; e \overset{\text{df}}{\longmapsto} P; e'}\ \textsc{df-Prim}$$

$$\frac{P; d \overset{\text{df}}{\longmapsto} P'; d'}{P; \mathsf{let}\ d\ \mathsf{in}\ e \overset{\text{df}}{\longmapsto} P'; \mathsf{let}\ d'\ \mathsf{in}\ e}\ \textsc{df-Let}$$

$\boxed{P; d \overset{\text{df}}{\longmapsto} P'; d'}$

$$\frac{P; e \overset{\text{df}}{\longmapsto} P'; e'}{P; x = e \overset{\text{df}}{\longmapsto} P'; x = e'}\ \textsc{df-Leaf} \qquad \frac{P; d_1 \overset{\text{df}}{\longmapsto} P'; d_1' \qquad P'; d_2 \overset{\text{df}}{\longmapsto} P''; d_2'}{P; d_1\ \mathsf{and}\ d_2 \overset{\text{df}}{\longmapsto} P''; d_1'\ \mathsf{and}\ d_2'}\ \textsc{df-Branch}$$

The rules DF-VAL and DF-NONE allow a transition to skip over a sub-expression, in the case of DF-VAL, because the sub-expression is already a value, and in case of DF-NONE because no computational resources remain. Note that DF-VAL can only be applied if at least one processor is available.

Primitive transitions are embedded using DF-PRIM, but unlike the non-deterministic semantics, this rule cannot be applied to any expression to which a primitive transition can be taken. Instead, the depth-first semantics must "pay" for any primitive transitions by decrementing the remaining number of processors available. This limits the number of primitive transitions that can occur in any parallel step.

Finally, DF-LET and DF-LEAF allow transitions on the single declaration or expression (respectively) appearing in an expression or declaration (respectively).

The depth-first semantics uses a "top-level" judgment, defined by the single inference rule shown here.

$$\frac{P; e \overset{\text{df}}{\longmapsto} P'; e'}{e \overset{P\text{-df}}{\longmapsto} e'}$$

This judgment defines a P-depth-first semantics, or a depth-first semantics that uses at most the processors in P. Note that P' is unconstrained in the premise, allowing an arbitrary number of processors to remain unused. When comparing the depth-first semantics with the other semantics used in this thesis, I will always use this top-level judgment.

3.3.2 Determinacy

Unlike the non-deterministic semantics, there is at most one depth-first transition that can be applied to any expression. Thus the depth-first semantics defines an algorithm for evaluating

expressions.

Theorem 3.5 (Determinacy of DF Evaluation)**.** If $P; e \xmapsto{\text{df}} P'; e'$ and $P; e \xmapsto{\text{df}} P''; e''$ then $P' = P''$ and $e' = e''$. Similarly, if $P; d \xmapsto{\text{df}} P'; d'$ and $P; d \xmapsto{\text{df}} P''; d''$ then $P' = P''$ and $d' = d''$.

Proof. By induction on the first derivation. The proof relies on the following two facts: first, because of the constraints on P at most one of DF-VAL and DF-NONE can be applied; and second, in no instance can both DF-LET and DF-PRIM be applied. □

Given this fact, we can consider for any expression e the unique result of a depth-first transition.

3.3.3 Progress

Unlike the non-deterministic semantics, which might repeatedly apply the ND-IDLE rule *ad infinitum*, the depth-first semantics will always apply a primitive transition when such an application is possible. Though the property used below is a non-standard definition of "progress," a more traditional form of progress as part of type safety could also be proved. The property below might better be called "irreflexivity" except that the judgment *is* reflexive in the case that the expression is a value or there are no processors.

Theorem 3.6. If $P; e \xmapsto{\text{df}} P'; e$ then e is a value or $P = P' = \epsilon$.

Proof. By induction on the derivation of $P; e \xmapsto{\text{df}} P'; e$.
Case DF-VAL**:** In this case, e is a value.
Case DF-NONE**:** In this case, $P = P' = \epsilon$.
Case DF-PRIM**:** Here an inspection of inference rules in Figure 3.2 shows that the primitive transition relation is irreflexive. Each expression steps from a non-value to a value, expands an expression by wrapping it in one or more let expressions, or transitions to a sub-expression. In the case of function application (P-APPBETA), note that an underlined expression steps to an expression that is *not* underlined.

The remaining cases (DF-LET, DF-LEAF, DF-BRANCH) follow immediately from the induction hypothesis. □

It should be noted that our language includes non-terminating terms, so there are infinite sequences of transitions in the depth-first semantics. However, there is no expression which has both infinite *and* finite sequences of transitions.

3.3.4 Soundness

The soundness and completeness results in this section serve to verify that there is at least one implementation adhering to the specification given by the cost semantics and, furthermore, that the cost semantics provides a accurate mechanism for reasoning about such implementations. As such, these results do not provide an significant insights but rather verify our intuitions about the depth-first semantics.

To prove soundness, I show that the depth-first semantics implements a (unspecified) scheduling policy. In the proof, I leverage the non-deterministic semantics. Since the non-deterministic semantics has been shown to be sound, we must simply show that the behavior of depth-first semantics always lies within the behavior of the non-deterministic semantics. This is proved in the following theorem.

Theorem 3.7. If $e \overset{P\text{-df}}{\longmapsto} e'$ then $e \overset{\text{nd}}{\longmapsto} e'$.

Proof. This theorem is proved by showing that each rule in the depth-first semantics is a constrained form of a rule in the non-deterministic semantics: whenever a depth-first rule is applied, a non-deterministic rule could also have been applied to yield the same result. The proof proceeds by induction on the derivation of the depth-first transition, in each step first applying the induction hypothesis (where applicable) and then one of rules from the table below.

When the original derivation used...	...now use the following instead:
$\dfrac{P \neq \epsilon}{P; v \overset{\text{df}}{\longmapsto} P; v}$ DF-VAL	$\dfrac{}{e \overset{\text{nd}}{\longmapsto} e}$ ND-IDLE
$\dfrac{}{\epsilon; e \overset{\text{df}}{\longmapsto} \epsilon; e}$ DF-NONE	$\dfrac{}{e \overset{\text{nd}}{\longmapsto} e}$ ND-IDLE
$\dfrac{e \longrightarrow e'}{P, p; e \overset{\text{df}}{\longmapsto} P; e'}$ DF-PRIM	$\dfrac{e \longrightarrow e'}{e \overset{\text{nd}}{\longmapsto} e'}$ ND-PRIM
$\dfrac{P; d \overset{\text{df}}{\longmapsto} P'; d'}{P; \mathsf{let}\ d\ \mathsf{in}\ e \overset{\text{df}}{\longmapsto} P'; \mathsf{let}\ d'\ \mathsf{in}\ e}$ DF-LET	$\dfrac{d \overset{\text{nd}}{\longmapsto} d'}{\mathsf{let}\ d\ \mathsf{in}\ e \overset{\text{nd}}{\longmapsto} \mathsf{let}\ d'\ \mathsf{in}\ e}$ ND-LET
$\dfrac{P; e \overset{\text{df}}{\longmapsto} P'; e'}{P; x = e \overset{\text{df}}{\longmapsto} P'; x = e'}$ DF-LEAF	$\dfrac{e \overset{\text{nd}}{\longmapsto} e'}{x = e \overset{\text{nd}}{\longmapsto} x = e'}$ ND-LEAF
$\dfrac{P; d_1 \overset{\text{df}}{\longmapsto} P'; d_1' \qquad P'; d_2 \overset{\text{df}}{\longmapsto} P''; d_2'}{P; d_1\ \mathsf{and}\ d_2 \overset{\text{df}}{\longmapsto} P''; d_1'\ \mathsf{and}\ d_2'}$ DF-BRANCH	$\dfrac{d_1 \overset{\text{nd}}{\longmapsto} d_1' \qquad d_2 \overset{\text{nd}}{\longmapsto} d_2'}{d_1\ \mathsf{and}\ d_2 \overset{\text{nd}}{\longmapsto} d_1'\ \mathsf{and}\ d_2'}$ ND-BRANCH

□

Theorem 3.8 (DF Soundness)**.** For any closed expression e, if $e \overset{P\text{-df}}{\longmapsto}{}^{*}\ v$ then there exist some g and h such that $\ulcorner e \urcorner \Downarrow v; g; h$

Proof. This follows from the previous theorem and the soundness of the non-deterministic semantics (Theorem 3.2). □

3.3.5 Completeness

In this section, I show that for each depth-first schedule, there is a sequence of transitions in the depth-first semantics. For source expressions (those expressions written by programmers), this can be stated simply as in the following theorem.

Theorem 3.9 (DF Completeness)**.** For any source expression e, if $e \Downarrow v; g; h$ and $\mathcal{S}$ is a depth-first schedule of g for processors P, then there exists a sequence of expressions $e_0, \ldots, e_k$ such that

- $e_0 = e$ and $e_k = v$,
- $\forall i \in [0, k).\ e_i \overset{P\text{-df}}{\longmapsto} e_{i+1}$, and
- $\forall i \in [0, k].\ \text{locs}(e_i) = \text{roots}_{h,v}(\widehat{N_i})$

where $N_0 = \emptyset$ and $N_1, \ldots, N_k = \text{steps}(\mathcal{S})$.

As in the case of the non-deterministic semantics, this theorem must be generalized to account for partially evaluated expressions (*i.e.*, those that contain locations). In addition, it must also be generalized over schedules which use different sets of processors at each step. This allows a schedule over processors P to be split into two schedules that, when run in parallel, each use only a portion of the P processors in a given step. The proof depends on the fact that any depth-first schedule can be split in this fashion, and moreover, that it can be split so that the left-hand side is allocated all the processors that it could possibly use. The theorem above follows immediately from this lemma by lifting these restrictions.

Lemma 3.7 (DF Completeness, Strengthened)**.** For any expression e, if $e \Downarrow v; g; h$ and $\mathcal{S}$ is a depth-first schedule of g restricted to processors $P_i \subseteq P$ at each step $i \in [0, k)$, then there exists a sequence of expressions $e_0, \ldots, e_k$ such that

- $e_0 = e$ and $e_k = v$,
- $\forall i \in [0, k).\ P_{i+1}; e_i \overset{\text{df}}{\longmapsto} P_i'; e_{i+1}$, and
- $\forall i \in [0, k].\ \text{locs}(e_i) = \text{roots}_{h,v}(\widehat{N_i} \cap \text{locs}(e))$

where $N_0 = \emptyset$ and $N_1, \ldots, N_k = \text{steps}(\mathcal{S})$ and P_i' is unconstrained.

Proof. The proof follows largely as the analogous proof for the non-deterministic semantics. We shall focus on the cases in that proof that used either the ND-IDLE or ND-BRANCH rules as these rules capture the substantial differences between the two semantics.

Case C-VAL**:** In this case $e = v$ so $\text{locs}(e) = \text{loc}(v)$. There is only one node in g, there is only one schedule, depth-first or otherwise. That schedule has a single step $N_1 = \{n\}$ and $P_1, p; e \overset{\text{df}}{\longmapsto} P_1', v$ by DF-VAL.

By Lemma 2.1, $\text{roots}_{h,v}(\widehat{N_o} \cup \text{loc}(v)) = \text{loc}(v)$. Because the roots of a set of locations always include the location of the resulting value and because n is chosen to be fresh $\text{roots}_{h,v}(\widehat{N_1} \cup \text{loc}(v)) = \text{loc}(v)$ as well.

Case C-FORK**:** Here e is $\{e^L, e^R\}$ and $g = g_1 \otimes g_2 \oplus [n]$. We first construct a sequence of expressions related by the depth-first semantics. Define e_1 to be let $x_1 = e^L$ and $x_2 = e^R$ in $\underline{(x_1,x_2)}$. We have e_0 transitions to e_1 by DF-PRIM and P-FORK using a single processor.

We have sub-derivations $e^L \Downarrow v_1; g_1; h_1$ and $e^R \Downarrow v_2; g_2; h_2$. If we restrict $\mathcal{S}$ to the nodes of g_1 and g_2, we obtain schedules $\mathcal{S}^L$ and $\mathcal{S}^R$ for these sub-graphs. By Corollary 3.2, these are both greedy, depth-first schedules. Inductively, we have that $e^L \overset{P\text{-df}}{\longmapsto}{}^{k_1} v_1$ using processors $P_1, \ldots, P_{k_1}$ as well as $P \backslash P_i; e_i^R \overset{\text{df}}{\longmapsto} P_i'; e_{i+1}^R$ for $i \in [1, k_2]$. Let $k' = \max(k_1, k_2)$ and define two new sequences of expressions of length k' and indexed by $[1, k' + 1]$ as follows. Define each e_i^L for $i \in [1, k_1]$ from the expressions given in the sequence derived inductively. If $k_1 < k'$ then

define the remaining e_i^L as v_1. For the first k_1, we will use the transitions given inductively except that we will replace the initial set of processors P with the set that is actually used. That is, if we have $P; e_i^L \overset{\text{df}}{\longmapsto} P_i'; e_{i+1}^L$ then let $P_i = P \backslash P_i'$. For the final $k' - k_1$ transitions, we use DF-VAL. For the right-hand side, we define a sequence of expressions using the unused processors from the left-hand side $P_1', \ldots, P_{k_1}'$ as a guide. Define e_1^R as e^R. If P_i' is empty then define e_{i+1}^R as e_i^R with transition DF-NONE. If P_i' is non-empty, then use the expression and transition given inductively.

For each $i \in [2, k'+1]$, define e_i using these sequences as $\mathsf{let}\ x_1 = e_i^L\ \mathsf{and}\ x_2 = e_i^R\ \mathsf{in}\ \underline{(x_1,x_2)}$ as in the proof for the non-deterministic semantics. We use the transitions above and apply rules DF-LET, DF-BRANCH, and DF-LEAF to the let expression at each step. Let $e_{k'+2}$ be $\underline{(v_1,v_2)}$ and note that $e_{k'+1}$ transitions to $e_{k'+2}$ by P-JOIN and DF-PRIM. Let $k = k' + 3$ and define e_k as $\langle v_1, v_2 \rangle^\ell$. It follows that $e_{k'+2} \overset{\text{nd}}{\longmapsto} e_k$ by rules P-PAIRALLOC and DF-PRIM.

Finally, we must show that the locations of each expression in this sequence match the roots of each step of the schedule. This follows as in the case of the non-deterministic semantics, using Lemma 2.1 initially and then the inductive results for each sub-expression. □

3.3.6 Greedy Semantics

The notion of greediness defined above can also be captured precisely using an implementation semantics. As an alternative to the DF-BRANCH rule, we might consider the following rule.

$$\frac{P_1; d_1 \mapsto P_1'; d_1' \qquad P_2; d_2 \mapsto P_2'; d_2'}{P_1, P_2; d_1 \ \mathsf{and}\ d_2 \mapsto P_1', P_2'; d_1' \ \mathsf{and}\ d_2'}$$

While more constrained than the non-deterministic rule ND-BRANCH above, it allows transitions to occur in either the left or the right branch so long as no more than $|P|$ primitive transitions are taken in parallel. This rule, taken together with the other depth-first rules in Figure 3.6, describes all implementations that are greedy. Like the depth-first semantics, the greedy semantics "pays" one processor each time a primitive transition is applied. Also like the depth-first semantics, the greedy semantics is only reflexive for values and when no processors remain: it does not include an idle rule as in the more general non-deterministic semantics.

A semantics such as the depth-first semantics can then be shown to be greedy using a soundness proof similar to that shown in Section 3.3.4 above.

3.4 Breadth-First Scheduling

While the depth-first semantics evaluates sub-expressions in the program according to a depth-first traversal of the corresponding nodes, we can also define a semantics corresponding to the *breadth-first* traversal of these nodes. A breadth-first schedule is sometimes called a round-robin schedule: if we view each serial chain of nodes in a graph as a thread, then the breadth-first schedule equally divides processors among the threads, scheduling each thread for one unit of time. To define breadth-first schedules, we must first define the depth of a node in a graph.

Figure 3.7 Syntax Extensions for Breadth-First Semantics. Cursors $_\wedge$ and stops $^\vee$ are used to maintain the state of the breadth-first scheduler.

$$\text{(expressions)} \quad e \quad ::= \quad \ldots \mid {}_\wedge e \mid e_\wedge \mid {}^\vee e \mid e^\vee$$

Definition (Depth)**.** The **depth** of a node n in g is the length of the shortest path from the start node of g to n.

I will write $\mathrm{depth}_g(n)$ (or simply $\mathrm{depth}(n)$ where the graph is evident from context) to denote the depth of a node. The breadth-first order is a total order on the nodes of a graph g that first compares nodes based on their depth and then, for nodes of equal depth, using the sequential order $<_*$.

Definition (Breadth-First Order)**.** The breadth-first order is the total order $<_{\text{bf}}$ of the nodes of g defined for all nodes $n_1, n_2 \in \mathrm{nodes}(g)$ as

$$n_1 <_{\text{bf}} n_2 \text{ iff } \mathrm{depth}(n_1) < \mathrm{depth}(n_2) \vee (\mathrm{depth}(n_1) = \mathrm{depth}(n_2) \wedge n_1 <_* n_2)$$

A **breadth-first** schedule is a greedy schedule based on the total order $<_{\text{bf}}$. As such, a breadth-first schedule will visit all the nodes at a given depth before proceeding to nodes at a greater depth. As in the case of depth-first schedules, the following corollaries follow immediately from Lemmata 3.5 and 3.6 and Theorem 3.4.

Corollary 3.4 (Sequential Breadth-First Decomposition)**.** If $\mathcal{S}$ is a breadth-first schedule of $g_1 \oplus g_2$ then $\mathcal{S}_1 = \mathcal{S} \cap \mathrm{nodes}(g_1)$ is a breadth-first schedule for g_1 and $\mathcal{S}_2 = \mathcal{S} \cap \mathrm{nodes}(g_2)$ is a breadth-first schedule for g_2.

Corollary 3.5 (Parallel Breadth-First Decomposition)**.** If $\mathcal{S}$ is a breadth-first schedule of $g_1 \otimes g_2$ then

- $\mathcal{S}_1 = \mathcal{S} \cap \mathrm{nodes}(g_1)$ is a breadth-first schedule of g_1 restricted to processors P_i in step i where P_i is defined as the set of processors p such that there is a node $n \in \mathrm{nodes}(g_1)$ and $\mathrm{visits}_{\mathcal{S}}(p, n)$ in step i, and
- $\mathcal{S}_2 = \mathcal{S} \cap \mathrm{nodes}(g_2)$ is a breadth-first schedule of g_2 and restricted to processors P'_i in step i, defined analogously.

Corollary 3.6 (Uniqueness of Breadth-First Schedules)**.** If $\mathcal{S}$ and $\mathcal{S}'$ are breadth-first schedules of g for processors P, then

$$\begin{aligned} \forall p, q \in P.\, \forall n_1, n_2 \in \mathrm{nodes}(g).\, (p, n_1) \leq_{\mathcal{S}} (q, n_2) \\ \Rightarrow \exists p', q' \in P.\, (p', n_1) \leq_{\mathcal{S}'} (q', n_2) \end{aligned}$$

3.4.1 Semantics

The semantics implementing a breadth-first schedule is more complex than that implementing a depth-first schedule because the breadth-first semantics must maintain some state between

parallel steps. While the depth-first semantics always starts with the left-most sub-expression, the breadth-first semantics picks up where it left off on the previous step. The structure of an expression does not encode any analogue of depth in the computation graph, so instead we will use a cursor $_\wedge$ to separate two sub-expressions that occur at the same depth in the syntax tree but correspond to nodes of different depth in the computation graph. This cursor will move throughout an expression during a parallel step.

The semantics must also keep track of any sub-expressions that have undergone transitions in the current step so as to avoid violating any sequential dependencies: if we have an edge (n_1, n_2) in g then the schedule cannot visit both n_1 and n_2 in the same step. A stop $^\vee$ will be used to mark the first sub-expression to take a transition in the current step and to prevent that sub-expression from taking any further transitions.

The cursor and stop are implemented as extensions of the language syntax as shown in Figure 3.7. Like other extensions of the syntax in this chapter, these extensions are never written by programmers and only appear in expressions that have been partially evaluated using the transition semantics described below.

Figure 3.8 shows the breadth-first equivalences and transition semantics. As the cursor marks a position *between* two sub-expressions, these equivalences are used to describe different ways of writing an expression with the cursor in equivalent positions. The equivalences $\equiv^{\text{bf}}$ are defined as the least symmetric, commutative, transitive relations satisfying the axioms in the figure. For example, given a declaration d_1 and d_2, if the cursor is currently positioned after d_1 then it can also be thought of as appearing just before d_2. This is implemented through the following axiom:

$$d_{1\wedge} \text{ and } d_2 \equiv^{\text{bf}} d_1 \text{ and } {}_\wedge d_2$$

The other important feature of these equivalences is the absence of the stop $^\vee$. This ensures that the cursor can never move over the stop. Thus, the stop is exactly what its name implies: a mechanism to block the movement of the cursor.

Now consider the transition rules that appear in the figure. The rule BF-NONE is identical to that found in the depth-first semantics: if no available processors remain in the current step then no more transitions are performed. The rule BF-PRIM includes primitive transitions and, like its counterpart in the depth-first semantics, consumes a processor. It also adds the additional constraint that the cursor must appear at the beginning of the expression. After the transition, the cursor is moved forward to mark the position after the expression. BF-EQUIV applies an equivalence before taking a transition. The next three rules, BF-LET, BF-STOPBEFORE, and BF-STOPAFTER, take a single transition on the only sub-declaration or sub-expression. The final rule for expressions, BF-NOCURSOR is applied in case of expressions that are not prefixed by the cursor and none of the other rules apply.

The transition rules for declarations are similar to those in the depth-first semantics: available processors are first passed to the left side of a parallel declaration, and any remaining ones are then passed to the right side. In addition, the breadth-first equivalence is applied between transitions of these two branches (to allow the cursor to move from one branch to the other).

The breadth-first semantics also makes use of a top-level judgment $e \overset{P\text{-bf}}{\longmapsto} e'$ that describes

Figure 3.8 Breadth-First Parallel Equivalences and Transition Semantics. These rules define an implementation of a breadth-first scheduling policy.

$\boxed{e \equiv^{\text{bf}} e'}$

$$ {}_{\wedge}\mathsf{let}\ d\ \mathsf{in}\ e \equiv^{\text{bf}} \mathsf{let}\ {}_{\wedge}d\ \mathsf{in}\ e \qquad\qquad \mathsf{let}\ d_{\wedge}\ \mathsf{in}\ e \equiv^{\text{bf}} (\mathsf{let}\ d\ \mathsf{in}\ e)_{\wedge} $$

$\boxed{d \equiv^{\text{bf}} d'}$

$$ {}_{\wedge}x = e \equiv^{\text{bf}} x = {}_{\wedge}e \qquad x = e_{\wedge} \equiv^{\text{bf}} (x = e)_{\wedge} \qquad {}_{\wedge}(d_1\ \mathsf{and}\ d_2) \equiv^{\text{bf}} {}_{\wedge}d_1\ \mathsf{and}\ d_2 $$

$$ d_{1\wedge}\ \mathsf{and}\ d_2 \equiv^{\text{bf}} d_1\ \mathsf{and}\ {}_{\wedge}d_2 \qquad\qquad d_1\ \mathsf{and}\ d_{2\wedge} \equiv^{\text{bf}} (d_1\ \mathsf{and}\ d_2)_{\wedge} $$

$\boxed{P; e \overset{\text{bf}}{\longmapsto} P'; e'}$

$$ \frac{}{\epsilon; e \overset{\text{bf}}{\longmapsto} \epsilon; e}\ \textsc{bf-None} \qquad \frac{e \longrightarrow e'}{P, p; {}_{\wedge}e \overset{\text{bf}}{\longmapsto} P; e'_{\wedge}}\ \textsc{bf-Prim} $$

$$ \frac{e \equiv^{\text{bf}} e' \qquad P; e' \overset{\text{bf}}{\longmapsto} P; e''}{P; e \overset{\text{bf}}{\longmapsto} P; e''}\ \textsc{bf-Equiv} \qquad \frac{P; d \overset{\text{bf}}{\longmapsto} P'; d'}{P; \mathsf{let}\ d\ \mathsf{in}\ e \overset{\text{bf}}{\longmapsto} P'; \mathsf{let}\ d'\ \mathsf{in}\ e}\ \textsc{bf-Let} $$

$$ \frac{P; e \overset{\text{bf}}{\longmapsto} P'; e'}{P; {}^{\vee}e \overset{\text{bf}}{\longmapsto} P'; {}^{\vee}e'}\ \textsc{bf-StopBefore} \qquad \frac{P; e \overset{\text{bf}}{\longmapsto} P'; e'}{P; e^{\vee} \overset{\text{bf}}{\longmapsto} P'; e'^{\vee}}\ \textsc{bf-StopAfter} $$

$$ \frac{e \not\equiv^{\text{bf}} {}_{\wedge}e' \qquad e \neq {}^{\vee}e' \qquad e \neq e'^{\vee} \qquad e \neq \mathsf{let}\ d\ \mathsf{in}\ e'}{P; e \overset{\text{bf}}{\longmapsto} P; e}\ \textsc{bf-NoCursor} $$

$\boxed{P; d \overset{\text{bf}}{\longmapsto} P'; d'}$

$$ \frac{P; e \overset{\text{bf}}{\longmapsto} P'; e'}{P; x = e \overset{\text{bf}}{\longmapsto} P'; x = e'}\ \textsc{bf-Leaf} $$

$$ \frac{P; d_1 \overset{\text{bf}}{\longmapsto} P'; d'_1 \qquad d'_1\ \mathsf{and}\ d_2 \equiv^{\text{bf}} d''_1\ \mathsf{and}\ d'_2 \qquad P'; d'_2 \overset{\text{bf}}{\longmapsto} P''; d''_2}{P; d_1\ \mathsf{and}\ d_2 \overset{\text{bf}}{\longmapsto} P''; d''_1\ \mathsf{and}\ d''_2}\ \textsc{bf-Branch} $$

Figure 3.9 Manipulating Cursors and Stops. The functions defined in this figure add and remove cursors and stops to programs as required by the breadth-first transition semantics.

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
[\![{}_{\wedge}e]\!]^{+} & = & {}^{\vee}{}_{\wedge}e & [\![{}^{\vee}e]\!]^{-} & = & e \\
[\![e_{\wedge}]\!]^{+} & = & e^{\vee}{}_{\wedge} & [\![e^{\vee}]\!]^{-} & = & e \\
[\![e]\!]^{+} & = & \ldots \text{(congruently otherwise)} & [\![e]\!]^{-} & = & \ldots \text{(congruently otherwise)}
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{erase}({}_{\wedge}e) & = & \mathrm{erase}(e) \\
\mathrm{erase}(e_{\wedge}) & = & \mathrm{erase}(e) \\
\mathrm{erase}({}^{\vee}e) & = & \mathrm{erase}(e) \\
\mathrm{erase}(e^{\vee}) & = & \mathrm{erase}(e) \\
\mathrm{erase}(e) & = & \ldots \text{(congruently otherwise)}
\end{array}
$$

breadth-first schedules with P processors. This judgment is defined by the following two rules.

$$
\frac{P; [\![e]\!]^{+} \overset{\text{bf}}{\longmapsto} P'; e' \qquad e' \not\equiv^{\text{bf}} e''_{\wedge}}{e \overset{P\text{-bf}}{\longmapsto} [\![e']\!]^{-}} \ \textsc{bf-NoWrap}
$$

$$
\frac{P; [\![e]\!]^{+} \overset{\text{bf}}{\longmapsto} P'; e' \qquad e' \equiv^{\text{bf}} e''_{\wedge} \qquad P'; {}_{\wedge}e'' \overset{\text{bf}}{\longmapsto} P''; e'''}{e \overset{P\text{-bf}}{\longmapsto} [\![e''']\!]^{-}} \ \textsc{bf-Wrap}
$$

Ignoring the brackets for the moment, the first rule BF-NOWRAP states that the top-level semantics is defined by starting with P processors and finishing with an arbitrary P' processors remaining. This rule can only be applied if the result is not equivalent to an expression with the cursor at the very end. Otherwise, the second rule BF-WRAP must be applied. This rule shifts the cursor back to the beginning of the expression and continues with any remaining processors. This corresponds to the case when the breadth-first traversal of the computation graph progresses from nodes at a given depth to those found at the next level of depth. At the same time, this traversal wraps around from the right side of the graph back to the left.

The brackets represent two functions that add $[\![-]\!]^{+}$ and remove $[\![-]\!]^{-}$ the stop as defined in Figure 3.9. Only two cases of each function are interesting, and in the remaining cases, the given function is applied to every sub-expression and sub-declaration congruently. For example, stops are added to pairs based on the following equation,

$$[\![(e_1,e_2)]\!]^{+} = ([\![e_1]\!]^{+},[\![e_2]\!]^{+})$$

The first function $[\![-]\!]^{+}$ adds the stop by walking over the input expression until the cursor is found. The stop is then added immediately before the cursor. The second function $[\![-]\!]^{-}$ removes the stop from the given expression.

The top-level semantics serves three purposes in the breadth-first case: to refresh the number of processors available at each step (as in the depth-first semantics), to allow the cursor to wrap around at the top level, and to update the position of the stop.

3.4.2 Soundness

As in the case of the depth-first semantics, I show that the breadth-first semantics implements a scheduling policy by showing that its behavior is contained within the behavior of the non-deterministic semantics. The proof has two parts: first I show any individual transition is a constrained form of a non-deterministic transition; I then do the same for any top-level breadth-first transition.

As I have added additional state to the breadth-first expressions (in the form of the cursor and stop), I define an erasure function that removes these additional syntactic forms (as shown in Figure 3.9).

Lemma 3.8. If $P; e \overset{\text{bf}}{\longmapsto} P'; e'$ then there exists an expression e'' such that $\text{erase}(e) \overset{\text{nd}}{\longmapsto} e''$ and $e'' = \text{erase}(e')$.

Proof. The proof proceeds by induction on the derivation of the breadth-first transition. For the rules BF-STOPBEFORE and BF-STOPAFTER, the result follows from the induction hypothesis. It is easy to show that if $e \equiv^{\text{bf}} e'$ then $\text{erase}(e) = \text{erase}(e')$ by induction on the structure of expressions. The case for BF-EQUIV follows immediately. The remaining rules follow from the following table.

When the original derivation used...	...now use the following instead:
$\dfrac{}{\epsilon; e \overset{\text{bf}}{\longmapsto} \epsilon; e}$ BF-NONE	$\dfrac{}{e \overset{\text{nd}}{\longmapsto} e}$ ND-IDLE
$\dfrac{e \not\equiv^{\text{bf}} {}_{\wedge}e' \qquad e \neq \ldots}{P; e \overset{\text{bf}}{\longmapsto} P; e}$ BF-NOCURSOR	$\dfrac{}{e \overset{\text{nd}}{\longmapsto} e}$ ND-IDLE
$\dfrac{e \longrightarrow e'}{P, p; {}_{\wedge}e \overset{\text{bf}}{\longmapsto} P; e'_{\wedge}}$ BF-PRIM	$\dfrac{e \longrightarrow e'}{e \overset{\text{nd}}{\longmapsto} e'}$ ND-PRIM
$\dfrac{P; d \overset{\text{bf}}{\longmapsto} P'; d'}{P; \mathsf{let}\ d\ \mathsf{in}\ e \overset{\text{bf}}{\longmapsto} P'; \mathsf{let}\ d'\ \mathsf{in}\ e}$ BF-LET	$\dfrac{d \overset{\text{nd}}{\longmapsto} d'}{\mathsf{let}\ d\ \mathsf{in}\ e \overset{\text{nd}}{\longmapsto} \mathsf{let}\ d'\ \mathsf{in}\ e}$ ND-LET
$\dfrac{P; e \overset{\text{bf}}{\longmapsto} P'; e'}{P; x = e \overset{\text{bf}}{\longmapsto} P'; x = e'}$ BF-LEAF	$\dfrac{e \overset{\text{nd}}{\longmapsto} e'}{x = e \overset{\text{nd}}{\longmapsto} x = e'}$ ND-LEAF
$\dfrac{\begin{array}{c}P; d_1 \overset{\text{bf}}{\longmapsto} P'; d'_1 \\ d'_1\ \mathsf{and}\ d_2 \equiv^{\text{bf}} d''_1\ \mathsf{and}\ d'_2 \\ P'; d_2 \overset{\text{bf}}{\longmapsto} P''; d'_2\end{array}}{P; d_1\ \mathsf{and}\ d_2 \overset{\text{bf}}{\longmapsto} P''; d'_1\ \mathsf{and}\ d'_2}$ BF-BRANCH	$\dfrac{d_1 \overset{\text{nd}}{\longmapsto} d'_1 \qquad d_2 \overset{\text{nd}}{\longmapsto} d'_2}{d_1\ \mathsf{and}\ d_2 \overset{\text{nd}}{\longmapsto} d'_1\ \mathsf{and}\ d'_2}$ ND-BRANCH

Lemma 3.9. If $e \overset{P\text{-bf}}{\longmapsto} e'$ then there exists an expression e'' such that $\text{erase}(e) \overset{\text{nd}}{\longmapsto} e''$ and $e'' = \text{erase}(e')$.

Proof. If the breadth-first transition was derived with the BF-NOWRAP rule, the result follows immediately from the lemma above. If the BF-WRAP rule applies, the two sub-derivations of the rule may be combined into a single derivation of $\text{erase}(e) \overset{\text{nd}}{\longmapsto} e''$. In that case, let $\mathcal{D}_1$

Figure 3.10 Syntax Extensions for Work-Stealing Semantics. Boxed expressions are used as an abstract representation of work-stealing task queues. In effect, a single processor takes responsibility for evaluating all sub-expressions within a box.

$$\begin{array}{lrcl} \text{(expressions)} & e & ::= & \ldots \mid \boxed{e} \\ \text{(declarations)} & d & ::= & \ldots \mid \boxed{d} \end{array}$$

and $\mathcal{D}_2$ be the two sub-derivations of BF-WRAP. Take $\mathcal{D}_1$ and replace all occurrences of BF-NOCURSOR with the non-deterministic rule corresponding to breadth-first rule used in the same sub-expression from $\mathcal{D}_2$. This will be either BF-NONE, BF-PRIM, or BF-EQUIV. Note that this sub-expression must appear in $\mathcal{D}_2$ since in remains unchanged in the derivation $\mathcal{D}_1$. In fact, every sub-expression that appears *before* the stop will remain unchanged in $\mathcal{D}_1$ and every sub-expression that appears *after* the stop will remain unchanged in $\mathcal{D}_2$. Thus we are able to stitch together these two derivations at the position of the stop. □

Theorem 3.10 (BF Soundness)**.** For any closed expression e, if $e \xmapsto{P\text{-bf}}{}^{*} v$ then there exists some g and h such that $e \Downarrow v; g; h$

Proof. This follows from the previous theorem and the soundness of the non-deterministic semantics (Theorem 3.2). □

3.5 Work-Stealing Scheduling

I can also use my framework to describe implementations related to **work-stealing** scheduling policies [Burton and Sleep, 1981, Blumofe and Leiserson, 1999]. Intuitively, work stealing places the burden of scheduling on those processors that have no work to do, allowing occupied processors to continue useful computation. Typically, work-stealing algorithms are presenting by describing the data structures used to maintain queues of waiting jobs (*i.e.* unevaluated sub-expressions). In this presentation, I use the syntax of the expression itself as the structure by which these queues are maintained. I extend the original syntax once again, marking particular sub-expressions or sub-declarations with a box.

My approach abstracts away from the details of how work is communicated and focuses only on the order in which tasks are evaluated. Because "stealing" is often used to describe the communication of work itself, my semantics might be better described using another phrase often applied to work-stealing schedulers due to Mohr et al. [1990]: "breadth-first until saturation, then depth-first." This level of abstraction will be sufficient, as in previous sections, to model how memory is used by lower-level implementations, including those that use explicit stealing to communicate and balance load among processors.

Boxed expressions and declarations (Figure 3.10) mark parts of an expression that are currently undergoing evaluation. An expression running on a P-way multi-processor will contain P boxed sub-expressions. In some sense, a boxed expression is owned by a particular processor, and this processor takes responsibility for evaluating all sub-expressions. When work is stolen, an expression is unboxed, and two of its sub-expressions are boxed. Thus, the work queues used

Figure 3.11 Work-Stealing Parallel Equivalences and Transition Semantics. These rules, taken together the DF implementation, define the parallel execution of an expression using a work-stealing scheduler where each processor uses a DF policy. The DF policy may be replaced with other per-processor policies by amending the WS-EXPBOX and WS-DECLBOX rules.

$\boxed{e \equiv^{\text{ws}} e'}$

$$\boxed{\mathsf{let}\ d\ \mathsf{in}\ e} \equiv^{\text{ws}} \mathsf{let}\ \boxed{d}\ \mathsf{in}\ e$$

$\boxed{d \equiv^{\text{ws}} d'}$

$$\boxed{x = e} \equiv^{\text{ws}} x = \boxed{e} \qquad \boxed{d_1\ \mathsf{and}\ d_2} \equiv^{\text{ws}} \boxed{d_1}\ \mathsf{and}\ d_2 \qquad \boxed{d_1\ \mathsf{and}\ d_2} \equiv^{\text{ws}} d_1\ \mathsf{and}\ \boxed{d_2}$$

$\boxed{e \overset{\text{df-ws}}{\longmapsto} e'}$

$$\frac{d \overset{\text{df-ws}}{\longmapsto} d'}{\mathsf{let}\ d\ \mathsf{in}\ e \overset{\text{df-ws}}{\longmapsto} \mathsf{let}\ d'\ \mathsf{in}\ e}\ \text{WS-LET} \qquad \frac{}{v \overset{\text{df-ws}}{\longmapsto} v}\ \text{WS-VAL} \qquad \frac{e \overset{\text{1-df}}{\longmapsto} e'}{\boxed{e} \overset{\text{df-ws}}{\longmapsto} \boxed{e'}}\ \text{WS-EXPBOX}$$

$\boxed{d \overset{\text{df-ws}}{\longmapsto} d'}$

$$\frac{e \overset{\text{df-ws}}{\longmapsto} e'}{x = e \overset{\text{df-ws}}{\longmapsto} x = e'}\ \text{WS-LEAF} \qquad \frac{d_1 \overset{\text{df-ws}}{\longmapsto} d_1' \qquad d_2 \overset{\text{df-ws}}{\longmapsto} d_2'}{d_1\ \mathsf{and}\ d_2 \overset{\text{df-ws}}{\longmapsto} d_1'\ \mathsf{and}\ d_2'}\ \text{WS-BRANCH}$$

$$\frac{d \overset{\text{1-df}}{\longmapsto} d'}{\boxed{d} \overset{\text{df-ws}}{\longmapsto} \boxed{d'}}\ \text{WS-DECLBOX}$$

in many implementations of work stealing are represented directly in the abstract syntax tree instead.

My work-stealing semantics is defined in two parts. First, I define a pair of equivalence relations on expressions and declarations that allow work to be stolen. Second, I define transition rules for parallel execution of expressions and declarations, including the boxed forms introduced in this section. Figure 3.11 gives the equivalences and transition rules for the work-stealing semantics. This semantics is parametrized by the scheduling policy used by individual processors. In effect, these transition rules describe a scheduling policy that allows local scheduling decisions to be made by each processor. The rules presented here use a depth-first policy (to reflect a common implementation choice), but other policies may substituted by changing the premises of the WS-EXPBOX and WS-DECLBOX rules.

When a processor has no work remaining (*i.e.*, its boxed expression is a value) it must interact with other processors to steal work. In this semantics, the process of communicating is implemented using a pair of equivalence relations on expressions and declarations. They allow

a box to be moved up or down the abstract syntax tree of an expression and declaration. These relations are defined as the smallest equivalences satisfying the equations in Figure 3.11. For example, the following sequence of equivalences shows how work can be stolen by an idle processor. Here, the processor that initially owns the value declaration δ on the right steals work from the processor on the left by dividing its work in two.

$$\begin{aligned}
\boxed{(d_1 \text{ and } d_2)} \text{ and } \boxed{\delta} &\equiv^{\text{ws}} (\boxed{d_1} \text{ and } d_2) \text{ and } \boxed{\delta} \\
&\equiv^{\text{ws}} \boxed{(\boxed{d_1} \text{ and } d_2) \text{ and } \delta} \\
&\equiv^{\text{ws}} \boxed{(\boxed{d_1} \text{ and } d_2)} \text{ and } \delta \\
&\equiv^{\text{ws}} (\boxed{d_1} \text{ and } \boxed{d_2}) \text{ and } \delta
\end{aligned}$$

A work-stealing execution is then defined by an interleaving of the application of these equivalences and the work-stealing transition rules.

$$\frac{e \equiv^{\text{ws}} e'' \quad e'' \overset{\text{df-ws}}{\longmapsto} e'}{e \overset{\text{ws}}{\longmapsto} e'}$$

To start the evaluation of an expression, we must introduce one box for each processor. Since we cannot break an arbitrary expression into a sufficient number of sub-expressions or sub-declarations to add a box for each processor, we use a **let** expression that binds variables to unit values and places boxes around those unit values (one for each processor after the first). For example, to evaluate an expression e with two processors, we start with the following expression.

$$\text{let } x_1 = \boxed{e} \text{ and } x_2 = \boxed{\langle\rangle} \text{ in } x_1$$

It is straightforward to prove that this is a valid parallel implementation, as stated by the following theorems. As in the case of the breadth-first semantics, I must define a function $\text{erase}(e)$ that yields the expression e with all boxes removed.

Theorem 3.11. If $e \overset{\text{ws}}{\longmapsto} e'$ then there exists an expression e'' such that $\text{erase}(e) \overset{\text{nd}}{\longmapsto} e''$ and $e'' = \text{erase}(e')$.

As in previous sections, this theorem is proved by showing that each work-stealing rule is a constrained form of a non-deterministic rule.

Theorem 3.12 (WS Soundness)**.** For any closed expression e, if $e \overset{\text{ws}}{\longmapsto}{}^* v$ then there exist some g and h such that $e \Downarrow v; g; h$

Proof. As above, this follows from the previous theorem and the soundness of the non-deterministic semantics (Theorem 3.2). □

This semantics is non-deterministic in how it applies the equivalences $\equiv^{\text{ws}}$. While implementations of work stealing typically choose a random processor as the "victim" of each steal, any

algorithm for selecting such a target can be expressed using this semantics. To make progress, these equivalences must be applied to yield an expression does not contain any boxed expressions which are themselves boxed. Nested boxes would correspond to a state where a given task was present on more than one queue simultaneously. To understand why, consider the rules for evaluating boxed expressions, WS-EXPBOX and WS-DECLBOX. Each of these rules requires the boxed expression or declaration to take a step using the depth-first semantics. The depth-first semantics, however, does not provide any rules for boxed expressions so any doubly boxed expression will be stuck. In addition, there is no unconstrained reflexive rule in this semantics, such as ND-IDLE. This implies that every non-value expression must be contained within some box.

As I have yet to give a definition of work-stealing schedules in terms of an ordering on the nodes of a computation graph, I leave a formal connection between this semantics and such an ordering as future work.

3.6 Summary

In this chapter, I have shown several implementation semantics for the parallel language studied in this thesis, including those that implement depth-first and breadth-first scheduling policies. While none of these semantics describe a practical implementation, they act as a stepping stone between the cost semantics and more realistic implementations. As a technical device, I have used a non-deterministic semantics to model all possible parallel executions. This semantics, despite being non-deterministic in how it takes transitions, is still confluent. I have also shown how a more constrained, but still non-deterministic, semantics can succinctly describe properties of a scheduling policy such as greediness.

3.7 Related Work

My implementation semantics bear some relation to previous work on abstract machines (*e.g.*, the SECD machine [Landin, 1964], the CEK machine [Felleisen and Friedman, 1986]). Rather than defining some additional machine state, I have extended the syntax of my language. The semantics in this chapter are also related to abstract machine models of call-by-need evaluation (*e.g.*, Sestoft [1997]) where variables are used to track the evaluation of sub-expressions. My declarations are similar to the stack frames used by Ennals [2004] in his work on hybrid evaluation strategies. Unlike work on call-by-need languages, none of my semantics perform any speculation.

Greiner and Blelloch [1999] give a formal implementation of a parallel language as a parallel version of the CEK abstract machine. As in my work, their goal is to give a high-level description of the behavior of a scheduling policy. However, their implementation is specific to a depth-first policy.

Chapter 4

Semantic Profiling

Profiling has long been used as a means for improving the performance of programs. In general, profiling means collecting some data about resource use while a program runs. Resources include time, memory, cache, and input from or output to the network or persistent storage. In some cases, these measurements are performed directly: the simplest way to determine how long a (terminating) program will take to terminate is simply to run it and wait. In other cases, profilers use sampling to reduce the overhead associated with these measurements (*e.g.*, Arnold and Ryder [2001]). I refer to these methods collectively as instrumented profiling, since in each case the program or runtime is instrumented to collect performance measurements.

Though useful, instrumented profiling has some limitations with respect to sequential programs, and more significant problems with respect to parallel programs.

- Instrumented profiling is not portable.

  Sequential programs may perform very differently on different computers, for example, depending on the cache size or the amount of memory available. As discussed in the introduction, locality provides an abstract of the performance effects of memory caches.

  For parallel programs, programmers hope that performance will vary with the number of processors and processor cores and that programs will run faster with more processors. However, in some cases performance may degrade as the available parallelism increases: for example, memory use may increase. It is also important to understand *how much* faster programs will run as the number of processors increases. These issues are not addressed by instrumented profiling, as the program must be run on each hardware configuration (or at least enough different configurations that trends can be inferred).

- The results obtained through instrumented profiling are specific to a given language implementation.

  Programs compiled with different compilers (or even different versions of the same compiler) may exhibit very different performance. Many language implementations make use of just-in-time (JIT) compilers and in these cases, the version of the compiler is determined not by the programmer, but by the user.

  These effects are magnified in the case of parallel programs. As I have discussed, the choice of scheduling policy can have asymptotic effects on performance. The performance given by one scheduling policy will, more often than not, indicate very little about the per-

formance of other scheduling policies. Scheduling policies can also interact with aspects of the compiler in subtle ways. For example, inlining can change the order in which some expressions are evaluated and therefore which expressions are evaluated in parallel.

I will address these problems by considering an alternative method of profiling. **Semantic profiling** is a form a profiling that relies only on the semantics of the source program. While instrumented profiling can be viewed as a set of measurements about a program together with a particular implementation of the language, semantic profiling gives results independently of any implementation.

To achieve this implementation-independence, the semantic profiler described in this chapter is based on the cost semantics presented in Chapter 2. Though this method loses some precision over an instrumented profiler, its results should predict asymptotic resource use for any compliant implementation of the language. In addition, a semantic profiler can be used to reason about the performance of scheduling policies or hardware configurations even without full implementations of such policies or access to such hardware.

Like instrumented profiling, semantic profiling gives answers specific to a particular input and, similarly, only yields results for inputs where the program terminates. For each closed program, the cost semantics is used to derive a pair of cost graphs. I have implemented my cost semantics as an interpreter in Standard ML [Milner et al., 1997] using a straightforward transcription of the inferences rules. These graphs are then interpreted using the techniques described in this chapter to present the programmer with more concrete measures of performance. I remind the reader that cost graphs, and therefore the results of the profiler, are not abstract in the sense of abstract interpretation [Cousot and Cousot, 1977]: they do not approximate the extensional behavior of programs. Instead, they are abstract in that they do not explicitly account for many aspects of the language implementation, including the compiler and garbage collector.

Memory Profiling. In this chapter, I will present two techniques for memory profiling using my cost semantics. Memory profiling has been used to great effect in improving the performance of sequential call-by-need programs [Runciman and Wakeling, 1993a]. Memory profilers can either measure the number and sizes of allocated values, or it can measure the total memory required to represent all necessary values at any given time[1]. Both forms of memory profiling can be implemented using a semantic profiler. As modern garbage collection techniques make short-lived objects relatively cheap, I will focus on the uses of values stored in memory and the total amount of memory required to store value at any given time.

4.1 Simulation

The first technique that can be used by programmers to profile memory using my cost semantics is simulation. In this technique, the cost graphs are used together with an implementation of the scheduling policy to compute the number of reachable values at each step.

[1]Like a garbage collector, a profiler must use some approximation of the set of necessary values, for example, one based on reachability.

As discussed previously, each schedule can be described by a parallel traversal of the computation graph. To implement different scheduling policies in the profiler, one must simply implement a function that, given a set of nodes visited in the previous step, determines which should be visited in the current step. Policies can usually be implemented with a few lines of code. For example, depth- and breadth-first scheduling policies can be implemented using essentially only functions from the Standard ML Basis [Gansner and Reppy, 2002] implementation of lists. (The work-stealing scheduler requires slightly more effort.)

As the profiler is a sequential program, implementing scheduling policies in the profiler avoids most of the complexity associated with implementing a scheduling policy as part of the runtime: an online scheduler implemented as part of a language runtime is an example of a *concurrent* program, since its behavior depends crucially on the interleaving of memory accesses by the hardware. Such an implementation must carefully use synchronization primitives to guard any state shared among different processors. While the profiler could easily be implemented as a *parallel* program, I leave such an implementation as future work. Note, however, that the extensional semantics of such a implementation would be identical, by definition, to the one I describe in this chapter. I should further note that my implementation is not a hardware simulator. It is an abstraction of the actual implementation: it does not account for the precise cost of visiting each node and it only approximates the interleaving of tasks that would occur in the runtime implementation. In the simulation, processors move in lock-step as they visit nodes and fully "synchronize" after each step.

Implementing Scheduling Policies. The depth-first scheduling policy is simulated using a list to represent the set of ready nodes. (Recall that a node is ready if all its parents have been visited, but it has not yet been visited.) To simulate a P processor execution, the first $|P|$ nodes are removed from the list. The profiler must then determine the set of nodes that are enabled in this step. This is accomplished by mapping a function defined by the edges of the computation graph across these nodes. The results are then concatenated together and added at the front of the list of ready nodes. These concatenated nodes must be keep in the same order as the corresponding nodes that were removed from the ready list to ensure a depth-first traversal of the graph [Blelloch et al., 1999]. Note that the profiler performs none of the actual computation of the program. The process of simulating a scheduling policy requires only nodes and edges of the computation graph. The breadth-first policy uses a similar implementation, but instead of adding new nodes to the beginning of the list, it adds them at the end.

The work-stealing scheduling requires slightly more effort: it must maintain a separate set of nodes for each processor. Each set is maintained using a list as in the case of the depth-first scheduler. If a list becomes empty, the list from which to steal is randomly chosen.

I have also implemented a variant of each of these policies that implement the coarser-grained schedules used by my runtime implementation. That is, the versions described above schedule at the granularity of individual nodes. Put another way, those versions allow threads to be preempted at arbitrary points in the execution of the program. In my runtime implementation (as described in Chapter 7), scheduling is performed cooperatively by processors, and thus a thread will only be interrupted if it makes a call to the scheduling library, for example, to spawn a new task. In terms of the cost graphs, that means that whenever a processor visiting a node n_1 enables

exactly one new node n_2 then that processor will visit n_2 in the next step, regardless of what nodes are enabled by other processors.

Determining Memory Use. Once the profiler has computed the set of nodes visited at a given step, the amount of memory required at this step can be determined by following the steps outlined in Chapter 2. In particular, the profiler uses the set of visited nodes to determine the set of roots (unvisited nodes that are the sink of a heap edge that leads from an unvisited node). The total memory use is determined by the set of nodes in the heap graph that are reachable from these roots. Because the number of nodes visited in each step is small compared to the total number of nodes in the computation graph, it is more efficient to maintain the set of edges that cross between unvisited and visited nodes. At each step, the profiler simply adds and removes edges from that set based on which nodes are visited.

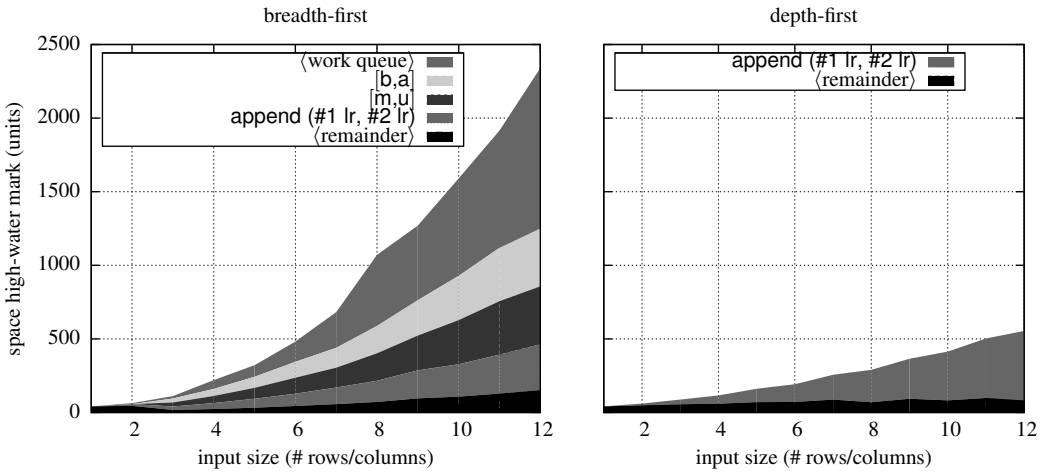
While it is possible to use this technique to show the total memory use as a function of time since the beginning of program execution (as in Runciman and Wakeling [1993a]), I have found it more useful to record the maximum amount of memory required at any point during an execution. I refer to this quantity as the high-water mark of memory use. This quantity may then be plotted as a function of input size to show trends in memory use.

The profiler is also able to attribute memory use to variables and expressions found in the source program. To do so, it uses an extended version of the cost semantics that associates a source expression or other annotation with each node on the heap graph. For example, the node corresponding to the allocation a (sequential) pair comprised of the values bound to variables `x` and `y` would be annotated with the source expression `(x,y)`. Nodes representing the allocation of function values are annotated with a list of variables that appear freely in the body of function (excluding the function argument). These lists are written within square brackets (*e.g.*, `[a,b,c]`). For uses of values (*e.g.*, in the applications of array primitives such as `append`) nodes are annotated with the source code of the application. Note that this means that the annotations of some nodes may be sub-strings of the annotations of other nodes. For example, there may be a node with an annotation such as `#1 lr` as well as one annotated `append (#1 lr, #2 lr)`. While it is possible that this could lead to very long annotations, this has not been a problem in my experience. In general, ML programmers tend to name intermediate results (as is good programming practice), and this limits the length of annotations. In addition, if we only consider the annotations of nodes at the sources or sinks of root edges (as we do in the subsequent development), there will be little duplication of source code in the annotations in which we are interested.

These annotations support profiling both the creation of heap values and the uses of those values. If we are interested in what sort of values are present in the heap, then we look at the annotations associated with the sinks of heap edges. As noted in Chapter 2, the sink of a heap edges always represents a value. When showing memory use (or "retainers" in the terminology of Runciman and Röjemo [1996]), we instead consider the sources of heap edges. These correspond both to values as well as uses of those values (*e.g.*, function applications, pair projections).

In this chapter, I focus on the uses of heap values (or retainers). Using annotations, my profiler is able to break down the total memory use by attributing part of this total to each root. As described previously, the total memory use is proportional to the total number of heap nodes reachable from the roots. For nodes that are reachable from more than one root, the memory

Figure 4.1 Matrix Multiplication Memory Use. These plots show the memory required to multiply two matrices using two different scheduling policies (breadth- and depth-first) as determined by my profiler. Each plot shows as the high-water mark of space use as a function of the number of rows or columns of the input matrices.



associated with that node is divided equally among all such roots. For example, if there are six nodes in the heap graph that are each reachable from three roots, then each root would be charged two units of memory.

In the examples considered in the section, I have attributed memory to parts of the source program according to the description given above. For larger programs, a profiler should also include information about which source file contains the given sub-expression and the position of the sub-expression within that source file.

4.1.1 Example: Matrix Multiplication

As an example, consider again the implementation of matrix multiplication shown in Figure 2.1. Figure 4.1 shows some results obtained using the profiler and the process described above. The two plots show the memory use using breadth-first (left) and depth-first scheduling (right) policies, both using two processors. In both cases, the input size (the number of rows or columns of the input matrices) varies between one and 12. On the vertical axis, I show the memory use high-water mark. The profiler does not account for the differences in the amounts of memory required to represent booleans, integers, or pairs, but instead assumes that all values can be represented using the same amount of memory. In addition, it does not account for the any overhead introduced by the garbage collector (*e.g.*, header words). As such, these plots should only be used as indicators of asymptotic memory use.

Memory use under the breadth-first scheduler is broken down into five parts. The first part, "⟨work queue⟩," is space associated with the schedulers internal data structures. In the implementation of breadth-first scheduling, this is determined by the length of the list of ready nodes. The next two components, $[\mathrm{b}, \mathrm{a}]$ and $[\mathrm{m}, \mathrm{u}]$, represent the memory used by closures allocated during the evaluation of the reducei (see Appendix A). The fourth entry is labeled

"append (#1 lr, #2 lr)." This corresponds to the allocation of the vector built by appending the two vectors returned from recursive calls to loop. Finally, all remaining memory use not attributed to one of these four entries is grouped together. In the case of depth-first scheduling, only the final two entries appear. That is, nearly all the memory use at the high-water mark can be attributed to vectors allocated by append.

From plots such as these, a programmer might look for the sources of performance problems in the source program. In the case of the depth-first evaluation, memory use increases as the input gets larger (as expected). Most of the memory use can be attributed to the point in the source code where the result of evaluation is allocated. In the breadth-first case, the total memory use is much higher. Moreover, this memory use is attributed to the schedule itself (in the form of the work queue) and to closures allocated during evaluation. These both indicate that this schedule policy is perhaps not a good choice for this program: more memory is used in storing the intermediate state of program evaluation than the values of the program itself. The performance of this example, especially with respect to the scheduling policy, will be considered in more detail below.

4.2 Visualization

The challenge of visualization in general is taking large and high-dimensional data sets and reducing their size and dimensionality so that users can glean some information from their presentation. This reduction must account for the kinds of conclusions that users wish to draw and how these conclusions relate to the original data.

In the case of cost graphs, even small programs may result in 10,000s of nodes. We might analyze these graphs while varying input size, scheduling policy, the time elapsed since the beginning of the execution, or by considering different sub-expressions that occur within a program. In the previous section, we fixed the scheduling policy and, for each input, picked a particular point in the execution. This enabled us to easily render memory use for different input sizes and show which parts of the program led to this memory use. However, those plots are not particularly useful for understanding how a scheduling policy affects space use. In this section, we choose to focus on a different set of dimensions. In particular, we will concentrate less on the effects of input size and instead use visualizations to understand the effects of the scheduling policy for a particular input.

To visualize the behavior of a scheduling policy, it seems natural to render some form of the computation and heap graphs: these graphs embody constraints about the order in which nodes are visited and how memory can be used. Unfortunately, as noted above, these graphs may easily contain 10,000s nodes. Given the resolution of current displays, this leaves only a handful of pixels to render each node (as well as the space between nodes) if we were to render the entire graph. In addition, generic tools used to layout these graphs for display cannot cope with such large inputs. While it might be possible to use more specialized algorithms to perform graph layout, this layout has some inherently non-local aspects (*e.g.*, when positioning one of two peer nodes, a layout algorithm must consider all children of *both* nodes to ensure adequate space). Thus it seems necessary to consider rendering graphs which are related, but not identical, to the original cost graphs. Transforming cost graphs derived from the semantics into something that

can be rendered on the screen will be discussed in detail below. As above, we shall only consider the point of execution at which the program reaches its high-water mark of memory use.

4.2.1 Distilling Graphs

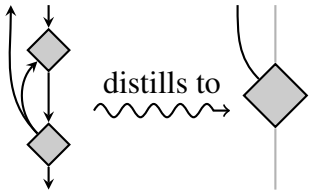
Cost graphs, as derived from the cost semantics, are too large to either be rendered efficiently or understood by programmers. In this section, I will describe a technique called distillation that mechanically reduces these large graphs. In addition, distillation also associates additional attributes with nodes and edges in the output graph, including size, color and relative position, that can be used to convey additional information to the programmer.

As the computation and heap graphs share nodes, I often show one superimposed over the other. All of the graphs shown in this chapter are mechanically generated from our cost semantics and an implementation of the following numbered rules. I determined these rules in part through experimentation, but in a large part upon the principles discussed below.

I am most interested in the parallel structure of program execution. Thus a series of nodes that describes sequential computation can be rendered as a single node. I use node size to indicate the amount of sequential computation and node position to indicate computational dependencies.

1. For each node n in the computation graph, if n has out-degree one, n has in-degree one, and the (sole) predecessor of n also has out-degree one, then coalesce n with its predecessor. The area of the resulting node is the sum of the area of the two coalesced nodes. Nodes in the original graph have unit area.
2. Remove edges between coalesced nodes in both the computation and heap graphs. (There are no self-edges in the result.)
3. Constrain the layout so that the vertical position of nodes respects the partial order determined by computation edges.
4. Constrain the layout so that the horizontal position of nodes reflects the sequential schedule: nodes that are visited earlier in the sequential schedule should appear further to the left.

In the output graph, I draw computation edges in a light gray as the structure of the computation graph can be roughly derived from the layout of the nodes: among vertically aligned nodes, those closer to the top of the graph must be visited first. Nodes are also drawn so that the sequential schedule follows a left-to-right traversal. Finally, I also omit arrowheads on edges as they add to visual clutter. An example of node coalescing is shown here.



Due to the structure of the cost graphs, coalescing will never create non-unique edges in the computation graph (*i.e.,* more than one edge between the same pair of nodes). On the other hand, it will often be the case that there are several *heap* edges between the same pair of nodes. We considered trying to represent these duplicate heap edges, for example, by weighting heap edges

in the output according to number of duplicates. This, however, ignores any sharing among these heap edges and may lead to confusing visual results (*e.g.,* a group of heavy edges that represents a relatively small amount of heap space). For graphs distilled independently of a particular point in time, duplicate heap edges are removed, and all heap edges are given the same weight.

If we restrict ourselves to a single moment in time, we can overcome the difficulties with representing sharing and also focus on the behavior of a specific scheduling policy and its effect on memory use. I use the color of both nodes and heap edges to highlight the behavior of the scheduling policy at step i, the moment when memory use reaches it high-water mark.

5. Fill nodes with black if they are visited at or before step i, and gray otherwise.
6. Draw heap edges that determine roots at step i in black, red or another visually distinct color. Draw other heap edges in gray.

We must be careful about which nodes we coalesce, as those heap edges that determine roots connect only visited and unvisited nodes.

7. Avoid coalescing two nodes if one has been visited at or before step i and the other has not.

Finally, now that we have restricted ourselves to a single moment in time, we can properly account for sharing in the heap graph.

8. Weight each heap edge according to its share of the total amount of heap space reachable from that edge at step i.

Thus the total memory use at the high-water mark may be determined from the total weight of the black heap edges (*i.e.,* those representing the roots). Generally, the above rules mean that the visual properties of distilled graphs can be interpreted as follows.

The greater the ...	then the more or longer the ...
graph height	sequential dependencies
graph width	possible parallelism
node size	computation
edge thickness	memory use

4.2.2 GraphViz

The graphs shown in this chapter are rendered with GraphViz [Gansner and North, 2000] an open-source suite of tools for visualization graphs.[2] GraphViz uses a text format called "dot" to describe graphs. The process described in the previous section showed how to use position, size and color to reduce the number of nodes in a graph; in this section, I will show how these graphs are described in a format that can be read by GraphViz.

Figure 4.2 shows portions of the dot file used to describe distilled cost graphs. The dot format consists of three type of entities: graphs, nodes, and edges. In this work, I will only use a single graph per dot file, but it is possible to use them to describe sub-graphs as well. Line 2 sets the spline style to "line:" this instructs GraphViz to draw edges as straight line segments instead of curved splines. Line 3 defines the ordering of edges and forces all nodes to be laid out so that the

[2] `http://www.graphviz.org/`

Figure 4.2 Example `dot` File. Files such as this one are produced by my profiler and then used by GraphViz to generate the visualizations shown in the other figures in this chapter.

```
1   graph G {
2     splines=line;
3     ordering=out;
4     node [label=‘‘’’,shape=diamond];
5     ...
6     // Nodes
7     node [width=0.048,height=0.048,fillcolor=black,style=‘‘filled,setlinewidth(4)’’] 100;
8     node [width=1.06,height=1.06,fillcolor=gray66,style=‘‘filled,setlinewidth(4)’’] 101;
9     ...
10    // Computation graph edges
11    100:s −− 101:n [color=gray66,style=‘‘setlinewidth(4)’’];
12    ...
13    // Heap graph edges
14    101 −− 100 [constraint=false,color=gray66,style=‘‘setlinewidth(3.4)’’];
15    102 −− 100 [constraint=false,color=red,style=‘‘setlinewidth(6.5)’’];
16    ...
17  }
```

left-to-right, depth-first traversal of these nodes corresponds the sequential schedule. The node specification on line 4 does not define an actual node, but sets defaults for all nodes. In particular, nodes should be drawn without labels and should be drawn as diamonds.

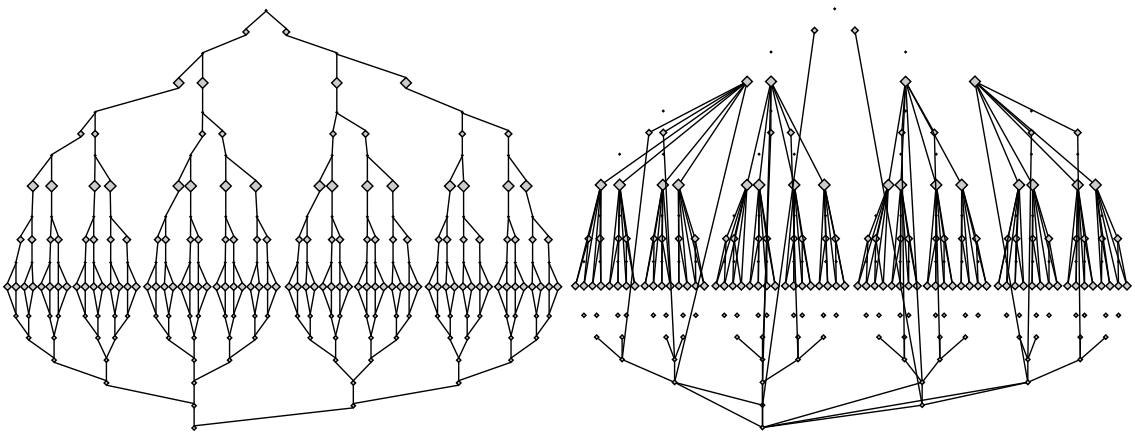
Lines 7 and 8 define two nodes. In both cases the node size is set using the height and width parameters, and the style is set so that nodes will be filled and drawn with an outline. The first node has been visited and is filled black while the second node is an example of an unvisited node and is therefore filled gray. The node names 100 and 101 are unique numbers generated during distillation.

An edge of the computation graph is described on line 11. The annotations ":s" and ":n" indicate that the edge should be drawn from the southernmost (or bottom) point of the first node to the northernmost (top) point of the second node. Computation edges are drawn in gray. Heap edges are described on lines 14 and 15. Heap edges are not used in determining the layout of nodes hence the attribute "constraint=false." If the graphs are to be drawn to illustrate a specific step in a schedule, then the color and width of head edges is determined based on whether or not the given edge is a root at that step.

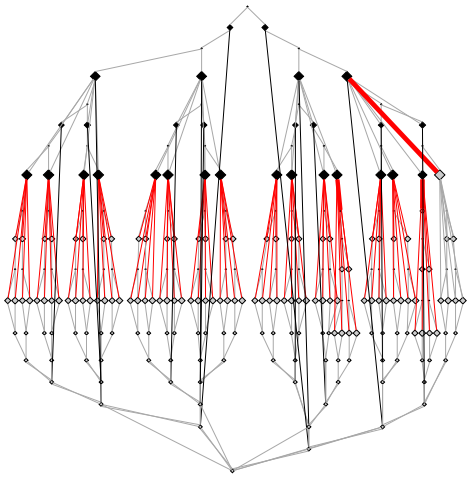
4.2.3 Example: Matrix Multiplication

Figure 4.3(a) shows the cost graphs for the matrix multiplication example run with inputs of four rows and columns. On the left side is the computation graph. This graph shows that the parallel structure of matrix multiplication is very regular and well-balanced. This is expected since each time we split a densely represented matrix in half, we split the work in half as well. On the

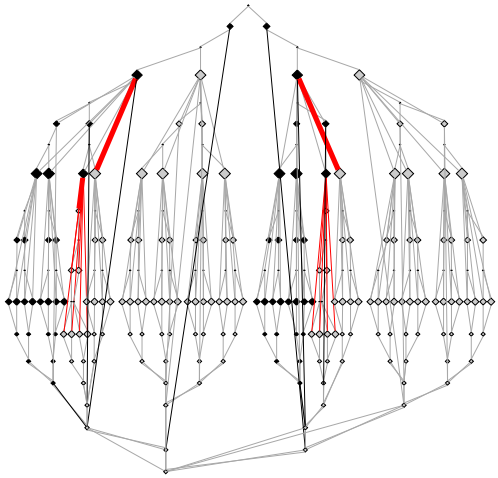
Figure 4.3 Matrix Multiplication Visualization. These graphs are produced by my profiler given the implementation of matrix multiplication shown in Chapter 2. The input matrices each have four rows and columns.



(a) Cost Graphs



(b) Breadth-First, $|P| = 2$



(c) Work-Stealing, $|P| = 2$

right side, the heap graph is drawn in such a way so that the node layout matches that in the computation graph. In subsequent examples, these two graphs will be drawn on the same set of nodes. Note that heap edges need not join two nodes of adjacent depth, while computation edges will only join nodes whose depth differs by exactly one.[3] The heap graph shows that most of the memory use can be attributed up to nodes that appear at the widest part of the graph. Nodes after this point contribute less to the overall use of memory.

Figure 4.3(b) shows the step of the breadth-first schedule with two processors that reaches the memory high-water mark. In this part of the figure, the computation and heap graphs are drawn on the same set of nodes. Note that this step occurs in a breadth-first schedule: every black node is either at a smaller depth than or, if not, then further to the left of every gray node. Heap edges whose sources are roots at this step are highlighted. This figure shows that the breadth-first schedule requires the most memory at the point at which the scheduler has (nearly) explored the full breadth of the graph.

The step that reaches the high-water mark under a work-stealing schedule with two processors is shown in Figure 4.3(c). First, note that this point can be described as "breadth-first until saturation, then depth-first" as described by Mohr et al. [1990]: the schedule implements a breadth-first schedule until it reaches a depth with more than two nodes and from that point forward, each processor follows a depth-first traversal of its own sub-graph. At the high-water mark, each of these processors uses memory in a manner similar to the right-most side of the breadth-first schedule. This is where the similarity ends, however, since the work-stealing schedule need not "pay" for partially evaluating all of the $O(n^3)$ threads.

These graphs can be used to explain the memory use shown in Figure 4.1. From looking at these graphs, we can see that the amount of memory required by the breadth-first schedule will always be proportional to the width of the graph $O(n^3)$ regardless of the number of processors. The amount of memory required by depth-first and work-stealing schedulers, on the other hand, will only be proportional to the size of the result and the number of processors.

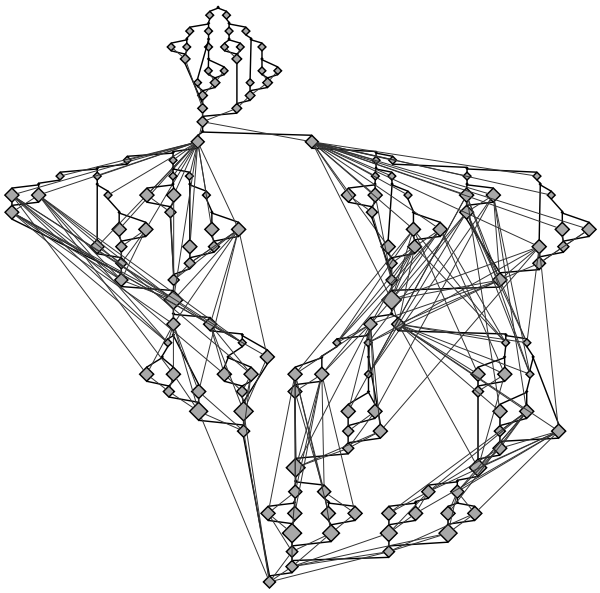
4.2.4 Example: Quickhull

We now consider a second example. The quickhull algorithm is a method for finding a convex hull for a set of points. Here, we consider an implementation of this algorithm for points in two dimensions. Like quicksort, this algorithm partitions input and recursively solves each partition before joining the results. The algorithm begins by finding the two points with the smallest and largest values for a single coordinate: these two points must lie on the convex hull. It proceeds by partitioning the input set, in parallel, by dividing the plane in half using the line connecting the two extreme points. It then recursively processes each half in parallel. The resulting set of points is simply the union of the sets of points returned from each recursive call.

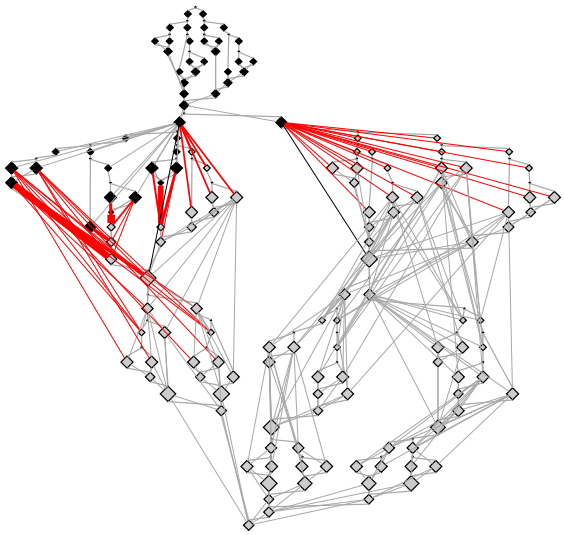
The cost graphs for this example are shown in Figure 4.4(a). Note that the structure is much more irregular than that of the matrix multiplication example. Also note the frequent widening and narrowing of the graph. These narrowing points correspond to synchronizations between the partition of the input points and the recursive calls.

[3] If we consider more general forms of parallelism, as for example in Chapter 6, computation edges will not be restricted in this manner.

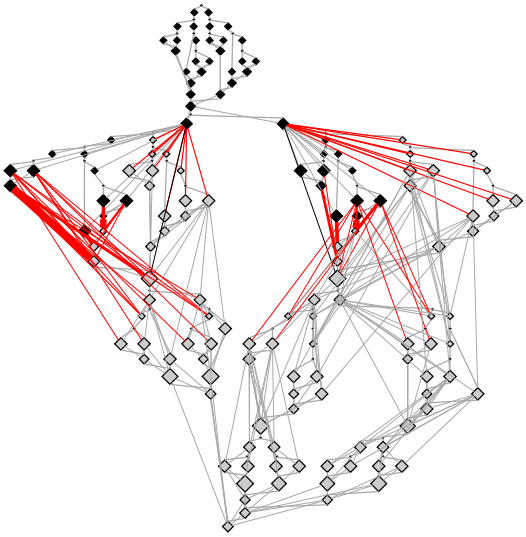
Figure 4.4 Quickhull Visualization.



(a) Cost Graphs



(b) Parallel Depth-First, $|P| = 2$



(c) Work-Stealing, $|P| = 2$

Figures 4.4(b) and 4.4(c) show points at which the depth-first and work-stealing scheduling policies reach their respective memory use high-water marks. Both represent executions with two processors. Again, nodes are colored to distinguish those nodes that have been visited from those that have not, and heap edges that point to roots are also highlighted.

In this case, the work-stealing policy performs more poorly than the depth-first policy because it starts, but does not finish, computing these partitions. At this point in time, the program is holding onto the entire original set plus half of the resulting partition. The same pattern appears for each processor. The depth-first scheduling policy does not exhibit this behavior.

I reached these conclusions in part by annotating nodes with the corresponding portion of the source code and using this information as part of the display of graphs. Unfortunately, I have found it difficult to present this information using GraphViz because the amount of text that must be displayed quickly clutters the available space. One possibility for future work is using an interactive tool for selectively displaying this information.

4.3 Summary

In this chapter, I have shown how a cost semantics can serve as the basis for a profiler. Semantic profilers yield results that are independent of any implementation of the language. As I shall discuss in Chapters 7 and 8, this kind of semantic profiling led me to the discovery of a memory leak caused by a bug in a compiler optimization.

In the case of parallel programs, the profiler must also account for the scheduling policy to make concrete predictions about program performance including memory use. Fortunately, at the level of abstraction used by a semantic profiler, implementing scheduling policies is straightforward and much less error-prone than doing so in a runtime implementation.

I have shown two ways that cost graphs can be used to present the programmer with a visual representation of program execution. In the first case, I showed how memory use varies with input size and the scheduling policy. In the second case, I have shown how the essential properties of cost graphs can be distilled and rendered using an off-the-shelf graph layout program.

The profiler implementation presented in this chapter is still a prototype and offers many opportunities for improvement. As a result, the programs I have presented are relatively small. To support to larger programs, the profiler must be extended to support a broader subset of Standard ML, including pattern matching and modules. To yield results efficiently, it may be necessary to "fuse" parts of the implementation of the cost semantics and graph distillation. That is, rather than generating the entire cost graphs and then distilling them, the profiler could support larger programs (and larger inputs) by distilling graph nodes as they are generated. I leave this and other extensions of my semantics profiler to future work.

4.4 Related Work

Profiling tools (*e.g.*, gprof [Graham et al., 2004]) play an integral role in developing programs. To my knowledge, all widely used profilers are based on the instrumentation of a particular compiler or runtime system. Hardware simulators offer even more precise results about programs or the

language implementations (*e.g.*, Chen et al. [2007]), but such results are obviously specific to the simulated architecture.

In their seminal work on memory profiling for call-by-need functional programs, Runciman and Wakeling [1993a] demonstrate the use of a profiler to reduce the memory use of a functional program by more than two orders of magnitude. They measure memory use by looking at live data in the heap, thus their tool is tied to a particular sequential implementation. Sansom and Peyton Jones [1993, 1995] extend this work with a notion of "cost center" that enables the programmer to designate how resource use is attributed to different parts of the source program.

Appel et al. [1988] describe a method for profiling for a call-by-value functional language. They instrument source code at an early stage of the compiler by inserting additional statements into the program. These statements that cannot be distinguished from ordinary program statements by later compiler phases. This allows the profiler to accurately assign costs to different parts of the source program, because the semantics of the added instrumentation must also be preserved by the rest of the compiler. It should be noted, however, that these later compiler phases may still affect the costs themselves.

There has been work on profiling methods and tools for parallel functional programs [Roe, 1991, Runciman and Wakeling, 1993b, Hammond et al., 1995, Charles and Runciman, 1998]. These works use a combination of instrumentation and simulation to determine whether or not a program expresses adequate parallelism. Conversely, they also note that too much parallelism can also lead to poor performance. Thus their goals are to help programmers find an appropriate granularity of parallel tasks rather than understanding how scheduling affects the resource requirement of an application. None of this work measures how different scheduling policies affect memory use.

In the course of their profiling work, Hammond and Peyton Jones [1992] considered two different scheduling policies. While one uses a LIFO strategy and the other FIFO, both use an evaluation strategy that shares many attributes with a work-stealing policy. These authors found that for their implementation, the choice of policy did not usually affect the running time of programs, with the exception in the case where they also throttled the creation of new parallel threads. In this case, the LIFO scheme gave better results. Except for the size of the thread pool itself, they did not consider the effect of scheduling policy on memory use.

Chapter 5

Vector Parallelism

In Chapter 3, I presented several online scheduling policies. These policies use information that only becomes available when the program is run, including the number of processor cores and the program input, to assign tasks and balance load. In this chapter, I consider a program transformation that instead performs part of this assignment during compilation.

While such offline scheduling policies must make scheduling decisions with less information than online policies, opportunities for parallel execution that can be distinguished at compile time can be tightly integrated with application code, lowering the computation overhead of scheduling. This becomes especially important as parallel tasks become very small, as in the case of hardware architectures that support vector parallelism. Here, processors are capable of executing multiple operations simultaneously, but unlike in the case of multi-core architectures, the kinds of simultaneous operations are tightly constrained. Single instruction multiple data (SIMD) architectures can perform a single operation in parallel on many data, for example by simultaneously computing the element-wise sums of the elements of two vectors. Such operations are the foundation of graphics processors and are also common in a limited form in many general-purpose processors.

This chapter will examine a program transformation called flattening [Blelloch and Sabot, 1990]. **Flattening** transforms nested parallelism into the form of data-level parallelism provided by the architectures described above.[1] Nested parallelism occurs when a function that is invoked in parallel in turn spawns further opportunities for parallel evaluation, for example as in divide-and-conquer algorithms. Many data-parallel languages prohibit nested parallelism (*e.g.*, High Performance Fortran Forum [1993]). As described below, flattening generates data-parallel versions of user-defined functions by composing primitive vector operations, such as vector addition. Flattening was originally described by Blelloch and Sabot [1990] and later implemented as part of the NESL programming language [Blelloch et al., 1994] and Data Parallel Haskell [Chakravarty and Keller, 2000, Peyton Jones et al., 2008].

The parallelism introduced by parallel pairs (Chapter 2) allows only a single new thread of parallel evaluation to be spawned at each step. This means that to take advantage of P execution units, a program execution must have a corresponding computation graph with depth of at least

[1] In this chapter, I will use the term "nested" in a slightly different, but related, sense. Previously, I used "nested" in the sense of "well-nested" and as opposed to more general forms of parallelism. In this chapter, I use it in a more strict sense to mean "well-nested but not flat" since flat parallelism is trivially well-nested.

$2 \log P$. This is a poor match for the SIMD architectures considered in this chapter: whenever there are P elements of data available, all such elements can be processed in parallel on a SIMD machine. Instead of pairs, we will consider vectors of elements of arbitrary length. Vectors are immutable just as the data structures introduced in previous chapters. I shall assume that the elements of a vector of length n are indexed from 1 to n.

In the sequel, I will reconsider the example of matrix multiplication in light of SIMD architectures and then describe the extensions to my language and semantics necessary to express these sorts of data-parallel programs. I will give an explanation of flattening, and finally, I will show how the behavior of flattened programs can be described using a form of labeling [Lévy, 1978].

5.1 Example: Matrix Multiplication

As an example, consider the following variation on parallel matrix multiplication. We shall again assume that both input matrices are the same size and that each is matrix is represented as a vector of vectors of elements. (To further simplify the presentation, we assume that the one matrix is in row-major form and the other in column-major.)

```
fun vv_mult' u v = sum (map op* (zip (u, v)))
fun mm_mult' (A, B) = map (fn u ⇒ map (vv_mult' u) B) A
```

This implementation leverages parallel operations over vectors, including sum, map, and zip, to provide a much more concise version than that found in Chapter 2. Notice that there are several levels of nested parallelism: in the definition of mm_mult' the anonymous function argument to map itself invokes map with the implementation of the vector dot product, which in turn uses parallel operations zip[2] and (again) map.

If we only consider parallelism at the leaves of the call tree, then we might only perform in parallel those scalar multiplications used to compute a single element of the resulting matrix. However, given adequate parallelism in the hardware, the scalar multiplications for several result elements might also be performed in parallel. Unfortunately, it is not obvious how to do so given the program as expressed above.

The intuition behind flattening is to transform the program so that all of these scalar multiplications are visible at the same time. The flattened form of this implementation can be understood in its essence through the following four steps. Assume that the input matrices each contain n^2 elements represented as vectors of vectors.

1. replicate both input vectors n times as follows:
   - for the first input vector A, put all n copies of the first row in sequence, followed by n copies of the second row, *etc.*
   - for the second input vector B, replicate the rows in order (*e.g.*, so that the second copy of the first row follows the first copy of the last row)
2. compute a point-wise product of the elements

[2] zip takes two lists and "zips" them together yielding a single list where each element consists of a pair of elements, one from each input list.

3. compute a segmented prefix sum of this vector
4. take the last element of each sub-vector

A prefix sum is a vector operation that, given a vector of k numbers, computes a vector where the i^{th} element of the result is the sum of the first i elements of the input. Such an operation can be implemented using a divide-and-conquer algorithm in $\log(k)$ parallel steps. A segmented prefix sum is a similar operation but where sums are not accumulated across segment boundaries. The other operations that appear in these steps (replicating and indexing elements to form new vectors) can all be computed in a constant number of parallel steps.

As example of these operations computing the result of a matrix multiplication, take the following two matrices.

A: $\begin{bmatrix} 1 & 3 \\ 0 & 4 \end{bmatrix}$ B: $\begin{bmatrix} 2 & 7 \\ 5 & 1 \end{bmatrix}$

Represented as vectors of vectors of elements (and recalling that B is column-major), we might illustrate them as follows.

A: [[1 3] [0 4]]

B: [[2 5] [7 1]]

1. The first step in the above sequence is to replicate each of the rows/columns in the input matrices.

   [[1 3] [1 3] [0 4] [0 4]]

   [[2 5] [7 1] [2 5] [7 1]]

   Though I have illustrated these matrices as nested vectors, flattening implementations use a technique called vector segmentation to represent the elements found in these vectors separately from the structure imposed by this nesting. This will be discussed in more detail below.

2. We then compute the element-wise scalar products.

   [[2 15] [7 3] [0 20] [0 4]]

   Note that a processor with 8-way parallel arithmetic can compute these products in a single step.

3. Next, we compute segmented prefix sums.

   [[2 17] [7 10] [0 20] [0 4]]

4. Finally, we take the last element of each sub-vector.

   [[17 10] [20 4]]

Written in matrix form, we have the desired result.

$\begin{bmatrix} 17 & 10 \\ 20 & 4 \end{bmatrix}$

Figure 5.1 Syntax Extensions For Nested Vector Parallelism.

(expressions)	e	::=	$\ldots \mid \{e_1, \ldots, e_k\} \mid c \mid i$
(constants)	c	::=	replicate | index | map | concat | length | sub | $\ldots$
(integers)	i, j, k	$\in$	$\mathbb{Z}$
(values)	v	::=	$\ldots \mid \langle v_1, \ldots, v_k \rangle^{\ell} \mid c \mid i$

The challenge of implementing flattening is to derive such alternative implementations mechanically. This process will be discussed in more detail below.

I have used dense matrix multiplication as an example because it can be presented in a relatively concise form. The real power of flattening, however, is in exposing additional parallelism in algorithms with less regular structure. For example, when computing the product of two sparsely represented matrices, some elements of the result may only require summing a small number of scalar products while others may be the sum of many such products. (In the dense case, each output element requires the sum of exactly n products.) Flattening implements a form of load balancing by combining the computation of sparsely populated rows and columns. In doing so, flattening plays a role similar to that played by online scheduling policies in ensuring that all processors have adequate work to perform.

5.2 Syntax

A vector expression of length n will be written $\{e_1, \ldots, e_n\}$. To simplify this presentation, I will only allow such expressions as intermediate forms that appear during evaluation. This is not a significant restriction because the real expressive power of the extensions described in this chapter is derived from the fact that the primitives described below introduce an unbounded amount of parallelism in a single step. In addition, any literal vector of fixed length can be generated using the other primitives described below. As with pairs, each value representing a vector carries a location uniquely identifying the memory associated with that value.

To facilitate the process of adding new operations for manipulating vectors to the language, I extend the syntax of expressions with a new category of constants c. Constants will be primitive operations implemented as part of the compiler and runtime system. The first constant replicate takes a pair of arguments and creates a new vector whose length is determined by the first argument and where each component is given by the second argument. The second constant, index, takes a single integer argument and generates a vector of the given length where each element is equal to its own index. The third constant map takes a pair of arguments. Like the familiar function that applies a function to each element of a list, map takes a function and a vector, and applies the function to each element in that vector to create a new vector of the same length. As will be shown in the cost semantics in the next section, replicate, index and map will be additional sources of parallelism.

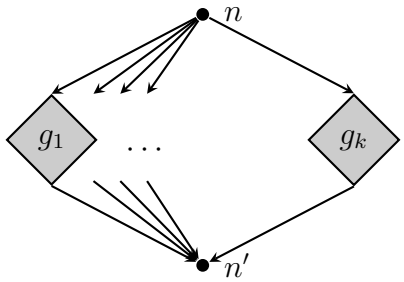
The constant concat also acts as a function taking a pair of arguments. In this case, both arguments must be vectors with elements of the same type; the result of applying concat to such

Figure 5.2 Graph Operations for Vector Parallelism.

k-ary Parallel Composition

$$\bigotimes_i g_i$$

$$g_i = (n_i; n_i'; E_i) \ \text{(for } i \in [1, k])$$



$$(n; n'; \bigcup_i E_i \cup \bigcup_i \{(n, n_i), (n_i', n')\})$$

$$n, n' \text{ fresh}$$

a pair is a new vector whose elements are the concatenation of elements of both arguments. The constants length and sub act as their sequential counterparts, yielding the length of a vector and the element found at a given index, respectively.

Though many programs may require additional primitive operations on vectors, these will be sufficient for the purposes of the current exposition. As constants, these new expressions will also be considered values, and as for unit values, we do not annotate constant values with labels.

Finally, I add integers to the syntax of the language, as these will be necessary to describe the semantics of constants such as index and length. In the examples used in this chapter, I shall further assume the addition of floating-point numbers as well as constants that implement arithmetic operations over both integers and floating-point numbers.

5.3 Semantics

As with other extensions of our language, our first task is to describe the extensional behavior of vector operations and to define the costs associated with these operations.

To describe vector parallelism using computation graphs, I must the extend the parallel composition of graphs as described in Chapter 2, to arbitrary (finite) sets of graphs. Figure 5.2 defines k-ary parallel composition. As for binary parallel composition, we add two additional nodes to the graph, one as the start node of the result and one as the end node. The edges of the resulting graph are determined by the union of the edges of each sub-graph, plus two additional edges for each sub-graph: one leading from the start node of the resulting graph and one leading to the end node of that result. Like the graphs resulting from binary composition, this graph states that nodes in any sub-graph may be visited simultaneously.

Figure 5.3 Cost Semantics For Vector Parallelism.

$$\frac{e_1 \Downarrow \mathsf{replicate}; g_1; h_1 \qquad e_2 \Downarrow \langle k, v\rangle^{\ell_2}; g_2; h_2 \qquad n, n'_i, \ell \text{ fresh}}{e_1\ e_2 \Downarrow \langle \underbrace{v, \ldots, v}_{k \text{ times}}\rangle^{\ell}; g_1 \oplus g_2 \oplus [n] \oplus (\bigotimes_i [n'_i]) \oplus [\ell]; h_1 \cup h_2 \cup \{(n, \ell_2), (\ell, \mathrm{loc}(v))\}}\ \textsc{c-Replicate}$$

$$\frac{e_1 \Downarrow \mathsf{index}; g_1; h_1 \qquad e_2 \Downarrow k; g_2; h_2 \qquad n_i, \ell \text{ fresh}}{e_1\ e_2 \Downarrow \langle 1, \ldots, k\rangle^{\ell}; g_1 \oplus g_2 \oplus (\bigotimes_i [n_i]) \oplus [\ell]; h_1 \cup h_2}\ \textsc{c-Index}$$

$$\frac{e_1 \Downarrow \mathsf{map}; g_1; h_1 \quad e_2 \Downarrow \langle v, \langle v_1, \ldots, v_k\rangle^{\ell_1}\rangle^{\ell_2}; g_2; h_2 \quad v\ v_i \Downarrow v'_i; g'_i; h'_i \quad (n, \ell \text{ fresh})}{\begin{array}{c} e_1\ e_2 \Downarrow \langle v'_1, \ldots, v'_k\rangle^{\ell}; g_1 \oplus g_2 \oplus [n] \oplus (\bigotimes_i g'_i) \oplus [\ell]; \\ h_1 \cup h_2 \cup \{(n, \ell_1), (n, \ell_2)\} \cup (\bigcup_i h'_i \cup (\ell, \mathrm{loc}(v'_i))) \end{array}}\ \textsc{c-Map}_k$$

$$\frac{e_1 \Downarrow \mathsf{concat}; g_1; h_1 \quad e_2 \Downarrow \langle \langle v_1, \ldots, v_k\rangle^{\ell_1}, \langle v'_1, \ldots, v'_{k'}\rangle^{\ell'_1}\rangle^{\ell_2}; g_2; h_2 \quad n, n'_i, \ell \text{ fresh}}{\begin{array}{c} e_1\ e_2 \Downarrow \langle v_1, \ldots, v_k, v'_1, \ldots, v'_{k'}\rangle^{\ell}; g_1 \oplus g_2 \oplus (\bigotimes_i [n'_i]) \oplus [n] \oplus [\ell]; \\ h_1 \cup h_2 \cup \{(n, \ell_1), (n, \ell'_1), (n, \ell_2)\} \cup (\bigcup_i \{(\ell, \mathrm{loc}(v_i))\}) \cup (\bigcup_i \{(\ell, \mathrm{loc}(v'_i))\}) \end{array}}\ \textsc{c-Concat}$$

$$\frac{e_1 \Downarrow \mathsf{length}; g_1; h_1 \qquad e_2 \Downarrow \langle v_1, \ldots, v_k\rangle^{\ell}; g_2; h_2 \qquad n \text{ fresh}}{e_1\ e_2 \Downarrow k; g_1 \oplus g_2 \oplus [n]; h_1 \cup h_2 \cup \{(n, \ell)\}}\ \textsc{c-Length}$$

$$\frac{e_1 \Downarrow \mathsf{sub}; g_1; h_1 \qquad e_2 \Downarrow \langle \langle v_1, \ldots, v_{k_1}\rangle^{\ell_1}, k_2\rangle^{\ell_2}; g_2; h_2 \qquad 1 \le k_2 \le k_1 \qquad n \text{ fresh}}{e_1\ e_2 \Downarrow v_{k_2}; g_1 \oplus g_2 \oplus [n]; h_1 \cup h_2 \cup \{(n, \ell_1), (n, \ell_2)\}}\ \textsc{c-Sub}$$

Figure 5.4 Primitive Transitions for Vector Parallelism.

$$\frac{(\ell \text{ fresh})}{\underline{\mathsf{replicate}\ \langle k, v\rangle^{\ell_1}} \longrightarrow \langle \underbrace{v, \ldots, v}_{k \text{ times}}\rangle^{\ell}}\ \textsc{p-Replicate} \qquad \frac{(\ell \text{ fresh})}{\underline{\mathsf{index}\ k} \longrightarrow \langle 1, \ldots, k\rangle^{\ell}}\ \textsc{p-Index}$$

$$\frac{}{\underline{\mathsf{map}\ \langle v, \langle v_1, \ldots, v_k\rangle^{\ell_1}\rangle^{\ell_2}} \longrightarrow \mathsf{let}\ x_1 = v\ v_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ x_k = v\ v_k\ \mathsf{in}\ \underline{\{x_1, \ldots, x_k\}}}\ \textsc{p-Map}$$

$$\frac{(\ell \text{ fresh})}{\underline{\mathsf{concat}\ \langle\langle v_1, \ldots, v_k\rangle^{\ell_1}, \langle v'_1, \ldots, v'_{k'}\rangle^{\ell'_1}\rangle^{\ell_2}} \longrightarrow \langle v_1, \ldots, v_k, v'_1, \ldots, v'_{k'}\rangle^{\ell}}\ \textsc{p-Concat}$$

$$\frac{}{\underline{\mathsf{length}\ \langle v_1, \ldots, v_k\rangle^{\ell}} \longrightarrow k}\ \textsc{p-Length} \qquad \frac{1 \le k_2 \le k_1}{\underline{\mathsf{sub}\ \langle\langle v_1, \ldots, v_{k_1}\rangle^{\ell_1}, k_2\rangle^{\ell_2}} \longrightarrow v_{k_2}}\ \textsc{p-Sub}$$

5.3.1 Cost Semantics

As the constants added to the language are also values, they can be evaluated using the rule C-VAL. Figure 5.3 shows the inference rules defining the cost semantics for these constants. As constants will appear as the left-hand sides of applications, each rule starts with an expression of the form $e_1\ e_2$.

In this presentation, the inference rule C-MAP$_k$ stands for a family of rules, indexed by the natural number k. In each such rule, i varies between 1 and k. Thus when the length of the vector argument is 10, there are 12 antecedents to the rule: one for each of e_1 and e_2 plus 10 additional instances for each element of that vector.

In all cases except for map, the depth of the resulting computation graph is constant. In the case of map, the depth of the computation graph is determined by the maximum depth of any graph derived from applying the given function to a vector element: all such applications may be performed in parallel. This is shown through the use of k-ary parallel composition of graphs.

5.3.2 Implementation Semantics

To integrate vector operations into the implementation semantics given in Chapter 3, we must define the primitive transitions for these operations. Figure 5.4 shows these transitions. Like P-APPBETA, each of these axioms describes a transition starting from an underlined application form. In all cases but P-MAP, a new vector is created or a scalar is extracted from a vector, in either case in a single step.

In the case of P-MAP, a new let expression is used to describe possible parallel evaluation. A new function application is built for each element of the given vector and these applications are combined into a single declaration. The associativity of and remains unspecified as all the implementation semantics described in this thesis behave independently of whatever associativity

might be chosen. It is only essential that all of these sub-expressions are exposed within the same let expression and that and is non-commutative.

With these axioms, no further changes are required to support vector parallelism using the scheduling policies implemented in Chapter 3 (*e.g.*, depth-first, breadth-first scheduling). Again, factoring the implementation semantics into primitive transitions and scheduling transitions enables us to express how language constructs are computed (including how they express potential parallelism) from how computational resources are used to tap that potential.

5.4 Flattening

As noted above, the goal of flattening is to avoid some of the overhead of online scheduling by exposing more parallelism during the compilation process and taking advantage of vector operations available on the target architecture. In the latter case, it is not feasible to schedule such parallel operations at runtime efficiently.

Given vectors u and v, the following code (from the example above) computes the element-wise product of those vectors.

map **op**$*$ (zip (u, v))

The flattened version of this code should compute this product using a hardware vector operation (assuming vectors of appropriate length). For every primitive operation on scalars, for example **op**$*$, we require a corresponding vector operation **vop**$*$. Thus flattening should replace every occurrence of map **op**$*$ with (**vop**$*$ **o** unzip). [3]

$$\begin{aligned} &= \text{(}\textbf{vop}* \textbf{ o } \text{unzip) (zip (u, v))} \\ &= \text{(}\textbf{vop}* \text{ (u, v))} \end{aligned}$$

The addition of unzip is required by the fact that map **op**$*$ has type (int $\times$ int) list $\rightarrow$ int list, but **vop**$*$ has type int list $\times$ int list $\rightarrow$ int list. Further discussion of the representation of products is beyond the scope of this thesis. I refer the reader to the references cited below for details.

This transformation is sufficient to take advantage of parallelism at the leaves of the call tree in the matrix multiplication example, but not across function calls. In that example, there are two other uses of map where, in each case, the body of function that is mapped might also be evaluated simultaneously. For example, the above expression appears in a function that is passed to one such call to map. If we expand the definition of that function and use the flattened form of the dot product, we have:

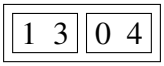
map (vv_mult' u) B = map (**fn** v $\Rightarrow$ (sum **o** **vop**$*$) (u, v)) B

Flattening is defined in such a way as to also eliminate this use of map. As written by the programmer, this anonymous function takes a vector and returns a scalar. The flattened version must take a vector of vectors and return a vector of scalars. Thus, in general, flattening will replace scalar operations with vector operations, and vector operations will be replaced with operations over vectors of vectors. Interestingly, only the original expression and this vectorized form will be necessary.

[3] I use the SML syntax **o** to denote infix function composition.

For a more complete description of flattening see Chapter 11 of Blelloch and Sabot [1990], Chakravarty and Keller [2000], Leshchinskiy et al. [2006], and especially Peyton Jones et al. [2008]. In the remainder of this section, I will give an overview of flattening and point out some of its salient features.

Segmented Vectors. A key aspect of flattening is separating the data of nested vectors from the structure of that nesting. A **segmented vector** is a nested vector that is represented as a pair of vectors. The first vector, or **data vector**, contains the elements of the nested vectors, while the second vector, or **segment descriptor**, records the length of each nested vector. The length of the data vector is equal to the number of values in the original nested vectors. The length of the segment descriptor is equal to the number of segments, while the sum of the elements of the segment descriptor is equal to the length of the data vector. As an example, consider a matrix represented by nested vectors from the beginning of this chapter.



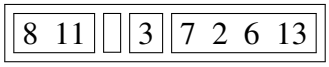
In segmented form, the matrix would be represented as the following pair of (flat) vectors.

Given this representation, it is easy to see how to implement operations such as element-wise product on two nested vectors: as long as the segment descriptors are equal, the result is simply the element-wise product of the data paired together with a segment descriptor.

```
fun nested_elwise_mult ((u_dat, u_sd), (v_dat, v_sd)) =
    if u_sd = v_sd then (vop* (u_dat, v_dat), u_sd)
    else (* raise error *)
```

The key insight is that the nesting structure does not change after an element-wise operation, thus we can strip off the segment descriptor, apply a primitive vector operation to the data, and then restore the segment descriptor.

Segment descriptors become more interesting in cases where the nesting structure is less regular. For example, the nested vector,



would be represented in segmented form as:

By separating information about nesting from the data itself, segmented vectors expose more opportunities for parallel evaluation, including the use of hardware-supported vector operations across the nested elements. In general, the segmented representation of vectors gives rise to a translation of the types of expressions, one case of which is shown below.

$$\mathcal{F}[(\tau \text{ vector}) \text{ vector}] \quad = \quad \mathcal{F}[\tau] \text{ vector} \times \text{int vector}$$

Vectorization. In the process of flattening programs, some functions originally written by the programmer in the context of a scalar input will be applied to a vector, for example, when such a function is an argument to map. In general, the compiler cannot determine whether a function will be invoked with a scalar or a vector argument, and in fact some functions may be invoked

both in scalar and vector contexts. To account for this, every function in the source code will be implemented as a pair of functions: one that is called with arguments as described by its original type and one that is called with vector arguments. The process of transforming each function into this pair of functions is called **vectorization**.[4] Like the segmented representation of vectors, vectorization also defines a translation of program types.

$$\mathcal{V}[\tau_1 \to \tau_2] \quad = \quad (\mathcal{V}[\tau_1] \to \mathcal{V}[\tau_2]) \times (\mathcal{F}[\tau_1] \to \mathcal{F}[\tau_2])$$

Continuing with our running example, we consider the vectorization of the expression vv_mult' u shown in its expanded form below.

$\mathcal{V}[$(**fn** v $\Rightarrow$ (sum **o** **vop**$*$) (u, v))$] = \ldots$

First, we must describe a pair of functions. The left-hand function is called when vv_mult' u is called in a scalar context.[5] I have already described this function in the discussion above. In addition to replacing map applied to a scalar operation with the corresponding vector operation, we must also project out the proper version of sum since (like all functions) it is implemented by a pair of functions.

= (**fn** v : float vector $\Rightarrow$ ((#1 sum) **o** **vop**$*$) (u, v), **fn** v : float vector vector $\Rightarrow \ldots$)

In the case of the matrix multiplication example, however, vv_mult' u is applied in a vector context: it is the function argument to map and so we must also consider the second component of the pair. We begin this case by noting that the argument v is a segmented representation of a nested vector. Thus, we might write the following type annotation and use a pattern to separately bind the data of the argument from the segment descriptor.

= ($\ldots$, **fn** (v_dat, v_sd) : float vector $\times$ int vector $\Rightarrow \ldots$)

One problem that is immediately apparent is that the free variable u no longer has the same type as v. To compute the element-wise product, we must first replicate u so that there are an adequate number of copies of it. This replication is part of the first step of the informal description of the flattened version of matrix multiplication from the beginning of this chapter.

= ($\ldots$, **fn** (v_dat, v_sd) : float vector $\times$ int vector $\Rightarrow \ldots$ (replicate (length v_sd, u), v_dat))

We can now apply the element-wise product and restore the segment descriptor.

= ($\ldots$, **fn** (v_dat, v_sd) $\Rightarrow \ldots$ (**vop**$*$ (replicate (length v_sd, u), v_dat), v_sd))

Finally we invoke sum to add all of the elements in each segment. Here, sum appears in a vector context, so we must call the right-hand side of the vectorized form of sum.

= ($\ldots$, **fn** (v_dat, v_sd) $\Rightarrow$ (#2 sum) (**vop**$*$ (replicate (length v_sd, u), v_dat), v_sd))

Once we have completely vectorized an expression, implementing applications of map become trivial, including applications to user-defined functions. As in the case of sum, we simply apply the right-hand side of the vectorized form of the argument.

map (vv_mult' u) B = map (**fn** v $\Rightarrow$ (sum **o** **vop**$*$) (u, v)) B
= #2 (**fn** v $\Rightarrow \ldots$, **fn** (v_dat, v_sd) $\Rightarrow \ldots$) B

[4] Blelloch [1990] uses the term "code replication" or "replication" while Chakravarty and Keller [2000] use the term "vectorization." I follow Chakravarty and Keller's terminology. Chakravarty and Keller use "replication" to refer only to behavior of the replicate function.

[5] I use "scalar context" to mean the application of a function at a type as it appeared in the original program text. Note that a function argument may actually have a vector type in either a scalar or vector context.

Figure 5.5 Syntax for Labeled Expressions.

(labels)	α	::=	$\epsilon \mid \alpha, i$
(natural numbers)	i	$\in$	$\mathbb{N}$
(expressions)	e	::=	$\ldots \mid \alpha{:}e$
(values)	v	::=	$\ldots \mid \alpha{:}v$

Branch Packing. As in previous chapters, the handling of conditional expressions presents an interesting case. When vectorizing a conditional expression, the compiler must consider the case when the program branches not on a single boolean, but a vector of booleans. In that case, some elements of the vector may be true and others may be false. This implies a form of control parallelism. While we would still like to exploit whatever data parallelism is available in each branch, we cannot simultaneously perform these operations on both branches. Therefore, we must separate the data used on one branch from that used on the other. The flattened code will collect together all the elements from the original array that satisfy the predicate and (separately) those that do not. Next, it will execute the first branch, exploiting whatever parallelism is available, followed by the second branch. From the point of view of a flattening compiler, *both* branches of the conditional are taken, but different elements are passed to each branch.

We add two additional primitives to the target language, pack and merge, with the following types.

$$\mathsf{pack} : \mathsf{bool\ vector} \times \tau\ \mathsf{vector} \rightarrow \tau\ \mathsf{vector}$$
$$\mathsf{merge} : \tau\ \mathsf{vector} \times \tau\ \mathsf{vector} \times \mathsf{bool\ vector} \rightarrow \tau\ \mathsf{vector}$$

The first, pack, takes two vectors of the same length and uses the first vector to select elements from the second. The result of the application pack (flags, xs) is to collect the elements of xs that appear at any index i such that the i^{th} element of flags is true. The length of the result will be equal to the number of elements in flags that are true.

The other primitive, merge, performs the inverse operation: it takes a two vectors to merge along with boolean vector that determines the interleaving of the result. The number of elements in the boolean vector should be equal to the sum of the number of elements in the other two vectors. The i^{th} element of the result will come from the first vector if the i^{th} element of the boolean vector is true and from the second vector otherwise.

5.5 Labeling

The flattening transformation is quite sophisticated. As with other examples of scheduling policies, it is my goal to describe the behavior of the flattening transformation abstractly. In the case of the online scheduling policies, I showed how an ordering on nodes in the computation graph is sufficient to describe those policies, and that implementation details including the queues and other structures used to manage ready tasks can be omitted. In the case of flattening, an ordering on nodes in the computation graph will not be sufficient: because flattening is a compiler transformation, it is more constrained than an online policy in how it can schedule tasks. For example, when a function is mapped across an array, we must ensure that any conditional sub-expressions

Figure 5.6 Labeling Expressions.

$$\frac{}{\alpha \vDash x \rightsquigarrow \alpha{:}x}\ \textsc{l-Var} \qquad \frac{}{\alpha \vDash c \rightsquigarrow \alpha{:}c}\ \textsc{l-Const}$$

$$\frac{\alpha, 1 \vDash e \rightsquigarrow e'}{\alpha \vDash \mathsf{fun}\ f(x) = e \rightsquigarrow \alpha{:}\mathsf{fun}\ f(x) = e'}\ \textsc{l-Fun} \qquad \frac{\alpha, 1 \vDash e_1 \rightsquigarrow e_1' \qquad \alpha, 2 \vDash e_2 \rightsquigarrow e_2'}{\alpha \vDash e_1\ e_2 \rightsquigarrow \alpha{:}(e_1'\ e_2')}\ \textsc{l-App}$$

$$\frac{\alpha, 1 \vDash e_1 \rightsquigarrow e_1' \qquad \alpha, 2 \vDash e_2 \rightsquigarrow e_2'}{\alpha \vDash \{e_1, e_2\} \rightsquigarrow \alpha{:}\{e_1', e_2'\}}\ \textsc{l-Fork} \qquad \frac{\alpha, 1 \vDash e_1 \rightsquigarrow e_1' \qquad \alpha, 2 \vDash e_2 \rightsquigarrow e_2'}{\alpha \vDash (e_1{,}e_2) \rightsquigarrow \alpha{:}(e_1'{,}e_2')}\ \textsc{l-Pair}$$

$$\frac{\alpha, 1 \vDash e \rightsquigarrow e'}{\alpha \vDash \#i\ e \rightsquigarrow \alpha{:}(\#i\ e')}\ \textsc{l-Proj}_i \qquad \frac{}{\alpha \vDash \mathsf{true} \rightsquigarrow \alpha{:}\mathsf{true}}\ \textsc{l-True}$$

$$\frac{}{\alpha \vDash \mathsf{false} \rightsquigarrow \alpha{:}\mathsf{false}}\ \textsc{l-False} \qquad \frac{\alpha, 1 \vDash e_1 \rightsquigarrow e_1' \qquad \alpha, 2 \vDash e_2 \rightsquigarrow e_2' \qquad \alpha, 3 \vDash e_3 \rightsquigarrow e_3'}{\alpha \vDash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rightsquigarrow \alpha{:}\mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'}\ \textsc{l-If}$$

of that function are properly sequentialized.

To capture these constraints, I define *labeled expressions* as shown in Figure 5.5. Labels α (as well as β and variants) will be defined as sequences of natural numbers. A labeled expression is written using a colon to separate the label from the expression. Each expression in the source program is given a label using the following judgment.

$$\alpha \vDash e \rightsquigarrow e'$$

This judgment is read, *assuming the current context is a label α then expression e is rewritten as labeled expression e'*. This judgment rewrites source terms into their labeled counterparts as shown in Figure 5.6. While these labels initially correspond to the tree address of each node in the abstract syntax tree, this property will not be maintained by evaluation. Note that each branch of a conditional expression is labeled starting with a different prefix.

With labeled expressions, we can now succinctly state the following constraint: data parallel implementations only evaluate expressions in parallel if those expressions have the same label. Since each label comes from a unique sub-expression of the original term, duplicate labels will only appear when they are introduced by the evaluation of terms such as map, as discussed below.

In general, any labeling that assigns a unique label to each sub-expression will be sufficient to enforce a SIMD-style parallel execution. However, I use sequences of natural numbers as labels to enable us to capture the effect of the flattening transformation. In particular, I will use an ordering based on the natural ordering of natural numbers to ensure that proper sequencing of conditional expressions.

Figure 5.7 Labeled Primitive Transitions. As for the breadth-first data parallel semantics, this semantics is a family of relations indexed by label. Most primitive transitions remain unchanged. The two rules shown in the figure are the exceptions: map replicates its label while application freshens the labels in the body of the function using label-indexed substitution.

$$\frac{}{\underline{\mathsf{map}\ \langle v, \langle v_1, \ldots, v_k\rangle^{\ell_1}\rangle^{\ell_2}} \longrightarrow_\alpha \mathsf{let}\ x_1 = \alpha{:}(v\ v_1)\ \mathsf{and}\ \ldots\ \mathsf{and}\ x_k = \alpha{:}(v\ v_k)\ \mathsf{in}\ \underline{\{x_1, \ldots, x_k\}}}\ \text{P}_\alpha\text{-MAP}$$

$$\frac{}{\underline{\langle f.x.e\rangle^\ell\ v_2} \longrightarrow_\alpha [\langle f.x.e\rangle^\ell/f, v_2/x]^\alpha e}\ \text{P}_\alpha\text{-APPBETA}$$

Labeled Substitution and Primitive Transitions. This parallel transition semantics makes use of an indexed primitive transition relation. In most cases, this index is ignored. The two exceptions are shown in Figure 5.7. First, when a function is mapped across an array, each application is given the same label. Second, when a function is applied in an ordinary context, the index of the relation is used when substituting the argument into the body of the function. The purpose of this substitution is to specialize the function body to the calling context. In particular, whenever a value is substituted into a labeled expression, that label of that expression is rewritten as shown by the following equation (where "," is overloaded here to mean the concatenation of sequences).

$$[v/x]^\alpha(\beta{:}e) = (\alpha, \beta){:}([v/x]^\alpha e)$$

That is, every label that appears inside a function body will be rewritten to include the label of the call site. The other interesting rule is that for substituting declarations $x = \alpha{:}v$. In this case, we are substituting a value representing an intermediate result. Here, we drop the label associated with the value.

$$[x = \alpha{:}v]e = [v/x]^\epsilon e$$

The intuition behind this case is that once we have a value, there is no need to constrain the transitions that apply to that value, since no further transitions will occur. The remainder of the label-indexed substitution rules follow from the standard definition of capture-avoiding substitution and are shown in Figure 5.8.

Breadth-First Labeled Semantics. We now define a transition semantics on labeled expressions that emulates the behavior of the flattened versions of these same expressions, as shown in Figure 5.9. This semantics is defined as a family of transition relations indexed by labels and based on the breadth-first semantics from Chapter 3. Each relation in this family will only apply primitive transitions to expressions with the corresponding label. The rule BF-WAIT is applied to all other sub-expressions, leaving them unchanged. The remaining rules either ignore labels or pass them up the derivation tree.

As noted above, I use an order on labels to further constrain the labeled semantics. This order $\lhd$ is a partial order derived from the standard ordering on natural numbers. Specifically, it

Figure 5.8 Labeled Substitution.

$$
\begin{array}{lcll}
[v/x]^\alpha y & = & [v/x]y & \\
[v/x]^\alpha c & = & c & \\
[v/x]^\alpha(\mathsf{fun}\ f(y) = e) & = & \mathsf{fun}\ f(y) = ([v/x]^\alpha e) & (\text{if } x \neq f, y) \\
[v/x]^\alpha(e_1\ e_2) & = & ([v/x]^\alpha e_1)\ ([v/x]^\alpha e_2) & \\
[v/x]^\alpha\{e_1, e_2\} & = & \{[v/x]^\alpha e_1, [v/x]^\alpha e_2\} & \\
[v/x]^\alpha(e_1,e_2) & = & ([v/x]^\alpha e_1,[v/x]^\alpha e_2) & \\
[v/x]^\alpha(\#i\ e) & = & \#i\ ([v/x]^\alpha e) & \\
[v/x]^\alpha\mathsf{true} & = & \mathsf{true} & \\
[v/x]^\alpha\mathsf{false} & = & \mathsf{false} & \\
[v/x]^\alpha(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) & = & \mathsf{if}\ ([v/x]^\alpha e_1)\ \mathsf{then}\ ([v/x]^\alpha e_2)\ \mathsf{else}\ ([v/x]^\alpha e_3) & \\
[v/x]^\alpha(\mathsf{let}\ d\ \mathsf{in}\ e) & = & \mathsf{let}\ [v/x]^\alpha d\ \mathsf{in}\ [v/x]^\alpha e & (\text{if } x \text{ is not bound in } d) \\
[v/x]^\alpha(\beta{:}e) & = & (\alpha, \beta){:}([v/x]^\alpha e) & \\
 & & & \\
[x = \alpha{:}v]e & = & [v/x]^\epsilon e & \\
[\delta_1\ \mathsf{and}\ \delta_2]e & = & [\delta_1][\delta_2]e &
\end{array}
$$

Figure 5.9 Breadth-First Labeled Transition Semantics. We describe the behavior of flattened nested parallelism using a family of transition relations, indexed by labels α. The expressions here are *not* flattened, but the behavior of flattened code is emulated by the constrained semantics: primitive transitions may only be applied if the label of the expression matches the index of the relation.

$\boxed{e \mapsto_\alpha e'}$

$$\frac{}{v \mapsto_\alpha v}\ \textsc{f-Val} \qquad \frac{e \longrightarrow_\alpha e'}{\alpha{:}e \mapsto_\alpha \alpha{:}e'}\ \textsc{f-Prim} \qquad \frac{\alpha \neq \beta}{\beta{:}e \mapsto_\alpha \beta{:}e}\ \textsc{f-Wait}$$

$$\frac{d \mapsto_\alpha d'}{\beta : (\mathsf{let}\ d\ \mathsf{in}\ e) \mapsto_\alpha \beta : (\mathsf{let}\ d'\ \mathsf{in}\ e)}\ \textsc{f-Let}$$

$\boxed{d \mapsto_\alpha d'}$

$$\frac{e \mapsto_\alpha e'}{x = e \mapsto_\alpha x = e'}\ \textsc{f-Leaf} \qquad \frac{d_1 \mapsto_\alpha d_1' \qquad d_2 \mapsto_\alpha d_2'}{d_1\ \mathsf{and}\ d_2 \mapsto_\alpha d_1'\ \mathsf{and}\ d_2'}\ \textsc{f-Branch}$$

is defined by the transitive closure of the following two rules.

$$\begin{array}{rcl} \alpha, i \lhd \alpha, j & \text{iff} & i < j \\ \alpha \lhd \beta, \beta' & \text{iff} & \alpha \lhd \beta \end{array}$$

The first rule states that two sequences that share a prefix are compared according to the first element at which they differ. The second rule states that one sequence comes before another if it comes before any prefix of that sequence. Note that a sequence α is incomparable with any sequence for α, β for which it is a prefix.

The complete evaluation of an expression is defined by a sequence of expressions $e_0, \ldots, e_k$ and an ordered sequence of labels $\alpha_0, \ldots, \alpha_{k-1}$, such that $\alpha_i \lhd \alpha_{i+1}$ and

$$e_i \mapsto_{\alpha_i} e_{i+1}$$

This ordering of labels ensures that the left branch of a conditional expression is always completely evaluated before the right branch is initiated. Without this ordering, this transition semantics would allow additional schedules that are not implemented by a flattening transformation. For example, without this constraint, an implementation might interleave evaluation of the two branches of a parallel conditional expression, rather than exhaustively executing the left branch first.

Relation To Flattened Code. Recall that, after flattening, two branches of a conditional (in a vector context) will be evaluated serially. (We will assume that there is at least one value to evaluate on each branch.) In the labeled semantics, whenever a function is bound to a variable that occurs more than once in the source code, the body of that function will be specialized to each context in which it is applied. In particular, if a function is applied in both branches of a conditional expression, each instance will be relabeled with a different set of labels, preventing them from executing in parallel. On the other hand, when a function body is duplicated as a result of map, every occurrence of the body will be specialized with the *same* label. As an example, consider the following expression.

```
map (fn x ⇒ if even x then x div 2 else (x + 1) div 2) (index 3)
```

A flattening compiler will lift the scalar integer operations (*e.g.,* even, div, $+$) to the analogous vector versions so that within each branch, these operations may be performed in parallel. The division operator is used in both branches, but these operations will *not* be performed in parallel.

An intermediate expression in the sequence of labeled transitions for this expression looks something like the following. (I have omitted some labels for the sake of clarity.)

```
let x1 = α:(α, β,3):((1 + 1) div 2)
  and x2 = α:(α, β,2):(2 div 2)
  and x3 = α:(α, β,3):((3 + 1) div 2)
in ...
```

First, note that each expression bound in the declaration is given the same outer label. As noted above, this is a result of the application of map. Up to this point in the evaluation of this expression, all of these expressions have transitioned in parallel. Second, note that the labels

that appear within each expression differ in the final component. Once the test of conditional expression is evaluated and a branch is taken, those sub-expressions corresponding to the "then" branch will end with the number 2, and those from the "else" branch will end with the number 3. (Here, β is the label of the **if** expression within the anonymous function.) The next step of this transition sequence must evaluate the "then" branches because $\alpha, \beta, 2 \lhd \alpha, \beta, 3$ and the labeled transition sequence must respect the order of labels $\lhd$.

5.6 Summary

In this chapter, I have described a method for compiling nested parallelism into the flat parallelism implemented by vector processors, graphics processors, and multimedia extensions of general-purpose processors. As this method determines the order of parallel tasks, it may be viewed as a scheduling policy. The behavior of this transformation seems to be captured by a breadth-first scheduling policy that is further constrained by a labeling of source terms. This characterization of flattening as a form of breadth-first scheduling agrees with previous experience with NESL [Blelloch et al., 1994]: examples such as matrix multiplication required so much memory in flattening implementations of NESL that they could not be evaluated for even moderately sized inputs, requiring the user to scale back parallelism explicitly. These memory use problems in NESL form an important part of the motivation for my work. A formal connection between this labeled scheduling policy and the flattening transformation is left to future work.

5.7 Related Work

Flat data parallelism is the focus of languages such as High Performance Fortran [High Performance Fortran Forum, 1993] and C* [Rose and Steele, 1987]. While these languages admit more straightforward implementations, it is unclear if they can support modular parallel programming without support for nested parallelism.

While most programming languages include support for some form of vector or array type, collection-oriented languages, including APL [Iverson, 1962], SQL [Chamberlin and Boyce, 1974], and Paralation Lisp [Sabot, 1988] among others, make collections a central part of the language. Flattening was introduced by Blelloch [1990] as a way of implementing the nested parallelism found in Paralation Lisp on a Connection Machine computer. It was later a critical part of the implementation of NESL [Blelloch et al., 1994]. Flattening was extended to languages with **while**-loops [Suciu and Tannen, 1994] and to Fortran [Au et al., 1997] as well as to languages with higher-order functions, recursive types, sum types, separate compilation [Keller and Chakravarty, 1998, Chakravarty and Keller, 2000] and arrays of functions [Leshchinskiy et al., 2006].

The labeled lambda-calculus [Lévy, 1978] has been used to study reduction strategies in several different contexts. Most closely related to my work is that of Abadi, Lampson, and Lévy [1996], which used labels to trace runtime dependencies back to source-level terms.

Chapter 6

Parallel Futures

In this chapter, I shall consider a more expressive form of parallelism called parallel futures [Halstead, 1985]. While nested parallelism enables programmers to express divide-and-conquer algorithms, it is not powerful enough to express the kind of parallelism found in programs that use parallel pipelining. Parallel futures allow programmers to create streams of values that are produced and consumed in parallel. Like nested parallelism, however, futures are deterministic: the answer given by a program is independent of how the scheduling policy maps parallel tasks to processors.

I will consider implementations of futures based on work stealing [Blumofe and Leiserson, 1999, Arora et al., 1998]. While work stealing has been studied in detail for the case of nested parallelism, less work has focused on understanding the performance with respect to parallel futures. Using cost graphs, I will prove tight bounds on several different types of overhead due to parallel scheduling.

In this thesis, I consider a form of futures that are evaluated *eagerly*. These futures do not support speculative evaluation,[1] but only provide a more flexible way of expressing parallelism. This means every parallel schedule will perform exactly the same amount of computation as the sequential schedule. I will briefly discuss the relationship to other forms of futures in Section 6.6 below.

Unlike the futures found in Multilisp, we use the static semantics of the language to distinguish between ordinary values (*e.g.* an integer, a list) and a future that will compute such values. Thus, as in the case of nested parallelism, programmers must specify opportunities for parallelism explicitly using the **future** keyword. In addition (and unlike in Multilisp), programmers must also explicitly demand the value of a future using **touch**.

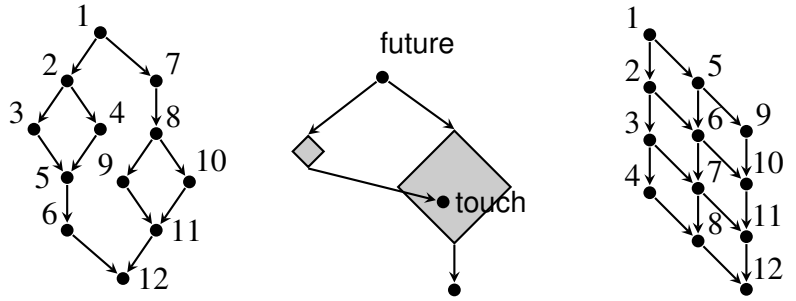
Parallel futures allow an expression to be computed in parallel with the context in which it appears. In contrast to nested parallelism, futures are asynchronous: a processor evaluating in parallel with a future will only block when the value of that future is required. The code below shows an example where an argument to a function is wrapped in a future.

$$(\textbf{fn}\ \mathsf{x} \Rightarrow e_1)\ (\textbf{future}\ e_2)$$

In this case, the body of the function e_1 may start evaluation immediately and continue in parallel

[1] Though, as discussed below, the form of futures considered in this chapter is called **fully speculative** by Greiner and Blelloch [1999].

Figure 6.1 Computation Graphs With Futures. These graphs demonstrate the differences between nested parallelism and parallel futures. Notably, parallelism is not well-nested in parts (b) and (c). Part (b) shows an example of a single future interleaved with nested parallelism and part (c) shows an example of a parallel pipeline. In parts (a) and (c), nodes are labeled according to the order determined by the sequential execution.



(a) nested parallelism (b) parallel future (c) pipelining

with e_2 until the argument is touched. In a second example below, an expression e may be computed in parallel with the uses of the anonymous function constructed in the body of the **let** expression.

let val y = **future** e **in** (**fn** g $\Rightarrow$ g (**touch** x)) **end**

This example shows how a future may "escape" from the scope at which it is defined.

Just as they originally appeared in Multilisp, we define parallel futures so that the body of the future begins evaluation before its enclosing context. Thus, in the absence of parallelism, the order of evaluation is identical to that of a program without **future** or **touch**.

Example: Producer-Consumer Pipelining. As an example of the use of parallel futures for pipelining, take the following program that produces a list of values (using the given function f) and consumes them (using g) in parallel.

```
datatype α flist = Nil | Cons of α * α flist future
fun produce (f : int → α option, idx : int) =
  case f idx
    of NONE ⇒ Nil
     | SOME x ⇒
          Cons (x, future produce (f, idx + 1))
fun consume (g : α * β → β, acc : β, xs : α flist) =
  case xs
    of Nil ⇒ acc
     | Cons (x, xs) ⇒ consume (g, g (x, acc), touch xs)
```

If we momentarily ignore those parts of the code relating to parallel execution, we have a pair of functions for generating and processing lists. (Some readers may recognize consume as foldl over lists.) Adding futures requires three changes. The type of lists is changed so that the tail of

Figure 6.2 Syntax Extensions for Futures. This figure describes the additional forms of expressions and values used as part of the language with parallel futures.

(expressions)	e	::=	$\ldots \mid$ **future** $e \mid$ **touch** e
(values)	v	::=	$\ldots \mid \langle v \rangle^{\ell}$

a list is now a future, indicating that the rest of the list might not yet be computed. The bodies of produce and consume must then be updated to indicate where parallel execution may begin and where it must end. Notice that occurrences of f in the body of produce are interleaved with uses of **future**. Similarly, occurrences of g are interleaved with uses of **touch** in the body of consume.

The effect of these changes is that the producer and the consumer can now run in parallel as long as values are produced before they are required by the consumer. If the consumer catches up with the producer then the processor running the consumer will look for other work to perform. This form of synchronization cannot be expressed with nested parallelism.

Finally, note that these changes will *not* affect the way that this code is executed by a single processor: with or without futures, a single processor will first run f repeatedly to generate the entire list and only then run g to consume it. Thus it may be helpful to think of a future as a heavy-weight function call whose result may be computed in parallel.

Relation to Nested Parallelism. Futures allow strictly more expressive forms of parallelism than parallel pairs. We can simulate the behavior of nested parallelism using futures as follows. Given a pair of expressions e_1 and e_2, the transformation below faithfully captures the parallelism described in previous chapters (where x does not appear free in e_2).

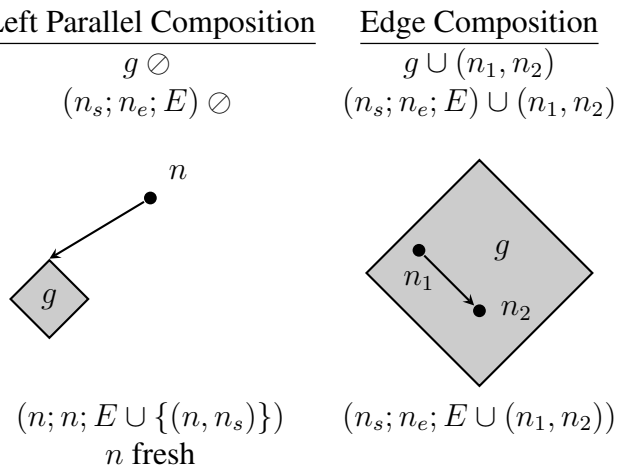
$\{e_1, e_2\}$ *... may be expressed as ...*

```
let val x = future e1
    val y = e2 in
  (touch x, y)
end
```

Conversely, however, there is no transformation for emulating arbitrary uses of futures using parallel pairs. The intuition behind this is twofold: first, futures syntactically separate the beginning of parallel execution from its end; second, futures may be embedded within data structures to create streams of values, as in the producer-consumer example above.

As an illustration of the differences between nested parallelism and futures, Figure 6.1 shows three examples of computation graphs. Part (a) shows a graph derived from nested-parallelism, while parts (b) and (c) show graphs derived from uses of futures. Note that in the latter two cases, parallelism is clearly not well-nested. Part (c) shows a computation graph derived from a parallel pipeline with three stages. In the next section, I will describe the extensions to the language, cost graphs, and semantics used to express and model the behavior of parallel futures.

Figure 6.3 Graph Operations for Futures. Futures also require additional ways of composing graphs, as shown in this figure.



6.1 Syntax and Semantics

The new syntax for futures is shown in Figure 6.2. Futures are defined by prefixing an expression with the **future** keyword. To obtain the value of a future, programmers use the **touch** operation. If that value has not yet been computed, then **touch** will suspend the current thread. I also add a form of value that is essentially a unary tuple. The use of these values will be explained in the discussion of the cost semantics below.

As in previous chapters, I will present the typing rules for the reader who is most comfortable thinking in terms of types. As we distinguish between futures and other values, we must extend the syntax of types to include a type τ **future** for any type τ.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{future}\ e : \tau\ \textbf{future}}\ \textsc{t-Future} \qquad \frac{\Gamma \vdash e : \tau\ \textbf{future}}{\Gamma \vdash \textbf{touch}\ e : \tau}\ \textsc{t-Touch}$$

From these rules, it should be clear that there is a simple erasure of programs with futures to sequential programs with the same extensional behavior.

Graphs. As futures are more expressive than nested parallelism, we must extend the sorts of graphs used to describe the costs of evaluating programs. Figure 6.3 shows two new operations on graphs added to those that appear in Figure 2.3.

The first operation, left parallel composition $\oslash$ is "half" of the parallel composition defined in Chapter 2 with the following exception: this is a unary operation that takes a graph and adds a single new node that acts as both the start and end node for the resulting graph. When further composed with other graphs, the sub-graph g will then appear on the left-hand side. For example, take the serial composition $(g \oslash) \oplus g'$ and assume that n is the new node added by the left parallel composition. Then n must be visited before any node in g', but any node in g may be visited simultaneously with any node in g'.

Figure 6.4 Cost Semantics for Futures. From these rules, it should be clear that futures have no effect on the extensional behavior of programs: creating a future adds a layer of indirection that is later removed by **touch**.

$$\frac{e \Downarrow v; g; h \qquad (\ell \text{ fresh})}{\textbf{future}\ e \Downarrow \langle v \rangle^{\ell}; (g \oplus [\ell]) \oslash ; h}\ \textsc{c-Future} \qquad \frac{e \Downarrow \langle v \rangle^{\ell}; g; h \qquad (n \text{ fresh})}{\textbf{touch}\ e \Downarrow v; g \oplus [n] \cup (\ell, n); h}\ \textsc{c-Touch}$$

The second operation, edge composition, shown on the left in the figure, adds one new edge (but no nodes) to a computation graph. As such, the resulting graph contains no additional computation, but this graph is more restrictive in terms of how nodes can be visited.

Cost Semantics. The rules defining the cost semantics for parallel futures are shown in Figure 6.4. If we first consider only the extensional behavior, we see that creating a future adds a layer of indirection that is subsequently removed by **touch**. This layer of indirection is reflected in many implementations: a common way of implementing futures is to create a memory cell that is passed both to the code computing the future and to the code using the future. This memory cell is then updated to hold the value of the future once it has been computed. In the cost semantics, however, there is no update to perform as the future is always computed "before" it is touched, just as in a sequential implementation of futures.

The potential parallel behavior of futures is captured in the computation graphs described in these two rules. The cost graph for **future** e is determined by the cost graph associated with e followed by single node ℓ that associates a location with the future itself. This graph is then left parallel composed so that it can be visited in parallel with the graph derived from the context in which the future is evaluated. The edge added during the left parallel composition (in the figure between n and g) is called a **future edge**.

The **touch** operation adds one new node to the computation graph. It also adds one additional edge to connect the final node of the graph derived from the future. This edge is called a **touch edge**. Thus the computation graph for **touch** depends not only on the computation graph from a sub-derivation, but on a location that appears in the *value* of a sub-derivation. This reflects the fact that the edge added at a **touch** reflects a form of data dependency, as opposed to a control dependency as in the case of all edges added to the computation graph in other rules. In the cases of both **future** and **touch** no additional heap edges are added: futures only affect the order in which nodes are visited.

Though edge composition allows arbitrary edges to be added to the computation graph, these edges will be added only in a restricted manner when building cost graphs for futures. In particular, the source ℓ of such an edge (ℓ, n) will always come from a graph that is left-composed.

The sequential order $<_*$ is defined on graphs derived from uses of futures as follows. Assume that $g = (n_s; n_e; E)$. First, the edge (n, n_s) added as part of left parallel composition $g \oslash$ defines an adjacent pair of nodes $n \prec_* n_s$. Let n' be the last node in g visited in the sequential schedule. Thus if we have $((n_s; n_e; E) \oslash) \oplus (n'_s; n'_e; E')$ the sequential order includes the adjacency $n' \prec_* n'_s$. Second, edges (ℓ, n) added through edge composition $\cup$ will *not* directly contribute to the

sequential order. This is safe since, as noted above, these edges are added only in cases where ℓ is part of a graph that is left composed with a graph that is visited before n.

In the remainder of this chapter, I make the simplifying assumption, as in previous work [Arora et al., 1998, Acar et al., 2000], that neither the in-degree nor out-degree of any node is greater than two. This can be achieved either by limiting each future to a single touch or by adding additional nodes, one for each touch, to the graph. (In the latter case this fits with our model of nodes as the instructions executed by the program, since presumably waking up each reader requires at least one instruction.)

6.2 Work Stealing

Work-stealing scheduling policies have appeared in previous chapters but only in so far as such policies specify the order in which parallel tasks are evaluated (or equivalently the order in which nodes are visited). While these orders are sufficient to understand how scheduling affects application memory use, in this chapter, I will consider other kinds of overhead, including the time required to schedule tasks and the effects of scheduling on cache misses incurred by the application. To understand how scheduling affects these forms of overhead, we must first consider how work stealing can be implemented. In this section, I will give a summary of the relevant work-stealing implementations and results that have appeared in previous work.

The principle of work stealing first appeared in early work on parallel implementations of functional programming languages [Burton and Sleep, 1981, Halstead, 1984]. This principle states that idle processors should take responsibility for finding useful work to do. It follows that if a processor *does* have work to do, then it should focus on that work, rather than trying to help distribute load among other processors. As discussed below, in the case of nested parallelism, processors rarely find themselves without work to do, so minimizing the overhead for busy processors leads to good performance overall.

Typically, work stealing is implemented by maintaining ready tasks using a set of double-ended queues (or **deques**). To allow processors to operate independently, each processor maintains its own deque. With respect to its queue, a processor is called a **worker**. When a worker's queue is empty, it becomes a **thief** and randomly selects another queue from which it will steal; the processor associated with this other queue is then called the **victim**.

6.2.1 Cilk

Cilk [Frigo et al., 1998] is an extension of the C programming language that supports nested parallelism. Cilk achieves an efficient implementation of work stealing by further exploiting this imbalance between idle and busy processors. I summarize two important aspects of this implementation: a strategy for specializing code for the common case and an asymmetric implementation of double-ended queues.

Compiling Clones. The Cilk compiler transforms each function into a pair of clones, known as the fast clone and the slow clone. The fast clone is used when a processor is working on a sequence of tasks independently of any other processors; this is the common case. The fast clone

contains only the minimum synchronization necessary to determine if work has been stolen. In the absence of a steal, tasks are executed sequentially by the fast clone, so no checks are needed to determine which sub-tasks have been executed.

The slow clone is used by the thief. Because a stolen thread is running in parallel with its parent thread, the slow clone must perform a check at each join point to determine if the parent thread has finished. While the slow clone incurs greater overhead than the fast clone, only the first task run by a thief must be run using the slow clone. Any sub-tasks spawned by a slow clone may use the fast clone. (Leveraging the asymmetry between the common sequential case and the less-frequent parallel case can be traced back to Mohr et al. [1990] as well as Goldstein et al. [1996].)

Double-Ended Queues. In Cilk, each deque supports only three operations: pushBottom, popBottom, and popTop. The first two are used only by the worker. From the worker's point of view, the deque is a FIFO stack. When a thief selects a queue as a target, it steals work from the top of the victim's queue using popTop. This is important for two reasons. First, because the worker and the thief use different operations, the synchronization overhead can be heavily skewed toward the thief. Cilk uses a shared-memory mutual-exclusion protocol designed to minimize overhead for the worker. Second, tasks at the top of the queue are older and usually correspond to larger amounts of work. This second fact is critical in bounding the expected number of steals.

6.2.2 Bounding Steals

Bounding the number of steals is an important measure of performance for nested-parallel programs since it enables us to bound the communication and computational overhead incurred by the scheduler. Previous work has shown that the work-stealing scheduler used by Cilk achieves good speed-up with a small number of steals.

Theoretical and empirical results [Blumofe and Leiserson, 1999, Blumofe et al., 1995] have shown that the expected number of steals is $O(PT_\infty)$ where P is the number of processors and T_∞ is the depth of the computation graph. To understand this bound, recall that thieves steal the oldest task on a victim's deque. Among all the tasks in the victim's deque, it is often the case that the oldest task will require the most work before joining with the victim's current thread. Put another way, the oldest task has the most *potential* in the sense that we expect it to keep the thief busy for the longest period of time. The number of steals can then be bounded by quantifying this notion of potential for a particular steal and for the program execution as a whole [Arora et al., 1998]. Since the potential never increases and each steal decreases it by a constant fraction with constant probability, there is a limit to how many steals we expect before all the potential is exhausted.

Locality. The number of steals is often used as a measure of communication costs. On a shared-memory multiprocessor, communication is implemented by the memory hierarchy. The bound on the number of steals allows us to bound the number of cache misses in a distributed cache relative to the number of misses in a sequential execution. The key insight is that additional

Figure 6.5 WS Scheduler. This abstract description of the WS scheduler for programs using parallel futures shows the order in which parallel tasks will be executed but does not include any optimizations.

```
/* Each processor i maintains a deque Q_i of tasks. */
Processor i repeatedly executes the following:
    If Q_i is empty, select another deque Q_j at random and steal the top task of Q_j (if any)
    Else
        Pop the bottom task t from Q_i and execute it
        If t forks a new task
            Push the continuation task onto the bottom of Q_i
            Push the spawned task onto the bottom of Q_i
        If t joins two tasks (as part of a fork-join)
            If both such tasks are finished, push the continuation task onto the bottom of Q_i
            Else, do nothing
        If t spawns a future
            Push the continuation task onto the bottom of Q_i
            Push the future task onto the bottom of Q_i
        Else if t touches a future but that future has not yet been computed
            Mark t as a task suspended on that future
        Else if t finishes the value of a future and some task is suspended on that future
            Push the suspended task onto the bottom of Q_i
            Push the continuation task (if any) onto the bottom of Q_i
        Else /* t is an ordinary task */
            Execute t and push the next task onto the bottom of Q_i
```

misses occur only after a steal. In the worst case, the entire cache must be flushed, thus if the number of cache misses in the sequential execution is C_* and the size of the cache is M then the expected number of misses using work stealing is $O(C_* + MPT_\infty)$ [Acar et al., 2000].

Efficiency. Bounding the number of steals is also important in ensuring the relatively low overhead of parallel scheduling. In Cilk, the scheduler is manifest predominately in the slow clone and in the parts of the implementation of the deques and the runtime system used by the slow clone. Since the slow clone is executed exactly once for each steal, we can safely push much of the scheduling overhead into these parts of the code without affecting overall performance. In fact, much of the Cilk design is motivated precisely by this observation.

6.2.3 Beyond Nested Parallelism

Arora et al. [1998] extend the work of Blumofe and Leiserson to bound the number of steals for more general forms of deterministic parallelism, including futures. Again, the expected number of steals is $O(PT_\infty)$. Though that work does not explicitly consider futures, we can consider how their algorithm would implement them. Figure 6.5 shows pseudo-code for a version of

this algorithm specialized to the case of futures. In the remainder of this chapter, I refer to this algorithm as WS.

As in other implementations of work stealing, there is one deque of tasks per processor. Each processor repeatedly pops and executes the bottom task in its deque. When a processor encounters a future, it pushes a task representing the continuation onto the bottom of its deque followed by the task implementing the body of the future. When a processor tries to touch a future but that future has not yet been computed, that processor will pop the next task from the bottom of its deque. When a processor finishes the value of a future and one or more tasks have been suspended waiting for that value, those tasks will be added to the bottom of the deque of the processor that finished computing the value of the future. If a processor's deque is empty, it selects another (non-empty) deque at random and steals the top task on that deque.

The implementation shown in the figure is an abstraction of a more realistic implementation and is intended only to show how the algorithm behaves with respect to the order in which tasks are evaluated. As this implementation uses the same three deque operations as a version for nested parallelism, it can be optimized using similar methods, including the fast/slow clone compilation technique described above. Despite this, and the bound on the number of steals, we must re-evaluate the effects of futures on performance. As I will show below, these suspending and resuming tasks may increase the number of cache misses and may require additional tasks to be run using slow clones *even when no additional steals occur*.

6.3 Deviations

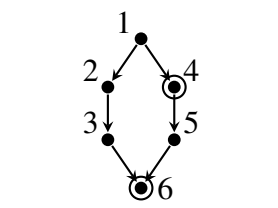
In Chapter 2, I defined a **deviation** as a point in a parallel execution where a processor diverges from the order in which tasks are evaluated by the sequential implementation. In this section, I will briefly review the kinds of deviations that can occur in nested parallelism before describing the additional kinds of deviations that occur in programs with parallel futures. I will then show how the time overhead of scheduling as well as the number of additional cache misses can be bounded in terms of the number of deviations.

Deviations in Nested Parallelism. Acar et al. [2000] show that for nested-parallel programs, each deviation incurred by WS is caused either directly or indirectly by a steal and that at most two deviations can be attributed to each steal.

Figure 6.6(a) shows a small parallel computation graph where nodes are labeled by the order in which they are evaluated in a sequential execution. Nodes at which deviations occur are circled. (Recall that graphs are drawn so that the sequential schedule corresponds to the left-to-right, depth-first traversal of the graph.) Part (b) shows one possible schedule using two processors. Maximal sequences of non-deviating execution are boxed, and deviations are labeled δ_1, δ_2, and so on. Note that while these deviations take up space in the illustrations, they do not correspond to a period of time during which a processor is idle.

In the schedule in Figure 6.6(b), processor p begins by executing tasks 1, 2 and 3. Processor q steals work from p and executes tasks 4 and 5. Since processor q does not execute task 3, this steal results in a deviation δ_1. A steal will always result in at least one deviation. A second deviation δ_2 will occur because q finishes task 5 before p finishes task 3. Though p itself does not steal

Figure 6.6 Example Deviations. (a) A computation graph and (b) a schedule for nested parallelism. (c) A computation graph and (d) a schedule for parallel futures. In part (c), the edge between nodes 3 and 7 corresponds to the touch of a future. Nodes in the graphs at which deviations occur are circled. Deviations occur in (b) either because of a steal (δ_1) or because of a join after a steal (δ_2). Deviations occur in (d) because of a steal (δ_3), when a task suspends while touching a future (δ_4), when executing a resumed task (δ_5), or indirectly after a touch or resume (δ_6, δ_7, δ_8).

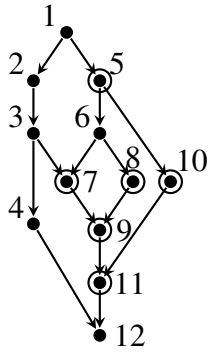


(a)

p | 1 2 3 | δ_2 | 6 |

q δ_1 | 4 5 |

(b)



(c)

p | 1 2 3 4 | δ_5 | 7 | δ_7 | 9 | δ_8 | 11 12 |

q δ_3 | 5 6 | δ_4 | 8 | δ_6 | 10 |

(d)

any work, we can attribute this deviation to the earlier steal. As noted above, for nested-parallel programs, WS will incur at most two deviations per steal.

6.3.1 Deviations Resulting From Futures

I now describe additional sources of deviations in programs that use parallel futures when using a work-stealing scheduler. Figure6.6(c) shows a computation graph that will be used as a running example in this section.

Suspend Deviations. A deviation can occur when a task attempts to touch a future that has not been computed yet. In Figure 6.6(d), task 7 represents the touch of the future that is computed by task 3. In the schedule shown in part (d), processor q executes task 6, but deviates at δ_4 since it cannot yet execute task 7. Note that this does not require an additional steal: in WS, task 8 sits at the bottom of q's deque.

Resume Deviations. A deviation also occurs when a suspended task is resumed. In the figure, task 7 is ready only after task 3 is completed. Thus after task 3 is executed, both tasks 7 and 4 will be pushed on to p's deque (in that order). When task 7 is popped and executed a deviation δ_5 will occur. In this example, there are no other tasks in p's deque when task 7 is executed, but it is possible that there could be more tasks in the deque above task 7. In that case, a second deviation would occur when the first of these tasks is popped and executed.

Indirect Deviations. The most subtle deviations occur through the use of multiple futures or a combination of nested parallelism and futures. Deviations can occur when a task is suspended waiting for another task which is itself suspended. In Figure 6.6, this occurs at δ_6 after task 8 is executed. This deviation cannot be attributed directly to a steal that has already occurred. Instead, it is caused indirectly by the suspension of task 7. As shown later in Section 6.4.2, a single steal followed by a single touch of a future can result in up to $T_\infty/2$ deviations (where T_∞ is the depth of the graph).

Indirect deviations can also occur on the processor that resumes a suspended touch. In this example, deviations occur at δ_7 and δ_8 on a processor that has neither stolen work nor touched a future. These deviations occur when a cascade of suspended tasks must be resumed once the value of a future is finished. In general, finishing the evaluation of the value of a future and resuming suspended tasks resulting from a single touch can also lead to up to $T_\infty/2$ deviations (see Section 6.4.2).

In the remainder of the text, I will refer to any deviation incurred by a processor after suspending on a touch as a suspend deviation (whether it is directly or indirectly caused by that suspend), and similarly for resume deviations. Just as at most one deviation occurs at a join for each steal in nested parallelism [Acar et al., 2000], deviations that occur as a result of futures can also be loosely paired.

Lemma 6.1. There is at least one steal or suspend deviation for every two resume deviations.

Proof. Assume that there is some node v that corresponds to a touch. We will show that if a resume deviation occurs when executing v there is at least one steal or suspend deviation and that the number of steal and suspend deviations related to v is no less than half the number of resume deviations related to v. Assume that processor p deviates when executing (and therefore, resuming) v. This implies that some processor q suspended rather than executing v and incurred either a steal or suspend deviation at that time. Processor p may also deviate after executing v and one or more of its descendants, so we have one or two resume deviations and one steal or suspend deviation. Now consider each descendant u of v at which p incurs an indirect resume deviation. This implies that some other processor (possibly q) already executed u's other parent but not u itself, at which time it incurred a suspend deviation. □

6.3.2 Performance Bounds

For nested-parallel computations, the number of steals can be used to bound the overhead of the scheduler, including scheduler time and cache misses.

Using deviations, we can extend these bounds to more general parallel computations. In the remainder of this section, I consider a scheduler called *generalized* WS that is a version of the scheduler shown in Figure 6.5 extended for unrestricted parallelism (and also defined in Arora et al. [1998]). The pseudo-code for the generalized version of the algorithm is shown in Figure 6.7. The performance bounds that follow are extensions of previous bounds on the overhead of work stealing. These results, together with results in the next section, show that for programs that use parallel futures, WS will incur overhead proportional to both the number

Figure 6.7 Generalized WS Scheduler. This abstract description of the generalization of WS scheduler to programs with unrestricted parallelism. This is algorithm that is presented by Arora et al. [1998].

```
/* Each processor i maintains a deque Q_i of tasks. */
Processor i repeatedly executes the following:
    If Q_i is empty, select another deque Q_j at random and steal the top task of Q_j (if any)
    Else
        Pop the bottom task t from Q_i and execute it
        For each task t' that t enables starting with the task corresponding to the right-most node,
            push t' onto Q_i /* This ensures that the left-most child will be on the bottom. */
```

of touches and the depth of the graph. Below, Δ indicates the number of deviations from the sequential execution.

Scheduler Time. Recall that Cilk compiles two versions of each function, the fast clone and the slow clone, and that the bulk of the synchronization overhead occurs during invocations of the slow clone. It can be shown for nested-parallel computations that the number of invocations of slow clones is bounded by the number of steals. For unrestricted parallel computations, the number of executions of slow clones can be bounded by the number of deviations.

Theorem 6.1. The number of invocations of slow clones using generalized WS is at most Δ the number of deviations.

Proof. A slow clone is invoked whenever a task is removed from a processor's deque *and* executed. Note that during the execution of a fast clone, tasks are removed from the deque, but since the identity of these tasks is known, they are discarded and the fast clone for the next task is run instead. Processor p removes some node v from a deque (not necessarily p's deque) only if the node u previously executed by p has no ready children. Let w be a child node of u. Since w is not ready, it must have in-degree two. Then u was either w's left parent or its right parent. If u is w's left parent then v was popped only to see if it had been stolen. If p successfully pops v, then v was not stolen and it will be executed next using the fast clone. (If it was stolen, the pop will fail because the deque is empty, and p will be forced to steal.) Alternatively, u is w's right parent. It follows that $u \prec_1 w$. In this case the slow clone for v will be run, but since $u \prec_p v$, we have $u \not\prec_p w$ and so a deviation occurred. □

Cache Misses. If we consider a shared memory architecture with a hardware-controlled cache, then we can bound the additional cache misses incurred during a parallel execution. We assume that each processor has a private cache with capacity C. Acar et al. [2000] bound the number of cache misses in nested-parallel computations in terms of the number of steals. To extend this to general parallelism, we must understand how data communicated between processors through uses of futures can affect cache behavior. In these results, I assume a fully associated cache with an LRU replacement policy.

Theorem 6.2 (extension of Theorem 3 in Acar et al. [2000])**.** The number of cache misses incurred using generalized WS is less than $M_*(C) + \Delta \cdot C$ where C is the size of the cache and $M_*(C)$ is the number of cache misses in the sequential execution and Δ is the number of deviations.

The original theorem is proved using the fact that the computation is race-free, *i.e.*, for two unrelated nodes in the computation graph, there is no memory location that is written by one node and either read or written by the other. All deterministic computations are race-free. For nested-parallel computations, race-free implies that any reads performed by the thief must have occurred before the steal. Therefore, after C misses, the thief's cache will be in the same state as that of the sole processor in the sequential schedule. From that point forward, such a processor will only incur misses that also occur during the sequential schedule.

In my proof, I consider an implementation that flushes the cache at each deviation. It is easy to show that such an implementation will incur no more than $\Delta \cdot C$ cache misses since after each flush it will take at most C memory accesses before the cache is in the same state as it would be in the sequential schedule. An alternative implementation that did not flush the cache after each deviation will incur no more misses.

Proof. Given a graph g and the schedule $\mathcal{S}$ determined by generalized WS, assume that the implementation flushes the cache after each deviation. Consider a node v that reads a memory location m and a second node u that writes to that memory location. Assuming the program is deterministic, then there is exactly one such u whose value should be read by v and there is at least one path from u to v in g.

If no deviation occurs between u and v then a cache miss occurs at v in $\mathcal{S}$ if and only if a cache miss occurs at v in the sequential schedule: either m must have remained in the cache since u was visited, or C other locations were accessed in the interval between u and v.

If a deviation does occur between u and v, then let w be the most recent deviation incurred by the processor q that executes v. Assume that no miss occurs at v in the sequential schedule, and further assume, for a contradiction, that q has already incurred C cache misses since w. This implies that there have been C distinct memory accesses since w. However, q visits all nodes between w and v in the same order as the sequential schedule, so the state of q's cache at v must be identical to that of the sequential schedule. Therefore, if no miss occurs in the sequential schedule at v then no miss occurs when v is visited by q, a contradiction. Therefore, q incurs at most C more misses than the sequential schedule. □

Lower Bounds. Deviations can also be used to establish lower bounds for some of these overheads. For example, a slow clone is executed after every steal and every suspend deviation (direct or indirect). By Lemma 6.1 this is a constant fraction of all deviations and thus the expected number of slow clone executions is $\Omega(\Delta)$ for arbitrary programs including those constructed only using parallel futures.

Acar et al. [2000] showed a family of graphs, which (though they do not claim it) can be constructed using futures, where $\Omega(RC)$ cache misses occur with two processors (where R is the number of touches), while only $O(C)$ cache misses occur when using a single processor. Moreover, each touch incurs a deviation so the number of misses with two processors can be expressed as $M_*(C) + \Omega(\Delta \cdot C)$.

6.4 Bounds on Deviations

In this section, we give the main results of this chapter: upper and lower bounds on the number of deviations that occur when using WS to schedule parallel programs that use futures. Given the results of the previous section, these bounds imply that using work stealing to schedule tasks in programs that use futures may incur high overheads.

6.4.1 Upper Bound

This bound is given in terms of the number of touches that appear in the computation graph. Note that this is not the number of touches that appear in the program source code, but the number of touches that occur dynamically in the program execution. The theorem is stated and proven below, preceded by several lemmata used in the proof.

Lemma 6.2. Given a computation graph g derived from an execution of a program using parallel futures and a node v with out-degree two and right child u, if y is a descendant of v and $y <_* u$, but y has a parent that is not a descendant of v, then y is a touch.

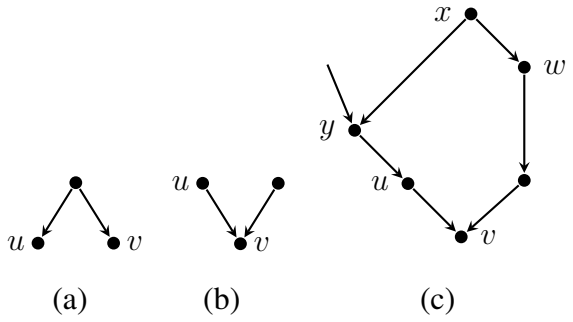
Proof. If y is a descendant of v but it also has a parent that is not a descendant of v, then y must have in-degree two. Therefore it is either a join or a touch. Suppose the other parent of y is x. As we have $y <_* u$ and $x <_* y$, it follows that $x <_* u$. Furthermore, x is not a descendant of v and so $x <_* v$. If x was executed before v in the sequential execution then x and v share a common predecessor w (possibly x). Since the two parallel paths between w and y are not well-nested with respect to v's two children, y must be a touch. □

Lemma 6.3. Given a node of out-degree two with children u and v, let V be the set of nodes executed in the sequential schedule between u and v (inclusive). If for some processor p in a parallel execution and for each touch $y \in V$, $x \prec_* y \Rightarrow x \prec_p y$, then for all $y, z \in V$, $y <_* z \Rightarrow y <_p z$.

Proof. Assume otherwise, for the sake of contradiction. Take the first pair of nodes $y, z \in V$ (ordered lexicographically according to the sequential order of execution) such that $y <_* z$, but $y \not<_p z$. There are three possibilities: either y or z is not executed by p (and thus p does not relate y and z) or $z <_p y$. If y is not executed by p (and similarly for z) then it must have a parent that is not a descendant of u. By Lemma 6.2, y is a touch, a contradiction, since we assumed that p executed all touches in V. Alternatively, we have $z <_p y$. If this is the case, then up to the point that p executes z, p has executed every node x such that $x <_* y$ (since otherwise y and z would not be the first offending pair). Let x be the last such node: $x \prec_* y$. If p does not execute y immediately after x, then y must have another parent that is not in V. This again (by Lemma 6.2) implies that y is a touch, a contradiction, since we assumed that $x \prec_* y$ implies $x \prec_p y$ for all touches in y in V. □

Lemma 6.4. When executing a graph of depth T_∞, the number of elements in any processor's deque is at most T_∞.

Figure 6.8 Example Graphs Used in Proof of Upper Bound. These graphs illustrate situations that arise in the proof of Theorem 6.3.



Proof. The number of elements on a processor's queue only increases when it executes a node with out-degree two. Thus if there are T_∞ elements in a processor's queue, then there must exist a path of length T_∞ composed of nodes of out-degree two. If such a path exists, then the depth of the graph is at least T_∞. □

Lemma 6.5. For a graph g with start node u and end node v, for any other node w in g, if there is a path from w to v that does not include any touch edges, then there is a path from u to w that does not include any future edges.

Proof. By induction on the length of the path from w to v. First take the case where there is an edge connecting w and v (a path of length one) that is not a touch edge. Edges in the computation graph are either (i) future or touch edges, (ii) are added to between two sub-graphs that must be computed serially, or (iii) are added to compose two sub-graphs as a nested-parallel computation. The edge from w to v cannot be a future edge (i) as a future edge cannot lead to the end node of a graph. If that edge was added as part of the serial composition of two sub-graphs (ii), then every path from u to v that does not include a future or touch edges must include w (and there is always at least one path from u to v that does not include either future or touch edges). If an edge was added as part of a nested-parallel composition (iii) (*i.e.* leading to a join), then there must be a corresponding fork with an edge that leads to w as well as a path without future edges that leads from u to that fork.

Now take the case where the length of the path from w to v is greater than one. Assume that there is a node y with an edge (w, y), a path from y to v, and that none of those edges is a touch edge. By induction, there is a path from u to y that does not include any future edges. As above, consider how the edge (w, y) was added to the graph. This edge cannot be a future edge (i) since if it was then every path from y to v must include a touch edge and we assumed at least one path that does not. If that edge was added during a serial composition (ii) then again every path from u to v that does not include a future or touch edges must include w. (iii) If the edge (w, y) is a join edge, then there must be a corresponding fork and that fork must lie on the path between u and y; it follows that there is path without future edges from that fork to w. Finally, if the edge (w, y) is a fork edge, then every path without future edges from u to y also goes through w. □

Theorem 6.3 (Upper Bound)**.** For computations with depth T_∞ and R touches, the expected number of deviations by WS on P processors is $O(PT_\infty + RT_\infty)$.

Proof. We consider each node in the graph and determine whether or not a deviation could occur at that node. We divide nodes into three classes based on the in-degree of each node and (possibly) the out-degree of its parent. Only the root node has in-degree zero, but a deviation never occurs at the root. If a node v has in-degree one and either (i) its parent has out-degree one, or (ii) its parent has out-degree two and v is the left child of its parent, then v will always be executed immediately after its parent and no deviation will occur.

Consider the case where v has in-degree one, its parent has out-degree two, and v is the right child of its parent as shown in Figure 6.8(a). In this case, there is some descendant x of u such that $x \prec_* v$. Assume that v is executed by processor p. A deviation occurs when executing v only if at least one of the following three conditions holds: (i) v is stolen, (ii) there is a touch at some node w that is executed between u and v in the sequential schedule along with a node y such that $y \prec_* w$ and p executes y but $y \not\prec_p w$, or (iii) there is a node x such that $x \prec_* v$ and x is executed by p, but p also executes at least one other node between x and v.

First, we show that these conditions are exhaustive. Assume that a deviation occurs when executing v but suppose, for the sake of contradiction, that none of these conditions is true. If v was not stolen then u was also executed by p and v was pushed onto p's deque. By assumption, for every touch w that appears in the sequential execution between u and v with $y \prec_* w$, we have $y \prec_p w$. It follows from Lemma 6.3 that for every pair of nodes y and z that appears in the sequential execution between u and v (inclusive), we have $y <_* z \Rightarrow y <_p z$. This implies $x <_p v$ for the node x such that $x \prec_* v$. Finally, by assumption, there is no other node that p executes between x and v and so $x \prec_p v$. Therefore, no deviation occurs at v, a contradiction.

Now we count the number of nodes that can be described by one of the three conditions above. First, from the proof of Theorem 9 in Arora et al. [1998], the expected number of steals is $O(PT_\infty)$, bounding the number of nodes that fall into the first category. Second, if there is a touch w such that $y \not\prec_p w$ and v was not stolen, then v was on p's deque when p executed y. Moreover, since $y \prec_* w$, there is exactly one such y for each touch. By Lemma 6.4, there are at most T_∞ elements on p's queue at any time, and thus $O(RT_\infty)$ nodes in the second category. Finally, there is exactly one node v for each node x such that $x \prec_* v$. The number of nodes that finish the value of a future is bounded by the number of touches. If p executes another node between x and v, then x is a future and the number of nodes such as x is $O(R)$. It follows that the number of nodes that immediately follow x is also $O(R)$. Thus the expected number of deviations occurring at nodes of in-degree one is $O(PT_\infty + RT_\infty)$.

Now take the case where v has in-degree two. Deviations only occur here if v is enabled by its left parent (*i.e.*, if its left parent is visited after its right parent), so consider that case (Figure 6.8(b)). Define $\hat{g}_x$ to be the sub-graph of g rooted at some node x formed by removing any future or touch edges as well as any nodes that are not reachable from x through the remaining edges. In this case, a deviation occurs when executing v only if one of the two following conditions holds: (i) there is a touch of a node y and another node x that is a predecessor of both v and y such that v lies on all paths from y to the sink of $\hat{g}_x$, or (ii) there exists a node w such that $u \prec_* w$ and w was stolen.

Assume that a deviation occurs when executing v but suppose again, for the sake of contra-

diction, that neither of the above conditions is true. First, v cannot be a touch. (If it were, then it would lie on all paths between itself and the sink of $\hat{g}_v$.) Therefore, v must be a join. Let x be the corresponding fork and w be the right child of x. It follows that $u \prec_* w$. By assumption, w was not stolen and was therefore executed by the same processor p that executed x. Since no deviation occurred, w is also executed before u. However, if p executes w before u, then there must be some node y that is a predecessor of u and a descendant of x but is not dominated by x. The graph might look something like Figure 6.8(c). If y is not dominated by x then it must be a touch. If there is more than one such touch, let y denote the one that is closest to u. Then any edges between y and u are not touch edges. By Lemma 6.5 there is a path from x to y that does not include any future edges. Thus y is in $\hat{g}_x$. Since $\hat{g}_x$ is series-parallel and y appears after x but before v, any path in $\hat{g}_x$ from y to the sink must go though v, a contradiction. Thus no deviation can occur unless one of the above conditions holds.

We now count the number of nodes that can fall into one of the two categories above: in the first case, the number of nodes such as v is bounded by the number of touches times the depth of $\hat{g}_x$ (which is in turn bounded by the depth of g); in the second case, there is exactly one node v for each steal of w such that v follows some node u with $u \prec_* w$. Thus the number of deviations occurring at nodes of in-degree two is $O(PT_\infty + RT_\infty)$, and the total number of expected deviations, including nodes with both in-degree one and two, is also $O(PT_\infty + RT_\infty)$. □

6.4.2 Lower Bound

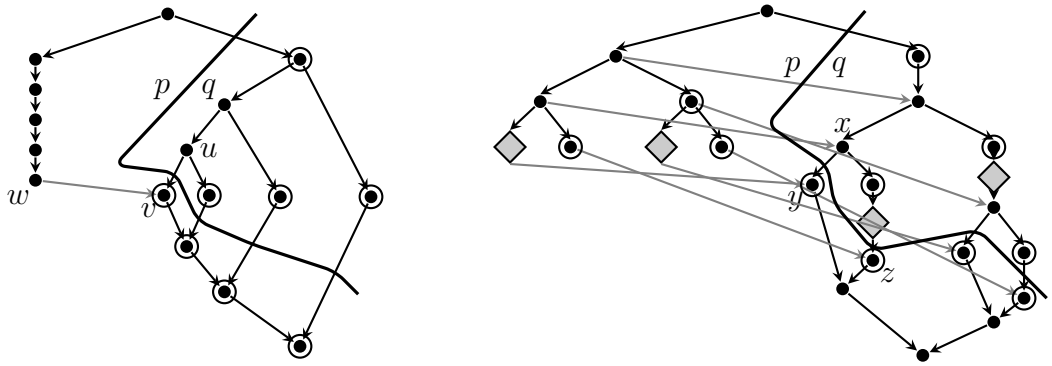
While this upper bound is high, we show that there are programs that use futures for which this upper bound is tight. We assume that the scheduler is parsimonious (*i.e.*, it maintains one task queue per processor and each processor only steals when its local queue is empty) and also greedy. This class of schedulers includes WS. In the computations we construct, each time we increase the depth by one, we can either double the number of touches (and the number of deviations incurred as a result of those touches) or we can cause each existing touch to incur an additional deviation.

Theorem 6.4 (Lower Bound)**.** There exists a family of parallel computations derived from programs that use parallel futures with depth T_∞ and R touches where $\Omega(PT_\infty + RT_\infty)$ deviations occur when executed on P processors with any parsimonious work-stealing scheduler and when assuming that $T_\infty > 8 \log R$ and $T_\infty > 2 \log P$.

Proof. The proof is given by constructing such a family of computations. Graphs that incur $\Omega(PT_\infty)$ deviations can be constructed using only nested parallelism, for example, by chaining together sequences of fork-join pairs. In such examples, T_∞ must be greater than $2 \log P$ to ensure there is adequate work for P processors. Below, we construct graphs with $\Omega(RT_\infty)$ deviations for two processors and assume that they can be combined (in parallel) with such chains yielding a family of graphs with the required number of deviations.

For ease of presentation, we will carry out this construction in stages. In the first step, we show that one steal and one touch is sufficient to generate $\Omega(T_\infty)$ deviations on a graph of depth T_∞ with two processors. Figure 6.9(a) shows the one such graph. The program starts by using a

Figure 6.9 Families of Computations Demonstrating Lower Bounds. In part (a), there is one future, one steal, and one touch leading to $\Omega(T_\infty)$ deviations on a graph of depth T_∞ with 2 processors. In part (b), one steal and R touches leads to $\Omega(R)$ deviations with 2 processors. A heavy line divides those nodes executed by processor p from those executed by q. Nodes at which deviations occur are circled. Each edge that leads to a touch is drawn in gray to improve readability.



(a) $\Omega(T_\infty)$ deviations

(b) $\Omega(R)$ deviations

future to split the computation into two parts. The value of the future is finished at node w and the edge from w to v is a touch edge.

Assume that processor p starts execution of the graph and the processor q steals work corresponding to the right-hand sub-graph. We assume that steals require unit time, but our examples can be extended to account for larger costs. This steal means that q immediately incurs a deviation. The left-hand sub-graph is a sequence of nodes that ensures that q finishes node u before p finishes w.

After q finishes u, it cannot continue sequentially to v. However, there are several nodes in q's deque. Since we assume the work-stealing scheduler is parsimonious, q will continue by removing elements from its queue, incurring a deviation each time. These deviations occur as indirect results of a suspension.

Meanwhile, p will finish the value of the future and continue by executing the v (again, since this node was on its own deque and it has no other nodes to execute). Then p will execute all of the nodes along the lower-left side of the right-hand sub-graph, incurring a deviation at each node. These are examples of indirect deviations that occur after resuming a suspended touch.

To generalize this graph to an arbitrary depth, we add an additional fork on the right-hand sub-graph and one more node on the left, ensuring there is sufficient work to delay p. This will increase both the depth and the number of deviations by two but will not change the number of steals or touches. (In fact, similar graphs can be constructed that will incur $\Omega(T_\infty)$ deviations for any greedy scheduler.)

In the second step, we show that one steal and R touches is sufficient to generate $\Omega(R)$ deviations on a graph of depth $\Omega(\log R)$ as in Figure 6.9(b). Graphs of this form are generated from a pair of functions which produce and consume a binary tree in parallel. Each node in this tree, both internal and external, is defined by a future.

These graphs are constructed so that p and q alternate between suspending on the touch of a future and resuming it. The gray sub-graphs (drawn as diamonds in the figure) are small sequences of nodes introduced to ensure this alternation. In the figure, each such sub-graph requires two or three nodes. Processor q incurs a deviation after executing node x. There are several nodes in q's deque, but the graph is constructed so the next touch will also result in suspend, no matter which node is chosen. This graph may be generalized to an arbitrary number of touches so long as the depth of the graph is at least $4 \log R$.

Finally, we define a family of computations with depth T_∞ that incur $\Omega(RT_\infty)$ deviations on two processors. We do so by first taking a graph as described in the second step (Figure 6.9(b)) of depth $T_\infty/2$. We then replace each touch in that graph (*e.g.*, nodes y and z in the figure) with an instance of the right-hand portion of the graph described in the first step (Figure 6.9(a)) of depth $T_\infty/2$. In addition, each diamond sub-graph on the producer-side of the graph must be extended with an additional delay of $T_\infty/2$ nodes. This results in a graph of depth T_∞ with R touches each of which will result in $\Omega(RT_\infty)$ deviations. Note that this construction requires that $T_\infty > 8 \log R$ so that there is adequate parallelism to produce and consume the tree in parallel. □

6.5 Alternative Implementations

Given that the performance effects of deviations are just as significant as those of steals, it seems worthwhile to consider implementations of work-stealing that avoid not just steals but deviations as well. The examples given above provide some clues as to how a scheduler might avoid deviating from the sequential order of execution.

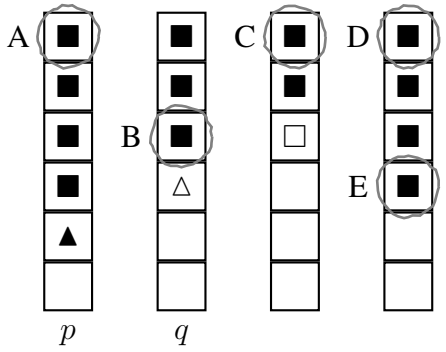
- In the schedule given in Figure 6.6(d), a series of deviations occurs after touching a single future. One might consider the problem to be that the suspending processor q remains "too close" to the suspended task and instead should look for work elsewhere. q (or another processor) would return to these tasks later, after the value of the future is likely to be computed.
- In this same schedule, p deviates from the sequential order when it executes task 7. This deviation was caused by the fact that p added task 7 to its own deque and thereby decreased the locality of tasks within its own deque.

Taking these examples as inspiration, we propose three design choices in implementing a work-stealing scheduler with support for futures. At the same time, we consider a snapshot of a work-stealing scheduler in Figure 6.10 to help illustrate these choices.

Suspend Entire Deques. When processor's task is suspended, that processor may have other ready tasks sitting in its deque. This is illustrated by the deque associated with q in Figure 6.10. This deque contains several ready tasks (■). At the bottom of the deque, the current task (△) is not filled to indicate that it has been suspended.

If a scheduler *suspends entire deques* then all of those tasks are also suspended. That is, the entire deque is set aside, and the processor must steal from another deque. This causes at least one deviation but hopefully avoids the cascading deviations as in the example above. This strategy creates the possibility that there are more deques than processors as shown in the figure.

Figure 6.10 Work-Stealing Deques. These double-ended deques represent one possible state of an alternative work-stealing scheduler. Each of the two deques on the left is owned by a processor. Older tasks are near the top of each deque. Ready tasks are solid, and suspended tasks are outlined. When q suspends waiting for the value of a future, there are several possibilities for which task it should work on next; the five possibilities that are circled are considered in the text.



Alternatively, when task is suspended, a processor may take a task from the bottom of its own deque as in WS. In this case, q would continue by executing task B. This is a deviation but not a steal.

Steal Entire Deques. If an implementation suspends entire deques then those deques not owned by any processor may be in one of two states, as shown by the two deques on the right in Figure 6.10. The deque at the second from the right has been suspended, but the bottommost task in that deque (□) has not yet been resumed. In contrast, the bottommost task of the deque on the far right has since been resumed.

If a scheduler *steals entire deques* then when stealing from deques such as the rightmost deque in Figure 6.10, then the thief claims the entire deque and begins work on the bottom most task (labeled E in the figure).

Alternatively, a scheduler might treat this deque like any other deque and only a single task from the top of the deque (task D in the figure).

Note that if a scheduler suspends entire deques, it must still consider suspended (but non-empty) deques as targets for theft. If not, it risks letting idle processors remain idle even if there are ready tasks to be executed. Thus even though the second deque from the right has a suspended task as its bottommost element, the topmost task C must still be considered (like task A) as a target for theft.

Resume Tasks in New Deques. If an implementation does *not* suspend entire deques, then there is no convenient way to reassociate a resumed task with its original deque (since that processor may have since moved on to other work).

If a scheduler *resumes tasks in new deques* then it creates a new deque that is not associated with any processor and adds the resumed task to that deque.

Alternatively, the resumed task may be placed at the bottom of resuming processor's own deque, as in the implementation of WS.

Table 6.1 Work-Stealing Implementations. Each of the four implementations supporting futures is described by one column in this table. The first column corresponds to the implementation described by Figure 6.5 and Arora et al. [1998].

	WS	WS2	WS3	WS4
suspend entire deques	no	no	yes	yes
steal entire deques	-	-	no	yes
resume tasks in new deques	no	yes	-	-

Four possible policies based on these design choices are shown in the columns of Table 6.1. The first policy WS the uses described above and by Arora et al. [1998]. The other three columns represent alternatives based on the discussion in this section. These policies will be compared empirically in Chapter 8.

6.6 Speculation

This chapter focuses on eager, or fully speculative, futures. As with the other forms of parallelism described in this thesis, eager futures simply enable the programmer to express opportunities for parallelism: the extensional behavior of programs remains the same. In addition, the total amount of computation performed by a program with futures is the same as that performed by the version of the program with all uses of futures erased.

An alternative formation of futures uses a lazy (or, in the terminology of Greiner and Blelloch [1999], partially speculative) semantics. Though the syntax remains the same, lazy futures combine parallelism with a way of delaying the computation of some sub-expressions. In a sequential, call-by-value language such as ML, laziness enables programmers to delay the evaluation of an expression until necessary or, in cases where that expression is unnecessary, to forgo evaluation of the expression entirely. Lazy futures combine both opportunities for parallelism and opportunities to avoid some unnecessary computation. Put another way, lazy futures use parallel hardware to evaluate parts of the program that *may* be required later. There is, however, a risk that it will later be discovered that this work was wasted. Thus, the total amount of computation depends on which futures are selected for evaluation.

This form of parallel can be expressed equivalently using a special form of binding such as the **pval** binding in Manticore [Fluet et al., 2008b]. Regardless of the syntax, the implementation of lazy futures must support some form of thread cancellation, at the very least because this will help to limit the amount of unnecessary work. Moreover, there is no guarantee in the case where the value of a future is not needed that the expression computing that value will ever terminate.

6.6.1 Glasgow Parallel Haskell

As a call-by-need language, it is natural to integrate parallelism into Haskell as a form of speculation. As an example, a Haskell implementation of the producer/consumer example from the beginning of this chapter is shown below. Here, I use a parallel implementation of Haskell called Glasgow Parallel Haskell (GpH) [Trinder et al., 1998].

```
produce f idx = case f idx of
                    [] → []
                    (x:xs) → x : (produce f (idx + 1))
consume g acc [] = acc
consume g acc (x:xs) = consume g (g (x, acc)) xs

let xs = produce 0
in force xs `par` consume xs
    -- Force each element of a list to weak head normal form
    where force [] = ()
          force (x:xs) = x `pseq` (force xs)
```

As Haskell's types implicitly include a notion of laziness, there is no need to define a new data type: in general, the tail of a list is not computed until its value is required. The definitions of produce and consume are almost identical to the versions in sequential ML (*i.e.*, without futures). Unlike the form of parallelism used above, GpH expresses opportunities for parallelism separately from the code that describes the extensional behavior of an algorithm. Thus rather than annotating some expressions with the **future** keyword, consume is run in parallel with force, a function that expresses a **strategy** for parallel evaluation. Note that force has no (extensional) effect on the result of this program. Instead it describes a way of visiting parts of a data structure, in this case a list, so that parts of that data structure will already be computed by the time that they are required by consume. Strategies are expressed using the following pair of operators.

par	:	$\alpha \to \beta \to \beta$	par e_1 e_2	=	e_2
pseq	:	$\alpha \to \beta \to \beta$	pseq e_1 e_2	=	$\bot$, if $e_1 = \bot$
				=	e_2 otherwise

The first operator par states that two expressions can be run in parallel. Only the result of the second argument is returned: the result of the first argument is discarded. In the example above, the strategy affects the performance of consume because the variable xs appears in both arguments to par. This corresponds to some shared state between the two parallel branches.

The second operator pseq is similar to the built-in Haskell operator seq, in that it is strict in both arguments. As discussed by Marlow et al. [2009], pseq also makes the operational guarantee that the first argument will be evaluated first.

Implementation. Recent versions of GpH use a work-stealing scheduler. When a par is encountered, the first argument is pushed on a deque so that it may be stolen by another processor. If argument is not stolen, then it is popped and discarded: the first argument is not evaluated if the second argument is already in weak head normal form. This is different than an implementation of eager futures, where tasks popped from the bottom of the queue must be evaluated. This points to an important difference in how partial speculation can be implemented. In this case, there is no need make a distinction between fast and slow clones because tasks pushed onto a deque will only be run after a steal: there is no equivalent of a fast clone. It is also unclear how to apply the method of reasoning about cache misses used above. With partial speculation, a parallel execution may run code that was never run during a sequential execution. Therefore the

number of cache misses incurred during the sequential execution may tell us very little about the number incurred during a parallel execution.

6.6.2 Cost Semantics

This leads us to the real challenge in programming with partial speculation: reasoning about program performance. The term "full speculation" is meant to remind programmers that they will pay (in the sense of spending resources) for *all* potential parallelism. With partial speculation it is difficult to say what code will be run. This is true for the partial speculation found in both call-by-value languages, such as Manticore, as well as call-by-need languages, such as Haskell. It is also true for parallel languages that support search, for example as in some parallel implementations of SAT. Here, it may be that a parallel implementation finds the required result very quickly not because it explores a large part of the search space but rather because it happens to visit the right part of that space very early in its execution. Addressing the questions about performance considered in this thesis for partial speculation is an interesting area that I leave for future work.

6.7 Summary

In this chapter, I have extended my language and cost semantics with parallel futures. Though parallel futures have appeared in the literature for nearly 30 years, there has been relatively little work in understanding how they can be implemented efficiently. By leveraging the concept of deviations, I have proved new bounds on the overhead of scheduling futures using existing work-stealing scheduling policies. These results suggest novel alternatives to existing policies that may incur less overhead. While I have addressed only the implementation of eager futures, related forms of parallelism, including partial speculation, present interesting opportunities for future work.

6.8 Related Work

Futures were originally proposed by Halstead [1985] as part of Multilisp and also appeared in work by Kranz et al. [1989]. Light-weight implementations of futures were described by Mohr et al. [1990] and Goldstein et al. [1996]. Parallel pipelining can also be implemented using a number of other language features such as I-structures in ID [Arvind et al., 1989] and streams in SISAL [Feo et al., 1990]. Blelloch and Reid-Miller [1997] give several example of applications where the use of futures can asymptotically reduce the depth of the computation graphs. They also give an upper bound on the time required to execute these computations by giving an implementation of a greedy scheduler that supports futures. Blelloch and Reid-Miller limit each future to a single touch to enable a more efficient implementation.

Static scheduling determines the mapping between parallel tasks and processors at compile time and has been used in the implementation of stream processing languages (*e.g.*, Gordon et al. [2006]) where the hardware platforms are determined in advance. Stream processing includes

limited forms of parallel pipelining, but static scheduling policies cannot support applications where the structure of the pipeline depends on the program input.

Greiner and Blelloch [1999] give a cost semantics for what fully speculative parallelism, but consider only an implementation using a depth-first scheduling policy. Partial speculation appears in languages such as Manticore [Fluet et al., 2008b] and Glasgow Parallel Haskell [Trinder et al., 1998, Marlow et al., 2009]. While there are many similarities in the kinds of programs that can be expressed using partial speculation, it is not obvious how to apply the methods described in this chapter to reason about the performance of these programs.

The principle of work stealing goes back at least to work on parallel implementations of functional programming languages in the early 1980's [Burton and Sleep, 1981, Halstead, 1984]. It is a central part the implementation of the Cilk programming language [Frigo et al., 1998]. Cilk-style work stealing has also been implemented in Java [Lea, 2000], X10 [Agarwal et al., 2007, Cong et al., 2008], and C++ [Intel Corporation, 2009].

Work stealing for nested-parallel programs (such as those in Cilk) was studied by Blumofe and Leiserson [1999] who bounded the execution time in terms of the number of steals as well as the additional space required. Agarwal et al. [2007] extend Blumofe and Leiserson's work on space bounds to *terminally strict* parallelism, a class of parallel programs broader than nested-parallel (fully strict) programs but still not sufficiently expressive to build parallel pipelines. Arora et al. [1998] give a work-stealing scheduler for unrestricted parallelism and bound the number of steals and execution time but not space. They also consider performance in a multi-programmed environment. Acar et al. [2000] bound the cache overhead of this scheduler using *drifted nodes* (which I have re-dubbed deviations) but only for nested-parallel programs.

Chapter 7

Implementation

The various semantics in previous chapters have served as abstract descriptions of how a parallel programming language could be implemented. These semantics have omitted details about many parts of the implementation, including the compiler, garbage collector, and hardware, to present a simple picture of how different scheduling policies behave. In this chapter, I will describe the implementation of a deterministic parallel language as part of an existing compiler and runtime system. While previous implementations have served to describe how programs may be evaluated, the purpose of this implementation is to improve the performance of programs through the use of multi-processor and multi-core hardware. This implementation serves as a validation of the abstractions used in previous chapters as well as a basis for an implementation that can be used by programmers to make programs run faster. It also provide a framework for implementing and experimenting with new scheduling policies.

My implementation is built as an extension of MLton [Weeks, 2006], a whole-program compiler for Standard ML (SML) [Milner et al., 1997]. I chose MLton because of its maturity and stability, as well as the quality of its generated code. The first public release of MLton was nearly 10 years ago, and it now supports five different architectures and eight operating systems. For small programs, MLton achieves performance comparable with other mature compilers for functional languages (*e.g.*, GHC, OCaml) and is within a factor of two of top compilers for any language [Weeks, 2006]. By leveraging MLton's aggressive optimizations and representation strategies, I was able to write much of the implementation of different scheduling policies in SML.

My implementation is described in the following three sections. First, I will discuss changes I made to the existing implementation of MLton, including changes to the native runtime system, the compiler, and the accompanying libraries. Second, I will discuss the primitive building blocks I designed and then used to implement nested fork-join and array parallelism, futures, and synchronization variables [Blelloch et al., 1997]. Finally, I will describe my implementations of three different scheduling policies.

One significant downside of choosing MLton over some other functional language implementations was that, at the time that I started my implementation, MLton was a uni-processor implementation. While MLton included support for concurrency in the form of a light-weight, user-level threading library, it did not support simultaneous execution of multiple threads. Therefore, a significant part of my implementation work was to extend MLton to support execution

on multiple processors simultaneously. To offset this effort, I made several decisions to limit the other implementation work that I would perform as part of this thesis, as discussed below.

No Parallel Garbage Collection. Parallel garbage collection is critical in achieving good performance in a parallel language. To understand why, consider a typical uni-processor application that spends 10% of its time reclaiming memory using a garbage collector. If this 10% cannot be parallelized then Amdahl's law [Amdahl, 1967] states that the best achievable speed-up on eight processors is 4.7.[1] However, the situation is even worse if we make the reasonable assumption that each processor is generating garbage at the same rate. In that case, the best speed-up achievable with eight processors is 1.1.[2]

Despite this, I have not implemented a parallel garbage collector as part of this thesis, for the following two reasons. First, I know of no reason why previous work on parallel collection for SML (*e.g.*, Cheng and Blelloch [2001]) could not be integrated into my parallel version of MLton. Thus, it is simply a matter of the engineering effort required to adapting this work to the data structures used in the MLton runtime. Second, it is easy to measure the time consumed by the garbage collector separately from the rest of the application. In the empirical results reported in the next chapter, I will only report the time used by the application.

C Backend. MLton includes several backends, including native support for both the x86 and AMD64 (also known as x86-64) architectures. It also includes a backend that generates platform-independent C that can be used on any of the SPARC, PowerPC, HPPA, x86, and AMD64 platforms. I chose to develop my extension to MLton as part of the C backend, because I planned to use both 32- and 64-bit Intel platforms and because I believed that the C backend would be easier to debug. Some cursory testing showed that for the benchmarks included in the MLton source distribution, the C backend was only about 30% slower than the native x86 backend. In the end, the only changes that I made that were specific to the C backend were a handful of lines used during thread initialization and to access thread-local state. It should be relatively straightforward to port these changes to the other backends.

No New Syntax. Though I have described nested parallelism and futures as extensions to the syntax of a functional language, I am mostly interested in how these features can be implemented. Rather than modifying the parser found in MLton, I have implemented these features as a library. Thus, an example such as, **future** produce (f, idx + 1) must be written as,

future (**fn** () ⇒ produce (f, idx + 1))

in my implementation. Extending the parser to support additional forms of expressions is trivial, and unless I change the static semantics of the language, there few reasons to do so. Using a library interface also allows programmers to debug programs using other implementations of SML with a stub implementation of that interface. In many cases, I used this technique to develop applications using SML/NJ [Appel and MacQueen, 1991], which offers fast, incremental recompilation, and then test the sequential version of an application. Because the same source

[1] In terms of the measure of efficiency used in the next chapter, each of the eight processors is 59% efficient.

[2] This yields only 14% efficiency.

code could also be used in MLton, I could then quickly switch to MLton and run a parallel version as well.

Limited Support For Other MLton Extensions. In addition to a threading library, MLton also includes support for interrupt handlers, finalizers (*cf.* [Boehm, 2003]), sample-based profiling, and a foreign function interface that allows SML code to invoke functions written in C. My parallel extension of MLton provides limited or no support for these features.

7.1 Changes To Existing Code

As noted above, MLton is a stable and mature implementation of SML. It also strives for simplicity and adherence, when feasible, to a functional style of programming. These both helped to facilitate my implementation work.

7.1.1 Runtime

The MLton runtime is comprised of a small amount of platform-specific code along with about 10,000 lines of C that is shared among all platforms. The garbage collector forms a large portion of that platform-independent code.

MLton maintains activation frames for ML function calls using contiguous stacks. Regardless of the backend, these stacks are maintained separately from the C call stack. These stacks are heap allocated, but in contrast to the suggestion offered by Appel [1987], individual frames may be reclaimed immediately after a function has returned. When there is not sufficient room for a new activation frame, a new, larger stack is allocated in the heap, and existing frames are copied to the new stack.

These stacks may be manipulated by the programmer using the MLton.Thread library. In particular, this library forms a basis for an implementation of call/cc.

Spawning Threads. I chose to use the POSIX threads (pthreads) interface to expose hardware parallelism. This means that my version of MLton supports parallel execution on any POSIX-compliant operating system (OS). I will use the term "pthread" to distinguish these heavy-weight threads from the user-level threads provided by MLton.Thread; other thread libraries would also suffice.

I have added a new run-time parameter to indicate the number of pthreads that should be spawned at the beginning of program execution. I rely on user-level threads (as implemented by MLton.Thread) to manage the interleaving of parallel tasks on a single processor, and so I only require one pthread per processor or processor core. User-level threads are significantly lighter weight than pthreads in that user-level context switches require only updating a few fields of the per-processor state and checking that there is adequate free heap space for the new thread to continue.

After all run-time parameters are parsed and the heap is initialized, a set of pthreads is initialized. These pthreads cannot immediately call into generated code, however, as none have an associated ML stack. Instead, they wait for the original OS-level thread to call into generated

code and finish any initialization code found in the Basis. Once this step is completed, that thread calls back into the runtime where its ML stack and associated data are duplicated, once for each additional pthread. These threads then invoke a well-known function in the scheduler library and wait for tasks to be added to the schedule queue. This initialization sequence is implemented by the following two lines of ML (along with the supporting code in the runtime).

```
val () = (_export "Parallel_run": (unit → empty) → unit;) return
val () = (_import "Parallel_init": unit → unit;) ()
```

These lines use MLton's foreign function interface (FFI) for importing C functions into ML and exporting ML functions into C. The first line exports a function of type unit → empty (empty is the empty type). In particular, it binds the function return, which invokes the main scheduler loop, to the C identifier Parallel_run. The second line exposes a C function called Parallel_init, which in C terminology takes no arguments and has return type void, and then immediately invokes it. The implementation of Parallel_init duplicates the current ML stack, as described above and then invokes Parallel_run. Note that these two lines must be evaluated in this order to avoid a race condition. The MLton compiler tracks the effectfulness (or inversely, purity) of each expression, and since parts of the FFI are considered effectful, the compiler will never reorder these two declarations.

On Linux systems, I use the Portable Linux Processor Affinity (PLPA) library[3] to set the affinity of each pthread. As I intend for there to be at most one pthread per processor, setting the affinity helps to limit uncertainty caused by the kernel migrating pthreads from one processor to another.

Exclusive Access. With relatively few changes to the compiler, these pthreads can then run generated code independently. Difficulties only arise when one or more threads must modify the state of some shared resource, for example as during a garbage collection or when allocating a large object. To protect access to these resources, I added a global synchronization barrier to the runtime. Whenever any pthread enters the runtime system, it first requests exclusive access. While my implementation might distinguish between different kinds or instances of resources, a single barrier is used for all exclusive access with the exception of allocation, as discussed below.

For a pthread to perform an operation such as a garbage collection, not only must all other pthreads avoid entering the runtime system, but they must also expose any pointers so that the collector can compute reachability correctly. MLton already uses safe points as part of its support for interrupt handlers. It uses a form of software polling [Feeley, 1993] so that when an interrupt is handled by the runtime, the allocation limit of the current thread is overwritten with a value (*e.g.* zero). The current thread will then call into the runtime as soon as a safe point is reached. A similar mechanism can be used to ensure that all pthreads reach the barrier in a timely manner. MLton supports two different compilation modes with respect to the allocation limit, depending on whether or not the program uses interrupt handlers. If the program does use interrupts, then extra checks of the limit are added to loops that perform no allocation. Similar, checks are made by programs that include uses of parallelism.

Once all pthreads have reached the barrier, one pthread at a time is allow to enter the critical

[3] `http://www.open-mpi.org/projects/plpa/`

section and perform its required operation (if any). Upon either entering or leaving the critical section, each pthread must ensure that any additional pointers are added to a globally accessible data structure. Pthreads that enter the runtime because of an allocation request also put the parameters of that request in a globally accessible data structure so that at most one garbage collection is performed in each synchronization round.

Allocation. Using such a heavy-weight synchronization mechanism for every allocation would be far too expensive. My implementation divides the unused portion of the heap and allows each pthread to allocate in that portion without further synchronization. Rather than assume that all pthreads will allocate at the same rate, however, free space is parceled out to processors in four kilobyte pages. Within a page, memory is allocated using the pointer-bumping scheme already present in MLton. When a page is exhausted, the pthread attempts to claim another empty page using an atomic compare-and-swap operation. Note that allocation pages are specific to a pthread and not a user-level thread. Though this may result in poorer locality among heap objects, it means maintaining less state for each user-level thread and less overhead in switching between user-level threads.

Pthread-Local State. Each pthread must maintain its own allocation pointer (a pointer to the next free memory cell) and limit (the end of the last free memory cell), as well as pointers to the current stack and thread data structures. This information was already stored in the GC_state structure, so I modified the runtime to allocate one such structure for each pthread. After each such structure is initialized, it is associated with a pthread using pthread_setspecific, a function that creates a map that, given a key, assigns a different value for each pthread. When a call is made into the runtime system, a call to pthread_getspecific returns a pointer to the GC_state associated with the current pthread. Once a reference to this state is on the C stack it is relatively easy to pass this state explicitly within the runtime either by adding or reusing a function parameter. Some information in the GC_state structure is common to all processors, however, and a more robust solution would be to split this structure into two parts, separating global state from processor-specific state. I leave such an implementation for future work: at the time that I began my implementation, it was not clear to me how to split this state and, though not as maintainable, duplicating the shared state was more expedient.

The generated C code also makes frequent access to parts of the GC_state structure, especially the allocation pointer and limit as well as the stack pointer and limit. When using the C backend, MLton compiles a program into a large switch statement with each function implemented (roughly) by a single **case**. Due to limitations in C compilers, this switch statement must be divided into many "chunks," where each chunk consists of a C function with a single switch statement. Each such function includes several local variables that act as virtual registers for holding the allocation and stack state. I have added an additional local variable and a call to pthread_getspecific to initialize this variable at the beginning of each chunk. Though this pthread-specific state could be passed explicitly between chunks (to avoid calls to pthread_getspecific), MLton already attempts to limit the number of inter-chunk calls (because they are more expensive) by moving mutually invoked ML functions into the same chunk. This has the effect of also limiting the number of calls to pthread_getspecific.

7.1.2 Compiler

The MLton compiler itself consists of approximately 140,000 lines of SML. Fortunately, nearly all of the compiler generates code that is safe for parallel evaluation. In this section, I will describe the changes I was required to make to ensure correct parallel evaluation, both in terms of the extensional behavior of programs and performance.

Much like the "virtual registers" used to store information about free allocation space and the stack, the C backend defines a set of global variables as temporaries that are used by the generated code. To support multiple processors, I have moved these temporaries into the functions defining each chunk as local variables. This has little effect on the overall memory use of the program since inter-chunk calls are made via a trampoline [Steele, 1978] rather than direct function calls. New frames are added to the C stack only to make FFI calls from native C functions into ML. Thus the total number of frames on the C stack, each of which includes space for these temporaries, will never exceed the number of interleaved calls from C to ML and vice versa.

MLton also uses global variables to store the arguments, results, and identity of ML functions exported through the FFI. The export portion of the FFI is implemented using a single function that is exported into the C name-space and a table on the ML side to dispatch calls accounting to one of these variables. In my extension of MLton, I moved the variable identifying which function is being invoked (the so-called "operation") into the processor-specific state. Thus in my version of MLton, only ML functions of type $\mathsf{unit} \rightarrow \mathsf{unit}$ may be safely exported. This is sufficient for the uses of the FFI used in my parallel implementation. Since I branched the MLton source code, the implementation of this portion of the FFI has changed to only require a single global variable. Therefore, a small change to my method would be sufficient to support the full range of exported ML functions.

One final example of a global variable that I was required to move into the processor-specific state is the variable holding the exception that is currently being raised.[4] To be more precise, this variable holds such an exception for the short duration between the moment when it is raised and when it is handled. If two processors were to both raise exceptions at the same time, then if one processor were to overwrite this variable in between a write and a read performed by the other processor, then both processors would, in effect raise the same exception. Like the case of the current FFI operation, moving this variable to the processor-specific state avoids this problem.

Reference Flattening. I will describe one optimization, reference flattening, that is implemented in MLton in more detail because it was not, as it was originally implemented, safe-for-space [Shao and Appel, 1994]. Analogously to tuple flattening, references that appear in heap-allocated data structures (e.g. records) may be flattened[5] or treated as a mutable field within that structure. At run time, such a reference is represented as a pointer to the containing structure. Accesses to the reference must computed using an offset from that pointer.

This optimization can save some memory by eliminating the memory cell associated with the reference. However, it can also increase the use of memory. As the reference is represented as a pointer to the entire structure, all of the elements of the structure will be reachable anywhere the

[4]This variable has the rather overly general name globalObjptrNonRoot and is meant to hold arbitrary pointers that need not be traced by the collector. Its only use in the current version of MLton is that described in the text.

[5]This is an entirely different use of the word "flattening" than that used in Chapter 5.

reference is reachable. If the reference would have outlived its enclosing structure then flattening will extend the lifetime of the other components.

To avoid asymptotically increasing the use of memory, MLton uses the types of the other components of the structure to conservatively estimate the size of each component. If any component could be of unbounded size then the reference is not flattened. The problem was that this analysis was not sufficiently conservative in its treatment of recursive types. Though a single value of a recursive type will only require bounded space, an unbounded number of these values may be reachable through a single pointer. Based on my reporting, this problem has been corrected in a recent version in the MLton source repository. I will also discuss the particular program that led me to find this bug in the next chapter.

7.1.3 Libraries

While most of the MLton implementation of the Basis library and MLton's extensions to the Basis is written in a functional style, some parts use mutable state either for reasons of efficiency or because the of the functionality that is provided. Fortunately, these instances are relatively easy to find by looking for occurrences of either the `ref` or `array` constructors.

An example of where mutable state is used to improve performance is in the implementation of integer-to-string conversion. Here, a single buffer is allocated during initialization and is then used to store the results of formatting an integer. This avoids allocating many small, short-lived objects in programs that make heavy use of integer formatting. A similar device is used in some of the floating-point number conversions. In these cases, global mutable state is protected against race conditions that may occur because of interrupts but not because of multi-processor execution.

Another use of mutable state occurs in the implementation of MLton's user-level thread library, `MLton.Thread`. In this case, mutable state is used to implement the creation of new user-level threads. To create a new thread, the function that will be executed by that thread must be specified. Given the interface to creating new threads provided by the runtime system, this function must be stored in a reference cell so that it can be communicated to the newly created thread. Obviously each processor requires its own such reference cell.

In my extension of MLton, I have addressed these uses of mutable state on an *ad hoc* basis and as necessary. However, a more robust solution would be to add a compiler extension that would provide a notion of processor-specific state in ML. In a program compiled for only a single processor, such state could be implemented using a reference cell, but in a multi-processor compilation, it would be implemented as an array. Uses of that array could then be automatically indexed by the current processor. I leave such a solution to future work.

7.2 Implementing Parallelism

The extensions to MLton described in the previous section enable the MLton runtime to run code on multiple processors simultaneously, but offer no additional language tools or features to help programmers manage this process. In fact, those extensions could be used to support a variety of different styles of multi-processor programming, including both parallelism and concurrency

(*e.g.*, CML [Reppy, 1999]).[6] In this section, I will describe a library that provides support for various deterministic parallel primitives in MLton.

This library is factored into three parts: a set of primitives for managing parallel tasks, an interface for writing parallel programs, and a set of scheduling policies. This architecture enables me to easily reuse code between different forms of parallelism and to quickly implement new scheduling policies. Moreover these primitives also serve to isolate the implementation of different forms of parallelism (*e.g.*, nested parallelism, futures) from the scheduling policy.

7.2.1 Managing Tasks

The set of primitives described are not meant to be used by programmers directly, but instead as a basis for building parallel language features such as synchronization variables, fork-join parallelism, and futures (as described below). As such, programs that use these primitives directly will not be *parallel* programs, but *concurrent* ones: programs where the result depends on the how the effects of different processors are interleaved. Note that I am not claiming that these are the only set of primitives for building concurrent programs, only that they provide a clean way of implementing the forms of parallelism described in the next section.

The two core primitives provide an interface for initiating new parallel tasks and ending them.

```
type empty (* the empty type *)
type task = unit → empty
(* tasks are forced to call return when they are done *)
val add : task → unit
val return : task
```

The add primitive takes a function as an argument that should be added to the set of ready tasks. To force users of these primitives into a discipline of explicitly ending parallel tasks, the signature requires that every job returns a value from the empty type. In fact, empty is defined to be equal to the unit type, but this equality is hidden using opaque signature ascription. The only way to convince the compiler that you have obtained such a value is to call return. When invoked, return jumps back into the scheduler loop to execute the next task.

These primitives also include a mechanism for delaying the execution of a piece of code.

```
type α susp (* type of suspensions waiting for an α *)
val suspend : (α susp → unit) → α
val resume : α susp * α → unit
```

The function suspend is used when the further execution of current task must be delayed (for example when a future that has not yet been computed is touched). It reifies the current context and passes that context to its argument (much like call/cc). This argument should store the suspension in some mutable data structure and return control to the scheduler so that the next task may be scheduled. In particular, this argument should *not* invoke resume. These primitives are implemented efficiently with a lightweight wrapper around MLton's user-level thread library.

[6] A group working under Suresh Jagannathan at Purdue University is currently implementing a version of CML based on my extension of MLton.

One subtle aspect of this implementation is that since callers of suspend will typically store the reified context in a *shared* data structure, there is often a potential race between the current processor and another processor that reads from that data structure and resumes the suspension. Recall that in MLton, the call stack is implemented using an empheral data structure; this call stack forms a large portion of the reified context described above. This problem was also noted by Li et al. [2007] in the context of designing concurrency primitives in Haskell.

I have implemented suspend and values of type α susp in such a way that users of suspend need not be concerned with these race conditions: the argument to suspend is not run using the same thread or stack as was used to call suspend. That is, instead of switching to a new thread *after* the invocation of the argument to suspend, this switch occurs *before* the argument is invoked. Rather than creating a new thread solely to execute the argument to suspend, the implementation takes the next task that will be executed from the scheduling policy and temporarily hijacks the thread that will be used to execute that task. (If there is no task to be executed then a new thread will be created to run the argument to suspend.) This avoids the overhead of creating a new thread as well as the additional synchronization that would be necessary to invoke the argument to suspend atomically.

I also provide a three variations of the add primitive described above. The first variation allows tasks to be removed from the set of ready tasks.

type token *(∗ identifies a task in the set of ready tasks ∗)*
val add' : task → token
val remove : token → bool

It returns a token which serves to uniquely identify this task in the ready set. A complementary primitive remove allows tasks to be removed from the ready set. It returns a boolean indicating whether or not the task was successfully removed. Note that remove does not cancel an existing task. Instead it only removes tasks which have not yet started execution. The primitive remove returns true if and only if the given task will not be executed.

The second variation of the add has the same type as the original but assigns a different priority to the given task.

val run : task → unit

The function run behaves similarly to add except that instead of adding its argument to the set of ready tasks, it adds a task containing the current continuation and begins running the argument instead: it prioritizes its argument over the continuation.

The final variation of the add primitive, called yield, adds a task to the ready set that represents the current continuation.

val yield : unit → unit

This primitive should be invoked if there is still work to be performed in the current task, but that task should yield to a higher priority task, if any. The full signature is shown in Figure 7.1.

7.2.2 From Concurrency to Parallelism

The primitives described in the previous section can be composed, along with a few other data structures and synchronization primitives, to provide implementations of nested parallelism and

Figure 7.1 Task Management Primitives. An implementation of this signature is used in implementing the various forms of parallelism found in this thesis, as described in the text.

```
signature TASK_SET =
sig
  type empty (* the empty type *)
  (* Tasks are forced to call return when they are done. *)
  type task = unit → empty

  (* Add a new task to the ready set. *)
  val add : task → unit
  val return : task

  (* Provide a call/cc−like mechanism for adding tasks. *)
  type α susp (* type of suspensions waiting for an α *)
  val suspend : (α susp → unit) → α
  val resume : α susp * α → unit

  (* Add a task in such a way that it can be removed from the ready set. *)
  type token (* identifies a task in the set of ready tasks *)
  val add' : task → token
  val remove : token → bool

  (* Add the current continuation to the set of ready tasks and continue with the given task. *)
  val run : task → unit

  (* Add the current continutation as a task if there is another task to run instead *)
  val yield : unit → unit
end
```

parallel futures. As noted above, doing so requires writing concurrent programs. I again draw an analogy to garbage collection. Just as a garbage collector manages memory explicitly so that, from the point of view of the applications, it is managed automatically, a similar role is played by my parallel library: I have implemented a library using concurrency so that programmers don't have to. In the remainder of this section, I will give implementations of both nested parallelism and parallel futures, along with an implementation of synchronization variables, which are used in the implementations of the other two.

Synchronization Variables. A synchronization variable [Blelloch et al., 1997] (or single cell I-structure [Arvind et al., 1989]) is a write-once memory cell where attempts to read the contents of the cell before anything is written force readers to block until a value is available: every reader will see the same value. Figure 7.2 shows a implementation synchronization variables based on that found in my extension of MLton. (I have omitted some instrumentation for the sake of clarity.) This implementation defines two types (including one auxiliary type) and three functions. First, I define a datatype that will be used to record the state of a synchronization variable: either there are zero or more readers waiting on the value of the variable or the value has already been written. A synchronization variable is a reference to a instance of this datatype. The first function var creates a new, empty synchronization variable with a new lock and an empty list of waiting tasks.

The read function first checks if to see if a value has been written. If that is the case then it simply returns that value. If not, then it takes the lock and re-reads to avoid a race. If the value still has not been written, then it calls suspend and adds the suspension to the list of tasks waiting for the value.

Ignoring the erroneous and impossible cases, the write function takes the lock, reads the list of waiting suspends, and writes the new value. It then resumes each of the readers that have suspending waiting for the value.

Unlike the other forms of parallelism implemented below, it is easy to create programs that misuse synchronization variables and, as a result, deadlock. For example, a program that reads from such a variable without ever writing to it, or a program with two synchronization variables where the write to each variable is guarded by a read to the other. It is also easy to write programs that incur runtime errors by writing to a future multiple times. For these reasons, I do not advocate building applications with them but describe them here only because they are useful in implementing the other forms of parallelism described below.

Fork-Join Parallelism. The first form of nested parallelism that I considered in this thesis was based on parallel pairs, or fork-join parallelism. Rather than writing $\{e_1, e_2\}$, programmers using my MLton implementation write fork (**fn** () $\Rightarrow e_1$, **fn** () $\Rightarrow e_2$). A simplified version of the implementation of fork is shown in Figure 7.3.

This implementation uses a synchronization variable to store the result of the right-hand branch and to synchronize between the two branches. Though simplified, it gives an example of how synchronization variables are used. Consider an execution on a uni-processor machine. At line 5, a new task is added to the ready set. The processor will continue by evaluating f (). It will attempt to read the value of the synchronization variable v at line 8. This will suspend,

Figure 7.2 Implementation of Synchronization Variables. This structure shows an implementation of synchronization variables based on locks and the primitives shown in Figure 7.1.

```
structure SyncVar =
struct
  datatype α state =
      Waiting of Lock.lock * α susp list
    | Done of α
  (* type of synchronization variables; abstract in signature *)
  type α var = α state ref

  fun var () = ref (Waiting (Lock.new (), nil))

  fun read v =
    case !v
      of Done a ⇒ a (* do the easy case first *)
       | Waiting (lk, _) ⇒
         let in (* take the lock *)
           Lock.lock lk;
           case !v (* read again, now with lock *)
            of Done a ⇒ (Lock.unlock lk; a)
             | Waiting (_, readers) ⇒
               suspend (fn k ⇒ (v := Waiting (lk, k::readers);
                                Lock.unlock lk))
         end

  fun write (v, a) =
      case !v of Done _ ⇒ raise ... (* can't have two writes *)
        | Waiting (lk, _) ⇒
          let (* first take the lock *)
            val () = Lock.lock lk
            (* now read the wait list *)
            val readers = case !v
                           of Waiting (_, readers) ⇒ readers
                            | Done _ ⇒ raise ... (* async write? *)
            val () = v := Done a
            val () = Lock.unlock lk
          in (* add readers to the queue *)
            app (fn k ⇒ resume (k, a)) readers
          end
end
```

Figure 7.3 Simple Implementation of Fork. This implementation does not properly handle exceptions nor implement any of several possible optimizations. A more sophisticated version is shown in Figure 7.4.

```
    fun fork (f, g) =
2     let
        (* create a shared variable for the right-hand side *)
4       val v = SyncVar.var ()
        val () = add (fn () ⇒ (SyncVar.write (v, g ());
6                              return ()))
        val a = f ()
8       val b = SyncVar.read v
      in
10      (a, b)
      end
```

since no processor has yet evaluated g. The processor will then evaluate the only ready task: the task added at line 5. Once g has been evaluated, it will resume the suspended task and then return to the schedule. At this point, there is again exactly one ready task, which will continue by building a pair of the results of calling f and g. This implementation, however, is lacking in several respects. A more complete version is shown in Figure 7.4 and discussed below.

First, the simplified implementation does not accurately reflect the parallelism that I have discussed in previous chapters. While it allows the current task to be preempted when it invokes add, it does not allow any other task to run between the invocation of g and code that runs after the join. To allow the scheduling policy to run a different task instead, I add several calls to yield including one just before the final result is constructed.

Second, I admit one limited form of effects: exceptions, despite the fact that I encourage programmers to write functional code. Given the implementation in the text above, any exception raised by g would be lost. Furthermore, if any exception were raised by g then the call to read at line 8 would never return. The implementation in the figure properly handles exceptions by catching any exceptions and storing either the normal result or the exception in the synchronization variable.

I should point out, however, there are some cases where the performance of this implementation may differ from an SML programmer's expectation. For example, while the following code will raise a Match exception as expected, if another processor starts evaluating on the second argument to fork, that processor will continue to loop for the remainder of the program execution.

```
fork (fn () ⇒ (. . . (* busy wait *); raise Match),
    let fun loop () = loop () in loop () end)
```

Handling this case more consistently with SML's sequential cost semantics would require a mechanism for canceling tasks. As I have not implemented such a mechanism, I advise programmers not to use exceptions within parallel tasks to guard against meaningless cases.

Finally, the implementation above misses an important opportunity for optimization. In many

Figure 7.4 Implementation of Fork-Join Parallelism. This figure shows a more realistic version of fork. This implementation handles exceptions properly and also avoids running tasks taken from the task queue if there is a more efficient alternative.

```
structure ForkJoin =
struct
  datatype α result =
      Finished of α
    | Raised of exn

  fun fork (f, g) =
      let
        (* Used to hold the result of the right−hand side in the case where
           that code is executed in parallel. *)
        val var = SyncVar.var ()
        (* Closure used to run the right−hand side... but only in the case
           where that code is run in parallel. *)
        fun rightside () = (SyncVar.write (var, Finished (g ()))
                                                handle e ⇒ Raised e);
                            return ())

        (* Offer the right side to any processor that wants it *)
        val t = add' rightside (* might suspend *)
        (* Run the left side *)
        val a = f ()
            handle e ⇒ (ignore (remove t); yield (); raise e)
        (* Try to retract our offer − if successful, run the right side
           ourselves. *)
        val b = if remove t then
                  (* no need to yield since we expect this work to be the next thing
                     in the queue *)
                  g ()
                  handle e ⇒ (yield (); raise e)
                else
                  case SyncVar.read var of (Finished b) ⇒ b
                                         | (Raised e) ⇒ (yield (); raise e)
      in
        yield ();
        (a, b)
      end
end
```

Figure 7.5 Signature For Parallel Vectors. This signature describes the operations on parallel vectors supported by my implementation.

```
signature PARALLEL_VECTOR =
sig
  type α vector = α Vector.vector
  (* The first argument of the next six functions indicates the maximum number of elements
    that will be processed sequentially. *)
  val tabulate : int → (int → α) → int → α vector
  val replicate : int → (int × α) → α vector
  val index : int → int → int vector
  val map : int → (α → β) → α vector → β vector
  val concat : int → (α vector × α vector) → α vector
  val reduce : int → (β × β → β) → β → (α → β) → α vector → β
  (* These two functions do not offer any parallelism: both require only constant work and
     depth. *)
  val length : α vector → int
  val sub : α vector × int → α
end
```

cases, the combined cost of synchronizing on the shared variable, removing a task from a deque, and running it is much higher than simply running the task directly (in this case calling g). This is the observation that leads to the fast/slow clone compilation technique used in the implementation of Cilk [Frigo et al., 1998] (*cf.*, Chapter 6).

The implementation given in the figure uses the add'/remove interface to remove unexecuted tasks from the ready set. That is, the task added at line 19 is, if no other processor has started to execute it, removed at line 25. When that task is removed successfully, g is invoked directly and no synchronization is necessary after g is completed. A similar optimization based on removing tasks is also used in Doug Lea's implementation of fork-join in Java.[7]

Parallel Vectors. Though I have not implemented the flattening transformation described in Chapter 5 as part of my MLton implementation, I still support the collection-oriented operations described in that chapter. Figure 7.5 shows a portion of that signature.

Each of the functions that processes multiple elements of a vector takes an integer argument that indicates the maximum number of elements that should be processed sequentially; this is the first argument to the first six functions that appear in the signature. This argument enables programmers to throttle the number of parallel tasks and should be set depending on the amount of computation associated with each vector element.

The functions tabulate, map, length, and sub should be familiar to functional programmers. replicate and index are described in Chapter 5 and can be implemented easily using tabulate.

The last function, reduce, is a generalization of the common foldl and foldr functions and

[7]This was described in Lea's talk at IBM T.J. Watson Research Center on November 3, 2008.

Figure 7.6 Implementation of Futures. This structure shows how parallel futures can be implemented using synchronization variables and the task management primitives shown previously.

```
structure Future =
struct
  datatype α result = Finished of α | Raised of exn
  type α future = α result SyncVar.var

  fun future f =
    let
      val v = SyncVar.var ()
    in
      run (fn () ⇒ (SyncVar.write (v,
                                   Finished (f ()) handle e ⇒ Raised e);
                    return ()));
      v (* return the variable *)
    end

  fun touch v = case SyncVar.read v
                  of Finished v ⇒ (yield (); v)
                   | Raised e ⇒ (yield (); raise e)
end
```

enables the programmer to aggregate the values in a vector. Unlike foldl and foldr, which traverse the vector in a specified order, reduce performs that aggregation in parallel. For example, the following code computes the sum of the elements of a vector v.

```
reduce maxSeq op+ 0 (fn i ⇒ i) v
```

One can think of the second argument like the multiplication operation of a group and the third argument as the unit. The fourth argument acts as injection function from elements of the vector into the group. If multiplication operation is associative and the unit is the identity element of the group (as required by the group laws), then reduce behaves (extensionally) according to the following equivalence:

```
reduce _ m u inj v = Vector.foldl (fn (a, b) ⇒ m (inj a, b)) u v
```

The functions in this signature are implemented using ForkJoin.fork. For each of these functions, if the input set is larger than the maximum number of elements that should be processed sequentially, the set of elements is split in half and each half is then considered in parallel using fork. Each half may be split again until the set becomes small enough to process sequentially.

Futures. Finally, I will consider an implementation of parallel futures in the style of those found in Multilisp [Halstead, 1985]. Figure 7.6 shows the source code of my implementation. Like the implementation of fork-join, I use a synchronization variable to communicate results between parallel tasks. In fact, a future is largely a wrapper around a synchronization variable

Figure 7.7 Signature For Scheduling Policy. This signature describes the functionality required from a scheduling policy. To add new scheduling policies to my implementation, one must provide an implementation of this signature.

```
signature SCHEDULING_POLICY =
sig
  (* processor identifier *)
  type proc = int
  type task = TaskSet.task

  (* the first argument is the identifier of the current processor *)
  (* add new task to the queue; highest priority appears first *)
  val enque : proc → task list → unit
  (* remove the next, highest priority task *)
  val deque : proc → task option
  (* mark the most recent task as completed *)
  val finish : proc → unit
  (* is there higher priority task for the given processor? *)
  val shouldYield : proc → bool
end
```

that accounts for exceptions and adds the necessary yields. In addition, because this synchronization variable is not bound in the body of the future, there is no way (without using some other effects) for a single task to read the variable before writing to it. This implies that there is no way for a program that uses futures (or a combination of futures and nested-parallelism) to deadlock. In addition, this use of synchronization variables is guaranteed to write to each variable at most once, avoiding any runtime errors that would occur due to multiple writes to the same variable.

7.3 Scheduling

Given that task synchronization and a simple programmer interface are implemented elsewhere, all that remains for the scheduling policy is to maintain the set of ready tasks and, when a processor becomes idle, find another task for it to execute.

The interface to the scheduling policy looks much like the interface for a priority queue. However, priorities are not assigned by the user of the queue but predominately by the queue itself. Users of a scheduling policy only assign relative priority when adding a new task: by choosing either add or run, the implementer determines whether the argument to these functions or the continuation will be run first. The scheduling policy determines the priority between tasks added at different times or by different processors. The signature implemented by scheduling policies is shown in Figure 7.7.

All the functions of a scheduling policy are parametrized by the current processor. Given the purpose of scheduling policies, the functions enque and deque should be self-explanatory.

The finish function is called once for each task removed from the queue. For many scheduling policies, finish does nothing. The final function shouldYield is used to avoid some non-trivial thread operations in cases where they are unnecessary. This function returns a boolean value that indicates whether or not there is any other ready task that has a higher priority (as assigned by the scheduling policy) than the task being evaluated by the current processor. If the result is true, then the current task should be suspended and enqueued. Otherwise, the current task can continue. This operation is discussed in more detail in the description of the work-stealing scheduler below. I present this interface as an SML signature; I believe that this abstraction would be useful for implementations of deterministic parallelism in other languages.

The main loop executed by each of the processors and implemented as part of the implementation of the TASK_SET signature consists of repeatedly calling deque, running the task that is returned, and then calling finish.

7.3.1 Breadth-First Scheduling

As has been noted in previous chapters, the breadth-first schedule often results in poor performance with respect to application memory use. Despite this, it is often the first (and occasionally only) scheduling policy to be implemented in a parallel or concurrent language. It is a "fair" scheduling policy in that it divides computational resources among all active threads and so is a reasonable choice for languages that support only a small number of threads or concurrent languages, where some notion of fairness is expected by programmers. The breadth-first policy is easy to implement and the simplest policy in my implementation.

I implemented a breadth-first policy in part to test other aspects of my parallel implementation and in part because I wanted to test my hypotheses about the performance of applications using a breadth-first schedule.

The breadth-first policy maintains tasks as a FIFO queue implemented using a doubly-linked list: new tasks are added to the end of the queue and tasks are always removed from the beginning of the queue. Access to the queue is protected by a single, global lock.

An integer reference cell is associated with each task in the queue. When a request to remove a task from the queue is made, the value of this cell is atomically updated to indicate that it should not be run. Conversely, when a task is about to be run, this cell is atomically updated to indicate that subsequent attempts to remove it should fail. While this could be implemented as a boolean, I have only added infrastructure supporting atomic updates on integer values.

More efficient implementations are possible, for example, using a finer-grained locking scheme or a ring buffer to avoid some allocation. However, as the purpose of this implementation was for the most part to test theories about memory use, I have not pursued these alternatives.

7.3.2 Depth-First Scheduling

The parallel depth-first policy [Blelloch et al., 1999] prioritizes tasks according to a left-to-right, depth-first traversal of the computation graph. While the simulation of the depth-first policy described in Chapter 4 used a queue that was implemented using only a linked list, a true multi-core implementation requires some more sophistication. While the simulation fully synchronizes

all processors after each step (since they are simulated using a sequential program), a multiprocessor implementation will execute tasks more independently. For example, consider the following scenario. Say that two processors p and q are working in parallel with p working on task that is "to the left" of the one upon which q is working. Furthermore, say that q adds a new task during the same period of time that p works on its task. If p subsequently interacts with the scheduler (providing an opportunity to switch tasks), it should *not* start working on any task added by q since, by definition, any such task is to the right of where p is working. Thus tasks should be added to the queue relative to their parent task. In this example, any task added by q should be prioritized lower that any task added by p.

Like the policy implemented as part of my profiling tools, my MLton implementation uses a single global queue and runs tasks from the front of the queue. To account for the situation described above, this queue is not maintained in a strictly LIFO manner: to ensure that my implementation obeys the global left-to-right ordering, the children of the leftmost task must be given higher priority than the children of nodes further to the right. Thus, in a sense, priority is inherited. To assign proper priorities, my implementation also maintains one "finger" for each processor that indicates where new tasks should be inserted into the queue. The finish function is used to clean up any state associated with this finger.

I also experimented with a scheduling policy without this complication (where the global queue is maintained as LIFO stack). However, as noted above, this policy is not faithful to a depth-first scheduling policy. There is little theoretical work addressing this form of scheduling, and I have not considered it in my experiments.

7.3.3 Work-Stealing Scheduling

My work-stealing implementation is based roughly on that implemented in Cilk [Frigo et al., 1998, Blumofe and Leiserson, 1999]. Like other work-stealing implementations, it maintains a set of deques. Locally, each queue is maintained using a LIFO discipline. However, if one of the processors should exhaust its own queue, it randomly selects another processor to steal from and then removes the oldest task from that queue. In the common case, each processor only accesses its own queue, so it can use a more finely-grained synchronization mechanism than in the other two scheduling policies to serialize concurrent access. This leads to less contention and significantly smaller overhead compared to the breadth- and depth-first schedulers, both of which use a single global queue.

Because a work-stealing policy favors local work, a deque that immediately follows an enque will always return the task that was added. My implementation avoids these two operations (and also avoids suspending the current thread) by always returning false as the result of shouldYield. The implementation of the task management primitives calls this function in cases where a deque will immediately follow an enque and, when it returns false, continues by executing the given task directly.

My work-stealing implementation is also parametrized to account for the design choices discussed in Chapter 6. These are specified as compile-time parameters.

7.4 Instrumentation

MLton already provides mechanisms for collecting data about allocation and how time is spent. To this, I have added instrumentation for recording the number of steals and deviations. As I am interested not only in the amount of data that is allocated but also the amount of live data at any point in time, I implemented the following novel technique for efficiently measuring memory use with a well-defined bound on the error.

Efficiently Measuring Memory Use. One method to measure memory use is to record the maximum amount of live data found in the heap at the end of any garbage collection. Using the collector to determine the high-water mark of memory use with perfect accuracy would require a collection after every allocation or pointer update. This would be prohibitively expensive. Unfortunately, the default behavior of most collectors is also a poor choice: most collectors, including that found in MLton, require time proportional to the amount of live data and are therefore designed to perform collections when memory use is relatively low. In other words, achieving efficient collection is directly at odds with performing accurate measurements of memory use.

In my early empirical work, I manually added explicit garbage collections at points I believed likely to correspond to the high-water mark. Such efforts on the part of the programmer (including me) are obviously error-prone, however. Furthermore, there is no way to understand the accuracy of these measurements.

To rectify this problem, I have modified MLton's garbage collector to measure memory use with bounded error and relatively small effects on performance. To measure the high-water mark within a fraction E of the true value, I restrict the amount of memory available for new allocations as follows. At the end of each collection, given a high-water mark of M bytes and L bytes of live data currently in the heap, I restrict allocation so that no more than $M * (1 + E) - L$ bytes will be allocated before the next collection. In the interim between collections (*i.e.*, between measurements) the high-water mark will be no more than $M * (1 + E)$ bytes. Since the collector will report at least M bytes, we will achieve the desired level of accuracy. This technique differs from most collector implementations, which use only the current amount of live data L in determining when to perform the next collection.

For example, suppose that during the execution of a program, we have already observed 10 MB of live data in the heap but at the present moment (just after completing a collection), there is only 5 MB of live data. If we are satisfied with knowing the high-water mark to within 20% of the true value, then we constrain the collector to allocate at most $10 * (1 + 0.2) - 5 = 7$ MB before performing another collection. Even if all of this newly allocated data was live at some point in time, the high-water mark would never exceed 12 MB. At the end of the second collection, the collector will report a high-water mark no less than 10 MB.

The two key properties of this technique are, first, that is slowly increases the threshold of available memory (yielding an accurate result) and second, that it dynamically sets the period between collections (as measured in bytes allocated) to avoid performing collections when doing so would not give any new information about the high-water mark. Continuing the example above, if the second collection found only 1 MB of live data, then there is no reason to perform another collection until at least 11 MB of new data have been allocated.

Note that this technique may also be combined with a generational garbage collector [Lieberman and Hewitt, 1983, Ungar, 1984], for example, as in MLton. In this case, limiting allocation simply means limiting the size of the nursery. As in an Appel-style generational collector [Appel, 1989], all of the remaining space may be assigned to the older generation. Thus, this technique need *not* increase the frequency of major collections. When reporting the high-water mark after a minor collection, the collector must assume that all of the data in the older generation is live.

As I report in the next chapter, this technique has enabled me to measure memory use with low overhead and predicable results without additional effort on the part of the programmer.

7.5 Summary

This chapter describes my parallel extension to MLton. This is the first extension of MLton to support execution on multiple processors or processor cores. The changes I have made to the MLton runtime system and compiler are not specific to the forms of parallelism described in this thesis and should be applicable to other parallel and concurrent extensions of MLton. I have divided the remainder of my implementation into three parts: a set of primitives for managing parallel tasks, functions that enable the programmer to write parallel programs, and the implementation of three scheduling policies. This architecture enables new parallel features and new scheduling policies to be easily added to the implementation. I have also added a novel mechanism for measuring the high-water mark of memory use with a bounded amount of error and relatively low overhead.

7.6 Related Work

There are a number of parallel implementations of functional languages including Multilisp [Halstead, 1985], MUL-T [Kranz et al., 1989, Mohr et al., 1990], ID Arvind et al. 1989, pH [Aditya et al., 1995]), and Glasgow Parallel Haskell [Trinder et al., 1998]. All of the these implementations include a single (possibly unspecified) scheduling policy.

Morrisett and Tolmach [1993] describe a portable extension of Standard ML of New Jersey [Appel and MacQueen, 1991] that supports multi-processor execution. They define primitives that implement processor-specific state and mutual exclusion and use them to implement a thread library and a version of Concurrent ML [Reppy, 1999]. Though the focus in that work was on portability, building portable multi-processor runtimes has become more straightforward since the introduction of standards such as POSIX and pthreads. Other Standard ML implementations that support multi-processor execution include MLj [Benton et al., 1998] and SML.NET [Benton et al., 2004], which are both built on top of virtual machines.

Many of the work-stealing techniques developed in Cilk [Frigo et al., 1998] have been used elsewhere, including JCilk [Danaher et al., 2006], X10 [Agarwal et al., 2007], Manticore [Fluet et al., 2008a], and Intel's Threading Building Blocks [Intel Corporation, 2009]. Narlikar and Blelloch [1999] implement a hybrid depth-first and work-stealing scheduler.

The scheduling policy is typically not as important in concurrent languages such as Concurrent ML [Reppy, 1999] and JoCaml [Conchon and Fessant, 1999] since there is typically a

smaller number of threads and the scheduling processes is partially managed by the programmer.

Scheduling and Concurrency Frameworks. Fluet et al. [2008a] make scheduling policies explicit using an intermediate language and support nested scheduling policies. These include a round-robin (breadth-first) scheduler, a gang scheduler supposed data-parallel arrays, and a work-stealing scheduler. My implementation of fork follows a similar pattern as is described in that work, though as a library instead of a translation. Fluet et al. also support task cancellation and speculation.

Li et al. [2007] undertake a similar effort for building concurrency libraries. They give a small set of primitives and define the behavior of these primitives using an operational semantics. Among other programs, they develop round-robin and work-stealing schedulers using these primitives.

Sulzmann et al. [2008] compare the performance of different primitives for implementing a concurrent linked list. This comparison should be useful in extending and improving the performance of my implementation.

Chapter 8

Evaluation

The empirical evaluation in this chapter serves to verify the following hypotheses about the scheduling and profiling work described in earlier chapters: the choice of scheduling policy has significant effects on performance and reasoning about performance at the level of abstraction provided by cost graphs offers accurate predictions about the performance of a multi-processor implementation.

I will show that the cost semantics is a good model of performance for the implementation described in the previous chapter. However, a skeptical reader might suppose that I designed that implementation solely to satisfy my model. To assuage that concern, I will also provide some measurements of the scalability of my implementation. Not only do measurements of memory use and deviations agree predictions made in previous chapters, but this implementation uses additional processors to make programs run faster.

I performed my experiments on a four-way dual-core x86-64 machine with 32 GiBs[1] of physical RAM running version 2.6.21 of the GNU/Linux kernel. Each of the four processor chips is a 3.4 GHz Intel Xeon, and together they provide eight independent execution units. In the remainder of this section, I will refer to each execution unit as a processor. As noted in the previous chapter, I have not implemented a parallel garbage collector. As such, the measurements of execution time and scalability do not include the time used by the collector. Below, I will briefly describe the applications used in my empirical evaluation. The source code for these applications is found in Appendix A.

Matrix Multiplication. Dense matrix multiplication is a classic example of a parallel application. When measuring scalability, I use a blocked version where each matrix is represented as a tree. Each internal node has four children, and each leaf node contains a 24×24 element block. This structure enables a straightforward recursive implementation using the fork-join primitive described in the previous chapter. When measuring memory use, I use a simpler version similar to the one presented at the beginning of Chapter 2.

Sorting. I have implemented several sorting algorithms including mergesort, quicksort, insertion sort, and selection sort. I use mergesort and a representation based on trees (with selection

[1] One GiB = 1024^3 bytes.

Table 8.1 Lines of Code in Benchmarks. This table shows the size of each of the benchmarks used in this dissertation. All benchmarks are written in Standard ML. This does not include library routines such as the implementation of parallel vectors. Note that there are two versions of quicksort and mergesort: one based on nested parallelism and one using futures.

name	lines of code
matrix multiply	165
mergesort	132
quicksort	29
selection sort	41
insertion sort	13
quickhull	68
barnes-hut	379
fibonacci	9
quicksort (futures)	88
mergesort (futures)	43
pipeline	24

sort for small sets of elements) for the scalability measurements. For the memory use measurements, I use quicksort and a representation based on lists (with insertion sort). While this implementation is not well-suited to fork-join parallelism, I emphasize that it is important to understand the performance of all programs, not just the "right" programs. Indeed, it is only through understanding the performance of these programs that they are shown to be poor implementation choices. I also consider an implementation of quicksort using futures as it appears in the work of Halstead [1985] and a version of mergesort that uses futures [Blelloch and Reid-Miller, 1997].

Convex Hull. The convex hull of a set of points is the subset of those points that forms a convex polygon enclosing all the remaining points. The quickhull algorithm [Bykat, 1978] first finds the two most extreme points along a given axis. (These two points must lie on the hull.) It draws a line connecting those two points and partitions the remaining points according to which side of that line they fall on. These two partitions are processed recursively and the results are merged together. In my experiments, I use a two dimensional version of this problem and the worst-case input, a set of points distributed uniformly around a circle.

n-Body Simulation. This benchmark computes the gravitational forces on a set of bodies (*e.g.*, stars in a galaxy) based on the effects of each of the other bodies. It uses the Barnes-Hut approximation [Barnes and Hut, 1986], a method based on the fact that the gravitational force of a group of distant bodies can be approximated by the effect of a point mass. The set of bodies is maintained using an octree, a spatial data structure where each node collects together all the bodies found in a given cubic volume of space. An internal node has up to eight children corresponding to the eight octants of its cube. Each node maintains the position and size of the center-of-mass of bodies found within its volume. Thus, when computing the effect on a given body, only nodes for nearby portions of space must be visited; traversal may be cutoff for distant groups. My im-

plementation computes forces of these groups using the quadrupole moment. This is the largest benchmark used in this thesis and is comprised of more than 400 lines of SML, not including the 3-vector library used throughout.

Fibonacci. It has become traditional (since the work of Frigo et al. [1998]) to use a naïve computation of the Fibonacci sequence as a benchmark measuring of the time overhead of the scheduler. Each recursive call is computed in parallel and no memoization or other optimizations are performed. As there is almost no actual computation in this application, nearly all the execution time can be attributed to the scheduler.

Pipelining. As a test of parallel futures, I have constructed a benchmark that enables me to experiment with pipelines of various lengths. The benchmark is a generalization of the producer-consumer example in Chapter 6 where each intermediate stage acts both as a consumer and a producer, transforming one stream of values into another stream. In the experiments described below, each element of each stream is a matrix, and each stage consists of a matrix multiplication. These multiplications in turn use nested parallelism to express further opportunities for parallel evaluation. The benchmark is meant to simulate many media processing pipelines, were several transformations are applied in sequence to a stream of data. The parallel structure of this benchmark is also representative of many dynamic-programming examples.

8.1 Scalability

Scalability measures the change in execution time for a fixed program and input with respect to the amount of parallelism available in the hardware platform. In other words, as we add more processors, scalability tells us how much faster programs will run. The goal of measuring scalability is to understand what kinds of algorithms can be made faster through parallel programming and how many processors these programs can effectively use.

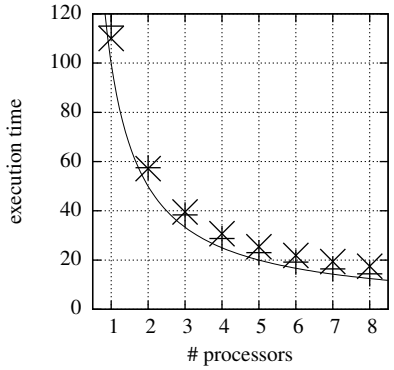
8.1.1 Plotting Scalability

All of the data reported in this section are derived from measurements of execution time. However, as in previous chapters, I am interested in finding ways of presenting these data that convey information visually.

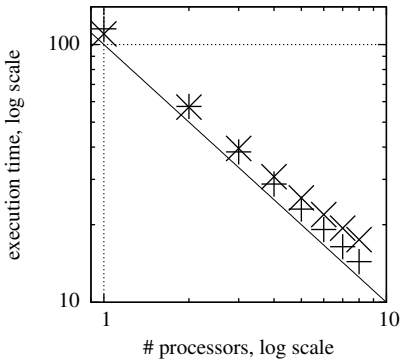
Figure 8.1 shows five different ways of presenting the same data. In each plot, two series of manufactured data are shown, corresponding to two different executions, for example using different scheduling policies. One series is illustrated using $\times$ and the other using $+$. Perfect scalability (or ideal speed-up) is shown using a solid line. Perfect scalability assumes that all the work performed by the sequential implementation is divided equally among the processors with no overhead.

The first plot (a) shows the execution time as a function of the number of processors. This is the most direct presentation of these data. This form of plot is perfect for answering questions such as, "how long will this algorithm take on seven processors?" However, it more is difficult to use this form of plot to ascertain trends in the data or to make comparisons between the two

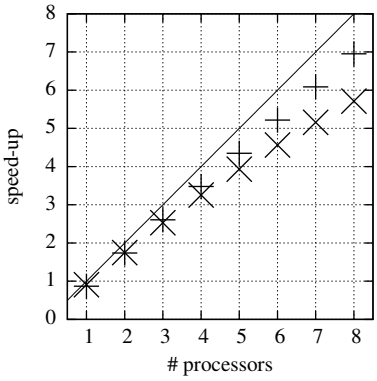
Figure 8.1 Measuring Scalability. These plots show the same manufactured data plotted five different ways. In each plot there are two series, denoted $+$ and $\times$. In each case, the solid line represents the perfect scalability.



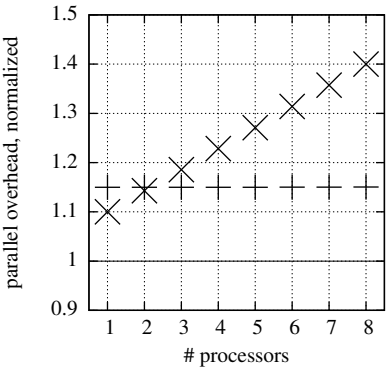
(a) time vs. # processors



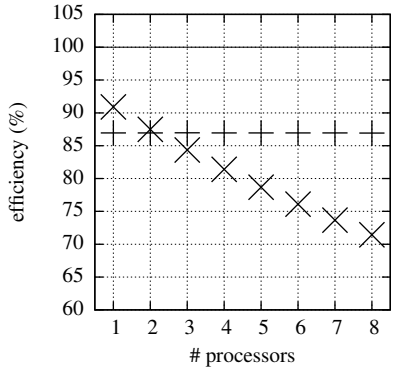
(b) time vs. # processors (log-log scale)



(c) speed-up vs. # processors



(d) overhead vs. # processors



(e) efficiency vs. # processors

series. In this case, we can see that the series plotted as $+$ performs slightly better for larger numbers of processors, but it is difficult to see how much better.

The second plot (b) shows these same quantities but plotted on a log-log scale. In terms of understanding trends in the data, this plot is an improvement over the previous one: it is now clear that the $+$ series consistently outperforms the $\times$ series and that the size of the gap between them is increasing. Unfortunately, in terms of efficient use of the space on the page, this plot is a poor choice: it uses only a small fraction of area available.

The third plot (c) has a similar look. This is a traditional speed-up plot that frequently appears in the literature. Given a sequential execution time T^* and an execution time of T_P on P processors, speed-up is defined as T^*/T_P. In such a plot, the ideal (or linear) speed-up is a line whose slope is unity, indicating that an execution with P processors is P times faster than the sequential version.[2] In this plot, it can be observed that the gap between the $+$ series and the ideal is increasing and that the performance gap between the two series is also increasing. Like the previous plot, this one also wastes considerable amounts of space: the upper-left half of the plot rarely contains any points. With these data, the lower-right corner is also largely wasted: while this plot helps to differentiate the behavior with many processors, it does so at the cost of sacrificing the area given to small numbers of processors. While it is clear in part (a) that the $\times$ series performs better with only one processor, it is difficult to see this in either parts (b) or (c).

The next plot (d) attempts to correct for this problem by normalizing with respect to the sequential execution time: this plot shows overhead, defined as $(T_P \cdot P)/T^*$. It makes a much better use of space than any of the previous plots, and the relative performance the two series can be easily compared at any number of processors. For example, it is clear from this plot that there is a 10% overhead for the $\times$ series using one processor, compared to 15% for the $+$ series. The values in this plot indicate the cost of adding more processors. We can see from the plot that the overhead of adding new processors in the $+$ series is a constant, while the overhead in the $\times$ seems to be a linear function of the number of processors.

The final plot (e) shows the **efficiency**, defined as $100 \cdot T^*/(T_p \cdot P)$. This is the reciprocal of the value plotted in part (d). Like the previous plot, it is easy to compare the performance of the two series for any number of processors, and this style also makes use of a large portion of the available plot area. Efficiency can be interpreted as the percentage of cycles that are being used to run the application rather than performing some form of communication or synchronization. Thus 100% efficiency is the ideal. For example, the series shown as $+$ is 87% efficient with eight processors, meaning that 87% of the cycles of each of the processor are being used to run the application.

I chose to plot efficiency in my results as it effectively conveys trends in how a program and scheduling policy use additional processors. It is easy to see how close a given point is to 100% efficiency. Moreover, it also makes clear an important difference between these two series: in the case of the $+$ series, efficiency remains constant with the addition of more processors, while in the case of the $\times$ series, it decreases with each additional processor. This trend is much more difficult to read in the execution time (a,b) or speed-up (c) plots. Like speed-up, however,

[2] I believe it is critical that speed-up plots use the same scale for both axes so that the ideal forms a 45° angle with the page edges. Though there no need for execution time or efficiency to be plotted with this constraint, I have plotted them using the same total area for the sake of consistency.

efficiency also has the advantage that larger is better, satisfying many readers' expectations that advantageous trends should go "up and to the right."

8.1.2 Results

I chose benchmarks for this set of measurements by looking for applications with a reasonable potential for parallel execution and not based on interesting memory use patterns. In each case the sequential execution time is the execution time on a single processor without any of the support required for multi-core execution. In my implementation, support for multi-core execution is controlled using a compile-time constant. Though I performed some experiments with variations in task granularity, I report on only a single granularity for each example. In each case, this granularity was chosen to limit the overhead when run on a single processor to an acceptable amount (typically no more than 10-20%).

Figure 8.2(a) shows overhead for a blocked version of matrix multiplication. Part (b) shows overhead for parallel mergesort on uniformly randomly distributed input. Part (c) shows the overhead of quickhull for points distributed uniformly a circle. Part (d) shows the overhead for the Barnes-Hut simulation with points distributed uniformly randomly. Though all of these applications show some potential for faster execution, some scale better than others. In addition, there are some clear differences in the performance trends for difference scheduling policies. For example, in the matrix multiplication example, the initial cost of adding parallelism is greater for the work-stealing scheduler when compared to the depth-first scheduler (leading to lower efficiency with one processor). This cost levels off as more processors are added: the work-stealing is more efficient for five or more processors. In this cases of mergesort and quickhull, there does not seem to be enough parallelism to (efficiently) leverage all of the cores available on this hardware. In all cases, however, using more cores leads to lower efficiency per core, most likely because of contention on the memory bus.

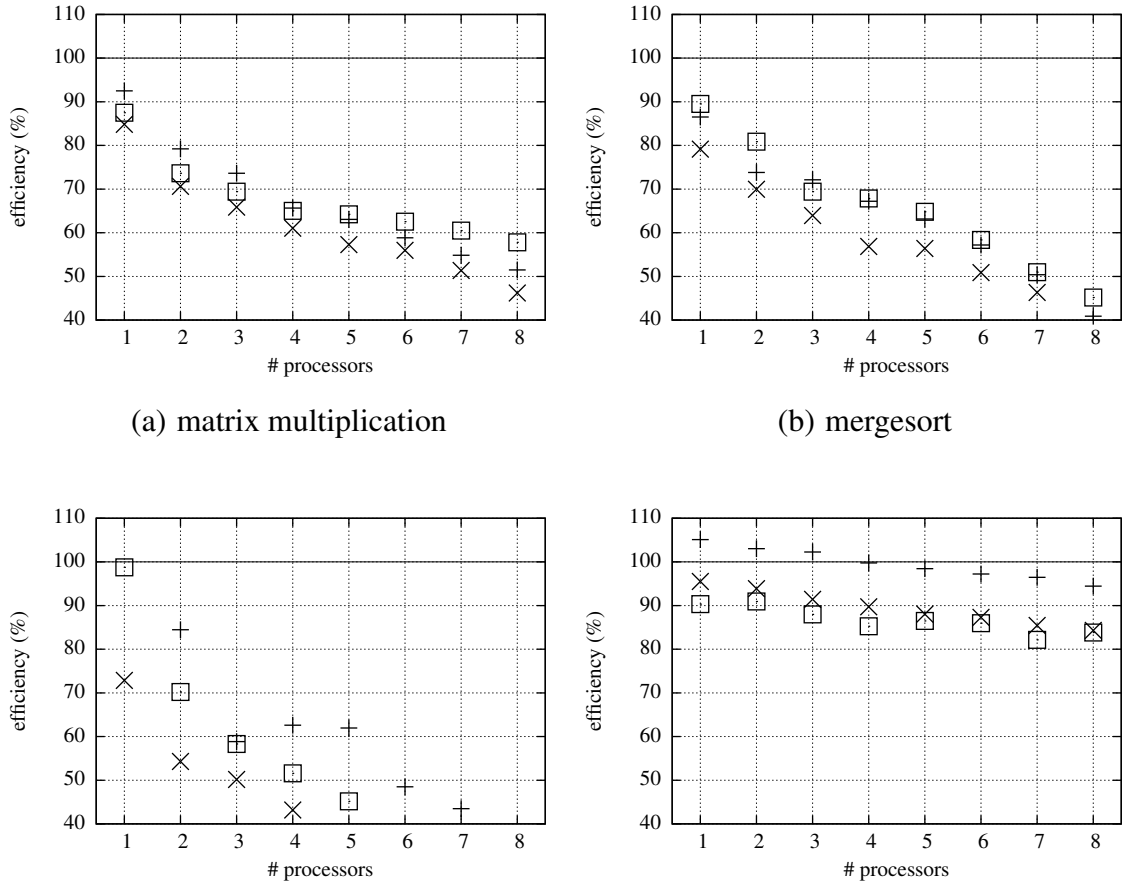
8.2 Memory Use

In this section, I present measurements of memory use for four applications. These measurements serve to demonstrate that the choice of scheduling policy has significant effects on memory use and that these effects are predicted by my semantic profiler. I also use these measurements to show how I found a memory leak introduced by one of the optimizations in MLton.

8.2.1 Measuring Memory Use

Measuring memory use precisely can be expensive. Using the technique described in the previous chapter, however, I can measure the high-water mark of the memory required to run an application within a fixed bound. Here, I compare the quality and cost of these measurements with those derived from a hand-instrumented version. In Figure 8.3(a), I show memory use for blocked matrix multiplication measured four different ways. All these data use the breadth-first scheduler and one processor. The first series (○) shows the measurements obtained when I added

Figure 8.2 Parallel Efficiency. These plots show parallel efficiency, defined as $100 \cdot T^*/(T_p \cdot P)$ with sequential execution time T^* and parallel execution time T_P on P processors. Equivalently, efficiency is equal to the speed-up divided by the number of processors, expressed as a percentage. 100% efficiency is the ideal. Three scheduling policies are shown: depth-first ($+$), breadth-first ($\times$), and work-stealing ($\boxdot$). These data do not include time consumed by garbage collection.



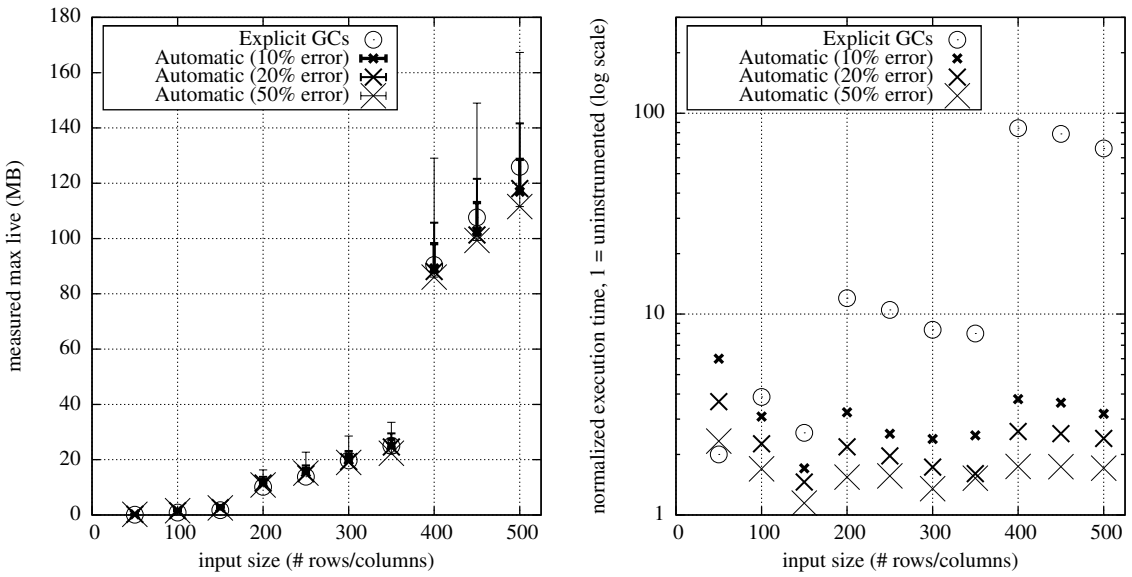
(a) matrix multiplication

(b) mergesort

(c) quickhull

(d) Barnes-Hut simulation

Figure 8.3 Measuring Memory Use. Four different measurements (a) and the cost of obtaining these measurements (b). Execution times are normalized to an uninstrumented version and shown on a logarithmic scale.



(a) measured memory use

(b) instrumentation overhead

explicit garbage collections to the source program. The other three series show the results using the restricted allocation technique with bounds of 10%, 20%, and 50%, respectively. These bounds are shown with error bars, but only positive errors are shown (as the true high-water mark cannot be smaller than the reported value). The reported values appear at the bottom of the indicated ranges.

I take the measurements derived from the explicit garbage collection to be the most accurate measurement of memory use. (In nearly every case, this technique reported the greatest values, and these values were always within the bounds of the other measurements.) The figure shows that the restricted allocation measurements are much more accurate than we might expect. For example, the 50% bound seems to be overly conservative: the actual measurements with this bound are within 15% of the measurement derived using explicit garbage collections. The jump in the memory use measurements between inputs of size 350 and 400 is due to the blocked representation and an additional level of depth in the trees used to represent the input and output matrices.

In addition to requiring less knowledge on the part of the programmer and yielding measurements with a bounded error, this technique requires less time to perform the same measurements. Figure 8.3(b) shows the execution time of these four instrumented versions. Values are normalized to the execution time of an uninstrumented version and shown on a logarithmic scale. Again the jumps in execution time are a result of the blocked implementation.

While the measurements with 10%, 20%, and 50% consistently require approximately $2\times$, $3\times$, and $4\times$ the execution time (respectively), the execution time with explicit measurements

may be 10s or a 100 times longer. This makes measurements using explicit collections for larger inputs prohibitively expensive. While it would be possible to build a more sophisticated instrumented version, such sophistication introduces the possibility of additional mistakes on the part of the programmer and still offers no bounds on the errors associated with the measurements.

8.2.2 Results

The results for my memory use experiments are shown in Figure 8.4 and described below.

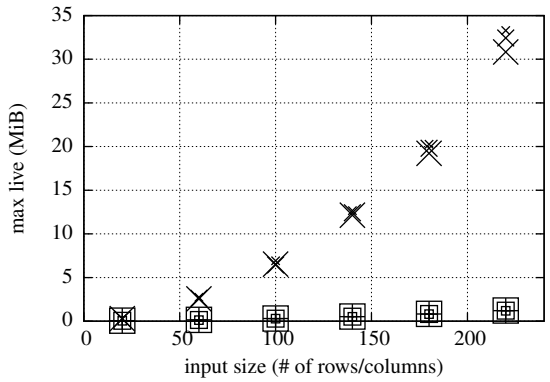
Matrix Multiplication. The simulations described in Chapter 4 (recall Figure 4.1) predict that the breadth-first scheduling policy uses asymptotically more memory than the depth-first policy. A similar analysis predicts that breadth-first will also require more memory than a work-stealing scheduler. Both these predictions are confirmed by the memory use in my implementation, as plotted in Figure 8.4(a).

Quicksort. Figure 8.4(b) shows the memory use of a functional implementation of quicksort where data are represented as binary trees with lists of elements at the leaves. At each step, the root element of the tree is chosen as the pivot. This plot shows the behavior for the worst-case input: the root elements appear in reverse order. While we would expect quicksort to take time quadratic in the size of the input in this case, it is perhaps surprising that it also requires quadratic space under some scheduling policies. This behavior is also predicted by the cost semantics. The plot in Figure 8.5(a) was generated using my semantic profiler and shows the consumers of memory for the depth-first policy at the high-water mark. To further understand why, consider the cost graphs that appear in Figure 8.5(b). These graphs depict the execution of this implementation of quicksort on the worst-case input and using the depth-first scheduling policy. The heavy edges are roots representing the input tree passed to each recursive call. As the scheduler explores this graph in a depth-first manner, it leaves a live reference to this input after each partition. As each of these trees contains up to n elements and there are n recursive calls, this requires $\Omega(n^2)$ memory cells.

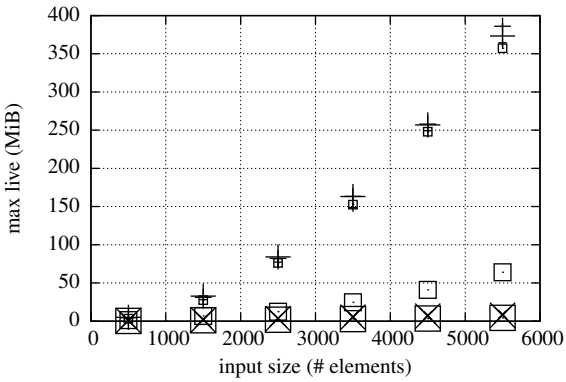
I chose this example because it was a case where my semantics predicts that using the breadth-first scheduler will require less memory than using the depth-first scheduler. While this is evident in the plot shown in the figure, my initial results for this benchmark were substantially different. In particular, my initial results showed both the depth-first and breadth-first scheduling policies using quadratic memory. Some investigation revealed that the problem was not in my semantics, profiler, or implementation, but in an existing optimization in MLton, the reference flattening optimization described in the previous chapter. As noted above, this leak has since been fixed.

Quickhull. This application computes the convex hull in two dimensions using the quickhull algorithm. I again show results for the worst-case input: the input points are arranged in a circle and so every point in the input is also in the hull. Figure 8.4(c) shows the high-water mark of memory use, which again matches my cost semantics-based predictions.

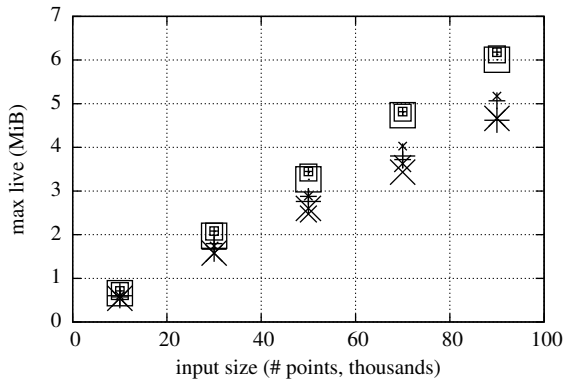
Figure 8.4 Memory Use. Each plot shows the high-water mark of memory use for one of four applications. I tested three scheduling policies, depth-first (+), breadth-first (×), and work-stealing (⊡), with one, two, and four processors. Larger symbols indicate that more processors were used. Different scheduling policies yield dramatically different performance, as discussed in the text.



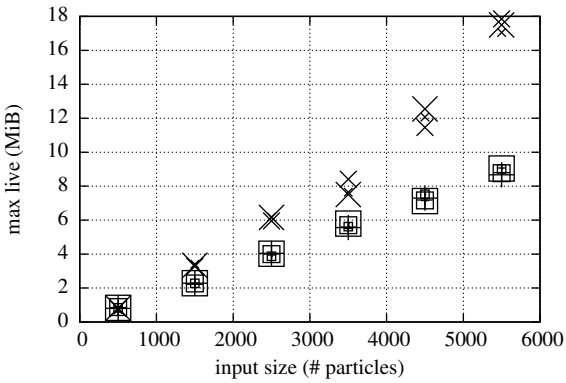
(a) matrix multiplication



(b) quicksort

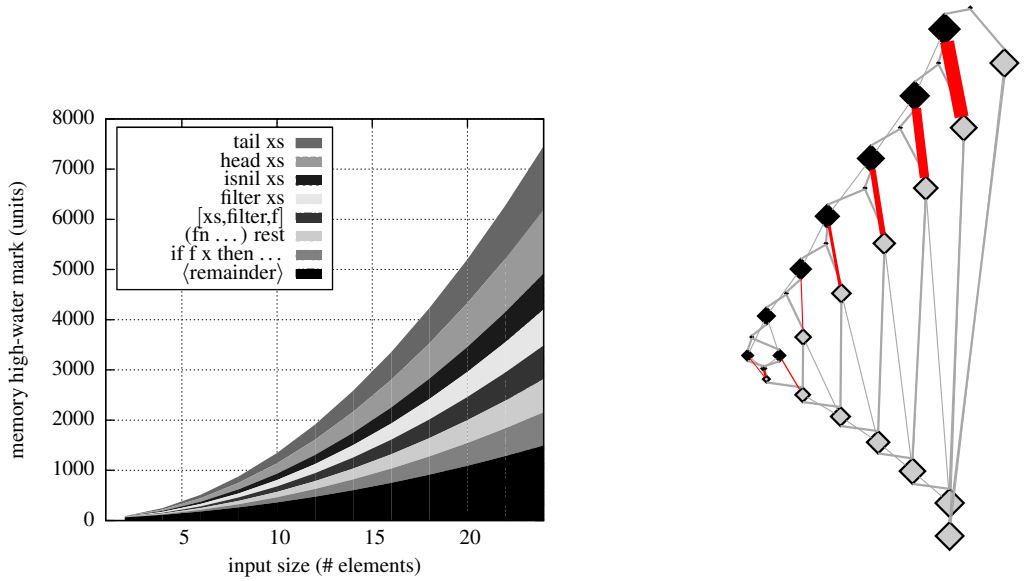


(c) quickhull



(d) Barnes-Hut simulation

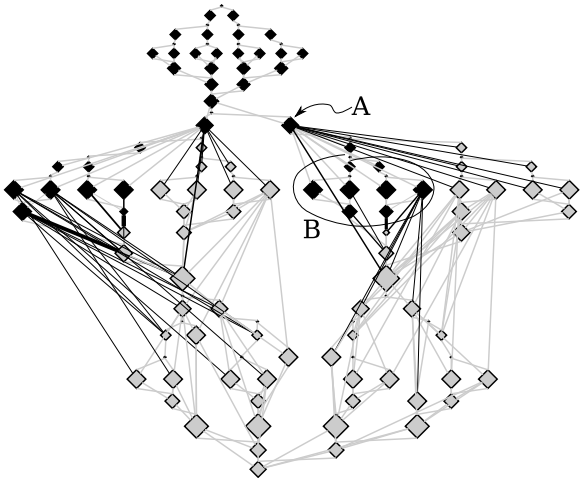
Figure 8.5 Profiling Results for Quicksort.



(a) memory use

(b) cost graphs

Figure 8.6 Profiling Results for Quickhull. The nodes labeled “A” and “B” are discussed in the text.



Quickhull offers more parallelism than the previous example, but this parallelism is still more constrained than that found in matrix multiplication. The algorithm proceeds by partitioning the point set by dividing the plane in half (in parallel) and then recursively processing each half (in parallel). Between these two phases there is a synchronization. This was shown through the widening and narrowing of the graphs in Figure 4.4 and again in Figure 8.6.

As previously described, nodes are filled to illustrate one point in the execution of the work-stealing scheduling policy with two processors. In this case, the work-stealing policy performs more poorly than either of the other two policies because it starts, but does not finish, computing these partitions. The point labeled "A" in Figure 8.6 represents the code that allocates one set of points. The black lines extending to the right of this point indicate part of the program that will compute one half of a partition of these nodes. The circled nodes labeled "B" also compute half a partition, but have already completed their work and have allocated the result. At this point in time, the program is holding onto the entire original set plus half of the resulting partition. The same pattern appears for each processor. Neither of the other two scheduling policies exhibit this behavior.

Barnes-Hut. Figure 8.4(d) shows memory use for my implementation of the Barnes-Hut approximation. This algorithm and representation are easily adapted to a parallel setting: not only can the forces on each particle be computed in parallel, but the individual components of this force can also be computed in parallel.

Due to its complexity (*e.g.*, use of modules, pattern matching), I have not implemented a version of this application that is compatible with my profiling tools. Applying the cost semantics informally, however, we expect the program to generate very wide cost graphs. Like the matrix multiplication example, the breadth-first scheduling policy performs poorly due to the large amount of available parallelism and the size of the intermediate results. Though its performance is not as bad as in the multiplication example, it is still significantly worse than the other two policies.

8.3 Scheduler Overhead

Though I continue to pursue work in improving the efficiency of my implementation, it is interesting to understand current performance bottlenecks and how my choice to implement scheduling policies in ML has affected performance. I have focused most of my efforts to improve the performance on the work-stealing scheduler and only for the case of nested parallelism. Of the three varieties of scheduling policies I implemented, the work-stealing schedulers have the lowest overhead. This seems largely due to the fact that each processor in a work-stealing scheduler maintains its own work queue and only interacts with other processors when a steal occurs.

In previous sections, I have selected or designed applications with relatively coarse task granularity: I was more interested in whether a particular application offered enough parallelism or how the scheduling policy affected memory use. In this section, I choose a benchmark with extremely fine granularity: a naïve implementation of the Fibonacci sequence. In the recursive case, each parallel task consists only of spawning a parallel task, making a function call, and adding the two resulting integers. The base case simply returns a integer.

Table 8.2 Scheduler Overhead. The overhead of the work-stealing scheduler is shown as a breakdown of the performance of the Fibonacci micro-benchmark run with a single processor. Some of the cost of adding elements to the queue (*) may be attributed to building the closure.

	% of total time	% of total allocation
hardware memory barriers	8.3	
foreign function calls	11.2	
core of Dekker's algorithm	15.1	
build closure	5.6	35.4
add to work queue	44.8*	43.6*
allocate synchronization variable	4.1	21.6
exception handling	3.9	
yield	3.4	
serial execution	3.6	<0.001

Table 8.2 shows a breakdown of the performance of this micro-benchmark running on a single processor. These data were gathered by progressively removing more and more of the implementation and measuring the execution time after each omission. As there was only a single processor, omitting various parts of the implementation had no effect on the parallel execution. Below, I explain each line of the table.

Memory barriers are necessary to ensure proper synchronization among processors. Omitting these instructions reduces the execution time by 8.3%. As neither MLton nor SML provide any way to use such instructions, using these barriers requires making a call across the foreign function interface into C, corresponding to 11.2% of the total execution time.

Dekker's algorithm [Dijkstra, 2002] is a mutual exclusion algorithm for two processors that is often used as part of a work-stealing implementation; it is very fast when there is little contention. To measure the cost of this algorithm without the overhead of calling into C, I also implemented a version in SML.

The next two lines of the table correspond to building a data structure representing the spawned task and adding it to the queue. It is difficult to tease apart these costs as MLton generates very different code if that closure is not added to a globally accessible data structure. These two lines should possibly be considered as one.

My implementation maintains the result of a task separately from that task itself. As described in the previous chapter, a synchronization variable is allocated to hold this result. Exception handling is also described in the previous chapter and ensures that exceptions raised in one parallel task will be properly propagated to the parent task.

Yields are added in the implementation of fork to support proper implementations of the depth- and breadth-first scheduling policies. In the work-stealing schedule, the implementation of the function consists solely (after inlining) of a foreign function call to get the current processor number. The result of this call is not used, but MLton makes no assumptions about the effects of foreign function calls and so retains the call.

Finally, the last line corresponds to the serial execution of the Fibonacci computation.

Comparison to Cilk. In summary, my implementation of fork using a work-stealing scheduler is about 28 times slower than a function call. Frigo et al. [1998] report that the equivalent function in Cilk is between 2 and 6 times slower than a function call, depending on the architecture. These differences can be explained as follows. First, Cilk (like C) does not support exceptions, and therefore some of the additional overhead in my implementation can be attributed to the fact that it offers more functionality. Also, some parts of my implementation are meant to support more general forms of parallelism and might be further optimized for the nested-parallel case. However, these are relatively small parts of the overhead. A larger part of the overhead is due to the lack of atomic memory operations in ML. This could be fixed relatively easily with a small extension to MLton.

The most significant part of the overhead, however, arises from the allocation of the data structures used internally by the scheduling infrastructure, independently of the management of the task queues by the work-stealing scheduler. I should note that the overhead here is not due to the cost of garbage collecting these data structures, but simply allocating them. This presents an interesting challenge for future work that I will touch upon in the next chapter.

Depth- And Breadth-First Scheduling Policies. Despite some attempts to improve its performance, my parallel depth-first schedule still incurs between 3 and 5 times the overhead of the work-stealing scheduler, meaning that my implementation of fork using a depth-first scheduler is more than 100 times as expensive as a function call. This is largely due to allocation and contention at the head of the shared queue.

As I have implemented the breadth-first scheduling policy as part of the validation of my semantics, and not as a practical scheduler for parallel programs, I have not performed any measurements of the overhead of this scheduler.

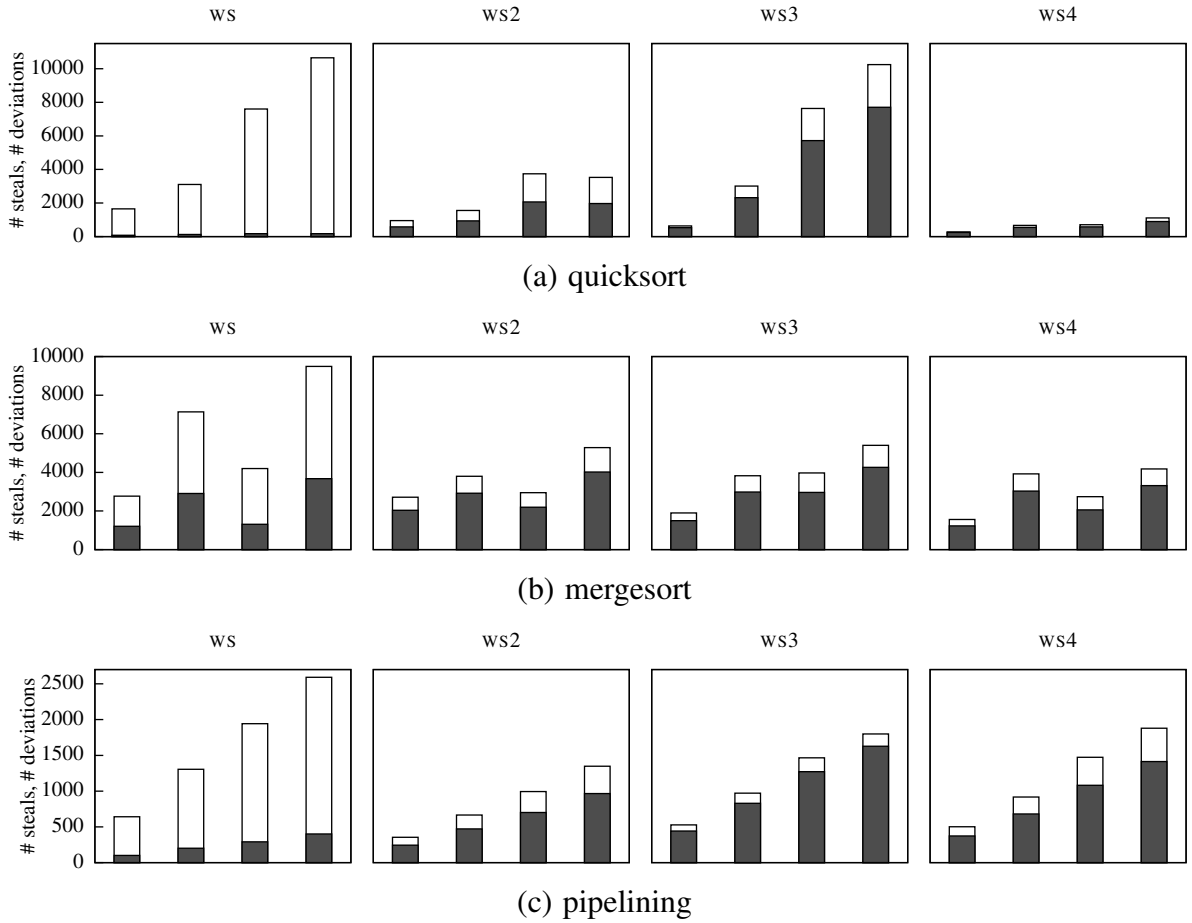
8.4 Deviations

To better understand when deviations occur in programs that use futures, I consider three benchmarks: an implementation quicksort that sorts streams of values [Halstead, 1985], an implementation of mergesort that uses trees but where merges may overlap [Blelloch and Reid-Miller, 1997], and a synthetic pipelining example where streams of matrices are transformed by matrix multiplication.

As deviations only occur during interactions with the scheduler, they can be measured by instrumenting the implementation of the parallel primitives. In my implementation, reading a synchronization variable also returns a boolean indicating whether the value was already available or if the current thread was suspended. In addition, the part of the parallel library used to manage tasks must also be modified to count the number of deviations.

The results for these three benchmarks are shown in Figure 8.7. For each benchmark, we report on the performance of each of the four work-stealing implementations described in Table 6.1. Each column of Figure 8.7 corresponds to one of these implementations, as labeled above each plot. In each of the twelve plots, we plot performance as a function of input size. For the sorting benchmarks, the input size is the number of elements to be sorted. For the pipelining

Figure 8.7 Deviations. These plots show steals and deviations as a function of input size. Each of the four columns shows a different work-stealing scheduler as described in Table 6.1. The height of each bar indicates the total number of deviations. The fraction that can be attributed to steals are shown as solid boxes.



benchmark, the input size is the number of elements generated by the producer stage. For each input size, we plot two metrics: the number of steals and the number of deviations.

The height of each bar indicates the number of deviations. The fraction of these that can be attributed to steals are shown as solid boxes. For example, the data point on the bottom right (smallest input, WS scheduler, pipelining benchmark) indicates that there were roughly 600 total deviations and 100 steals.

I draw two conclusions from these data. First, the number of deviations is often dramatically larger than the number of steals. In some cases, the number of deviations can increase asymptotically faster than the number of steals. The disparity between steals and deviations is particularly pronounced in the WS scheduler (the implementation based on the work of Arora et al. [1998]). This is consistent with the fact that this scheduler is designed to avoid steals but not deviations.

Second, the choice among the variations of work stealing described in Chapter 6 has a significant effect on performance. From this limited set of benchmarks, WS2 and WS4 seem like the best choices, but more evaluation is required before any stronger conclusions are drawn.

Recall that WS3 suspends entire deques but does not steal entire deques. It incurs many deviations (in the form of steals) because once a deque is suspended, every task in that deque must be stolen individually. This also means that many ready tasks appear in deques that are not associated with any processor.

The WS scheduler incurs many deviations because it continues to work in the same part of the program after a task suspends on a future and because it adds resumed tasks to the current processor's deque. In this implementation, the number of deques is always the same as the number of processors. In some sense, WS tries to maintain the set of ready tasks too compactly.

Both WS2 and WS4 implement some sort of compromise. Notably, in both cases the number of deques may be greater than the number of processors, but both attempt to limit the number of tasks that appear in those deques that are not associated with any processor. Given the results in Chapter 6, I expect these implementations to incur fewer cache misses and less overhead, but I leave these measurements for future work.

8.5 Summary

These empirical results show that the cost semantics and a profiler based on it provide accurate predictions of performance despite the many transformations used by the MLton compiler. They also show that the choice of scheduling policy can have dramatic effects on performance; I have considered memory use in the case of three schedulers for nested parallelism and the number of deviations for variations of work stealing that support parallel futures.

I have also described some results showing that the technique for measuring the high-water mark of memory use described in the previous chapter offers an efficient and reliable method of doing so. Finally, with the help of my profiling tools, I found a memory leak in one of MLton's optimizations.

As an aside, these results also show that performance of even well-optimized code can be improved with parallelism. I have implemented three scheduling policies in a high-level language, and through the use of a powerful optimizer, have achieved reasonably good performance. A breakdown of the scheduler overhead suggests some opportunities to improve performance.

Chapter 9

Conclusion

In the final chapter of this dissertation, I will describe some avenues for future work, summarize my results, and provide some concluding remarks.

9.1 Future Work

9.1.1 Cost Semantics

The cost semantics I have described gives a precise account of memory use using heap edges. This accurately captures the effects of different scheduling policies on memory use while abstracting away many details of the compiler and runtime. It does not, however, abstract away any details of the program itself. As I have discussed, previous work has shown how the computation graph can be summarized using work and depth. These two values discard nearly all the information about a program, but they offer succinct metrics for comparing different algorithms. It remains to be seen whether similar properties exist for heap graphs. This line of future work could be described as a form of abstract interpretation inspired by heap graphs.

We might also consider reasoning more abstractly about cost graphs. For example, when costs are simply integer values, one can solve recurrence relations to define costs as functions of input size. Performing a similar analysis seems to require a richer algebraic theory of graphs.

9.1.2 Profiling

My profiling framework also presents a number of areas for future work.

Larger Programs. In my work, I have used my profiler on only small programs and small inputs, in large part because of its performance. To reason about the memory use of a program, I run that program using an interpreter, simulate a scheduling policy, and then trace the heap graph much like a garbage collector would. For the sake of simplicity, I have implemented each of these aspects as a separate step. However, if we are only interested in a particular scheduling policy, for example, an optimized version of this process may offer much better performance.

For example, the interpreter could be specialized by essentially inlining the subsequent stages into it.

An alternative approach would be to perform the simulation of scheduling policies on distilled graphs instead of the original ones. As I have yet to prove any form of cost preservation for the rules I used to distill graphs, it is unclear what a simulation using the distilled graph would tell us. Such a theorem, however, would be proof that distillation is a reasonable abstraction of program behavior. A more efficient implementation would eagerly distill the graph as it was generated by the interpreter, rather that performing this process as a separate step.

Interactive Profiling. A second approach for profiling larger programs would be to build cost graphs lazily. This would be especially useful for interactive presentations of these graphs. One possibility is that the user is initially presented with a collapsed graph: a single node. Then user can then explore the graph by selectively expanding or collapsing nodes. Each node could be annotated with the source expression or a summary of the memory use associated with that node.

Sampling-Based Profiling. The overhead of online profilers can often be reduced by using sampling to measure an approximation of the desired value. My method of measuring memory use in MLton (Chapter 7) is in example of a sampling-based method that uses the rate of allocation to determine the interval between samples. A similar method could be used to improve the performance of my simulation and profiling framework. In addition, it would be interesting to determine whether the simulation of scheduling policies might be omitted altogether: given the cost graphs and the abstract definitions of scheduling policies in Chapter 3, the profiling could determine the intermediate state of a scheduling policy (including roots and memory use) without simulating each of the steps that led up to that point.

9.1.3 Scheduling

In Chapter 6, I proved results about parsimonious work-stealing scheduling policies and offered three alternative policies that are not parsimonious. I have yet, however, to prove any results on the number of deviations incurred by these alternatives. I should further note that these alternatives are by no means exhaustive but merely meant to address what seem to be possible shortcomings in previous work.

I have not considered the performance of depth-first scheduling policies that support futures. Narlikar and Blelloch [1999] present a hybrid scheduling policy that combines both work-stealing and depth-first policies for nested-parallel programs. It would be interesting to see how such a hybrid relates to the other alternatives I have proposed.

9.1.4 Implementation

While SML was not designed as a language for implementing low-level concurrent algorithms, my implementation of scheduling policies in SML has been a pleasant experience. In particular, I have found it useful to be able to distinguish between variable binding and memory references. Despite this, my implementation suggests several opportunities for future work.

Controlling Allocation. The greatest challenge in improving the performance of my implementation of work stealing lies in controlling allocation, and to a large extent, the allocation of closures. In my implementation of nested parallelism, the problem arises from the fact that the closure that used to represent a spawned task includes many values that are also present on its parent stack. In nested parallelism, this task will never outlive its parent's stack and so this copy is unnecessary. From the point of view of the compiler, however, copying these values is the correct behavior: the child task is referenced from a globally-accessible queue and the compiler cannot determine the relative lifetimes of the parent and child tasks.

Manipulating Stacks. The sharing of values between a child task and its parents can be taken one step further by eliminating the data structure used to represent queued tasks, as in lazy task creation [Mohr et al., 1990]. This implementation strategy efficiently represents tasks, task queues, and the call stack all using a single data structure. In previous work, this was accomplished through explicit support for parallelism in the compiler and runtime.

MLton already supports limited forms of stack manipulation, allowing programmers to suspend and resume threads. It would be interesting to see if this could be extended to a form of introspection, where stack frames could be accessed as a (specialized) form of record. This could, in turn, be used to streamline the creation of new tasks. While any form of lazy task creation would still require some changes to the compiler and runtime, these changes would be consistent with other aspects of my implementation: making only general-purpose extensions to the compiler and runtime, and then using these extensions within libraries implemented in ML to build additional language features.

Atomic Updates. Another performance bottleneck in my implementation stems from the lack of atomic memory updates in SML. Adding primitives, such as atomic compare-and-swap and test-and-set, for integer types would be straightforward. However, another interesting extension would be to add similar primitives for other types, including datatypes. As an example, consider the implementation of synchronization variables shown in Figure 7.2. This code uses a lock (implemented as an integer reference) to guard a reference to a datatype, as shown in the excerpt below.

```
let
  val () = Lock.lock lk
  val Waiting readers = !v
  val () = v := Done a
  val () = Lock.unlock lk
...
```

This code can be written more simply and concisely using an atomic operation to update the reference holding the datatype itself.

```
let
  val Waiting readers = !v
in
  if compareAndSwap (v, Waiting readers, Done a) then
```

. . .

If this compareAndSwap operation were to be implemented using hardware support for atomic updates, this would not only allocate one fewer reference cell, but it would also save several memory operations. However, it raises the issue of what the programmer means by "compare." In this case, there are two instances of the expression Waiting readers. These expressions are equal according to SML's built-in (structural) definition of equality, but as we would like to use hardware support for compare-and-swap, we must use the definition of equality determined by the hardware. How to integrate such notions of equality remains a challenge, but such an extension would enable very clean implementations of many shared-memory concurrent algorithms.

Alternative Architectures. Finally, I have considered an implementation only for shared-memory multi-processors and multi-core architectures. Another important class of parallel architectures entering mainstream computing includes graphics processors and multimedia extensions of general purpose processors. Graphics processors have for years included a variety of vector operations but until recently, uses of these operations have been limited to rendering graphics. The current generation of these processors offers much more flexibility, and along with multimedia extensions of general purpose processors, they make an attractive target for the forms of vector parallelism described in Chapter 5.

9.2 Summary and Remarks

Computer science is largely the study of abstraction: the study of models that describe programs and computers and how those models relate to each other and to computers themselves. In this thesis, I have defined a model for parallel programs, in the form of a cost semantics, that describes where parallel evaluation can occur and how memory is used. This model abstracts away from many aspects of a language implementation, including the compiler and garbage collector, but still provides enough detail to reason about different scheduling policies and compare these policies to the sequential implementation. In particular, it enables programmers to establish expectations about program performance using only the source program and a high-level description of a scheduling policy. It is critical that, when considering new parallel language features, language designers and implementers consider the cost model that programmers will use when developing programs.

This model forms a basis for a suite of profiling tools. Unlike other profiling tools, these tools are based not on an instrumented version of the compiler or runtime system but directly on a cost semantics instead. While this abstraction implies some loss of precision, it means that profiling results are obtained independently of any particular language implementation.

I have considered three forms of implementation of a parallel language and shown how these implementations are constrained by my cost semantics. First, I have given a high-level, semantic presentation of online scheduling and defined several of the scheduling policies that I later considered in more detail. Second, I have shown how flattened nested parallelism, a compiler transformation, can be modeled using a labeled version of one of these semantics. Third, and finally, I have described my parallel extension of MLton and given an example of where a change

to that implementation was necessary to adhere to the specification provided by my cost semantics. This was bug was not discovered over the course of many years of development with MLton. Furthermore, this behavior could not be considered incorrect until there was a specification, in the form of my cost semantics, that established the correct behavior.

My extension of MLton demonstrates that programmers can improve performance not only by adding threads, communication, and synchronization, but merely by expressing opportunities for parallelism. In doing so, the programmer makes *fewer* assumptions about how a program will be evaluated: for example, in Standard ML, elements of a tuple are always evaluated from left to right, but in my extension, they may be evaluated in an arbitrary order. This stands in contrast to the notion that programmers must make *more* assumptions about the language implementation and underlying architecture to improve performance.

There is a fallacy among programmers that better performance is achieved by writing programs in "low-level" languages that are "closer to the metal." In fact, better (and more predictable) performance is achieved by using languages with well-defined cost models. It happens that many programmers have a reasonably accurate cost model for microprocessors and that this model can also be used for many imperative languages including C and Java. Unfortunately, this model does not easily extend to parallel architectures without requiring programmers to also understand (and implement) load balancing, communication, and synchronization among tasks.

In fact, there is no contradiction in achieving good performance with high-level (or highly abstract) languages, so long as there is a well-defined cost semantics for such languages. This claim is supported not only by the work in this thesis but also by an example given in the introduction: the use of locality to predict the performance of the memory accesses. While the cost semantics in this thesis does not provide a mechanism as powerful as locality, it represents an important step in reasoning about the performance, including memory use, of parallel programs.

Models, such as a cost semantics, are what distinguishes computer *science* from computer *programming*. These models give us theories which, through the use of formal proofs and empirical tests, can be verified or refuted. These models also reveal something of human character of the study of abstraction: that science, as a human endeavor, reflects our desire to seek out simplicity and beauty in the world around us, including truths about computation.

Appendix A

Source Code

This chapter includes the source code for the benchmarks used in this dissertation as well as a library function used in one of the examples.

A.1 Library Functions

A.1.1 Parallel Array Reduction

This following function is used in the example in Chapter 2.

```
(* Reduce an array a in parallel. Assume an associative operator mul which is * used to
 * combine intermediate results. reducei is extensionally equivalent * to the following
 * expression.
 *
 * reduce mul inj uni a = Array.foldli (fn (i, v, b) => mul (v, inj (i, b))) uni a
 *
 * This implementation assumes MAX_SEQ is equal to the largest number of elements that
 * should be processed sequentially.
 *)
fun reducei mul inj uni a =
  let
    fun wrap i l =
      if l <= MAX_SEQ then
          let
            val stop = i + l
            fun loop j v = if j = stop then v
                           else loop (j + 1) (mul (v, inj (j, Array.sub (a, j))))
          in
            loop i uni
          end
      else
        let
```

```
        val l' = l div 2
      in
        f (fork (fn () ⇒ wrap i l',
                 fn () ⇒ wrap (i + l') (l − l')))
      end
  in
    wrap 0 (Array.length a)
  end
```

A.2 Matrix Multiplication

A.2.1 Unblocked Multiplication

```
structure Nonblocked : DENSE =
struct
  type v = real vector
  type m = v vector

  fun vvm _ (a, b) =
    let in
      foldInt (length a) op+ (fn i ⇒ sub (a, i) ∗ sub (b, i)) 0.0 (length a)
    end

  fun mvm maxSeq (m, a) =
    let
      val def = ~1.0
    in
      tabulate maxSeq (fn i ⇒ vvm maxSeq (sub (m, i), a))
              def (length m)
    end

  fun mmm maxSeq (m, n) =
    let
      val def = tabulate maxSeq (fn _ ⇒ ~1.0) ~1.0 0
    in
      tabulate maxSeq (fn i ⇒ mvm maxSeq (n, sub (m, i)))
              def (length m)
    end
end
```

A.2.2 Blocked Multiplication

The blocked multiplication is functorized over the implementation of multiplication that should be used on each block.

```
functor Blocked (structure D: DENSE) : DENSE =
struct
  exception Blocked of string
  type v = real vector
  type m = v vector

  (* Blocked matrix structured as tree *)
  datatype tree = Leaf of v t
                | Branch of { height : int, width : int, (* dim of UL used in unwrap *)
                              ul : tree, ur : tree, ll : tree, lr : tree }

  fun vva maxSeq (a, b) =
      tabulate maxSeq (fn i => sub (a, i) + sub (b, i)) ~1.0 (length a)

  fun mma maxSeq (Branch {ul = ul1, ur = ur1, ll = ll1, lr = lr1, height, width},
                  Branch {ul = ul2, ur = ur2, ll = ll2, lr = lr2, ...}) =
      let
        val ((ul, ur), (ll, lr)) = (mma maxSeq (ul1, ul2), mma maxSeq (ur1, ur2),
                                    mma maxSeq (ll1, ll2), mma maxSeq (lr1, lr2))
      in
        Branch {ul = ul, ur = ur, ll = ll, lr = lr, height = height, width = width}
      end

    | mma maxSeq (Leaf m1, Leaf m2) =
      let
        val def = tabulate maxSeq (fn _ => ~1.0) ~1.0 0
      in
        Leaf (tabulate maxSeq (fn i => vva maxSeq (sub (m1, i), sub (m2, i)))
                       def (length m1))
      end
    | mma _ _ = raise Blocked "blocked matrices must have the same structure"

  fun mma' maxSeq (f, g) () = mma maxSeq (f (), g ())

  fun mmm' maxSeq (Branch {ul = ul1, ur = ur1, ll = ll1, lr = lr1, ...},
                   Branch {ul = ul2, ur = ur2, ll = ll2, lr = lr2, ...}) () =
      let
        val ((ul, ur), (ll, lr)) =
            fork (fn () => fork (mma' maxSeq (mmm' maxSeq (ul1, ul2),
                                              mmm' maxSeq (ur1, ll2)),
                                 mma' maxSeq (mmm' maxSeq (ul1, ur2),
                                              mmm' maxSeq (ur1, lr2))),
                  fn () => fork (mma' maxSeq (mmm' maxSeq (ll1, ul2),
                                              mmm' maxSeq (lr1, ll2)),
```

```
                                mma' maxSeq (mmm' maxSeq (ll1, ur2),
                                             mmm' maxSeq (lr1, lr2))))

        fun dims (Leaf m) = (length m, length (sub (m, 0)))
          | dims (Branch { ul, ur, ll, ... }) =
            let
              val (h, w) = dims ul
              val (h', w') = (#1 (dims ll), #2 (dims ur))
            in
              (h + h', w + w')
            end

        val (height, width) = dims ul
      in
        Branch {ul = ul, ur = ur, ll = ll, lr = lr, height = height, width = width}
      end

    | mmm' maxSeq (Leaf m1, Leaf m2) () = Leaf (D.mmm maxSeq (m1, m2))

    | mmm' _ (m1, m2) () = raise Blocked "blocked matrices must have the same structure"

fun mmm maxSeq (m1, m2) =
    let
      (* convert from vector to tree representation *)
      fun wrap trans m depth (x, y) (height, width) =
          if depth = 0
          then
            let
              val def = tabulate maxSeq (fn _ => ~1.0) ~1.0 0
              val k = if y + width > length (sub (m, x)) then NONE else SOME width
              val m = tabulate maxSeq (fn i => slice (sub (m, x + i), y, k))
                                      def (if x + height > length m then (length m) - x else height)
            in
              Leaf m
            end
          else
            let
              (* round up *)
              val height' = height div 2 + (if height mod 2 > 0 then 1 else 0)
              val width' = width div 2 + (if width mod 2 > 0 then 1 else 0)
              val depth' = depth - 1
            in
              if trans then
                Branch {ul = wrap trans m depth' (x, y) (height', width'),
```

```
                    ur = wrap trans m depth' (x + height', y) (height − height', width'),
                    ll = wrap trans m depth' (x, y + width') (height', width − width'),
                    lr = wrap trans m depth' (x + height', y + width')
                                (height − height', width − width'),
                    width = width', height = height'}
        else
          Branch {ul = wrap trans m depth' (x, y) (height', width'),
                    ur = wrap trans m depth' (x, y + width') (height', width − width'),
                    ll = wrap trans m depth' (x + height', y) (height − height', width'),
                    lr = wrap trans m depth' (x + height', y + width')
                                (height − height', width − width'),
                    width = width', height = height'}
      end

(* convert from tree representation back to ordinary vectors *)
fun unwrap m len =
    let
      val debug = ref false

      val def = tabulate maxSeq (fn _ ⇒ ~1.0) ~1.0 0
      fun mySub (Leaf m) (x, y) = sub (sub (m, x), y)
        | mySub (Branch {ul, ur, ll, lr, height, width}) (x, y) =
          if x < height then
            if y < width then
              mySub ul (x, y)
            else
              mySub ur (x, y − width)
          else
            if y < width then
              mySub ll (x − height, y)
            else
              mySub lr (x − height, y − width)
    in
      tabulate maxSeq (fn i ⇒ tabulate maxSeq
                                (fn j ⇒ (mySub m (i, j))) ~1.0 len)
                def len
    end

fun computeDepth x =
    if x <= maxSeq then 0
    else 1 + (computeDepth ((x div 2) + (if x mod 2 > 0 then 1 else 0)))
val depth = computeDepth (Int.max (length m1, length (sub (m1, 0))))

val (m1', m2') = (wrap false m1 depth (0, 0) (length m1, length (sub (m1, 0))),
```

```
                    wrap true m2 depth (0, 0) (length m2, length (sub (m2, 0))))

      val m' = mmm' maxSeq (m1', m2') ()
    in
      unwrap m' (length m1)
    end
end
```

A.3 Sorting

Implementations of sorting implement the following signature. It defines a the type of collections t as well as the sort function itself. In addition to a collection to sort, this function also takes an alternative sorting implementation and a cut-off that indicates when that cut-off should be used. Whenever the input collection is smaller than the cut-off, the alternative implementation should be used instead.

```
signature SORT =
sig
  type t
  val sort : { cutoff : int, fallback : t -> t } -> t -> t
end
```

A.3.1 Quicksort

```
structure QuickSort : SORT =
struct
  fun sort { cutoff, fallback } a =
    let
      val c = ref 0
      fun next () = !c before c := !c + 1

      fun loop a =
        let
          val l = length a
        in
          if l < cutoff then [fallback a]
          else
            let
              val == = Real.== infix 4 ==
              val x = sub (a, 0)

              val (b, (c, d)) =
```

```
                fork (fn () ⇒ (loop (filter cutoff (fn y ⇒ y < x) a)),
                      fn () ⇒ (fork (fn () ⇒ [filter cutoff (fn y ⇒ y == x) a],
                                     fn () ⇒ loop (filter cutoff (fn y ⇒ y > x) a))))
            in
              b @ c @ d
            end
        end
    in
      concat (loop a)
    end
end
```

A.3.2 Mergesort

Mergesort depends on two implementations of merge, shown below.

```
functor MergeSort (structure M1: MERGE
                   structure M2: MERGE) : SORT =
struct
  fun sort { cutoff, fallback } a =
    let
      val merge = M1.merge { cutoff = 0, fallback = fn _ ⇒ raise Sort }
                           { cutoff = 0, fallback = fn _ ⇒ raise Sort }
      val split = M1.split { cutoff = 0, fallback = fn _ ⇒ raise Sort }
      val merge = M2.merge { cutoff = cutoff, fallback = split }
                           { cutoff = cutoff, fallback = merge }

      fun loop a =
        let
          val l = length a
        in
          if l <= 1 then a
          else if l < cutoff then fallback a
          else
            let
              val (b, c) = halve a
              val (b, c) = fork (fn () ⇒ loop b,
                                 fn () ⇒ loop c)
            in
              merge (b, c)
            end
        end
    in
      loop a
    end
```

```
end

structure MergeSeq : MERGE =
struct
  (* takes a sorted slice and an element x; returns two lists of A.t such that
     every element in the first is <= x and those in the second >= x *)
  fun split _ x c =
    let
      fun loop c =
          let
            val l = length c
          in
            if l = 0 then ([empty], [empty])
            else if l = 1 then
              if sub (c, 0) <= x then ([c], [empty]) else ([empty], [c])
            else
              let
                val == = Real.== infix 4 ==
                val (c1, c2) = halve c
                val y = sub (c2, 0)
              in
                if y == x then ([c1], [c2])
                else if y < x then
                  let
                    val (c21, c22) = loop c2
                  in
                    (c1 :: c21, c22)
                  end
                else (* y > x *)
                  let
                    val (c11, c12) = loop c1
                  in
                    (c11, c12 @ [c2])
                  end
              end
          end
    in
      loop c
    end

  (* merges two sorted inputs sequentially *)
  fun merge _ _ (b, c) =
    let
      val b_length = length b
```

```
      val c_length = length c

      fun loop (k, (i, j)) =
          if i = b_length then
            (sub (c, j), (i, j+1))
          else if j = c_length then
            (sub (b, i), (i+1, j))
          else
            let
              val x = sub (b, i)
              val y = sub (c, j)
            in
              if x < y then
                (x, (i+1, j))
              else
                (y, (i, j+1))
            end
    in
      #1 (unfoldi NONE (b_length + c_length, (0, 0), loop))
    end
end

structure MergePar : MERGE =
struct
  (* takes a sorted slice and an element x; returns two lists of A.t such that
     every element in the first is <= x and those in the second >= x *)
  fun split { cutoff, fallback } x c = fallback x c

  (* merges two sorted inputs in parallel *)
  fun merge split_cutofffallback { cutoff, fallback } (b, c) =
    let
      fun loop (b, c) =
        let
          val l = length b
        in
          (* assumes cutoff > 0 *)
          if l < cutoff orelse length c < cutoff then [fallback (b, c)]
          else
            let
              val (b1, b2) = halve b
              val x = sub (b2, 0)
              val (c1s, c2s) = split split_cutofffallback x c
              val (c1, c2) = (concat c1s, concat c2s)
            in
```

```
            if l = 1 then
               [c1, b, c2]
            else
              let
                val (a1, a2) = fork (fn () ⇒ loop (c1, b1),
                                     fn () ⇒ loop (c2, b2))
              in
                a1 @ a2
              end
          end
      end
    in
      concat (loop (b, c))
    end
end
```

A.3.3 Insertion Sort

```
structure InsertionSort : SORT =
struct
  fun sort _ a =
      let
        (* take an element and a sorted list and return a new list with that
         element in the appropriate place *)
        fun insert (x, nil) = [x]
          | insert (x, y::ys) = if x <= y then x::y::ys
                                else y::(insert (x, ys))
      in
        fromList NONE (foldl insert nil (toList a))
      end
end
```

A.3.4 Selection Sort

```
structure SelectionSort : SORT =
struct
  datatype mode =
      Repeat of real * int
   (* this could be nullary since we have the index of the last
      element added to the output *)
   | Larger of real

  fun sort _ a =
```

```
      let
        (* return the first occurrence of smallest element > z and it's idx *)
        fun select (_, Larger z) =
            let
              val (x, i) = foldli (fn (i, x, (y, j)) =>
                                      if x > z andalso x < y
                                      then (x, i)
                                      else (y, j))
                                  (Real.posInf, 0) a
            in
              (x, Repeat (x, i))
            end
          (* find the next occurance of x after index i *)
          | select (k, Repeat (x, i)) =
            let
              val == = Real.== infix 4 ==
              (* return index >= j such that sub (a, j) = x *)
              fun loop j =
                  if j = length a then NONE
                  else if sub (a, j) == x then SOME j
                  else loop (j + 1)
            in
              case loop (i + 1) of
                NONE => select (k, Larger x)
              | SOME j => (x, Repeat (x, j))
            end

        val b = #1 (unfoldi NONE (length a, Larger Real.negInf, select))
      in
        b
      end
end
```

A.3.5 Quicksort (futures)

```
structure QuickSortFutures : SORT =
struct
  fun sort { cutoff, fallback } xs =
    let
      fun qs' (nil, rest) = rest
        | qs' (x::xs, rest) =
          let
            fun partition nil = (nil, nil)
              | partition (y::ys) =
```

```
            let
              val (ls, rs) = partition ys
            in
              if x > y then
                (y::ls, rs)
              else
                (ls, y::rs)
            end
        val (ls, rs) = partition xs
      in
        qs' (ls, x::(qs' (rs, rest)))
      end

datatype 'a flist = Nil | Cons of 'a * 'a flist future

fun partition (x, ys, yss) =
      let
        fun part (y::ys) =
            let
              val (ls, rs) = part ys
            in
              if x > y then (y :: ls, rs)
              else (ls, y :: rs)
            end
          | part nil = (nil, nil)
        val (ls, rs) = part ys
      in
        case touch yss
         of Cons (ys, yss) =>
            let
              val parts = future (fn () => partition (x, ys, yss))
            in
              (ls, rs,
               future (fn () =>
                           let
                             val (ls, _, lss, _) = touch parts
                           in
                             Cons (ls, lss)
                           end),
               future (fn () =>
                           let
                             val (_, rs, _, rss) = touch parts
                           in
                             Cons (rs, rss)
```

```
                                    end))
                    end
                  | Nil ⇒ (ls, rs, future (fn () ⇒ Nil), future (fn () ⇒ Nil))
            end

        fun qs (Nil, rest) : real list flist' = rest
          | qs (Cons (nil, xss), rest) = qs (touch xss, rest)
          | qs (Cons (x::xs, xss), rest) =
            let
              val (ls, rs, lss, rss) = partition (x, xs, xss)
            in
              case touch rss
               of Nil ⇒
                  qs (Cons (ls, lss),
                      Cons' (qs' (x::rs, nil), future (fn () ⇒ rest)))
                | _ ⇒
                  qs (Cons (ls, lss),
                      Cons' ([x], future (fn () ⇒ (qs (Cons (rs, rss), rest)))))
            end

        fun toStream xs =
            let
              fun loop nil = Nil
                | loop xs = Cons (List.take (xs, cutoff),
                                  future (fn () ⇒ loop (List.drop (xs, cutoff))))
                            handle Subscript ⇒ Cons (xs, future (fn () ⇒ Nil))
            in
              loop xs
            end
        fun fromStream Nil = nil
          | fromStream (Cons (xs, xss)) = xs @ (fromStream (touch xss))
      in
        fromList NONE (fromStream (qs (toStream (toList xs), Nil)))
      end
end
```

A.3.6 Mergesort (futures)

```
structure MergeSortFutures : SORT =
struct
  fun sort { cutoff, fallback } a =
    let
      datatype 'a flist = Nil | Cons of 'a * 'a flist future
```

```
      fun merge (xss, yss, 0, zs) =
          Cons (rev zs, future (fn () ⇒ merge (xss, yss, cutoff, nil)))

        | merge (Nil, yss, _, zs) = Cons (rev zs, future (fn () ⇒ yss))
        | merge (xss, Nil, _, zs) = Cons (rev zs, future (fn () ⇒ xss))

        | merge (Cons (nil, xss), yss, count, zs) = merge (touch xss, yss, count, zs)
        | merge (xss, Cons (nil, yss), count, zs) = merge (xss, touch yss, count, zs)
        | merge (xsxss as Cons (x::xs, xss),
                 ysyss as Cons (y::ys, yss), count, zs) =
          if x < y then
            merge (Cons (xs, xss), ysyss, count − 1, x::zs)
          else
            merge (xsxss, Cons (ys, yss), count − 1, y::zs)

      fun loop a =
        let
          val l = length a
        in
          if l < cutoff then Cons (toList (fallback a), future (fn () ⇒ Nil))
          else
            let
              val (b, c) = halve a
              val (b, c) = (loop b, loop c)
            in
              merge (b, c, cutoff, nil)
            end
        end

      fun fromStream Nil = nil
        | fromStream (Cons (xs, xss)) = xs @ (fromStream (touch xss))
    in
      fromList (SOME cutoff) (fromStream (loop a))
    end
end
```

A.4 Miscellaneous

A.4.1 Barnes-Hut Simulation

```
structure BarnesHut : SIM =
struct
  exception Sim of string
                    (* #particles C.O.M. moment bounding children *)
```

```
datatype t = Node of int * pointmass * Matrix.t * box * children
and children = Octants of t list (* at least one child has a particle *)
             | Particles of particle list (* will never be split, >= 2 particles *)
             | Particle of particle (* for boxes with exactly one particle *)
             | Empty (* contains no particles *)

fun fold fb fp x (Node (_, _, _, box, children)) =
    let
      val x = fb (box, x)
    in
      fold' fb fp x children
    end
and fold' fb fp x (Octants ts) =
    foldl (fn (t, x) => fold fb fp x t) x ts
  | fold' _ fp x (Particles ps) =
    foldl fp x ps
  | fold' _ fp x (Particle p) = fp (p, x)
  | fold' _ _ x Empty = x

fun empty box = Node (0, { position = Vector.zero, mass = 0.0 },
                         Matrix.zero, box, Empty)

(* Make the i'th octant of box *)
fun octant (box : box) i =
    let
      val op* = Vector.scale

      val (x, y, z) = #position box
      val (dimension as (dx, dy, dz)) = 0.5 * (#dimension box)

      fun myEmpty position = empty { position = position,
                                     dimension = dimension }
    in
      case i of 0 => myEmpty (x, y, z)
              | 1 => myEmpty (x + dx, y, z)
              | 2 => myEmpty (x, y + dy, z)
              | 3 => myEmpty (x + dx, y + dy, z)
              | 4 => myEmpty (x, y, z + dz)
              | 5 => myEmpty (x + dx, y, z + dz)
              | 6 => myEmpty (x, y + dy, z + dz)
              | 7 => myEmpty (x + dx, y + dy, z + dz)
              | _ => raise Sim "can only build 8 octants"
    end
```

```
fun contains (box as { position, dimension }) pos =
    let
      val op+ = Vector.+
      val op< = Vector.<
      val op<= = Vector.<=
    in
      position <= pos andalso
      pos < position + dimension
    end

fun makeNode (box, children as Particle { position, mass, ... }) =
    let in
      Node (1, { position = position, mass = mass },
            Matrix.zero, box, children)
    end
  | makeNode (box, children as Particles ps) =
    let
      val op@ = Vector.+ infix 6 @
      val op* = Vector.scale

      val (mass, weighted_pos) =
          foldl (fn ({ position, mass, ... } : particle,
                     (sum_of_mass, weighted_pos)) =>
                    (mass + sum_of_mass, mass * position @ weighted_pos))
                (0.0, Vector.zero) ps
      val position = (1.0 / mass) * weighted_pos
    in
      Node (length ps, { mass = mass, position = position },
            Matrix.zero, box, children)
    end
  | makeNode (box, Empty) = empty box
  | makeNode (box, Octants ts) =
    let
      val op@ = Vector.+ infix 6 @
      val op* = Vector.scale
      val op- = Vector.-
      val x = Vector.outer infix 7 x
      val dot = Vector.dot infix 7 dot

      val (count, mass, weighted_pos) =
          foldl (fn (Node (count, { position, mass }, _, _, _),
                     (total, sum_of_mass, weighted_pos)) =>
                    (count + total, mass + sum_of_mass,
                     mass * position @ weighted_pos))
```

```
                (0, 0.0, Vector.zero) ts
    val position = (1.0 / mass) * weighted_pos

    (* calculate quadrupole for one node based on children ts *)
    val moment =
        let
          val op+ = Matrix.+
          val op* = Matrix.scale
          val op~ = Matrix.- infix 6 ~

          fun Q_child (Node (_, { position = child_pos, mass }, moment, _, _)) =
              let
                val del_r = child_pos - position
              in
                moment + (mass * (3.0 * (del_r x del_r)
                                                  ~ del_r dot del_r * Matrix.I))
              end
        in
          foldl (fn (n, acc) => Q_child n + acc) Matrix.zero ts
        end
  in
    if count = 0 then
      (* Prune empty trees *)
      empty box
    else if count = 1 then
      (* Prune trees with only one particle as well. *)
      let
        fun findParticle (p, NONE) = SOME p
          | findParticle (_, SOME _) =
            raise Sim "found 2 particles after count = 1"

        val p = case fold' (fn (_, y) => y) findParticle NONE (Octants ts) of
                    NONE => raise Sim "found 0 particles after count = 1"
                  | SOME p => p
      in
        Node (count, { position = position, mass = mass },
              Matrix.zero, box, Particle p)
      end
    else
      Node (count, { position = position, mass = mass },
            moment, box, Octants ts)
  end

(* Insert a particle into the given pseudo-tree
```

```
    (* pseudo−tree = tree without aggregate data (e.g. moment) *)
    invariant: the particle must lie within the given bounding box *)
and insertParticle (p, (box, Particles ps)) =
    let in
      (* This is a "small" box, so we don't split *)
      (box, Particles (p :: ps))
    end
  | insertParticle (p, (box, Empty)) =
    let in
      (box, Particle p)
    end
  | insertParticle (p, (box, Particle p')) =
    if Vector.< (#dimension box, epsilon) then
      (* Small box −− don't split *)
      insertParticle (p, (box, Particles [p']))
    else
      (* Split! *)
      let
        (* Make new children *)
        val ts = List.tabulate (8, octant box)
        (* Add the two particles *)
        val (box, children) = insertParticle (p, (box, Octants ts))
        val (box, children) = insertParticle (p', (box, children))
      in
        (box, children)
      end
  | insertParticle (p as { position, ... } : particle,
                    (box, Octants ts)) =
    let
      (* Find the octant that should contain the particle and recur *)
      val (ts', ts) = List.partition (fn (Node (_, _, _, box, _)) =>
                                         contains box position) ts
      val (box', children') =
          case ts' of [Node (_, _, _, box', children')] => (box', children')
                    | nil => raise Sim ("insertParticle found 0 "
                                        ^ "children containing particle "
                                        ^ (particleToString p))
                    | _ => raise Sim ("insertParticle found 2+ "
                                      ^ "children containing particle")
      val t' = makeNode (insertParticle (p, (box', children')))
    in
      (box, Octants (t' :: ts))
    end
```

```
(* Builds the Barnes−Hut tree with the addition of a new particle. That
 particle need not lie within the bounding box of the given tree *)
fun addParticle (p as { position, ... } : particle,
                 Node (_, _, _, box, children)) =
    let
      fun add (box, children) =
          if contains box position then
            (* If the new particle falls within the tree then add it *)
            makeNode (insertParticle (p, (box, children)))
          else
            (* Expand the tree to cover a larger volume *)
            let
              val op* = Vector.scale
              (* old box *)
              val (x0, y0, z0) = #position box
              (* particle position *)
              val (px, py, pz) = position

              (* Extend the box in the direction of the new point. NB, this
                 may not include the new point, but if not, we will double
                 again when we recur. Also, remember which octant the old
                 tree will fall in. *)
              val (dx, dy, dz) = #dimension box
              val (x, i) = if px < x0 then (x0 − dx, 1) else (x0, 0)
              val (y, i) = if py < y0 then (y0 − dy, i + 2) else (y0, i)
              val (z, i) = if pz < z0 then (z0 − dz, i + 4) else (z0, i)

              (* Double each dimension *)
              val box' = { position = (x, y, z), dimension = 2.0 * #dimension box }

              (* Make the other seven octants *)
              val ts = List.tabulate (8,
                                      fn j ⇒ if i = j then (* Use the original *)
                                                 makeNode (box, children)
                                             else (* Make a new one *)
                                                 octant box' j)

            in
              (* Then try to add the particle to the expanded tree *)
              add (box', Octants ts)
            end

      val t = add (box, children)
      (* possibly "prune" from the top *)
```

```
      val t = case t of Node (count, _, _, _, Octants ts) ⇒
                        (case List.find (fn (Node (count', _, _, _, _)) ⇒
                                            count = count') ts of
                           SOME t' ⇒ t'
                         | NONE ⇒ t)
                      | _ ⇒ t
    in
      t
    end

fun calculateForce maxSeq (p as { position = part_pos, ... } : particle) t =
  let
    val op− = Vector.−

    (* tests if a cell is well−separated from a particle *)
    fun separatedFrom (Node (count, { position = mass_pos, ... }, _,
                                    { dimension, ... }, _)) =
        if count = 1 then true
        else
          let
            val dot = Vector.dot infix 7 dot

            val del_r = mass_pos − part_pos
            val d = Math.sqrt (del_r dot del_r)
            val s = max3 dimension
          in
            s < theta ∗ d
          end

    (* calculates force on a particle due to a cell *)
    fun calculateFromNode (Node (count, { mass, position = mass_pos }, moment, _, _)) =
        if count = 0 then Vector.zero
        else
          let
            val del_r = mass_pos − part_pos
            val == = Vector.== infix 4 ==
          in
            if del_r == Vector.zero
            then Vector.zero
            else
              let
                val scale = Vector.scale infix 7 scale
                val dot = Vector.dot infix 7 dot
                val mul = Matrix.vec infix 7 mul
```

```
                val r_squared = del_r dot del_r
                val abs_r = Math.sqrt r_squared
                val mrcubed = mass / (r_squared * abs_r)
                val acc = mrcubed scale del_r
              in
                if count = 1 then acc else
                let
                  val del_r5inv = 1.0 / (r_squared * r_squared * abs_r)
                  val moment_del_r = moment mul del_r
                  val phi_moment = ~2.5 * (del_r dot moment_del_r) * del_r5inv / r_squared
                in
                  (acc - (phi_moment scale del_r))
                    - (del_r5inv scale moment_del_r)
                end
              end
          end

    val op+ = Vector.+
  in
    if separatedFrom t then
      calculateFromNode t
    else
      case t of
        Node (count, _, _, _, Octants ts) =>

        if false (* count > maxSeq * 10 *) then
          A.fold 1 Vector.+
                 (fn i => calculateForce maxSeq p (List.nth (ts, i)))
                 Vector.zero (length ts)
        else

          foldl (fn (child, acc) => calculateForce maxSeq p child + acc)
                Vector.zero ts
      (* Reached a leaf *)
      | _ => calculateFromNode t
  end

fun update maxSeq del_t t =
    let
      fun updateParticle (p as { position, mass, velocity }) =
          let
            val op@ = Vector.+ infix 6 @
            val scale = Vector.scale infix 7 scale
```

```
      val F = calculateForce maxSeq p t
      val a = mass scale F
      val v = velocity @ del_t scale a
    in
      { position = position @ del_t scale v,
        velocity = v,
        mass = mass }
    end

fun updateNode (t as Node (_, _, _, box, Empty)) = (t, nil)
  | updateNode (Node (_, _, _, box, Particle p)) =
    let
      val p = updateParticle p
    in
      if contains box (#position p) then
        (makeNode (insertParticle (p, (box, Empty))), nil)
      else
        (empty box, [p])
    end
  | updateNode (Node (_, _, _, box, Particles ps)) =
    let
      val ps = map updateParticle ps
      val (ps, ps') = List.partition (fn ({ position, ... } : particle) ⇒
                                          contains box position) ps
      val (box, children) = foldl insertParticle (box, Empty) ps
      val t = makeNode (box, children)
    in
      (t, ps')
    end
  | updateNode (Node (count, _, _, box, Octants ts)) =
    let
      val (ts, ps) =
          if count > maxSeq then
            A.fold 1 (fn ((ts, ps), (ts', ps')) ⇒
                                (ts @ ts', ps @ ps'))
                   (fn i ⇒
                       let val (t, ps) = updateNode (List.nth (ts, i))
                       in ([t], ps) end)
                   (nil, nil)
                   (length ts)
          else
            foldl (fn (t, (ts, ps)) ⇒
                      let val (t, ps') = updateNode t
```

```
                    in (t :: ts, ps' @ ps) end)
                (nil, nil)
                ts

          (* Some particles might have moved between our children... split
             those out *)
          val (ps, ps') = List.partition (fn ({ position, ... } : particle) =>
                                          contains box position) ps

          (* Insert those particles which still fall within our box *)
          val (box, children) = foldl insertParticle (box, Octants ts) ps
          val t = makeNode (box, children)
        in
          (t, ps')
        end

      val (t, ps) = updateNode t
      (* Add the remaining particles back in *)
      val t = foldl addParticle t ps
    in
      t
    end
end
```

A.4.2 Quickhull

```
structure QuickHull : HULL =
struct
  structure V = Vector

  (* Given two points on the hull (p1, p2), return the list of points on the
     hull between p1 (inclusive) and p2 (exclusive) in order. *)
  fun split maxSeq (ps, (p1, p2)) =
      let
        (* compute the "distance" from the line as well as the points above
           the line *)
        fun finddist (p, (ps, (p0, d0))) =
            let
              val d = V.cross (p, (p1, p2))
            in
              (if d > 0.0 then p::ps else ps,
               if Real.> (d, d0)
               then (p, d)
               else (p0, d0))
```

```
        end
    (* convert from list to seq *)
    fun seqToPar (ps, (p, d)) = (fromList maxSeq ps, (p, d))
    (* take the larger "distance" and merge the lines *)
    fun mergedist ((ps1, (p1, d1)), (ps2, (p2, d2))) =
        (append maxSeq (ps1, ps2),
         if Real.> (d1, d2)
         then (p1, d1)
         else (p2, d2))
    (* points above the line, plus the point furthest above the line *)
    val (ps, (p0, _)) = fold maxSeq
                            mergedist
                            seqToPar
                            finddist
                            (nil, (p1, 0.0))
                            ps
    (* include p1 *)
    val ps = append maxSeq (singleton p1, ps)
    val l = size ps
  in
    if l <= 2 then ps
    else if l <= maxSeq then
      append maxSeq (split maxSeq (ps, (p1, p0)),
                     split maxSeq (ps, (p0, p2)))
    else
      append maxSeq (fork (fn () ⇒ split maxSeq (ps, (p1, p0)),
                           fn () ⇒ split maxSeq (ps, (p0, p2))))
  end

fun hull maxSeq ps =
  let
    (* Find the extreme points in the x−dimension *)
    fun mergeminmax ((p1 as (min1, _),
                      p1' as (max1, _)),
                     (p2 as (min2, _),
                      p2' as (max2, _))) =
        (if min1 < min2 then p1 else p2,
         if max1 > max2 then p1' else p2')

    val (min, max) = fold maxSeq
                          mergeminmax
                          (fn x ⇒ x)
                          (fn (p, minmax) ⇒ mergeminmax ((p, p), minmax))
                          ((Real.posInf, 0.0), (Real.negInf, 0.0))
```

```
                                    ps
      in
        append maxSeq (fork (fn () ⇒ split maxSeq (ps, (min, max)),
                                    fn () ⇒ split maxSeq (ps, (max, min))))
      end
end
```

A.4.3 Fibonacci

```
fun fib n =
    if n <= 1 then 1
    else
      let
        val (a, b) = fork (fn () ⇒ fib (n − 1),
                                    fn () ⇒ fib (n − 2))
      in
        a + b
      end
```

A.4.4 Pipelining

```
structure Pipeline =
struct
  open FutureList
  fun run stages numSteps (init, last, finish) =
      let
        val inputs = tabulate (numSteps, init)
        fun loop (i, stage::stages, inputs) =
            let
              val inputs = mapi stage inputs
            in
              loop (i + 1, stages, inputs)
            end
          | loop (i, nil, inputs) = (i, inputs)

        val (i, inputs) = loop (1, stages, inputs)

        val outputs = foldl (fn (input, (j, output)) ⇒
                                        (j + 1, last (input, output))))
                            (0, finish)
                            inputs
      in
        #2 outputs
      end
end
```

Bibliography

Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proc. of the Int. Conf. on Funct. Program.*, pages 83–91, New York, NY, USA, 1996. ACM.

H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams Iv, D. P. Friedman, E. Kohlbecker, G. L. Steele, Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised report on the algorithmic language scheme. *Higher Order Symbol. Comput.*, 11(1):7–105, 1998. ISSN 1388-3690.

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proc. of the Symp. on Parallel Algorithms and Architectures*, pages 1–12, New York, NY, USA, 2000. ACM.

Shail Aditya, Arvind, Jan-Willem Maessen, and Lennart Augustsson. Semantics of pH: A parallel dialect of Haskell. Technical Report Computation Structures Group Memo 377-1, MIT, June 1995.

Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna K. Shyamasundar, and Katherine Yelick. Deadlock-free scheduling of x10 computations with bounded resources. In *Proc. of the Symp. on Parallel Algorithms and Architectures*, pages 229–240, New York, NY, USA, 2007. ACM.

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of the 1967 Spring Joint Comput. Conf.*, pages 483–485, New York, NY, USA, 1967. ACM.

Andrew W. Appel. Garbage collection can be faster than stack allocation. *Inf. Process. Lett.*, 25 (4):275–279, 1987.

Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989. URL `citeseer.nj.nec.com/appel88simple.html`.

Andrew W. Appel and David B. MacQueen. Standard ml of new jersey. In *Prog. Lang. Implementation and Logic Programming*, volume 528, pages 1–13. Springer-Verlag, 1991.

Andrew W. Appel, Bruce Duba, and David B. MacQueen. Profiling in the presence of optimization and garbage collection. Technical Report CS-TR-197-88, Princeton University, November 1988.

Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proc. of the Conf. on Program. Lang. Design and Implementation*, pages 168–179,

New York, NY, USA, 2001. ACM.

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proc. of the Symp. on Parallel Algorithms and Architectures*, pages 119–129, New York, NY, USA, 1998. ACM.

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, 1989.

K. T. P. Au, M. M. T. Chakravarty, J. Darlington, Y. Guo, S. Jähnichen, M. Köhler, G. Keller, W. Pfannenstiel, and M. Simons. Enlarging the scope of vector-based computations: Extending Fortran 90 by nested data parallelism. In *Proc. of the Advances in Parallel and Distrib. Comput. Conf.*, page 66, Washington, DC, USA, 1997. IEEE Computer Society.

Hendrik Pieter Barendregt. *The lambda calculus : its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North Holland, 1984.

J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(4), Dec 1986.

Nick Benton, Andrew Kennedy, and George Russell. Compiling standard ml to java bytecodes. In *Proc. of the Int. Conf. on Funct. Program.*, pages 129–140, New York, NY, USA, 1998. ACM.

Nick Benton, Andrew Kennedy, and Claudio V. Russo. Adventures in interoperability: the sml.net experience. In *Proc. of the Int. Conf. on Principles and Pract. of Declarative Program.*, pages 215–226, New York, NY, USA, 2004. ACM.

G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, 1990.

Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proc. of the Int. Conf. on Funct. Program.*, pages 213–225, May 1996.

Guy E. Blelloch and Margaret Reid-Miller. Pipelining with futures. In *Proc. of the Symp. on Parallel Algorithms and Architectures*, pages 249–259, Newport, RI, USA, 1997. ACM.

Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.*, 8(2):119–134, 1990.

Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21 (1):4–14, 1994.

Guy E. Blelloch, Phillip B. Gibbons, Girija J. Narlikar, and Yossi Matias. Space-efficient scheduling of parallelism with synchronization variables. In *Proc. of the Symp. on Parallel Algorithms and Architectures*, pages 12–23, New York, NY, USA, 1997. ACM.

Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, 1999.

Robert D. Blumofe. Scheduling multithreaded computations by work stealing. In *Symp. on Foundations of Comput. Sci.*, pages 356–368, 1994.

Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proc. of the Symp. on Theory of Comput.*, pages 362–371, 1993.

Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. of the Symp. on Principles and Pract. of Parallel Program.*, pages 207–216, New York, NY, USA, 1995. ACM.

Hans-J. Boehm. Destructors, finalizers, and synchronization. In *Proc. of the Symp. on Principles of Program. Lang.*, pages 262–272, New York, NY, USA, 2003. ACM.

F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proc. of the Conference on Funct. Program. Lang. and Comput. Architecture*, pages 187–194, New York, NY, USA, 1981. ACM.

A. Bykat. Convex hull of a finite set of points in two dimensions. *Inf. Process. Lett.*, 7(6): 296–298, 1978.

Manuel M. T. Chakravarty and Gabriele Keller. More types for nested data parallel programming. In *Proc. of the Int. Conf. on Funct. Program.*, pages 94–105, New York, NY, USA, 2000. ACM.

Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: a structured english query language. In *Proc. of the Workshop on Data Description, Access and Control*, pages 249–264, New York, NY, USA, 1974. ACM.

Nathan Charles and Colin Runciman. An interactive approach to profiling parallel functional programs. In *Selected Papers of the Workshop on the Implementation of Funct. Lang.*, pages 20–37, London, UK, September 1998.

Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *Proc. of the Symp. on Parallel Algorithms and Architectures*, pages 105–115, New York, NY, USA, 2007. ACM.

Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. *SIGPLAN Not.*, 36(5): 125–136, 2001.

Silvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for objective-caml. In *Proc. of the Int. Symp. on Agent Syst. and Applications*, page 22, Washington, DC, USA, 1999. IEEE Computer Society.

Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *Proc. of the Intl. Conf. on Parallel Processing*, pages 536–545, Washington, DC, USA, 2008. IEEE Computer Society.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the Symp. on Principles of Program. Lang.*, pages 238–252, New York, NY, USA, 1977. ACM.

John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. Programming with exceptions in jcilk. *Sci. Comput. Program.*, 63(2):147–171, 2006. ISSN 0167-6423.

Edsger W. Dijkstra. Cooperating sequential processes. *The origin of concurrent programming:*

from semaphores to remote procedure calls, pages 65–138, 2002.

Robert Ennals. *Adaptive Evaluation of Non-Strict Programs*. PhD thesis, University of Cambridge, 2004.

Marc Feeley. Polling efficiently on stock hardware. In *Proc. of the Conf. on Funct. Program. Lang. and Comput. Architecture*, pages 179–187, New York, NY, USA, 1993. ACM.

Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine and the λ-calculus. *Formal Description of Programming Language Concepts III*, pages 193–217, 1986.

John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the sisal language project. *J. Parallel Distrib. Comput.*, 10(4):349–366, 1990.

Matthew Fluet, Michael Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *Proc. of the Int. Conf. on Funct. Program.*, pages 241–252, New York, NY, USA, 2008a. ACM.

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in manticore. In *Proc. of the Int. Conf. on Funct. Program.*, pages 119–130, New York, NY, USA, 2008b. ACM.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proc. of the Conf. on Program. Lang. Design and Implementation*, pages 212–223, New York, NY, USA, 1998. ACM.

Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proc. of the Symp. on Foundations of Comput. Sci.*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.

Emden R. Gansner and John H. Reppy, editors. *The Standard ML Basis Library*. Cambridge University Press, jul 2002.

Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: implementing a fast parallel call. *J. Parallel Distrib. Comput.*, 37(1):5–20, 1996. ISSN 0743-7315.

Michael Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Int. Conf. on Architectural Support for Program. Lang. and Operating Syst.*, San Jose, CA, USA, October 2006.

Ronald L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical J.*, 45: 1563–1581, 1966.

Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, 2004. ISSN 0362-1340.

John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Trans. on Program. Lang. and Syst.*, 21(2):240–285, 1999.

Jörgen Gustavsson and David Sands. A foundation for space-safe transformations of call-by-need programs. In *Proc. of Workshop on Higher Order Operational Techniques in Semantics*, number volume 26 of Electronic Notes in Theoretical Computer Science, September 1999.

Robert H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proc. of the Symp. on LISP and Funct. Program.*, pages 9–17, New York, NY, USA, 1984. ACM.

Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

Kevin Hammond and Simon L. Peyton Jones. Profiling scheduling strategies on the GRIP multiprocessor. In *Int. Workshop on the Parallel Implementation of Funct. Lang.*, pages 73–98, RWTH Aachen, Germany, September 1992.

Kevin Hammond, Hans-Wolfgang Loidl, and Andrew S. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In A. P. Wim Böhm and John T. Feo, editors, *High Performance Functional Computing*, pages 208–221, 1995.

Kevin Hammond, Jost Berthold, and Rita Loogen. Automatic skeletons in template haskell. *Parallel Processing Letters*, 13(3):413–424, September 2003.

High Performance Fortran Forum. High performance fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, Center for Research on Parallel Computation, Houston, TX, USA, 1993.

Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.

John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: towards embedded ml programming. In *Proc. of the Int. Conf. on Funct. Program.*, pages 70–81, New York, NY, USA, 1999. ACM.

John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proc. of the Symp. on Principles of Program. Lang.*, pages 410–423, New York, NY, USA, 1996. ACM.

Intel Corporation. Intel Threading Building Blocks. `www.threadingbuildingblocks.org`, 2009.

Kenneth E. Iverson. *A Programming Language*. Wiley, New York, NY, USA, 1962.

C. Barry Jay, Murray Cole, M. Sekanina, and Paul Steckler. A monadic calculus for parallel costing of a functional language of arrays. In *Proc. of the Int. Euro-Par Conf. on Parallel Processing*, pages 650–661, London, UK, 1997. Springer-Verlag.

Gabriele Keller and Manuel M. T. Chakravarty. Flattening trees. In *Proc. of the Int'l Euro-Par Conf. on Parallel Processing*, pages 709–719, London, UK, 1998. Springer-Verlag.

David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-t: a high-performance parallel lisp. In *Proc. of the Conf. on Program. Language Design and Implementation*, pages 81–90, New York, NY, USA, 1989. ACM.

Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6, Jan 1964.

Doug Lea. A java fork/join framework. In *Proc. of the Conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM.

Roman Lechtchinsky, Manuel M. T. Chakravarty, and Gabriele Keller. Costing nested array

codes. *Parallel Processing Letters*, 12(2):249–266, 2002.

Roman Leshchinskiy, Manuel M. T. Chakravarty, and Gabriele Keller. Higher order flattening. In Vassil N. Alexandrov, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Proc. of the Int. Conf. on Comput. Sci.*, volume 3992 of *Lecture Notes in Computer Science*, pages 920–928. Springer, 2006.

Jean-Jacques Lévy. *Reductions Correctes et Optimales dans le Lambda Calcul*. PhD thesis, University of Paris 7, 1978.

Peng Li, Simon Marlow, Simon Peyton Jones, and Andrew Tolmach. Lightweight concurrency primitives for ghc. In *Proc. of the Workshop on Haskell*, pages 107–118, New York, NY, USA, 2007. ACM.

Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

Hans-Wolfgang Loidl and Kevin Hammond. A sized time system for a parallel functional language. In *Proc. of the Glasgow Workshop on Funct. Program.*, Ullapool, Scotland, July 1996.

Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. Submitted to ICFP'09, March 2009.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, USA, 1997.

Yasuhiko Minamide. Space-profiling semantics of the call-by-value lambda calculus and the cps transformation. In Andrew D. Gordon and Andrew M. Pitts, editors, *Proc. of the Int. Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.

Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proc. of the Conf. on LISP and Funct. Program.*, pages 185–197, New York, NY, USA, 1990. ACM.

J. Gregory Morrisett and Andrew Tolmach. Procs and locks: a portable multiprocessing platform for standard ml of new jersey. In *Proc. of the Symp. on Principles and Pract. of Parallel Program.*, pages 198–207, New York, NY, USA, 1993. ACM.

Joel Moses. The function of function in lisp or why the funarg problem should be called the environment problem. *SIGSAM Bull.*, (15):13–27, 1970.

Girija J. Narlikar and Guy E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Trans. on Program. Lang. and Syst.*, 21(1):138–173, 1999.

Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In *Foundations of Software Technology and Theoretical Computer Science*, 2008.

Álvaro J. Rebón Portillo, Kevin Hammond, Hans-Wolfgang Loidl, and Pedro B. Vasconcelos. Cost analysis using automatic size and time inference. In Ricardo Pena and Thomas Arts, editors, *Revised Selected Papers of the Int. Workshop on Implementation of Funct. Lang.*, volume 2670 of *Lecture Notes in Computer Science*, pages 232–248, Madrid, Spain, September 2002. Springer.

John H. Reppy. *Concurrent programming in ML*. Cambridge University Press, New York, NY, USA, 1999.

Paul Roe. *Parallel Programming Using Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, 1991.

J. R. Rose and G. L. Steele, Jr. C*: An extended C language for data parallel programming. Technical Report PL 87-5, Thinking Machines Corporation, 1987.

Mads Rosendahl. Automatic complexity analysis. In *Proc. of the Int. Conf. on Funct. Program. Lang. and Computer Architecture*, pages 144–156, New York, NY, USA, 1989. ACM.

Colin Runciman and Niklas Röjemo. New dimensions in heap profiling. *J. Funct. Program.*, 6 (4):587–620, 1996.

Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *J. Funct. Program.*, 3(2):217–245, 1993a.

Colin Runciman and David Wakeling. Profiling Parallel Functional Computations (Without Parallel Machines). In *Functional Programming, Glasgow '93*, pages 236–251. Springer-Verlag, 1993b.

Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. The MIT Press, Cambridge, MA, USA, 1988.

David Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, Department of Computing, Imperial College, University of London, September 1990.

Patrick M. Sansom and Simon L. Peyton Jones. Profiling lazy functional programs. In *Proc. of the Glasgow Workshop on Funct. Program.*, pages 227–239, London, UK, 1993. Springer-Verlag.

Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *Proc. of the Symp. on Principles of Program. Lang.*, pages 355–366, New York, NY, USA, 1995. ACM.

Peter Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3):231–264, 1997.

Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proc. of the Conf. on LISP and Funct. Program.*, pages 150–161, New York, NY, USA, 1994. ACM.

Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *Proc. of the Int. Conf. on Funct. Program.*, pages 253–264, New York, NY, USA, 2008. ACM.

Guy L. Steele, Jr. Rabbit: A compiler for scheme. Technical Report 474, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

Dan Suciu and Val Tannen. Efficient compilation of high-level data parallel algorithms. In *Proc. of the Symp. on Parallel Algorithms and Architectures*, pages 57–66, New York, NY, USA, 1994. ACM.

Martin Sulzmann, Edmund S.L. Lam, and Simon Marlow. Comparing the performance of concurrent linked-list implementations in haskell. In *Proc. of the Workshop on Declarative Aspects of Multicore Program.*, pages 37–46, New York, NY, USA, 2008. ACM.

Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. Funct. Programm.*, 8(1):23–60, January 1998.

David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proc. of the Software Engineering Symp. on Practical Software Development Environments*, pages 157–167. ACM Press, 1984. doi: http://doi.acm.org/10.1145/800020.808261.

Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1996.

Stephen Weeks. Whole-program compilation in mlton. In *Proc. of the Workshop on ML*, page 1, New York, NY, USA, 2006. ACM.