Bottom-Up Logic Programming for Multicores

Flavio Cruz[†‡], Michael P. Ashley-Rollman[†], Seth Copen Goldstein[†], Ricardo Rocha[‡], Frank Pfenning[†]

[†]Carnegie Mellon University, Pittsburgh, PA 15213
{fmfernan,mpa,seth,fp}@cs.cmu.edu
[‡]CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
ricroc@dcc.fc.up.pt

Abstract

In the Claytronics project, we have used Meld, a logic programming language suitable for writing scalable and concise distributed programs for ensembles. Meld allows declarative code to be compiled into distributed code that can be executed in thousands of computing units. We are now using Meld to program more traditional algorithms that run on multicore machines. We made several modifications to the core language, to the compiler and to the runtime system to take advantage of the characteristics of the target architecture. Our experimental results show that the new compiler and runtime system are capable of exploiting implicit parallelism in programs such as graph algorithms, neural networks and belief propagation algorithms.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Logic Programming, Parallel Programming

1. Introduction

In the context of the Claytronics project [1], a new programming language, called Meld [2], was created to declaratively program *ensembles* of modular robots. Ensembles are distributed systems composed of thousands or millions of processing units that can interact with each other and with the world. Meld makes ensemble programming easier by employing a declarative style of programming. Meld programs are compiled to fully distributed code and then run individually on each node.

An ensemble is a highly dynamic and massively distributed system, where the topology of the processing units can change very often and faulty units are relatively common. Multicore processors differ from ensembles by having a static topology, lower fault rates and by having significantly fewer computing units. Starting from the point of view of ensembles, we made several modifications to the Meld language to make it suitable for multicores.

Meld is a logic programming language heavily based on Datalog and uses the concepts of declarative networking that originated with P2 [3]. We use *structural facts* to represent the graph topology and to implicitly allow messaging between nodes through rule splitting so that rules can be executed in a distributed fashion.

In a multicore setting, instead of seeing the distributed system as a graph of processing units as in Claytronics, we see the distributed system as a graph data structure, with several threads or cores working on nodes. In the extreme case, if we have as many cores as nodes in the graph then we get the 1-to-1 mapping of ensembles. We have implemented several programs in Meld such as graph algorithms and machine learning problems using this analogy. For example, in a neural network program, each neuron is a node in the graph data structure and connections between neurons are represented as structural facts. We implemented a compiler and a virtual machine that runs on multicore architectures. Our preliminary experimental results show impressive scalability results.

The rest of the paper is organized as follows. First, we present the syntax and semantics of Meld, followed by how distribution is achieved through localization. Next, we give an overview of execution strategies that our virtual machine employs and then we present scalability results for those strategies. Finally, we close this paper by discussing current directions of this work.

2. Meld Language

Meld is a forward-chaining logic programming language based on Datalog. Each program corresponds to a set of logical rules that work on a database of facts. Facts are an association between a *predicate* and a tuple of values. Rules may be decomposed into *head* and *body*. The body indicates the prerequisites needed in order to instantiate the head. They may include facts from the database, expression constraints and variable assignments.

Meld programs are evaluated in a bottom-up fashion, that is, we consider the set of rules and the current database and look for rules that can be applied to create new facts. This process works by finding some substitution that satisfies the rule's body constraints and facts, and then instantiating the head with this substitution so that new facts are created and added to the database.

All predicates (and thus facts) are typed. Meld includes basic types such as floating point numbers and integers, node addresses and lists of basic types. We have four types of facts: *structural facts* to describe an underlying graph model; *computation facts* to be used as computational state; *action facts* which perform some kind of action outside the logical world; and *sensing facts* to sense and represent the outside world. From these, only computation and structural facts are stored in the database. In the context of ensembles, action facts force robots to perform actions in the real world and sensing facts give information about the world (e.g., read ambient temperature). In the case of multicores, action facts perform input/output and sensing facts detect when computation has finished.

2.1 Distribution

Like in the P2 system [3], facts are distributed across *nodes* by forcing the first argument of each predicate to be typed as a node address. This, in conjunction with structural facts, gives rise to an underlying graph model where nodes store facts and structural facts describe the connections between nodes.

Since facts are dispersed through the graph structure, we restrict rules to *local rules* and *link-restricted rules*. Local rules are rules where facts refer to the same node. Link-restricted rules are rules where the facts refer to different nodes but those nodes must be connected using of structural facts (therefore disallowing reference to arbitrary nodes). We want every rule to be evaluated locally, therefore we need to split link-restricted rules into smaller rules that can be evaluated locally, in a process called *localization*. Such smaller rules are called *communication rules*, because they only partially match the original body and such partial results must be sent to a connected node, where the rest of the original body can be matched next.

For instance, in this program that computes connectivity between nodes, the second path rule is a link-restricted rule:

```
type path(node, node).
type route edge(node, node).

path(A, B) :- edge(A, B).
path(A, B) :- edge(A, C), path(C, B).
```

Using localization, we get two new rules, marked here with "@comm". Note that communication rules force the transmission of instantiated head facts from the node where the rule was matched to other nodes. Localization is thus the crucial aspect that makes Meld distributed.

```
path(A, B) :- edge(A, B).
path(A, B) :- __remote(A, B).

__edge(X, Y)@comm :- edge(Y, X).
__remote(A, B)@comm :- __edge(C, A), path(C, B).
```

2.2 Aggregates

While Meld does not allow negation, it supports the concept of *aggregates*. Aggregates can be used to combine several facts into one aggregated fact by applying a function to a certain argument of the predicate. Meld supports functions such as min, max and sum.

Due to the distributed nature of Meld, it is sometimes difficult to know when it is safe to generate an aggregated value. Our compiler classifies aggregates into two classes: *safe aggregates* and *unsafe aggregates*. Safe aggregates include aggregates that the compiler knows in which situations they can be generated safely because we have all the required values. These include aggregates that only depend on local rules, called *local aggregates*, and *neighborhood aggregates*. Neighborhood aggregates depend on source code annotations that tell the compiler that it is safe to generate the aggregate when we have all relevant facts produced by all neighboring nodes. For example, in a PageRank computation we are interested in the sum of the rank of all neighbors, therefore when we have a rank for each neighbor then it is safe to compute the sum.

Some aggregates are unsafe because there is not enough local information available in the node to make a decision. In other situations, we may have recursive rules and stratification is impossible just by doing syntactical analysis of the source code. Unsafe aggregates are thus problematic since they may require re-computation and global synchronization.

2.3 Deletion

Because unsafe aggregates may be computed with the wrong collection of facts, other facts that depend on wrong aggregates may be computed. If the original collection of facts changes during execution, we delete every fact that depends on the invalid aggregate and then use the newly computed aggregate. This process is known as *deletion* and works like the creation of new facts, except we mark derivations as deletions.

3. Parallel Execution

For ensembles, a node in the graph is a single computing unit that performs communication with other nodes. In the multicore case, the node is just part of a graph data structure and each core performs computation on multiple nodes. Our execution model thus consists in a set of *workers* and a mapping between workers and nodes in the graph. In our implementation, a worker is a POSIX thread.

At the node level, each node has a queue of new facts to process and computation proceeds by taking a fact F from this queue. Next, we select all the *candidate rules* where F may match and then we try such rules with F and the database of facts. When a rule succeeds, we instantiate all the head facts and we add them to the node's queue. With communication rules, facts generated at other nodes are sent to the corresponding node. This pipelined execution increases throughput by allowing immediate processing of facts.

In order to make unsafe aggregates deterministic in a parallel setting, we introduce the concept of *computation round*. In each round, we derive facts and safe aggregates and between rounds we derive unsafe aggregates. Each round proceeds as follows:

- Workers process all regular facts and safe aggregates;
- When worker has no work to do, it enters into the *IDLE* state and waits for other workers;
- Once all workers are *IDLE*, they synchronize using a *termination barrier*;
- Unsafe aggregates are generated. If unsafe aggregates were previously computed incorrectly we derive deletion derivations;
- If new facts are generated, they are inserted into the node's queue to be processed during the next round;
- If no new facts were generated, then execution terminates;

3.1 Schedulers

The details of how nodes are assigned to workers is one aspect of the runtime system that has great impact on parallel performance. Our virtual machine implements three different *schedulers* that use radically different approaches to node scheduling.

3.1.1 Static Division (SD)

The most obvious way to schedule work is to statically divide the nodes in the graph between the available workers. However, this division must be done in such a way that it reduces inter-worker communication and increases intra-worker locality. We explore this in Section 3.2.

In the SD scheduler, each worker W has a *queue of active nodes* where *active nodes* owned by W are pushed into to be processed. An active node is characterized by having new facts on its queue ready to be processed. Whenever a worker needs to fetch work, it looks into the queue of active nodes and pops an active node N. Then, the worker processes facts in the node's queue until the queue is empty. A node becomes an *inactive node* when it has no more facts to be processed.

Inactive nodes may become active nodes in two situations: (1) if a fact is generated by a local rule and (2) if a communication rule forces a new fact to be sent to the inactive node. When the node is made active, it goes back to the queue of active nodes. Event (1) is only made possible by the worker that owns the node, while event (2) can be made possible by either the owner or some other worker, therefore each queue of active nodes needs synchronization during the push operation.

Finally, if the queue of active nodes is empty, the corresponding worker enters into the *IDLE* state, where it waits for the other workers to become *IDLE* or for one of its nodes to become active.

3.1.2 Dynamic Division (DD)

Similarly to SD, in DD we also start with a static division of nodes across workers, however we allow nodes to change owners through work stealing to improve load balancing. Each worker has the same queue of active nodes as before and processes active nodes in the exact same way. However, when workers enter into the *IDLE* state, a worker may select a random worker to steal nodes from.

The disadvantage of DD when compared to SD, is that the queue of active nodes now needs to be synchronized both during push and pop operations.

3.1.3 Dynamic Division Without Ownership (DDWO)

DDWO can be seen as an extreme instance of the DD scheduler. Here, instead of workers having a more or less stable set of nodes, nodes have no owner at all. There is only one queue of active nodes from where all the workers can pop and push nodes. There is no work stealing, since workers can perform computation in any node of the graph, thus decreasing intra-worker locality. However, the costs of selecting a random worker to steal work from are nonexistent and workers can start work immediately as soon as there is some active node in the system.

3.2 Graph Clustering

The method used to distribute nodes in the schedulers SD and DD is important. In order to reduce communication costs between threads, we need to avoid firing communication rules between nodes that are located in different threads. Ideally, we should group closer nodes into clusters and then assign a cluster to a worker.

All the structural facts that describe the graph structure need to be written as Meld axioms so that the compiler can reconstruct the graph and build the corresponding clusters. Since the number of workers used is arbitrary, we simply give each node a specific ordering address from 0 to N, where N is the number of nodes in the graph. This allows an efficient computation of each node's worker just by looking at its address. For example, to determine if node n was assigned to worker W, we compare W to $\mathtt{min}(n/(N/T), N-1)$, where T is the number of workers.

The clustering method used is the breadth-first algorithm. We pick an arbitrary node in the graph and assign it the address 0, then we mark all its neighbors to be processed next to define their ordering as $1, ..., N-1$. This is done recursively until all nodes have been visited.

4. Experimental Results

We have implemented four main algorithms[1]: Belief Propagation (Fig. 1), which performs image de-noising using naive loopy belief propagation; Neural Network (Fig. 2), which trains a neural network using back-propagation to recognize some letters from the alphabet; All-Pairs (Fig. 3), to compute the shortest path between all pairs of nodes in the graph representing the 500 biggest airports in the USA[2]; and PageRank (Fig. 4), that computes the rank of webpages[3].

The Belief Propagation program shows an almost linear speedup for the three schedulers. In this program, each node exchanges messages with their neighbors and each node does an equal amount of work. Moreover, the graph structure is a matrix, therefore the computation can be efficiently distributed statically, which makes SD and DD perform very similarly. In all the others benchmarks, SD does not perform very well since the graph structure is not regular.

In the Neural Network program, DDWO shows the biggest relative advantage over the other schedulers. The computational pattern in this program may be decomposed into two phases: (1) training data is sent from the input layer to the output layer, and (2) weight adjustments between neurons is then performed from the output layer to the input layer. If we also take into account that the number of neurons is small, graph clustering can greatly affect SD and DD. As the number of cores used increases, the number of neurons assigned per core gets smaller, which makes DDWO more efficient than DD, since we remove the costs of work stealing.

For the All-Pairs program, we observe slightly lower speedups than the other benchmarks. However, this is expected since this program is the only one that uses unsafe aggregates, therefore it requires more barrier operations between several computation rounds. In the PageRank program, only SD performs badly, but this is due to the irregular nature of the graph used.

[1] All programs are accessible here: `https://github.com/flavioc/meld/tree/master/benchs/progs/sources/`

[2] Dataset obtained from `http://toreopsahl.com/datasets/`

[3] Dataset obtained from `http://www.cs.toronto.edu/~tsap/experiments/download/download.html`

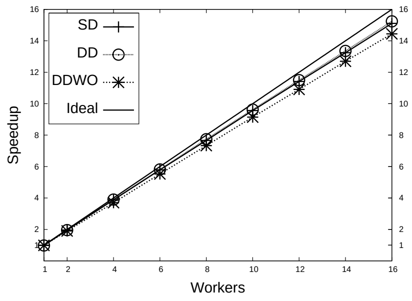

Figure 1: Scalability for the Belief Propagation program.

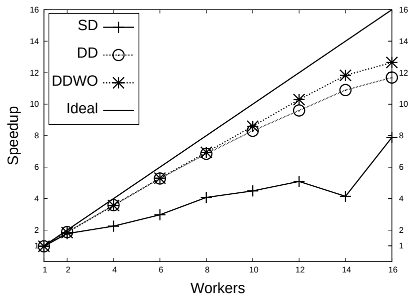

Figure 2: Scalability for the Neural Network program.

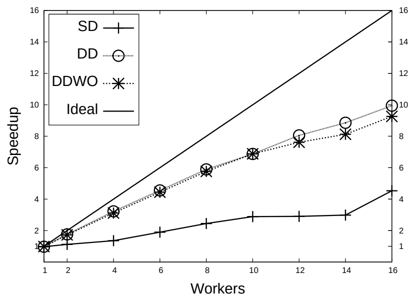

Figure 3: Scalability for the All Pairs program.

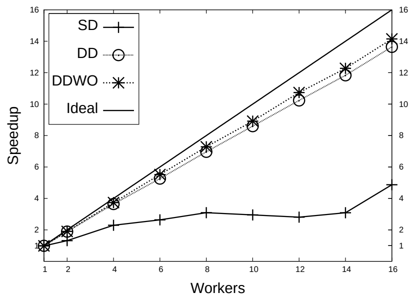

Figure 4: Scalability for the PageRank program.

5. Current Work

One of the issues in the current Meld design is the problem of state. We have been able to solve some of those problems by detecting sets of rules with a *stage argument*, that is, rules where the second argument of each predicate corresponds to the current iteration level. For example, all the programs in the previous section, except the All-Pairs, perform iterative refinements by indexing facts by the iteration number. Inspired by XY-stratification [4], we detect XY cliques and then insert delete operations to delete facts in older iterations, thus reducing the memory usage of such programs.

While the previous method solves memory issues, it is still awkward to express stateful programs in Meld. We are currently investigating how to model state directly in the language by leaving the limitations of classical logic and moving to a more expressive logical foundation. Our current direction is to use linear logic [5]. By using the resource interpretation of linear logic in a distributed context, we hope to implement new classes of algorithms in a declarative high-level manner. A (non-distributed) example of a logic programming language with a bottom-up semantics is provided by LolliMon [6]. LolliMon also integrates backward chaining for local, deterministic programs, which may make sense in our context.

We have also considered using temporal logic as our logical foundation. The language Dedalus [7], for instance, uses time as a source of mutability, where facts not derived in the next time step are considered as being ”deleted”. A disadvantage of the temporal approach is the so-called *frame problem*: when time changes, we need to forward everything that is supposed to remain unchanged to the next moment in time by using *frame axioms*. This can be cumbersome and can make programs much less modular. In linear logic, the frame axioms are unnecessary because the corresponding property is an intrinsic property of the logic itself.

```
type route edge(node, node).
type source(node).
type sink(node, int).
type linear pluggedIn(node, node).
type linear unplugged(node).
type linear load(node, float).

unplugged(Sink) :- !sink(Sink, _).
load(Source, 0) :- !source(Source).

pluggedIn(Sink, Source),
load(Source, OldAmt + Amt) :-
   unplugged(Sink),
   !sink(Sink, Amt),
   !edge(Sink, Source),
   !source(Source),
   load(Source, OldAmt).

pluggedIn(Sink, NewSource),
load(OldSource, OldSourceAmt - Amt),
load(NewSource, NewSourceAmt + Amt) :-
   pluggedIn(Sink, OldSource),
   !edge(Sink, NewSource),
   load(OldSource, OldSourceAmt),
   OldSourceAmt > 1,
   load(NewSource, NewSourceAmt),
   !sink(Sink, Amt),
   NewSourceAmt + Amt < OldSourceAmt || rand().

terminate() :- proved(pluggedIn) > MAX_ITERATIONS.
```

Figure 5: Power Grid problem in Linear Meld.

A potential application of linear logic are randomized and approximation algorithms. In approximation algorithms, we want an approximate result that is much faster to compute than the optimal result. Some examples are the asynchronous PageRank [8] and certain classes of belief propagation algorithms such as SplashBP [9]. An example of a randomized algorithm is presented in Fig. 5. This program solves the power grid problem, where we have a set of sinks and a set of sources and we want to connect each sink to a source in such a way that no source is overloaded. We first derive a linear fact `unplugged` for each sink and `load` fact for each source to mark the load as 0. Then, the third rule randomly picks a source for each sink, thus consuming the initial `unplugged` fact. The fourth rule selects an overloaded source and switches one of the connected sinks to a different source in order to reduce the source's load. This rule is applied several times and the program will converge to a state where the load is equally distributed. Finally, we use the `proved` sensing fact to detect if the `pluggedIn` fact has been derived enough times already (the program could run forever) and a `terminate` action fact may be generated to halt execution.

Our intuition tells us that linear logic is suitable for these kinds of problems. However, many issues are still left to be investigated. For example, how can we stratify linear logic programs in the presence of aggregates? And how can resources navigate efficiently through the graph structure?

6. Conclusions

We have presented Meld, a bottom-up logic programming language inspired by the principles of declarative networking, where rules are restricted to enable distribution in an underlying graph model. Meld has been applied successfully in extreme distributed systems such as ensembles. From this starting point, we have modified Meld for multicore architectures. The basic idea is to view execution as multiple threads performing computation on a graph data structure.

We have implemented algorithms such as Belief Propagation, Neural Networks and PageRank and have measured the scalability of our runtime system. The experimental results show impressive scalability performance even when using a relatively large number of cores. Currently, we are trying to extend Meld with linear logic in order to increase the applicability of the language for more interesting and real-world programs, particularly randomized and approximation algorithms.

Acknowledgments

This work has been partially supported by the FCT research projects HORUS (PTDC/EIA-EIA/100897/2008) and LEAP (PTDC/EIA-CCO/112158/2009).

References

[1] S. C. Goldstein, J. D. Campbell, and T. C. Mowry, “Programmable matter,” *IEEE Computer*, vol. 38, no. 6, pp. 99–101, June 2005.

[2] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell, “A language for large ensembles of independently executing nodes,” in *Proceedings of the International Conference on Logic Programming (ICLP ’09)*, July 2009.

[3] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, “Declarative networking: language, execution and optimization,” in *Proc. of the 2006 ACM SIGMOD int’l conf. on Management of data*. New York, NY, USA: ACM Press, 2006, pp. 97–108.

[4] C. Zaniolo, N. Arni, and K. Ong, “Negation and aggregates in recursive rules: the LDL++ approach,” in *Deductive and Object-Oriented Databases*, 1993, pp. 204–221.

[5] J.-Y. Girard, “Linear logic,” *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–101, 1987.

[6] P. López, F. Pfenning, J. Polakow, and K. Watkins, “Monadic concurrent linear logic programming,” in *Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming (PPDP’05)*, A.Felty, Ed. Lisbon, Portugal: ACM Press, Jul. 2005, pp. 35–46.

[7] P. Alvaro, W. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. C. Sears, “Dedalus: Datalog in time and space,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-173, Dec 2009.

[8] B. Lubachevsky and D. Mitra, “A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius,” *J. ACM*, vol. 33, pp. 130–150, January 1986.

[9] J. Gonzalez, Y. Low, and C. Guestrin, “Residual splash for optimally parallelizing belief propagation,” in *In Artificial Intelligence and Statistics (AISTATS)*, Clearwater Beach, Florida, April 2009.