Collection Oriented Programming

Guy Blelloch
Carnegie Mellon University

A bit of History

The Connection Machine (1985)

- 64K single-bit processors
- Richard Feynman, Steve Wolfram, Jack Schwartz, Guy Steele

Realization

- Traditional parallel constructs (e.g. cobegin), were not adequate
- Several data parallel languages developed:
  C*, *Lisp, CM-Lisp, CM-fortran

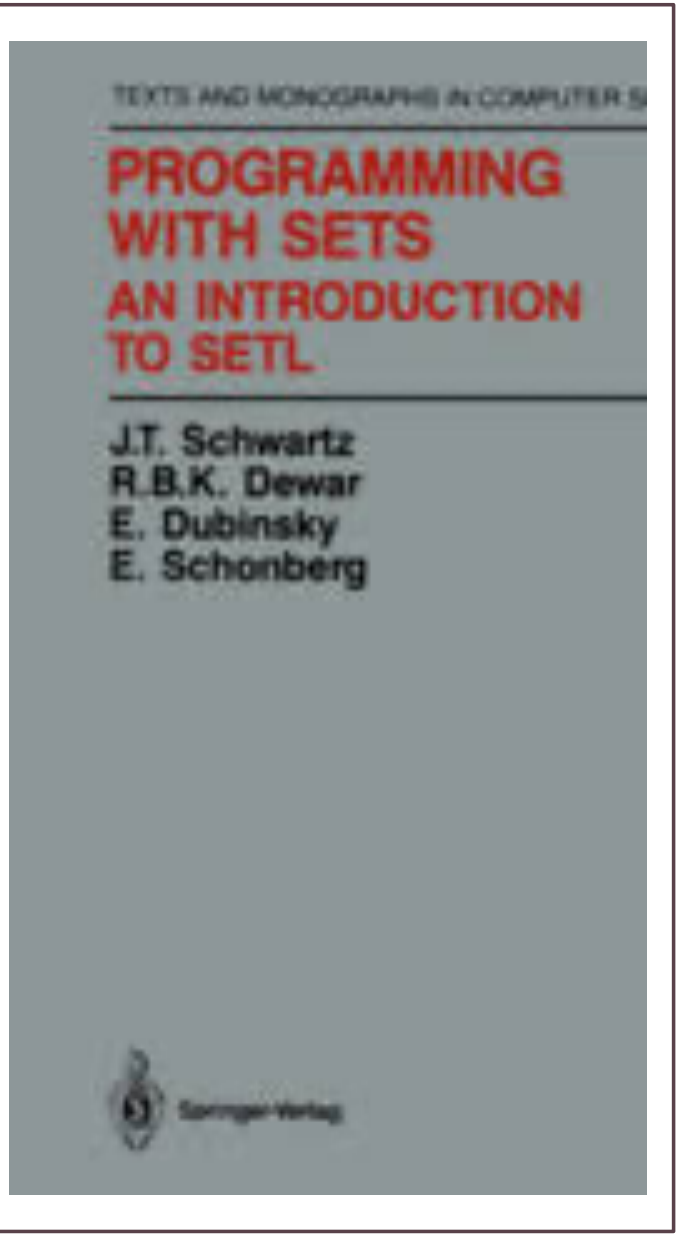
Collection Oriented Programming

Realization that similar to some existing “very-high-level” languages: APL (1966), SETL (1968), FP (1977), SQL (relational algebra, 1970)

- Work on collections, think ”parallel”
- Largely functional style programming

Many variants of core ideas:

- Nested vs. Flat
- Implicit vs. explicit map
- Collections types: sequence, map, set, relation
- Collection operations: map, reduce, filter, ….



Many languages since:

Nesl, Hadoop (map-reduce), Spark, LINQ, Matlab, Python (comprehensions), clojure, R, parallel Haskell, Lua, …, ???

Implicit vs Explicit

APL, Fortran90: A + 1

CMLisp: (α+ A α1)

SETL, Nesl : {a + 1 : a in A} -- “comprehensions”

Implicit is particularly bad for nested collections:

  e.g. reverse([[1, 2], [3,4]]) = ?

but even for flat collections, with overloading:

  e.g. square([2,3]).

Implicit: APL, Matlab, Fortran90, C*, R,

Explicit: Nesl, CMLisp, SETL, Hadoop, Spark,

Type of Collections

Sequences : Nesl, Spark, Parallel Haskell, R

Arrays : APL, Matlab, Fortran90, R

Sets/Maps: CM-lisp, SETL, python

Relations: SQL, Paralation-Lisp, …

Others? :

Nested vs Flat:

Nested Collections: Elements can be collections:

e.g.: [[2,3,4], [5,6], [8,9]]

- **Flat**: APL, SQL, C*, *Lisp, Hadoop, Spark
- **Nested**: APL2, FP, CM-Lisp, Nesl, parallel Haskell

Nested parallelism:

```
function quicksort(S) =
 if (#S <= 1) then S
 else let a = S[rand(#S)];
          S1 = {e in S | e < a}
          S3 = {e in S | e > a};
          R = {quicksort(v) : v in [S1, S2]};
      in R[0] ++ [a] ++ R[1];
```

Some languages (e.g. SQL) support ”nested parallelism” without nested tables,

e.g: `{sum({o.value : o in orders | o.nation = n}`
`: n in nations}`

Operations

Basic: Map, Reduce, Filter (pack), Tabulate, Scan
Ordering: Sort, Merge, Kth-Smallest
Grouping by: GroupBy (collect), groupByReduce,
Batch updates: Inject, multi-insert, …
Nested: Flatten, Split, partition
Sets/Tables: Union, Intersect, Difference
Relations/Dataframes: join, semi-join
Matrices: Multiply, inverse
Strings: tokens, toString,
Other: remove duplicates, append, subseq

Some Key Operations : Scan

Scan(f, s, a) -> [s, f(s,a[0]), f(f(s,a[0]),a[1]), …

e.g. scan('+, 0, [1,1,1,1]) -> [0, 1, 2, 3]

- f must be associative

Why Important:

- needed to capture "loop carried dependences"

Appications:

- Filter
- Partitioning a sequence
- Carry propagation
- Finite state automata
- Linear recurrences (x_i = a_i * x_{i-1} + b_i)
- Tokenizing a string
- Flattening nested parallelism (segmented scans)

Parallel Filter

```
{e in S | e < a};
```

S = [2, 1, 4, 0, 3, 1, 5, 7]

F = S < 4 = [1, 1, 0, 1, 1, 1, 0, 0]

I = addscan(F) = [0, 1, 2, 2, 3, 4, 5, 5]

where F

R[I] = S = [2, 1, 0, 3, 1]

Each element gets sum of previous elements. Seems sequential?

Some Key Operations: GroupBy

groupBy([(3,a), (5,b), (3,c), (1,d), (3,e)])

-> [(3, [a,c,e]), (5, [b]), (3, [e])]

- Called ”collect” in some languages (e.g. Nesl)
- groupByReduce is also useful

Why important:

- Easy to do by hand sequentially (e.g. hash tables, linked lists), hard to do by hand efficiently in parallel.

Applications:

- Hadoop Map Reduce
- Histograms
- Counting by type
- Very wide variety of database queries (~50% of TCP-H)
- Indexing
- Bucketing

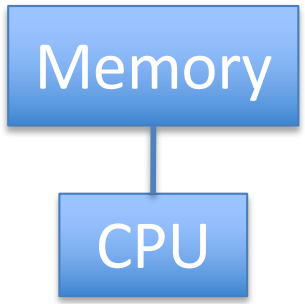
What about costs?

- How expensive is my code (roughly)?
- Is algorithm A or B better?
- Why is my code taking 10 hours to run, and when will it finish?

Not talking about precise times, but rather relative times, and order-of-magnitude times.

In the Sequential World

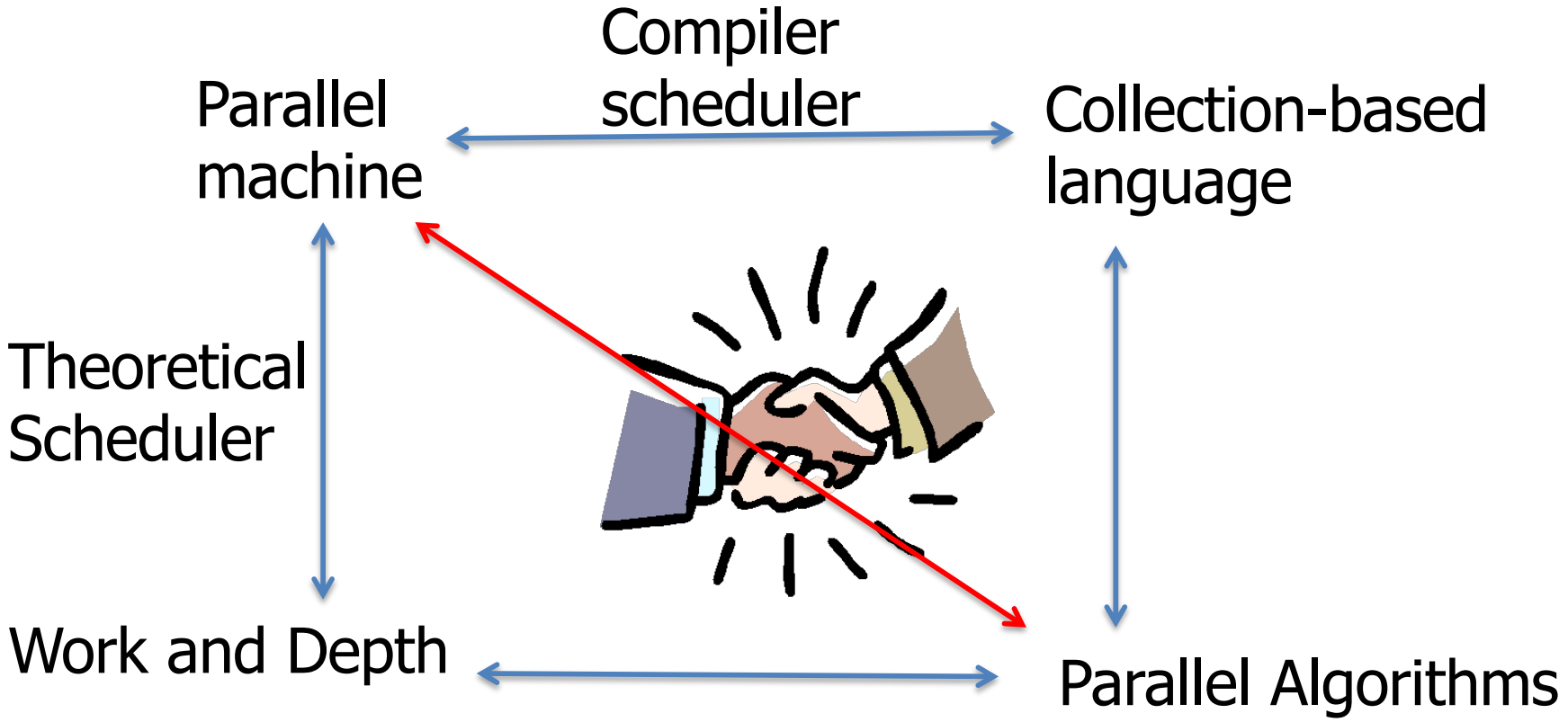
Programming Languages
C, Java, Pascal, …



Random Access
Machine (RAM)

Algorithms and Analysis
-Mergesort = O(n log n)
-BFS = O(n)
-MxM = $O(n^{2.376})$
-…

In the Parallel World



Work = total number of instructions

Depth = Longest Dependence Path

Scan code

```
function scan(g,s,A) =
if (#A <= 1) then [s]
else let
  sums = {g(A[2*i], A[2*i+1]) : i in [0:#a/2]};
  evens = addscan(sums);
  odds = {g(evens[i] + A[2*i]) : i in [0:#a/2]};
in interleave(evens,odds);
```

W(n) = W(n/2) + O(n) = O(n)

D(n) = D(n/2) + O(1) = O(log n)

Nesl in some more detail.

ParlayLib

A library for C++ that supports “collection oriented programming”.

- Shared memory machines
- Sequence data type (similar to std::vector, but much better support for parallelism).
- All the standard operations: map, reduce, scan, groupBy, sort, merge, partition, tokens, ….
  All are optimized.
- Allows nesting
- Integrates with Ligra (graph processing), and PAM (sets and tables)
- Only possible since C++11 due to lambda (makes heavy use of)
- Has its own work-stealing scheduler, but can also work with Cilk scheduler
- Supports delayed sequences – important for efficiency for supporting “loop fusion”.
- Still most functional style, but within C++

Delayed Sequences

```
A = reduce(map(S, [] (int a) {a + 1;}))
```

Would create an intermediate sequence, which is passed to reduce. Instead:

```
A = reduce(delayed_map(S, [] (int a) {a + 1;}));
```

This is about 3x faster, and saves memory. Delayed sequences can be used anywhere an immutable sequence can be used.

Example : Primes

```
sequence<long> prime_sieve(long n) {
  if (n < 2) return sequence<long>();
  long sqrt = sqrt(n);
  auto primes_sqrt = prime_sieve(sqrt);      // recursive call
  sequence<bool> flags(n+1, true);
  flags[0] = flags[1] = false;
  parallel_for(0, primes_sqrt.size(), [&] (long i) {
        long prime = primes_sqrt[i];
        parallel_for(2, n/prime + 1, [&] (long j) {
            flags[prime * j] = false; }); });
  return pack<long>(iota(n+1), flags);}
```

Iota(n) is a lazy sequence of the integers, [0, 1, …, n-1].

Total work is O(n log log n), depth is O(log n).

Example: Counting Words

```
auto word_counts(sequence<char> str) {
 sequence<sequence<char>> words = tokens(str, is_space);
 return group_by_and_count(words);}
```

```
e.g.
  "this is a test of a test"
  -> [("this",1), ("is",1), ("test",2), ("of",1), ("a",2)]
```

Uses “small string optimization”. i.e. if a sequence fits into 15 bytes, then don’t allocate space for it.

Above code can process 1G string in .5 seconds on 72 cores. This is 50x faster than best sequential code, which takes about 25 seconds.

Example: Breadth First Search

```
auto BFS(graph const &g, vertex start) {
  long n = g.num_vertices();
  sequence<vertex> Parents(n, n);
  update = [&] (vertex s, vertex d) {
     return atomic_compare_and_swap(&Parents[d], n , s); };
  cond = [&] (vertex d) { return (Parents[d] == n); }

  Parents[start] = start;
  vertex_subset frontier(start); //creates initial frontier
  long levels = 0, visited = 0;
  while(!frontier.is_empty()) { //loop until frontier is empty
    visited += frontier.size();
    levels++;
    frontier = edge_map(g, frontier, update, cond);}
  return levels;}
```

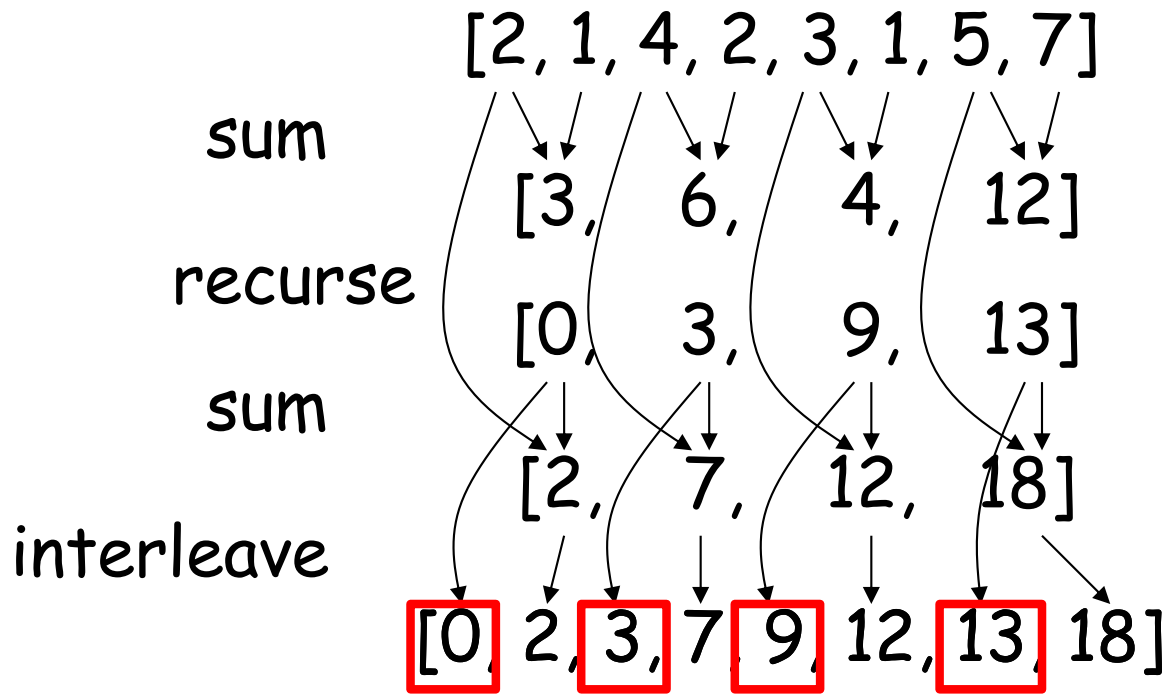
Uses ligra interface for graphs, which is built on parlaylib.

Very fast (almost fastest available). Applied to graphs with more than 100Billion edges.

Conclusion

Collection-oriented programming is fun and fast.

Scan


[2, 1, 4, 2, 3, 1, 5, 7]
sum
[3, 6, 4, 12]
recurse
[0, 3, 9, 13]
sum
[2, 7, 12, 18]
interleave
[0, 2, 3, 7, 9, 12, 13, 18]