Algorithm Engineering with PRAM Algorithms

Bernard M.E. Moret
moret@cs.unm.edu

Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
Measuring and Parallel Speedup

- Measuring Parallel Execution
- Algorithm Engineering for Parallel Algorithms (briefly)
- Message-Passing Computations (briefly)
- Shared-Memory Computations: Can We Finally Put to Use 30 Years of Research in PRAM Algorithms?
What to Measure

In a parallel execution, measure *wallclock time*!
- takes into account all processors (all must finish!) as well as system overhead
- but requires dedicated system and "warm-up" runs to allow threads to migrate and stabilize on their processors
How to Compare

- Compare to the best *sequential* algorithm (though it will make the parallel code look bad when run on few CPUs).
How to Compare

- Compare to the best *sequential* algorithm (though it will make the parallel code look bad when run on few CPUs).
- Compare speedup against (a cleaned-up version of) the parallel code run on a single processor—gives a better idea of scaling.

Average over lots of instances of various types to avoid data biases that may favor specific numbers of CPUs.

In an MP environment, ensure the single-CPU version has enough memory.
How to Compare

- Compare to the best *sequential* algorithm (though it will make the parallel code look bad when run on few CPUs).
- Compare speedup against (a cleaned-up version of) the parallel code run on a single processor—gives a better idea of scaling.
- Average over lots of instances of various types to avoid data biases that may favor specific numbers of CPUs.
How to Compare

- Compare to the best *sequential* algorithm (though it will make the parallel code look bad when run on few CPUs).
- Compare speedup against (a cleaned-up version of) the parallel code run on a single processor—gives a better idea of scaling.
- Average over lots of instances of various types to avoid data biases that may favor specific numbers of CPUs.
- In an MP environment, ensure the single-CPU version has enough memory.
Parallel Algorithm Engineering

- Does not exist...
Parallel Algorithm Engineering

- Does not exist...
- Can reuse techniques of algorithm engineering for sequential code—most gains will be found in the sequential part.

In distributed memory (or virtual shared-memory) systems, need to focus on transmission of information.
In true shared-memory systems, need to focus on synchronization and cache utilization.
Memory hierarchy is deeper and more complex.
Parallel Algorithm Engineering

- Does not exist...
- Can reuse techniques of algorithm engineering for sequential code—most gains will be found in the sequential part.
- In distributed memory (or virtual shared-memory) systems, need to focus on transmission of information.
Parallel Algorithm Engineering

- Does not exist.
- Can reuse techniques of algorithm engineering for sequential code—most gains will be found in the sequential part.
- In distributed memory (or virtual shared-memory) systems, need to focus on transmission of information.
- In true shared-memory systems, need to focus on synchronization and cache utilization.
Parallel Algorithm Engineering

- Does not exist. . .
- Can reuse techniques of algorithm engineering for sequential code—most gains will be found in the sequential part.
- In distributed memory (or virtual shared-memory) systems, need to focus on transmission of information.
- In true shared-memory systems, need to focus on synchronization and cache utilization.
- Memory hierarchy is deeper and more complex.
Message-Passing Systems

- Communication is the bottleneck.
- Load-balancing (data migration) is a problem.
- PRAM algorithms have little in common with efficient message-passing code.
- BSP or virtual shared-memory cannot overcome the basic facts, but they do allow the programmer to use PRAM-style algorithms, although with a mandatory slowdown.
Shared-Memory Programming

- Symmetric Multiprocessor (SMP) architectures
- Uniform-Memory-Access (UMA) SMPs
- PRAMs vs. UMA SMPs
- A UMA SMP programming model
- Our running example: Ear decomposition
- Implementation
- Experimental setup
- Experimental results
Symmetric Multiprocessor Architectures

- SMP: several processors integrated into one machine, using shared-memory with concurrent read.
- Small SMPs ($\leq 4$ CPUs) are commodity items.
- Larger SMPs are the choice for servers (large memory, high throughput, redundancy)
- Largest SMP is the Sun Starcat with 105 CPUs and 210GB memory. Clusters of large SMPs (IBM Netfinity) form the next-generation terascale architecture.
Uniform-Memory-Access SMPs

- Early SMPs (SGI Origin) had variable memory access times, so efficiency depended on keeping items local to each processor—just as in message-passing style.
- Sun and IBM Netfinity SMPs have true uniform-access shared-memory.
- UMA SMPs open the way to efficient parallel computing for complex, irregular problems.
Memory Access on Sun SMPs

- Large SMPs have a deep memory hierarchy—10ns to 600ns in the Sun E10K and 3ns to 250ns in the Starcat.
- The 250ns worst case is still \textit{two orders of magnitude} faster than message-passing.
- We tested memory access from a single processor to the full addressable memory on our lab’s E4500 and on the San Diego Supercomputing Center’s E10K. Results clearly show cache sizes (L1 and L2).
Sun Enterprise Results

E 4500

E 10,000
PRAMs and UMA SMPs

- Similarities are striking:
  - all processors can access any memory location
  - concurrent read capability
  - no need for messages

- Two major differences remain:
  - few processors: up to 100, not $O(n^{O(1)})$
  - no lockstep synchronization: one must use software barriers
Validating the Model

- A methodology for designing practical shared-memory algorithms on UMA shared-memory machines,
- A fast and scalable shared-memory implementation of ear decomposition demonstrating the first significant parallel speedup for this class of problems.
- An example of experimental performance analysis for a nontrivial parallel implementation.
Linear Speedups on Irregular Graph Problems

Problem Size ($\log_2 n$)

Time (ms)

graph type fixed, graph size and # processors variable
Linear Speedups on Irregular Graph Problems

Size fixed (8192), # processors and types of sparse graphs variable
Methodology: PRAM to SMP

- how to partition the tasks (and data) among the very limited number of processors available
- how to optimize the use of caches
- how to minimize the work spent in synchronization (barrier calls)

Good data and task partitioning, coupled with cache-sensitive coding; barriers are trickier, because of the need to flush caches.
Methodology: Complexity Model

matching complexity model (Helman and Jájá 99) captures contention at processors and contiguous vs. noncontiguous memory accesses.

analysis reports triple \((M_A, M_E, T_C)\), where

- \(M_A\): number of noncontiguous accesses to main memory
- \(M_E\): amount of data exchanged with main memory
- \(T_C\): upper bound on the local computational work
Ear Decomposition

- The ear decomposition of an undirected graph $G = (V, E)$ is a partition of $E$ into an ordered collection of simple paths such that each endpoint of a path is contained in an earlier path, but no interior point of a path is contained in any other path.

- Useful in its own right (structural analysis) and a good decomposition step for complex graph algorithms (in lieu of dfs, for instance).

- Has a simple linear-time sequential algorithm based on dfs.
Ear Decomposition

Challenging to implement in parallel
- Requires prefix sum, prefix product, list-ranking, least-common ancestor, tree raking, pipelined sorting, and spanning tree construction.
- Except for prefix computation, none of these tasks has been shown to scale on message-passing architectures.
Ear Decomposition: An Example
find a spanning tree
root tree and compute level of each node
each non-tree edge corresponds to a distinct ear, so find least common ancestor (LCA) of endpoints of each non-tree edge and label the edge by the level of its LCA
label each tree edge by choosing the smallest label of any non-tree edge whose cycle contains it
labels are sorted
SMP Implementation and Analysis

- find a spanning tree by successive grafting; each takes $O((m + n)/p)$ time, so total time is $T(n, p) = O(1, \frac{n}{p}, ((m + n)/p) \log n)$
- root the tree and compute the level of each node with the Euler tour technique in $O(n/p)$ time and $O(n/p)$ noncontiguous memory accesses
- computing the edge labels takes $O(n/p)$ time and $O(n/p)$ noncontiguous memory accesses
- total algorithm runs in $O\left(\frac{n}{p}, \frac{n}{p}, \frac{m+n}{p} \log n\right)$
Experimental Setup

- **Input**: sparse graphs (regular and not, planar and not) from $2^8$ to $2^{18}$ vertices.
- **Environment**: the SMP node library of SIMPLE (by D. Bader), built on top of POSIX threads.
- **Machines**: our laboratory’s E4500 (14 CPUS, 14GB memory) and the SDSC’s E10K (64 CPUS, 64GB memory), both with 450MHz Sparc II processors, each with 16KB on-chip direct-mapped L1 cache and 8MB L2 cache.
Classes of Input Graphs

- Regular 4-connected mesh of $\left\lfloor \sqrt{n} \right\rfloor \times \left\lfloor \sqrt{n} \right\rfloor$ vertices
- Regular triangulated mesh, obtained from above by adding an edge connecting a vertex to its down-and-right neighbor, if any
- Random planar graphs of varying density, from very sparse (few, if any, cycles) to nearly triangulated
- Constrained Delaunay triangulation of $n$ random points in the unit square
Experimental Results

We plot efficiency: the ratio of the measured speedup to the number of processors used (which is the ideal speedup).

A perfect implementation of a perfect parallel algorithm has a constant efficiency of 1.

We expect increases in efficiency as the size of the problem increases—due to relatively smaller influence of overhead.

Conversely, we expect decreases in efficiency as the number of processors increases (for the same problem size).
Experimental Results

Sparse random graphs on the NPACI Sun E10K:

Efficiency vs. Number of Processors ($p$)
Experimental Results

Sparse random graphs on the NPACI Sun E10K:

Efficiency

Problem Size (log₂n)
SMPs: Conclusions

- first practical linear speed-up for a difficult combinatorial problem
- good match of model predictions and experimental results
- transfer of PRAM algorithms to practical applications is possible on UMA SMPs
- promise for a whole new range of applications for parallel computation
Parallel Alg. Eng.: Conclusions

We can and should develop parallel algorithm engineering:

It will improve both the sequential and parallel parts of the code and yield new insights into the memory hierarchy, compiler development, and related topics.

Shared-memory implementations will be more challenging (more parameters), but we have promising results already.