On the Architectural Requirements for Efficient Execution of Graph Algorithms

David A. Bader*
Department of Electrical and Computer Engineering
University of New Mexico
dbader@ece.unm.edu

Guojing Cong
IBM T.J. Watson Research Center
Yorktown Heights, NY
gcong@us.ibm.com

John Feo
Cray, Inc.
feo@sdsc.edu

Abstract

Combinatorial problems such as those from graph theory pose serious challenges for parallel machines due to non-contiguous, concurrent accesses to global data structures with low degrees of locality. The hierarchical memory systems of symmetric multiprocessor (SMP) clusters optimize for local, contiguous memory accesses, and so are inefficient platforms for such algorithms. Few parallel graph algorithms outperform their best sequential implementation on SMP clusters due to long memory latencies and high synchronization costs. In this paper, we consider the performance and scalability of two graph algorithms, list ranking and connected components, on two classes of shared-memory computers: symmetric multiprocessors such as the Sun Enterprise servers and multithreaded architectures (MTA) such as the Cray MTA-2. While previous studies have shown that parallel graph algorithms can speedup on SMPs, the systems' reliance on cache microprocessors limits performance. The MTA's latency tolerant processors and hardware support for fine-grain synchronization makes performance a function of parallelism. Since parallel graph algorithms have an abundance of parallelism, they perform and scale significantly better on the MTA. We describe and give a performance model for each architecture. We analyze the performance of the two algorithms and discuss how the features of each architecture affects algorithm development, ease of programming, performance, and scalability.

Keywords: List ranking, Connected Components, Graph Algorithms, Shared Memory, Multithreading.

*This work was supported in part by NSF Grants CAREER ACI-00-93039, ITR ACI-00-81404, DEB-99-10123, ITR EIA-01-21377, Biocomplexity DEB-01-20709, DBI-0420513, ITR EF/BIO 03-31654; and DARPA Contract NBCH30390004.

1. Introduction

The enormous increase in processor speed over the last decade from approximately 300 MHz to over 3 GHz has far out-paced the speed of the hardware components responsible for delivering data to processors. For many large-scale applications, performance is no longer a function of how many operations a processor can perform per second, but rather the rate at which the memory system can deliver bytes of data. The conventional approach to ameliorating the memory bottleneck is to build hierarchical memory systems consisting of several levels of cache and local and remote memory modules. The first level cache can usually keep pace with the processor; but, fetching data from more remote memory causes the processor to stall. Since data is moved to the L1 cache in lines, reading data in sequence (i.e., with spatial locality) maximizes performance.

Combinatorial problems such as those from graph theory pose serious challenges for parallel machines due to non-contiguous, concurrent accesses to global data structures with low degrees of locality. The hierarchical memory systems of clusters are inefficient platforms for such algorithms. In fact, few parallel graph algorithms outperform their best sequential implementation on clusters due to long memory latencies and high synchronization costs.

A parallel, shared memory system is a more supportive platform. These systems typically have higher-bandwidth, lower-latency networks than clusters, and direct access to all memory locations avoids the overhead of message passing. Fast parallel algorithms for graph problems have been developed for such systems. List ranking [11, 31, 32, 23] is a key technique often needed in efficient parallel algorithms for solving many graph-theoretic problems; for example, computing the centroid of a tree, expression evaluation, minimum spanning forest, connected components, and planarity testing. Helman and JáJá [19, 20] present an efficient list ranking algorithm with implementation on SMP servers that achieves significant parallel speedup. Using this implementation of list ranking, Bader *et al.* have designed fast parallel algorithms and demonstrated speedups compared with the best sequential implementation for graph-theoretic problems such as ear decomposition [2], tree contraction and expression evaluation [3], spanning tree [4], rooted spanning tree [13], and minimum spanning forest [5]. Many of these algorithms achieve good speedups due to algorithmic techniques for efficient design and better cache performance. For some of the instances, e.g., arbitrary, sparse graphs, while we may be able to improve the cache performance to a certain degree, there are no known general techniques for cache performance optimization because the memory access pattern is largely determined by the structure of the graph.

In this paper, we discuss the architectural features necessary for efficient execution of graph algorithms by investigating the performance of two graph algorithms, list ranking and connected components, on two classes of shared memory systems: symmetric multiprocessors (SMP) such as the Sun Enterprise servers and multithreaded architectures (MTA) such as the Cray MTA-2. While our SMP results confirm the results of previous studies, we find the systems' reliance on cache microprocessors limits performance. For the MTA, we find its latency tolerant processors and hardware support for fine-grain synchronization make performance primarily a function of parallelism. Since graph algorithms often have an abundance of parallelism, these architectural features lead to superior performance and scalability.

The next section presents a brief overview of SMPs and a detailed description of the Cray MTA-2. We give a performance cost model for each machine. Sections 3 and 4 present SMP and MTA algorithms for list ranking and connected components, respectively. The SMP algorithms minimize non-contiguous memory accesses, whereas, the MTA algorithms maximize concurrent operations. Section 5 compares the performance and scalability of the implementations. In the final section, we present our conclusions and ideas for future work. In particular, we summarize how different architectural features affect algorithmic development, ease of programming, performance, and scalability.

2. Shared-Memory Architectures

In this section, we give a brief overview of two types of modern shared-memory architectures: symmetric multiprocessors and multithreaded architectures. While both allow parallel programs to access large globally-shared memories, they differ in significant ways as we discuss next.

2.1. Symmetric Multiprocessors (SMPs)

Symmetric multiprocessor (SMP) architectures, in which several processors operate in a true, hardware-based, shared-memory environment and are packaged as a single machine, are commonplace in scientific computing. Indeed, most high-performance computers are clusters of SMPs having from 2 to over 100 processors per node. Moreover, as supercomputers increasingly use SMP clusters, SMP computations play a significant and increasing role in supercomputing and computational science.

The generic SMP processor is a four-way super-scalar microprocessor, 32 to 64 hardware registers, and two levels of cache. The L1 cache is small (64 to 128 KB) and on chip. It can issue as many words per cycle as the processor can fetch and latency is a few cycles. The size of the L2 cache can vary widely from 256 KB to 8 MB. Bandwidth to the processor is typically 8 to 12 GB per second and latency is 20 to 30 cycles. The processors are connected to a large shared memory (4 to 8 GB per processor) by a high-speed bus, crossbar, or a low-degree network. The bandwidth to main memory falls off to 1 to 2 GB per second and latency increases to hundreds of cycles.

Caching and prefetching are two hardware techniques often used to hide memory latency. Caching takes advantage of spatial and temporal locality, while prefetching mechanisms use data address history to predict memory access patterns and perform reads early. If a high percentage of read/write operations are to L1 cache, the processor stays busy sustaining a high execution rate; otherwise, it starves for data. Prefetching may substantially increase the memory bandwidth used, and shows limited or no improvement in cache hits for irregular codes where the access patterns cannot be predicted, as is often the case in graph algorithms. Moreover, there is no hardware support for synchronization operations. Locks and barriers are typically implemented in software either by the user or via system calls. (Some newer systems do provide atomic memory operations such as compare-and-swap that may be used to build these features.) While an SMP is a shared-memory architecture, it is by no means the PRAM used in theoretical work — synchronization cannot be taken for granted, memory bandwidth is limited, and performance requires a high degree of locality. The significant features of SMPs are that the input can be held in the shared memory without having to be partitioned and they provide much faster access to their shared-memory (an order of magnitude or more) than an equivalent message-based architecture. As such SMPs provide a reasonable execution platform for graph algorithms. As noted above, parallel graph algorithms that execute faster than sequential algorithms do exist for this class of architecture.

To analyze SMP performance, we use a complexity model similar to that of Helman and JáJá [20] which has been shown to provide a good cost model for shared-memory algorithms on current symmetric multiprocessors [19, 20, 2, 3]. The model uses two parameters: the problem's input size n, and the number p of processors. For instance, for list ranking, n is the number of elements in the list, and for connected components, n is the number of vertices in the input graph. Running time $T(n,p)$ is measured by the triplet $\langle T_M(n,p)\ ;\ T_C(n,p)\ ;\ B(n,p)\rangle$, where $T_M(n,p)$ is the maximum number of non-contiguous main memory accesses required by any processor, $T_C(n,p)$ is an upper bound on the maximum local computational complexity of any of the processors, and $B(n,p)$ is the number of barrier synchronizations. This model, unlike the idealistic PRAM, is more realistic in that it penalizes algorithms with non-contiguous memory accesses that often result in cache misses and algorithms with more synchronization events.

We tested our SMP implementations in this paper on the Sun E4500, a uniform-memory-access (UMA) shared memory parallel machine with 14 UltraSPARC II 400MHz processors and 14 GB of memory. Each processor has 16 Kbytes of direct-mapped data (L1) cache and 4 Mbytes of external (L2) cache. We implement the algorithms using POSIX threads and software-based barriers.

2.2. Multithreaded Architectures (MTAs)

The Cray MTA is a flat, shared-memory multiprocessor system. All memory is accessible and equidistant from all processors. There is no local memory and no data caches. Parallelism, and not caches, is used to tolerate memory and synchronization latencies.

An MTA processor consists of 128 hardware streams and one instruction pipeline. The processor speed is 220 MHz. A stream is a set of 32 registers, a status word, and space in the instruction cache. An instruction is three-wide: a memory operation, a fused multiply-add, and a floating point add or control operation. Each stream can have up to 8 outstanding memory operations. Threads from the same or different programs are mapped to the streams by the runtime system. A processor switches among its streams every cycle, executing instructions from non-blocked streams in a fair manner. As long as one stream has a ready instruction, the processor remains fully utilized.

The interconnection network is a partially connected 3-D torus capable of delivering one word per processor per cycle. The system has 4 GBytes of memory per processor. Logical memory addresses are hashed across physical memory to avoid stride-induced hotspots. Each memory word is 68 bits: 64 data bits and 4 tag bits. One tag bit (the full-and-empty bit) is used to implement synchronous load/store operations. A synchronous load/store operation retries until it succeeds or traps. The thread that issued the load or store remains blocked until the operation completes; but the processor that issued the operation continues to issue instructions from non-blocked streams.

Since the MTA is a shared-memory system with no data cache and no local memory, it is comparable to an SMP where all memory reference are remote. Thus, the cost model presented in the previous section can be applied to the MTA with the difference that the magnitudes of $T_M(n,p)$ and $B(n,p)$ are reduced via multithreading. In fact, if sufficient parallelism exists, these costs are reduced to zero and performance is a function of only $T_C(n,p)$. Execution time is then a product of the number of instructions and the cycle time.

The number of threads needed to reduce $T_M(n,p)$ to zero is a function of the memory latency of the machine, about 100 cycles. Usually a thread can issue two or three instructions before it must wait for a previous memory operation to complete; thus, 40 to 80 threads per processor are usually sufficient to reduce $T_M(n,p)$ to zero. The number of threads needed to reduce $B(n,p)$ to zero is a function of intra-thread synchronization. Typically, it is zero and no additional threads are needed; however, hotspots can occur. Usually these can be worked around in software, but they do occasionally impact performance.

The MTA is close to a theoretical PRAM machine. Its latency tolerant processors, high bandwidth network, and shared memory, enable any processor to execute any operation and access any word. Execution time can reduce to the product of the number of instructions and the machine's cycle time. Since the MTA uses parallelism to tolerate latency, algorithms must often be parallelized at very fine levels to expose sufficient parallelism to hide the latencies. Fine levels of parallelism require fine grain synchronization that would cripple performance without some near zero-cost synchronization mechanism, such as the MTA's full-and-empty bits.

3. List Ranking

List ranking and other prefix computations on linked lists are basic operations that occur in many graph-based algorithms. The operations are difficult to parallelize because of the non-contiguous structure of lists and asynchronous access of shared data by concurrent tasks. Unlike arrays, there is no obvious way to divide the list into even, disjoint, continuous sublists without first computing the rank of each node. Moreover, concurrent tasks may visit or pass through the same node by different paths, requiring synchronization to ensure correctness.

List ranking is an instance of the more general prefix problem. Let X be an array of n elements stored in arbitrary order. For each element i, let $X(i).\mathit{value}$ be its value and $X(i).\mathit{next}$ be the index of its successor. Then for any binary associative operator $\oplus$, compute $X(i).\mathit{prefix}$ such that $X(\mathit{head}).\mathit{prefix} = X(\mathit{head}).\mathit{value}$ and $X(i).\mathit{prefix} = X(i).\mathit{value} \oplus X(\mathit{predecessor}).\mathit{prefix}$, where *head* is the first element of the list, i is not equal to *head*, and *predecessor* is the node preceding i in the list. If all values are 1 and the associative operation is addition, then prefix reduces to list ranking.

Our SMP implementation uses the Helman and JáJá list ranking algorithm [19] that performs the following main steps:

1. Find the head h of the list which is given by $h = (n(n-1)/2 - Z)$ where Z is the sum of successor indices of all the nodes in the list and n is the number of elements in the list.

2. Partition the input list into s sublists by randomly choosing one node from each memory block of $n/(s-1)$ nodes, where s is $\Omega(p \log n)$ and p is the number of processors. Create the array *Sublists* of size s. (Our implementation uses $s = 8p$.)

3. Traverse each sublist computing the prefix sum of each node within the sublists. Each node records its sublist index. The input value of a node in the *Sublists* array is the sublist prefix sum of the last node in the previous *Sublists*.

4. The prefix sums of the records in the *Sublists* array are then calculated.

5. Each node adds its current prefix sum value (value of a node within a sublist) and the prefix sum of its corresponding *Sublists* record to get its final prefix sums value. This prefix sum value is the required label of the leaves.

For $n > p^2 \ln n$, we would expect in practice the SMP list ranking to take
$T(n,p) = (M_M(n,p); T_C(n,p)) = \left(\frac{n}{p}, \mathrm{O}\left(\frac{n}{p}\right)\right)$. For a detailed description of the above steps refer to [19].

Our MTA implementation (described in high-level in the following four steps and also given in detail in Alg. 1) is similar to the Helman and JáJá algorithm.

1. Choose NWALK nodes (including the head node) and mark them. This step divides the list into NWALK sublists and is similar to steps 1 and 2 of the SMP algorithm.

2. Traverse each sublist computing the prefix sum of each node within the sublist (similar to step 3 of the SMP algorithm).

3. Compute the rank of each marked node (similar to step 4 of the SMP algorithm).

```
int list[NLIST+1], rank[NLIST+1];

void RankList(list, rank)
  int *list, *rank;
{ int i, first;
  int tmp1[NWALK+1], tmp2[NWALK+1];
  int head[NWALK+1], tail[NWALK+1], lnth[NWALK+1], next[NWALK+1];

#pragma mta assert noalias *rank, head, tail, lnth, next, tmp1, tmp2


  first = 0;
#pragma mta use 100 streams
  for (i = 1; i <= NLIST; i++) first += list[i];

  first = ((NLIST * NLIST + NLIST) / 2) - first;

  head[0] = 0; head[1]     = first;
  tail[0] = 0; tail[1]     = 0;
  lnth[0] = 0; lnth[1]     = 0;
  rank[0] = 0; rank[first] = 1;

  for (i = 2; i <= NWALK; i++) {
      int node = i * (NLIST / NWALK);
      head[i]    = node;
      tail[i]    = 0;
      lnth[i]    = 0;
      rank[node] = i;
  }

#pragma mta use 100 streams
#pragma mta assert no dependence lnth
  for (i = 1; i <= NWALK; i++) {
      int j, count, next_walk;

         count = 0;
         j     = head[i];
         do {count++; j = list[j];} while (rank[j] == -1);

         next_walk = rank[j];

         tail[i]         = j;
         lnth[next_walk] = count;
         next[i]         = next_walk;
  }

  while (next[1] != 0) {

#pragma mta assert no dependence tmp1
      for (i = 1; i <= NWALK; i++) {
          int n   = next[i];
          tmp1[n] = lnth[i];
          tmp2[i] = next[n];
      }

      for (i = 1; i <= NWALK; i++) {
          lnth[i] += tmp1[i];
          next[i]  = tmp2[i];
          tmp1[i]  = 0;
  }   }

#pragma mta use 100 streams
#pragma mta assert no dependence *rank
  for (i = 1; i <= NWALK; i++) {
      int j, k, count;
      j     = head[i];
      k     = tail[i];
      count = NLIST - lnth[i];
      while (j != k) {
        rank[j] = count; count--; j = list[j];
      }
  }
}
```

Algorithm 1: The MTA list ranking code.

4. Re-traverse the sublists incrementing the local rank of each node by the rank of the marked node at the head of the sublist (similar to step 5 of the SMP algorithm).

The first and third steps are $O(n)$. They consist of an outer loop of O(NWALK) and an inner loop of O(*length of the sublist*). Since the lengths of the local walks can vary, the work done by each thread will vary. We discuss load balancing issues below. The second step is also O(NWALKS) and can be parallelized using any one of the many parallel array prefix methods. In summary, the MTA algorithm has three parallel steps with NWALKS parallelism. Our studies show that by using 100 streams per processor and approximately 10 list nodes per walk, we achieve almost 100% utilization—so a linked list of length $1000p$ fully utilizes an MTA system with p processors.

Since the lengths of the walks are different, the amount of work done by each thread is different. If threads are assigned to streams in blocks, the work per stream will not be balanced. Since the MTA is a shared memory machine, any stream can access any memory location in equal time; thus, it is irrelevant which stream executes which walk. To avoid load imbalances, we instruct the compiler via a pragma to dynamically schedule the iterations of the outer loop. Each stream gets one walk at a time; when it finishes its current walk, it increments the loop counter and executes the next walk. A machine instruction, *int_fetch_add*, is used to increment the shared loop counter. The instruction adds one to a counter in memory and returns the old value. The instruction takes one cycle.

Alg. 1 gives our new source code for the MTA list ranking algorithm. **The fully-documented source codes for the SMP and MTA implementations of list ranking are freely-available from the web by visiting `http://www.ece.unm.edu/~dbader` and clicking on the *Software* tab.**

4. Connected Components

Let $G = (V, E)$ be an undirected graph with $|V| = n$ and $|E| = m$. Two vertices u and v are *connected* if there exists a path between u and v in G. This is an equivalence relation on V and partitions V into equivalence classes, i.e., connected components. Connectivity is a fundamental graph problem with a range of applications and can be building blocks for higher-level algorithms. The research community has produced a rich collection of theoretic deterministic [28, 21, 30, 26, 8, 9, 7, 18, 24, 34, 1, 12, 14] and randomized [17, 29] parallel algorithms for connected components. Yet for implementations and experimental studies, although several fast PRAM algorithms exist, to our knowledge there is no parallel implementation of connected components (other than our own [4, 6]) that achieves significant parallel speedup on sparse, irregular graphs when compared against the best sequential implementation.

Prior experimental studies of connected components implement the Shiloach-Vishkin algorithm [16, 22, 25, 15] due to its simplicity and efficiency. However, these parallel implementations of the Shiloach-Vishkin algorithm do not achieve any parallel speedups over arbitrary, sparse graphs against the best sequential implementation. Greiner [16] implemented several connected components algorithms (Shiloach-Vishkin, Awerbuch-Shiloach, "random-mating" based on the work of Reif [33] and Phillips [30], and a hybrid of the previous three) using NESL on the Cray Y-MP/C90 and TMC CM-2. On random graphs Greiner reports a maximum speedup of 3.5 using the hybrid algorithm when compared with a depth-first search on a DEC Alpha processor. Hsu, Ramachandran, and Dean [22] also implemented several parallel algorithms for connected components. They report that their parallel code runs 30 times slower on a MasPar MP-1 than Greiner's results on the Cray, but Hsu *et al.*'s implementation uses one-fourth of the total memory used by Greiner's hybrid approach. Krishnamurthy *et al.* [25] implemented a connected components algorithm (based on Shiloach-Vishkin) for distributed memory machines. Their code achieved a speedup of 20 using a 32-processor TMC CM-5 on graphs with underlying 2D and 3D regular mesh topologies, but virtually no speedup on sparse random graphs. Goddard, Kumar, and Prins [15] implemented a connected components algorithm (motived by Shiloach-Vishkin) for a mesh-connected SIMD parallel computer, the 8192-processor MasPar MP-1. They achieve a maximum parallel speedup of less than two on a random graph with 4096 vertices and about one-million edges. For a random graph with 4096 vertices and fewer than a half-million edges, the parallel implementation was slower than the sequential code.

In this paper, we compare implementations of Shiloach-Vishkin's connected components algorithm (denoted as SV) on both SMP and MTA systems. We chose this algorithm because it is representative of the memory access patterns and data structures in graph-theoretic problems. SV starts with n isolated vertices and m PRAM processors. Each processor P_i (for $1 \leq i \leq m$) grafts a tree rooted at vertex v_i (represented by v_i, in the beginning, the tree contains only a single vertex) to the tree that contains one of its neighbors u under the constraints $u < v_i$ or the tree represented by v_i is only one level deep. Grafting creates $k \geq 1$ connected subgraphs, and each of the k subgraphs is then shortcut so that the depth of the trees reduce at least by half. The approach continues to graft and shortcut on the reduced graphs until no more grafting is possible. As a result, each supervertex represents a connected graph. SV runs on an arbitrary CRCW PRAM in $O(\log n)$ time with $O(m)$ processors. The formal description of SV can be found in Alg. 2.

```
Input: 1. A set of $m$ edges $(i, j)$ given in arbitrary order
       2. Array $D[1..n]$ with $D[i] = i$
Output: Array $D[1..n]$ with $D[i]$ being the component
        to which vertex $i$ belongs
begin
    while true do
        1.for $(i, j) \in E$ in parallel do
            if D[i]=D[D[i]] and D[j]<D[i] then
            $D[D[i]] = D[j]$;
        2.for $(i, j) \in E$ in parallel do
            if $i$ belongs to a star and D[j]≠D[i] then
            $D[D[i]] = D[j]$;
        3.if all vertices are in rooted stars then exit;
        for all $i$ in parallel do
            $D[i] = D[D[i]]$
end
```

Algorithm 2: The Shiloach-Vishkin algorithm for connected components.

```
 while (graft) {
    graft = 0;
#pragma mta assert parallel
   1. for (i=0; i<2*m; i++) {
        u = E[i].v1;
        v = E[i].v2;
        if (D[u]<D[v] && D[v]==D[D[v]]) {
          D[D[v]] = D[u];
          graft = 1;
        }
      }
#pragma mta assert parallel
   2. for(i=0; i<n; i++)
      while (D[i] != D[D[i]]) D[i]=D[D[i]];
 }
```

Algorithm 3: SV on MTA. `E` is the edge list, with each element having two fields, `v1` and `v2`, representing the two endpoints.

SV can be implemented on SMPs and MTA, and the two implementations have very different performance characteristics on the two architectures, demonstrating that algorithms should be designed with the target architecture in consideration. For SMPs, we use appropriate optimizations described by Greiner [16], Chung and Condon [10], Krishnamurthy *et al.* [25], and Hsu *et al.* [22]. SV is sensitive to the labeling of vertices. For the same graph, different labeling of vertices may incur different numbers of iterations to terminate the algorithm. For the best case, one iteration of the algorithm may be sufficient, and the running time of the algorithm will be $O(\log n)$. Whereas for an arbitrary labeling of the same graph, the number of iterations needed will be from one to $\log n$. We refer the reader to our previous work [4] for more details on the SMP connectivity algorithm and its analysis (presented next).

In the first "graft-and-shortcut" step of SV, there are two non-contiguous memory accesses per edge, for reading $D[j]$ and $D[D[i]]$. Thus, first step costs $T(n,p) = \langle T_M(n,p) \; ; \; T_C(n,p) \; ; \; B(n,p) \rangle = \left\langle 2\frac{m}{p} + 1 \; ; \; O\left(\frac{n+m}{p}\right) \; ; \; 1 \right\rangle$. In the second step, the grafting is performed and requires one non-contiguous access per edge to set the parent, with cost $T(n,p) = \left\langle \frac{m}{p} + 1 \; ; \; O\left(\frac{n+m}{p}\right) \; ; \; 1 \right\rangle$. The final step of each iteration runs pointer jumping to form rooted stars to ensure that a tree is not grafted onto itself, with cost $T(n,p) = \left\langle \frac{n \log n}{p} \; ; \; O\left(\frac{n \log n}{p}\right) \; ; \; 1 \right\rangle$. In general, SV needs multiple iterations to terminate. Assuming the worst-case of $\log n$ iterations, the total complexity for SV is $T(n,p) = \langle T_M(n,p) \; ; \; T_C(n,p) \; ; \; B(n,p) \rangle \le \left\langle \frac{n \log^2 n}{p} + \left(3\frac{m}{p} + 2\right)\log n \; ; \; O\left(\frac{n \log^2 n + m \log n}{p}\right) \; ; \; 4 \log n \right\rangle$.

On the other hand, programming the MTA is unlike programming for SMPs, and code for the MTA looks much closer to the original PRAM algorithm. The programmer no longer specifies which processor works on which data partitions, instead, his/her job is to discover the finest grain of parallelism of the program and pass the information to the compiler using directives. Otherwise the compiler relies on the information from dependence analysis to parallelize the program. The implementation of SV on MTA is a direct translation of the PRAM algorithm, and the C source code is shown in Alg. 3. Alg. 3 is slightly different from the description of SV given in Alg. 2. In Alg. 3 the trees are shortcut into supervertices in each iteration, so that step 2 of Alg. 2 can be eliminated, and we no longer need to check whether a vertex belongs to a star which involves a significant amount of computation and memory accesses. Alg. 3 runs in $O(\log^2 n)$, and the bound is not tight. The directives in Alg. 3 are self-explanatory, and they are crucial for the compiler to parallelize the program as there is obvious data dependence in each step of the program.

5. Performance Results and Analysis

This section summarizes the experimental results of our implementations for list ranking and connected components on the SMP and MTA shared-memory systems.

For list ranking, we use two classes of list to test our algorithms: **Ordered** and **Random**. Ordered places each element in the array according to its rank; thus, node i is the i^{th} position of the array and its successor is the node at position $(i + 1)$. Random places successive elements randomly in the array. Since the MTA maps contiguous logical

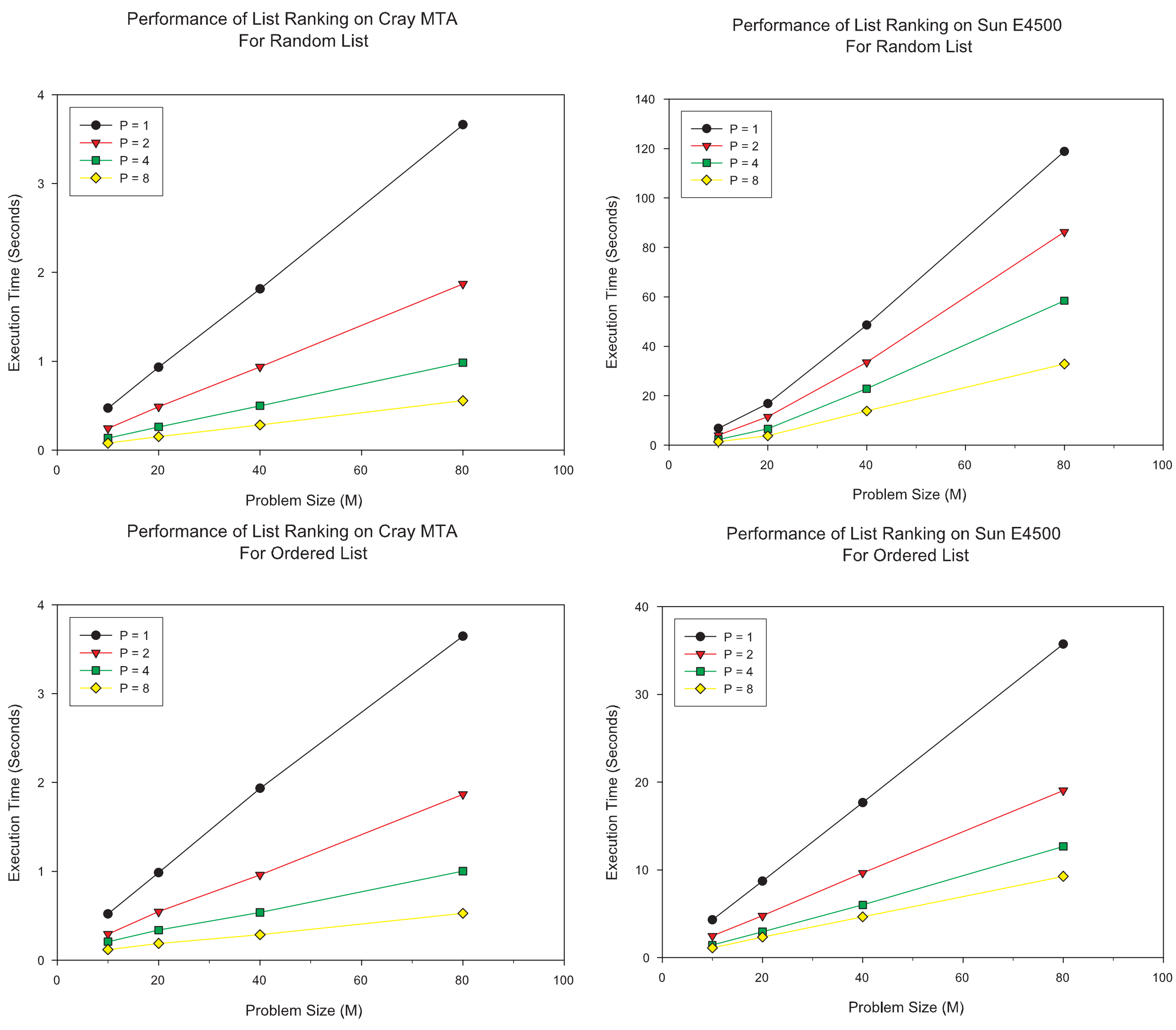


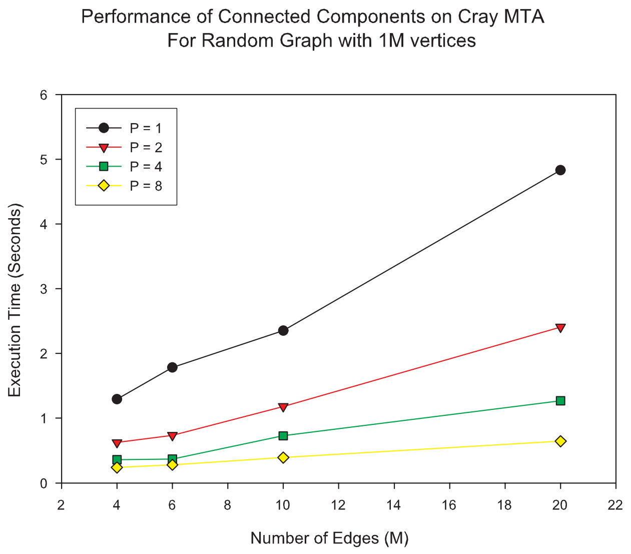
Figure 1. Running Times for List Ranking on the Cray MTA (left) and Sun SMP (right) for $p = 1, 2, 4$ and 8 processors.

addresses to random physical addresses the layout in physical memory for both classes is similar. We expect, and in fact see, that performance on the MTA is independent of order. This is in sharp contrast to SMP machines which rank Ordered lists much faster than Random lists. The running times for list ranking on the SMP and MTA are given in Fig. 1. First, all of the implementations scaled well with problem size and number of processors. In all cases, the running times decreased proportionally with the number of processors, quite a remarkable result on a problem such as list ranking whose efficient implementation has been considered a "holy grail" of parallel computing. On the Cray MTA, the performance is nearly identical for random or ordered lists, demonstrating that locality of memory accesses is a non-issue; first, since memory latency is tolerated, and second, since the logical addresses are randomly assigned to the physical memory. On the SMP, there is a factor of 3 to 4 difference in performance between the best case (an ordered list) and the worst case (a randomly-ordered list). On the ordered lists, the MTA is an order of magnitude faster than this SMP, while on the random list, the MTA is approximately 35 times faster.

For connected components, we create a random graph of n vertices and m edges by randomly adding m unique edges to the vertex set. Several software packages generate random graphs this way, including LEDA [27]. The running times for connected components on the SMP and MTA are given in Fig. 2 for a random graph with $n = 1M$ vertices and from $m = 4M$ to $20M$ edges. (Note that throughout this paper $M = 2^{20}$.) Similar to the list ranking results, we see that both shared-memory systems scale with problem size and number of processors for finding the connected components of a sparse, random graph. This is also a truly remarkable result noting that no previous parallel implementations have exhibited parallel speedup on arbitrary, sparse graphs for the connected components problem. (Note that we give speedup results for the SMP approach in [4, 6].) In comparison, the MTA implementation is 5 to 6 times faster than the SMP implementation of SV connected components, and the code for the MTA is quite simple and similar to the PRAM algorithm, unlike the more complex code required for the SMP to achieve this performance.

Number of Processors	List Ranking		Connected Components
	Random List	Ordered List	
1	98%	97%	99%
4	90%	85%	93%
8	82%	80%	91%

Table 1. Processor Utilization for List Ranking and Connected Components on the Cray MTA.



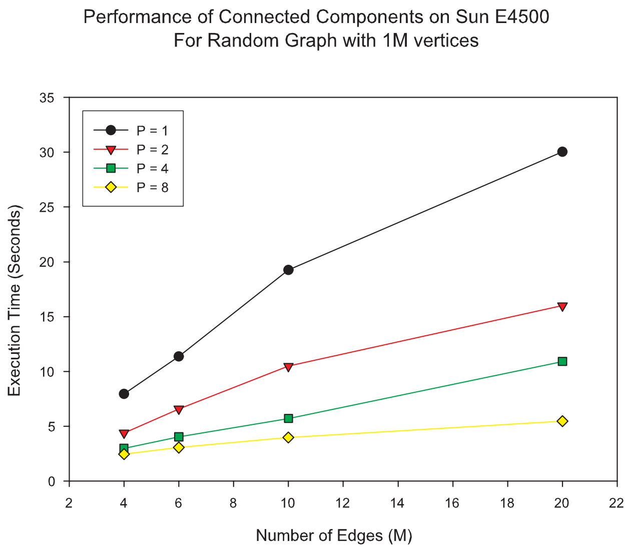


Figure 2. Running Times for Connected Components on the Cray MTA (left) and Sun SMP (right) for $p = 1, 2, 4$ and 8 processors.

On the Cray MTA, we achieve high-percentages of processor utilization. In Table 1 we give the utilizations achieved for the MTA on List Ranking of a $20M$-node list, and Connected Components with $n = 1M$ vertices and $m = 20M(\approx n \log n)$ edges.

6. Conclusions

In summary, we show that fast, parallel implementations of graph-theoretic problems such as list ranking and connected components are well-suited to shared-memory computer systems. We confirm the results of previous SMP studies and present the first results for multithreaded architectures. The latter highlights the benefits of latency-tolerant processors and hardware support for synchronization. In our experiments, the Cray MTA achieved high utilization rates for performing both list ranking and connected components. In addition, the MTA, because of its randomization between logical and physical memory addresses, and its multithreaded execution techniques for latency hiding, performed extremely well on the list ranking problem, no matter the spatial locality of the list.

Although both are shared memory machines, the programming model presented to the user by the two machines is different. The Cray MTA allows the programmer to focus on the concurrency in the problem, while the SMP server forces the programmer to optimize for locality and cache. We find the latter results in longer, more complex programs that embody both parallelism and locality.

We are currently developing additional graph algorithms for the MTA. In particularly, we are investigating whether the technique used in the list ranking program is a general technique. In that program, we first compacted the list to a list of super nodes, performed list ranking on the compacted list, and then expanded the super nodes to compute the rank of the original nodes. The compaction and expansion steps are parallel, $\mathrm{O}(n)$, and require little synchronization; thus, they increase parallelism while decreasing overhead.

In 2005, Cray will build a third generation multithreaded architecture. To reduce costs, this system will incorporate commodity parts. In particular, the memory system will not be as flat as in the MTA-2. We will reconduct our studies on this architecture as soon as it is available.

References

[1] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, C-36(10):1258–1263, 1987.

[2] D. Bader, A. Illendula, B. M. Moret, and N. Weisse-Bernstein. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In G. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, editors, *Proc. 5th Int'l Workshop on Algorithm Engineering (WAE 2001)*, volume 2141 of *Lecture Notes in Computer Science*, pages 129–144, Århus, Denmark, 2001. Springer-Verlag.

[3] D. Bader, S. Sreshta, and N. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In S. Sahni, V. Prasanna, and U. Shukla, editors, *Proc. 9th Int'l Conf. on High Performance Computing (HiPC 2002)*, volume 2552 of *Lecture Notes in Computer Science*, pages 63–75, Bangalore, India, Dec. 2002. Springer-Verlag.

[4] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, NM, Apr. 2004.

[5] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, NM, Apr. 2004.

[6] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 2005. to appear.

[7] F. Y. Chin, J. Lam, and I.-N. Chen. Efficient parallel algorithms for some graph problems. *Commununications of the ACM*, 25(9):659–665, 1982.

[8] K. Chong and T. Lam. Finding connected components in $O(\log n \log \log n)$ time on the EREW PRAM. *J. Algorithms*, 18:378–402, 1995.

[9] K. W. Chong, Y. Han, and T. W. Lam. Concurrent threads and optimal parallel minimum spanning tree algorithm. *Journal of the ACM*, 48:297–323, 2001.

[10] S. Chung and A. Condon. Parallel implementation of Borůvka's minimum spanning tree algorithm. In *Proc. 10th Int'l Parallel Processing Symp. (IPPS'96)*, pages 302–315, Apr. 1996.

[11] R. Cole and U. Vishkin. Faster optimal prefix sums and list ranking. *Information and Computation*, 81(3):344–352, 1989.

[12] R. Cole and U. Vishkin. Approximate parallel scheduling. part II: applications to logarithmic-time optimal graph algorithms. *Information and Computation*, 92:1–47, 1991.

[13] G. Cong and D. A. Bader. The Euler tour technique and parallel rooted spanning tree. In *Proc. Int'l Conf. on Parallel Processing (ICPP)*, pages 448–457, Montreal, Canada, Aug. 2004.

[14] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, 1991.

[15] S. Goddard, S. Kumar, and J. Prins. Connected components algorithms for mesh-connected parallel computers. In S. N. Bhatt, editor, *Parallel Algorithms: 3rd DIMACS Implementation Challenge October 17-19, 1994*, volume 30 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 43–58. American Mathematical Society, 1997.

[16] J. Greiner. A comparison of data-parallel algorithms for connected components. In *Proc. 6th Ann. Symp. Parallel Algorithms and Architectures (SPAA-94)*, pages 16–25, Cape May, NJ, June 1994.

[17] S. Halperin and U. Zwick. An optimal randomised logarithmic time connectivity algorithm for the EREW PRAM. In *Proc. 7th Ann. Symp. Discrete Algorithms (SODA-96)*, pages 438–447, 1996. Also published in J. Comput. Syst. Sci., 53(3):395–416, 1996.

[18] Y. Han and R. A. Wagner. An efficient and fast parallel-connected component algorithm. *Journal of the ACM*, 37(3):626–642, 1990.

[19] D. R. Helman and J. JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *Algorithm Engineering and Experimentation (ALENEX'99)*, volume 1619 of *Lecture Notes in Computer Science*, pages 37–56, Baltimore, MD, Jan. 1999. Springer-Verlag.

[20] D. R. Helman and J. JáJá. Prefix computations on symmetric multiprocessors. *Journal of Parallel and Distributed Computing*, 61(2):265–278, 2001.

[21] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commununications of the ACM*, 22(8):461–464, 1979.

[22] T.-S. Hsu, V. Ramachandran, and N. Dean. Parallel implementation of algorithms for finding connected components in graphs. In S. N. Bhatt, editor, *Parallel Algorithms: 3rd DIMACS Implementation Challenge October 17-19, 1994*, volume 30 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 23–41. American Mathematical Society, 1997.

[23] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, New York, 1992.

[24] D. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} |v|)$ parallel time for the CREW PRAM. In *Proc. of the 32nd Ann. IEEE Symp. on Foundations of Computer Science*, pages 688–697, San Juan, Puerto Rico, 1991.

[25] A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick. Connected components on distributed memory machines. In S. N. Bhatt, editor, *Parallel Algorithms: 3rd DIMACS Implementation Challenge October 17-19, 1994*, volume 30 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–21. American Mathematical Society, 1997.

[26] C. Kruskal, L. Rudolph, and M. Snir. Efficient parallel algorithms for graph problems. *Algorithmica*, 5(1):43–64, 1990.

[27] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[28] D. Nash and S. Maheshwari. Parallel algorithms for the connected components and minimal spanning trees. *Information Processing Letters*, 14(1):7–11, 1982.

[29] S. Pettie and V. Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. Comput.*, 31(6):1879–1895, 2002.

[30] C. Phillips. Parallel graph contraction. In *Proc. 1st Ann. Symp. Parallel Algorithms and Architectures (SPAA-89)*, pages 148–157. ACM, 1989.

[31] M. Reid-Miller. List ranking and list scan on the Cray C-90. In *Proc. 6th Ann. Symp. Parallel Algorithms and Architectures (SPAA-94)*, pages 104–113, Cape May, NJ, June 1994.

[32] M. Reid-Miller. List ranking and list scan on the Cray C-90. *J. Comput. Syst. Sci.*, 53(3):344–356, Dec. 1996.

[33] J. Reif. Optimal parallel algorithms for integer sorting and graph connectivity. Technical Report TR-08-85, Harvard Univ., Boston, MA, Mar. 1985.

[34] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algs.*, 3(1):57–67, 1982.