31

Design of Multithreaded Algorithms for Combinatorial Problems

31.1 Introduction 31-1
31.2 The Cray MTA-2 31-2
31.3 Designing Multithreaded Algorithms for the MTA-2 31-3
Expressing Parallelism • Synchronization • Analyzing Complexity
31.4 List Ranking 31-4
Multithreaded Implementation • Performance Results
31.5 Graph Traversal 31-8
Related Work • A Multithreaded Approach to Breadth-First Search • *st*-Connectivity and Shortest Paths • Performance Results
31.6 Algorithms in Social Network Analysis 31-16
Centrality Metrics • Algorithms for Computing Betweenness Centrality • Parallel Algorithms • Performance Results
31.7 Conclusions 31-24
Acknowledgments 31-25
References 31-25

David A. Bader
Kamesh Madduri
Georgia Institute of Technology

Guojing Cong
IBM T.J. Watson Research Center

John Feo
Microsoft Corporation

31.1 Introduction

Graph theoretic and combinatorial problems arise in several traditional and emerging scientific disciplines such as VLSI design, optimization, databases, and computational biology. Some examples include phylogeny reconstruction [65,66], protein–protein interaction networks [89], placement and layout in VLSI chips [59], data mining [52,55], and clustering in semantic webs. Graph abstractions are also finding increasing relevance in the domain of large-scale network analysis [28,58]. Empirical studies show that many social and economic interactions tend to organize themselves in complex network structures. These networks may contain billions of vertices with degrees ranging from small constants to thousands [14,42].

The Internet and other communication networks, transportation, and power distribution networks also share this property. The two key characteristics studied in these networks are *centrality* (which nodes in the graph are best connected to others, or have the most influence) and *connectivity* (how nodes are connected to one another). Popular metrics for analyzing these networks, like betweenness centrality [18, 43], are computed using fundamental graph algorithms such as breadth-first search (BFS) and shortest paths.

In recognition of the importance of graph abstractions for solving large-scale problems on high performance computing (HPC) systems, several communities have proposed graph theoretic computational challenges. For instance, the 9th DIMACS Implementation Challenge [38] is targeted at finding shortest paths in graphs. The DARPA High Productivity Computer Systems (HPCS) [35] program has developed a synthetic graph theory benchmark called SSCA#2 [9, 11], which is composed of four kernels operating on a large-scale, directed multigraph.

Graph theoretic problems are typically memory intensive, and the memory accesses are fine-grained and highly irregular. This leads to poor performance on cache-based systems. On distributed memory clusters, few parallel graph algorithms outperform their best sequential implementations due to the high memory latency and synchronization costs. Parallel shared memory systems are a more supportive platform. They offer higher memory bandwidth and lower latency than clusters, as the global shared memory avoids the overhead of message passing. However, parallelism is dependent on the cache performance of the algorithm and scalability is limited in most cases. Whereas it may be possible to improve the cache performance to a certain degree for some classes of graphs, there are no known general techniques for cache optimization, because the memory access pattern is largely dependent on the structure of the graph.

In this chapter, we present fast parallel algorithms for several fundamental graph theoretic problems, optimized for multithreaded architectures such as the Cray MTA-2. The architectural features of the MTA-2 aid the design of simple, scalable, and high-performance graph algorithms. We test our implementations on large scale-free and sparse random graph instances, and report impressive results, both for algorithm execution time and parallel performance. For instance, BFS on a scale-free graph of 200 million vertices and 1 billion edges takes less than 5 seconds on a 40-processor MTA-2 system with an absolute speedup of close to 30. This is a significant result in parallel computing, as prior implementations of parallel graph algorithms report very limited or no speedup on irregular and sparse graphs, when compared to the best sequential implementation.

This chapter is organized as follows. In Section 31.2, we detail the unique architectural features of the Cray MTA-2 that aid irregular application design. Section 31.3 introduces the programming constructs available to express parallelism on the MTA-2. In Section 31.4, we compare the performance of the MTA-2 with shared memory systems such as symmetric multiprocessors (SMP) by implementing an efficient shared memory algorithm for list ranking, an important combinatorial problem. In the subsequent sections, we present multithreaded algorithms and performance results for BFS, *st*-connectivity and shortest paths, and apply them to evaluate real-world metrics in Social Network Analysis (SNA).

31.2 The Cray MTA-2

The Cray MTA-2 [34] is a high-end shared memory system offering two unique features that aid considerably in the design of irregular algorithms: fine-grained parallelism and zero-overhead synchronization. The MTA-2 has no data cache; rather than using a memory hierarchy to hide latency, the MTA-2 processors use hardware multithreading to tolerate the latency. The low-overhead synchronization support complements multithreading and makes performance primarily a function of parallelism. Since combinatorial problems often have an abundance of parallelism, these architectural features lead to superior performance and scalability.

The computational model for the MTA-2 is *thread-centric*, not processor-centric. A thread is a logical entity comprised of a sequence of instructions that are issued in order. An MTA processor consists of 128 hardware *streams* and one instruction pipeline. A stream is a physical resource (a set of 32 registers, a status word, and space in the instruction cache) that holds the state of one thread. 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 eight 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 nonblocked streams. As long as one stream has a ready instruction, the processor remains fully utilized. No thread is bound to any particular processor. System memory size and the inherent degree of parallelism within the program are the only limits on the number of threads used by a program.

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

Cray's XMT [94,95] system, formerly called the Eldorado, is a follow-on to the MTA-2 that showcases the massive multithreading paradigm. The XMT is anticipated to scale from 24 to over 8000 processors, providing over one million simultaneous threads and 128 terabytes of globally shared memory. The basic building block of the XMT, the Threadstorm processor, is very similar to the thread-centric MTA processor.

31.3 Designing Multithreaded Algorithms for the MTA-2

The MTA-2 is closer to a theoretical PRAM machine than a shared memory SMP system. Since the MTA-2 uses parallelism to tolerate latency, algorithms must often be parallelized at very fine levels to expose sufficient parallelism. However, it is not necessary that all parallelism in the program be expressed such that the system can exploit it; the goal is simply to saturate the processors. The programs that make the most effective use of the MTA-2 are those that express the parallelism of the problem in a way that allows the compiler to best exploit it.

31.3.1 Expressing Parallelism

The MTA-2 compiler automatically parallelizes *inductive* loops of three types: parallel loops, linear recurrences, and reductions. A loop is inductive if it is controlled by a variable that is incremented by a loop-invariant stride during each iteration, and the loop-exit test compares this variable with a loop-invariant expression. An inductive loop has only one exit test and can only be entered from the top. If each iteration of an inductive loop can be executed completely independently of the others, then the loop is termed parallel. The compiler does a dependence analysis to determine loop-carried dependencies, and then automatically spawns threads to parallelize the loop. To attain the best performance, we need to write code (and thus design algorithms) such that most of the loops are implicitly parallelized.

There are several compiler directives that can be used to parallelize various sections of a program. The three major types of parallelization schemes available are

1. *Single-processor (fray) parallelism*: The code is parallelized in such a way that just the 128 streams on the processor are utilized.
2. *Multiprocessor (crew) parallelism*: This has higher overhead than single-processor parallelism. However, the number of streams available is much larger, bounded by the size of the whole machine rather than the size of a single processor. Iterations can be statically or dynamically scheduled.
3. *Future parallelism*: The *future* construct (detailed below) is used in this form of parallelism. This does not require that all processor resources used during the loop be available at the beginning of the loop. The runtime growth manager increases the number of physical processors as needed. Iterations are always dynamically scheduled. We illustrate the use of the future directive to handle nested parallelism in the subsequent sections.

A *future* is a powerful construct to express user-specified explicit parallelism. It packages a sequence of code that can be executed by a newly created thread running concurrently with other threads in the program. Futures include efficient mechanisms for delaying the execution of code that depends on the computation within the future, until the future completes. The thread that spawns the future can pass information to the thread that executes the future through parameters. Futures are best used to implement task-level parallelism and the parallelism in recursive computations.

31.3.2 Synchronization

Synchronization is a major limiting factor to scalability in the case of practical shared memory implementations. The software mechanisms commonly available on conventional architectures for achieving synchronization are often inefficient. However, the MTA-2 provides hardware support for fine-grained synchronization through the full-empty bit associated with every memory word. The compiler provides a number of generic routines that operate atomically on scalar variables. We list a few useful constructs that appear in the algorithm pseudocodes in subsequent sections:

- The `int_fetch_add` routine (`int_fetch_add`(&v, i)) atomically adds integer i to the value at address v, stores the sum at v, and returns the original value at v (setting the full-empty bit to full). If v is an empty sync or future variable, the operation blocks until v becomes full.
- `readfe`(&v) returns the value of variable v when v is full and sets v empty. This allows threads waiting for v to become empty to resume execution. If v is empty, the read blocks until v becomes full.
- `writeef`(&v, i) writes the value i to v when v is empty, and sets v back to full. The thread waits until v is set empty.
- `purge`(&v) sets the state of the full-empty bit of v to empty.

31.3.3 Analyzing Complexity

To analyze algorithm performance, we use a complexity model similar to the one proposed by Helman and JáJá [50], which has been shown to provide a good cost model for shared-memory algorithms on current SMP systems [10, 13, 49, 50]. The model uses two parameters: the problem's input size n, and the number p of processors. 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 noncontiguous 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 noncontiguous memory accesses that often result in cache misses, and also considers synchronization events in algorithms.

Since the MTA-2 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 Helman–JáJá model can be applied to the MTA with the difference that the magnitudes of $T_M(n,p)$ and $B(n,p)$ are reduced through 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, 40–80 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 intrathread 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.

31.4 List Ranking

List ranking [30,53,77,78] 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 [49, 50] 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. [10] have designed fast parallel algorithms and demonstrated speedups compared with the best sequential implementation for graph-theoretic problems such as ear decomposition, tree contraction and expression evaluation [13], spanning tree [6], rooted spanning tree [31], and minimum spanning forest [7].

31.4.1 Multithreaded Implementation

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 $X(i).value$ be its value and $X(i).next$ be the index of its successor. Then for any binary associative operator $\oplus$, compute $X(i).prefix$ such that $X(head).prefix = X(head).value$ and $X(i).prefix = X(i).value \oplus X(predecessor).prefix$, 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.

The operations are difficult to parallelize because of the noncontiguous 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.

The MTA implementation (described in high-level in the following four steps and also given in detail in Algorithm 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.
2. Traverse each sublist computing the prefix sum of each node within the sublist.
3. Compute the rank of each marked node.
4. Retraverse the sublists incrementing the local rank of each node by the rank of the marked node at the head of the sublist.

The first and third steps are $O(n)$. They consist of an outer loop of $O(\text{NWALK})$ and an inner loop of $O(\textit{length of the sublist})$. Since the lengths of the local walks can vary, the work done by each thread will vary. The second step is also $O(\text{NWALKS})$ 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 through 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. The `int_fetch_add` instruction is used to increment the shared loop counter.

31.4.2 Performance Results

This section summarizes the performance results of our list ranking implementations on SMP and the MTA-2 shared-memory systems. Our SMP implementation is also based on the Helman–JáJá algorithm [49, 50] and is detailed in [8]. We tested our implementation on the Sun E4500, a uniform-memory-access (UMA) shared memory parallel machine with 14 UltraSPARC II 400 MHz processors and 14 GB of memory. Each processor has 16 KB of direct-mapped data (L1) cache and 4 MB of external (L2) cache. We implement the algorithms using POSIX threads and software-based barriers.

```
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.

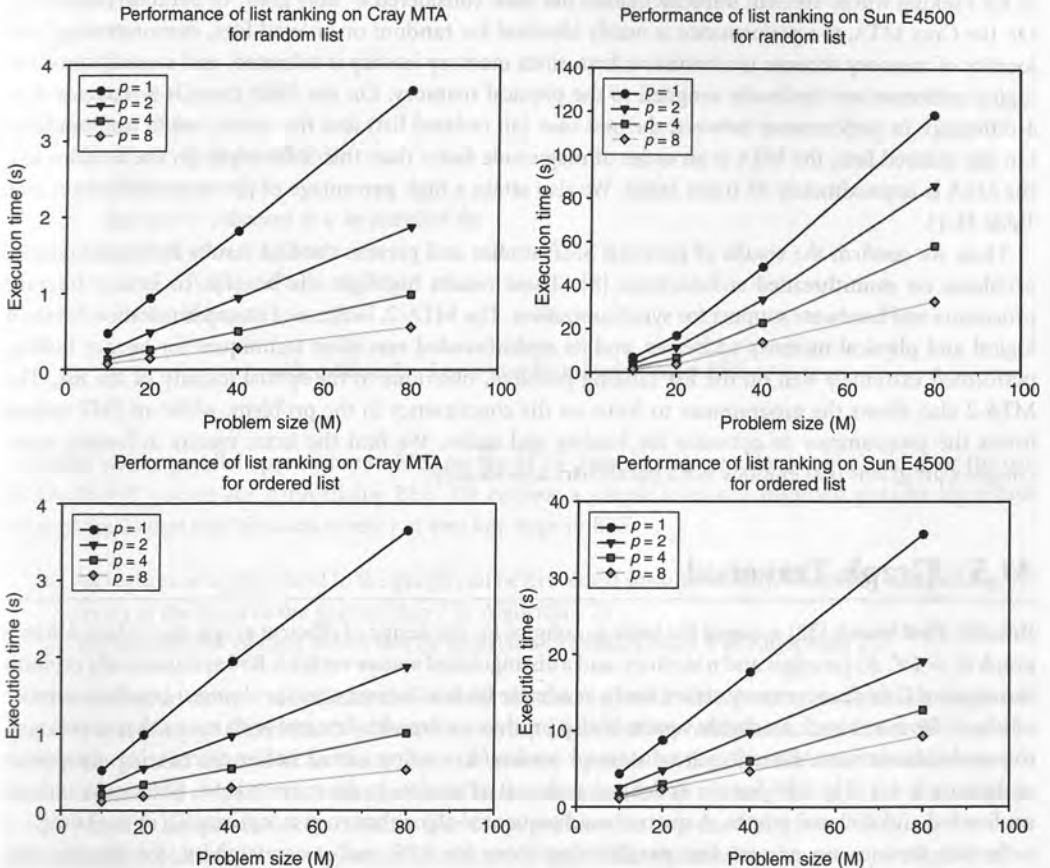

FIGURE 31.1 Running times for list ranking on the Cray MTA (left) and Sun SMP (right) for $p = 1, 2, 4$, and 8 processors.

For list ranking, we use two classes of lists 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 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 that rank Ordered lists much faster than Random lists. The running times for list ranking on the SMP and MTA are given in Figure 31.1. First, all of the

TABLE 31.1 Processor Utilization for List Ranking on the Cray MTA-2

	List Ranking (%)	
Number of processors	Random list	Ordered list
1	98	97
4	90	85
8	82	80

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 nonissue; 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 random 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. We also attain a high percentage of processor utilization (see Table 31.1).

Thus, we confirm the results of previous SMP studies and present the first results for combinatorial problems on multithreaded architectures [8]. These results highlight the benefits of latency-tolerant processors and hardware support for synchronization. The MTA-2, 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, oblivious to the spatial locality of the list. The MTA-2 also allows the programmer to focus on the concurrency in the problem, while an SMP system 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.

31.5 Graph Traversal

Breadth-First Search [32] is one of the basic paradigms for the design of efficient graph algorithms. Given a graph $G = (V, E)$ (m edges and n vertices) and a distinguished source vertex s, BFS systematically explores the edges of G to *discover* every vertex that is reachable from s. It computes the *distance* (smallest number of edges) from s to each reachable vertex. It also produces a *breadth-first tree* with root s that contains all the reachable vertices. All vertices at a distance k (or *level* k) are first visited, before discovering any vertices at distance $k + 1$. The *BFS frontier* is defined as the set of vertices in the current level. BFS works on both undirected and directed graphs. A queue-based sequential algorithm runs in optimal $O(m + n)$ time.

In this section, we present fast parallel algorithms for BFS and st-connectivity, for directed and undirected graphs, on the MTA-2. We extend these algorithms to compute single-source shortest paths, assuming unit-weight edges. The implementations are tested on four different classes of graphs—random graphs generated on the basis of the Erdős–Rényi model, scale-free graphs, synthetic sparse random graphs that are hard cases for parallelization, and SSCA#2 benchmark graphs. We also outline a parallel implementation of BFS for handling high-diameter graphs.

31.5.1 Related Work

Distributed BFS [4,92] and st-connectivity [15,45] are both well-studied problems, with related work on graph partitioning and load balancing schemes [5,86] to facilitate efficient implementations. Other problems and algorithms of interest include shortest path variants [26,33,39,63,83,85] and external memory

algorithms and data structures [1,22,62] for BFS. Several PRAM and BSP [36] algorithms have been proposed to solve this problem. However, there are very few parallel implementations that achieve significant parallel speedup on sparse, irregular graphs when compared against the best sequential implementations.

31.5.2 A Multithreaded Approach to Breadth-First Search

Input: $G(V, E)$, source vertex s
Output: Array $d[1..n]$ with $d[v]$ holding the length of the shortest path from s to $v \in V$, assuming unit-weight edges

```
1  for all v ∈ V in parallel do
2      d[v] ← −1;
3  d[s] ← 0;
4  Q ← φ;
5  Enqueue s ← Q;
6  while Q ≠ φ do
7      for all u ∈ Q in parallel do
8          Delete u ← Q;
9          for each v adjacent to u in parallel do
10             if d[v] = −1 then
11                 d[v] ← d[u] + 1;
12                 Enqueue v ← Q;
```

Algorithm 2: Level-synchronized parallel BFS.

Unlike earlier parallel approaches to BFS, on the MTA-2 we do not consider load balancing or the use of distributed queues for parallelizing BFS. We employ a simple level-synchronized parallel algorithm (Algorithm 2) that exploits concurrency at two key steps in BFS:

1. All vertices at a given *level* in the graph can be processed simultaneously, instead of just picking the vertex at the head of the queue (step 7 in Algorithm 2).
2. The adjacencies of each vertex can be inspected in parallel (step 9 in Algorithm 2).

We maintain an array d to indicate the level (or distance) of each visited vertex and process the global queue Q accordingly. Algorithm 2 is, however, a very high-level representation, and hides the fact that thread-safe parallel insertions to the queue and atomic updates of the distance array d are needed to ensure correctness. Algorithm 3 details the MTA-2 code required to achieve this (for the critical steps 7–12), which is simple and very concise. The loops will not be automatically parallelized as there are dependencies involved. The compiler can be forced to parallelize them using the *assert parallel* directive on both the loops. We then note that we have to handle and exploit the nested parallelism in this case. We can explicitly indicate that the iterations of the outer loop can be handled concurrently, and the compiler will dynamically schedule threads for the inner loop. We do this using the compiler directive *loop future* (see Algorithm 3) to indicate that the iterations of the outer loop can be concurrently processed. We use the low-overhead instructions `int_fetch_add`, `readfe()`, and `writeef()` to atomically update the value of d, and insert elements to the queue in parallel.

Once correctness is assured, we optimize the code further. Note that we used the *loop future* directive on the outer loop in Algorithm 3 to concurrently schedule the loop iterations. Using this directive incurs an overhead of about 200 instructions. So we do not use it when the number of vertices to be visited at a given level is <50 (experimentally determined figure for this particular loop). Clearly, the time taken to spawn threads must be considerably less than the time spent in the outer loop.

```
/* While the Queue is not empty */
#pragma mta assert parallel
#pragma mta loop future
for (i = startIndex; i < endIndex; i++) {
  u = Q[i];

  /* Inspect all vertices adjacent to u */
#pragma mta assert parallel
  for (j = 0; j < degree[u]; j++) {
    v = neighbor[u][j];

    /* Check if v has been visited yet? */
    dist = readfe(&d[v]);

    if (dist == -1) {
      writeef(&d[v], d[u] + 1);
      /*Enqueue v */
      Q[int_fetch_add(&count, 1)] = v;
      } else {
      writeef(&d[v], dist);
      }
  }
}
```

Algorithm 3: MTA-2 parallel C code for steps 7–12 in Algorithm 2.

High-degree vertices pose a major problem. Consider the case when the majority of vertices at a particular level are of low-degree (<100), but a few vertices are of very high-degree (order of thousands). If Algorithm 2 is applied, most of the threads will be done processing the low-degree vertices quickly, but only a few threads will be assigned to inspect the adjacencies of the high-degree nodes. The system will be heavily underutilized then, until the loop finishes. To prevent this, we first need to identify high-degree nodes at each level and work on them sequentially, but inspect their adjacencies in parallel. This ensures that work is balanced among the processors. We can choose the low-degree cutoff value appropriately so that parallelization of adjacency visits would be sufficient to saturate the system. We take this approach for BFS on Scale-free graphs. In general, given an arbitrary graph instance, we can determine which algorithm to apply based on a quick evaluation of the degree distribution. This can be done either during graph generation stage (when reading from an uncharacterized data set and internally representing it as a graph) or in a pre-processing phase before running the actual BFS algorithm.

We observe that the above parallelization schemes will not work well for high-diameter graphs (for instance, consider a chain of vertices with bounded degree). For arbitrary sparse graphs, Ullman and Yannakakis offer high-probability PRAM algorithms for transitive closure and BFS [87] that take $\tilde{O}(n^{\epsilon})$ time with $\tilde{O}(mn^{1-2\epsilon})$ processors, provided $m \geq n^{2-3\epsilon}$. The key idea here is as follows. Instead of starting the search from the source vertex s, we expand the frontier up to a distance d in parallel from a set of randomly chosen n/d *distinguished* vertices (that includes the source vertex s also) in the graph. We then construct a new graph whose vertices are the distinguished vertices, and we have edges between these vertices if they were pair-wise reachable in the previous step. Now a set of n/d^2 *superdistinguished* vertices are selected among them and the graph is explored to a depth d^2. After this step, the resulting graph would be dense and we can determine the shortest path of the source vertex s to each of the vertices. Using this information, we can determine the shortest paths from s to all vertices.

31.5.3 *st*-Connectivity and Shortest Paths

st-connectivity is a related problem, also applicable to both directed and undirected graphs. Given two vertices s and t, the problem is to decide whether or not they are connected, and determine the shortest path between them, if one exists. It is a basic building block for more complex graph algorithms, has linear time complexity, and is complete for the class SL of problems solvable by symmetric, nondeterministic, log-space computations [60].

Input: $G(V, E)$, vertex pair (s, t)
Output: The smallest number of edges *dist* between s and t, if they are connected

```
1  for all v ∈ V in parallel do
2      color[v] ← WHITE;
3      d[v] ← 0;
4  color[s] ← RED; color[t] ← GREEN; Q ← φ; done ← FALSE; dist ← ∞;
5  Enqueue s ← Q; Enqueue t ← Q;
6  while Q ≠ φ and done = FALSE do
7      for all u ∈ Q in parallel do
8          Delete u ← Q;
9          for each v adjacent to u in parallel do
10             color ← readfe(&color[v]);
11             if color = WHITE then
12                 d[v] ← d[u] + 1;
13                 Enqueue v ← Q;
14                 writeef(&color[v], color[u]);
15             else
16                 if color ≠ color[u] then
17                     done ← TRUE;
18                     tmp ← readfe(&dist);
19                     if tmp > d[u] + d[v] + 1 then
20                         writeef(&dist, d[u] + d[v] + 1);
21                     else
22                         writeef(&dist, tmp);
23                 writeef(&color[v], color);
```

Algorithm 4: *st*-connectivity (STCONN-FB): concurrent BFSes from s and t.

We can easily extend the BFS algorithm for solving the *st*-connectivity problem too. A naïve implementation would be to start a BFS from s, and stop when t is visited. However, we note that we could run BFS concurrently both from s and to t, and if we keep track of the vertices visited and the expanded frontiers on both sides, we can correctly determine the shortest path between s and t. The key steps are outlined in Algorithm 4 (termed STCONN-FB), which has both high-level details as well as MTA-specific synchronization constructs. Both s and t are added to the queue initially, and newly discovered vertices are either colored RED (for vertices reachable from s) or GREEN (for vertices that can reach t). When a *back edge* is found in the graph, the algorithm terminates and the shortest path is evaluated. As in the previous case, we encounter nested parallelism here and apply the same optimizations.

The pseudocode for STCONN-FB is elegant and concise, but it is also very easy to introduce race conditions and potential deadlocks. Figure 31.2 illustrates a subtle race condition if we do not update *dist* atomically in Algorithm 4. Consider the directed graph presented in the figure and the problem

Input: $G(V, E)$, vertex pair (s, t)
Output: The smallest number of edges *dist* between s and t, if they are connected

```
1  for all v ∈ V in parallel do
2      color[v] ← WHITE;
3      d[v] ← 0;
4  color[s] ← GRAY; color[t] ← GRAY; Qs ← φ; Qt ← φ;
5  done ← FALSE; dist ← −1;
6  Enqueue s ← Qs; Enqueue t ← Qt; extentS ← 1; extentT ← 1;
7  while (Qs ≠ φ or Qt ≠ φ) and done = FALSE do
8      Set Q appropriately;
9      for all u ∈ Q in parallel do
10         Delete u ← Q;
11         for each v adjacent to u in parallel do
12             color ← readfe(&color[v]);
13             if color = WHITE then
14                 d[v] ← d[u] + 1;
15                 Enqueue v ← Q;
16                 writeef(&color[v], color[u]);
17             else
18                 if color ≠ color[v] then
19                     dist ← d[u] + d[v] + 1;
20                     done ← TRUE;
21                 writeef(&color[v], color);
22     extentS ← |Qs|; extentT ← |Qt|;
```

Algorithm 5: *st*-connectivity (STCONN-MF): alternate BFSes from *s* and *t*.

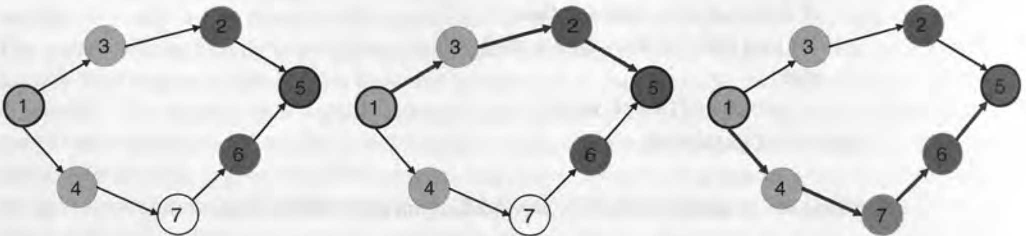

FIGURE 31.2 An illustration of a possible race condition to be avoided in *st*-connectivity Alg. STCONN-FB.

of determining whether vertices 1 and 5 are connected. After one iteration of the loop and concurrent expansion of frontiers from both 1 and 5, we have the vertices colored as shown in the leftmost diagram. The shortest path is clearly $(1, 3, 2, 5)$ and of length 3. Let us assume that the search from 2 finds 3 (or vice-versa), updates the *dist* value to 3, and mark *done* TRUE. Now vertex 4 will find 7 and color it, and then vertex 6 will encounter 7 colored differently. It may then update the value of *dist* to 4 if we do not have the conditional statement at line 19.

We also implement an improved algorithm for *st*-connectivity (STCONN-MF, denoting *minimum frontier*) applicable to graphs with a large percentage of high degree nodes, detailed in Algorithm 5. In this case, we maintain two different queues Q_s and Q_t and expand the smaller frontier (Q in Algorithm 5 is either Q_s or Q_t, depending on the values of *extentS* and *extentT*) on each iteration. This algorithm

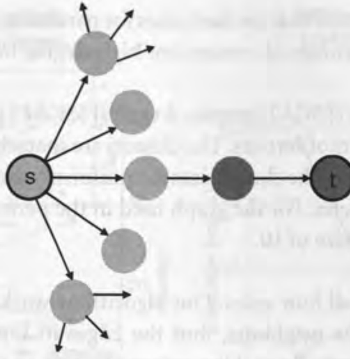

FIGURE 31.3 A case when *st*-connectivity Alg. STCONN-MF will be faster.

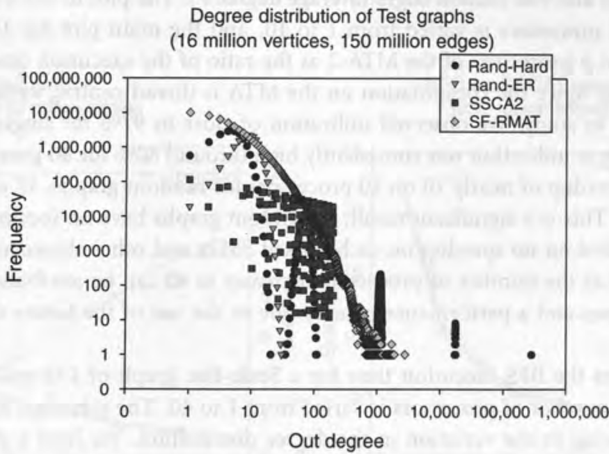

FIGURE 31.4 Degree distributions of the four test graph classes.

would be faster for some graph instances (see Figure 31.3) for an illustration). Intuitively, we try to avoid inspecting the adjacencies of high-degree vertices, thus doing less work than STCONN-FB. Both Algorithms 4 and 5 are discussed in more detail in [12].

31.5.4 Performance Results

This section summarizes the experimental results of our BFS and *st*-connectivity implementations on the Cray MTA-2. We report results on a 40-processor MTA-2, with each processor having a clock speed of 220 MHz and 4 GB of RAM. From the programmer's viewpoint, the MTA-2 is, however, a global shared memory machine with 160 GB memory.

We test our algorithms on four different classes of graphs (see Figure 31.4):

1. Random graphs generated based on the Erdős–Rényi $G(n, p)$ model (Rand-ER): A random graph of m edges is generated with $p = m/n^2$ and has very little structure and locality.
2. Scale-free graphs (SF-RMAT), used to model real-world large-scale networks: these graphs are generated using the R-MAT graph model [25]. They have a significant number of vertices of very high degree, although the majority of vertices are low-degree ones. The degree distribution plot on a log–log scale is a straight line with a heavy tail, as seen in Figure 31.4.

3. Synthetic sparse random graphs that are hard cases for parallelization (Rand-Hard): As in scale-free graphs, a considerable percentage of vertices are high-degree ones, but the degree distribution is different.
4. DARPA SSCA#2 benchmark (SSCA2) graphs: A typical SSCA#2 graph consists of a large number of highly interconnected clusters of vertices. The clusters are sparsely connected, and these intercluster edges are randomly generated. The cluster sizes are uniformly distributed and the maximum cluster size is a user-defined parameter. For the graph used in the performance studies in Figure 31.4, we assume a maximum cluster size of 10.

We generate directed graphs in all four cases. Our algorithms work for both directed and undirected graphs, as each vertex stores all its neighbors, and the edges in both directions. In this section, we report results for the undirected case. By making minor changes to our code, we can analyze directed graphs also.

Figure 31.5(a) plots the execution time and speedup attained by the BFS algorithm on a random graph of 134 million vertices and 940 million edges (average degree 7). The plot in the inset shows the scaling when the number of processors is varied from 1 to 10, and the main plot for 10–40 processors. We define the *Speedup* on p processors of the MTA-2 as the ratio of the execution time on p processors to that on one processor. Since the computation on the MTA is thread-centric, system utilization is also an important metric to study. We observed utilization of close to 97% for single processor runs. We also note that the system utilization was consistently high (around 80% for 40 processor runs) across all runs. We achieve a speedup of nearly 10 on 10 processors for random graphs, 17 on 20 processors, and 28 on 40 processors. This is a significant result, as random graphs have no locality and such instances would offer very limited on no speedup on cache-based SMPs and other shared memory systems. The decrease in efficiency as the number of processors increases to 40 can be attributed to two factors: hot spots in the BFS queue, and a performance penalty due to the use of the future directive for handling nested parallelism.

Figure 31.5(b) gives the BFS execution time for a Scale-free graph of 134 million vertices and 940 million edges, as the number of processors is varied from 1 to 40. The speedups are slightly lower than the previous case, owing to the variation in the degree distribution. We have a preprocessing step for high-degree nodes as discussed in the previous sections; this leads to an additional overhead in execution time (when compared to random graphs), as well as insufficient work to saturate the system in some cases. Figures 31.5(c) and (d) summarize the BFS performance for SSCA#2 graphs. The execution time and speedup (Figure 31.5(c)) are comparable to random graphs. We also varied the user-defined cluster size parameter to see how BFS performs for dense graphs. Figure 31.5(d) shows that the dense SSCA#2 graphs are also handled well by our BFS algorithm.

Figure 31.5(e) and (f) show the performance of BFS as the edge density is varied for Rand-ER and Rand-Hard graphs. We consider a graph of 2.147 billion edges and vary the number of vertices from 16 to 536 million. In case of Rand-ER graphs, the execution times are comparable as expected, since the dominating term in the computational complexity is the number of edges, 2.147 billion in this case. However, in case of the Rand-Hard graphs, we note an anomaly: the execution time for the graph with 16 million vertices is comparatively more than the other graphs. This is because this graph has a significant number of vertices of very large degree. Even though it scales with the number of processors, since we avoid the use of nested parallelism in this case, the execution times are higher.

Figure 31.6 summarizes the performance of *st*-connectivity. Note that both the *st*-connectivity algorithms are based on BFS, and if BFS is implemented efficiently, we would expect *st*-connectivity also to perform well. Figure 31.6(a) shows the performance of STCONN-MF on random graphs as the number of processors is varied from 1 to 10. Note that the execution times are highly dependent on (s, t) pair we choose. In this particular case, just 45,000 vertices were visited in a graph of 134 million vertices. The *st*-connectivity algorithm shows near-linear scaling with the number of processors. The actual execution time is bounded by the BFS time, and is dependent on the shortest path length and the degree distribution of the vertices in the graph. In Figure 31.6(b), we compare the performance of

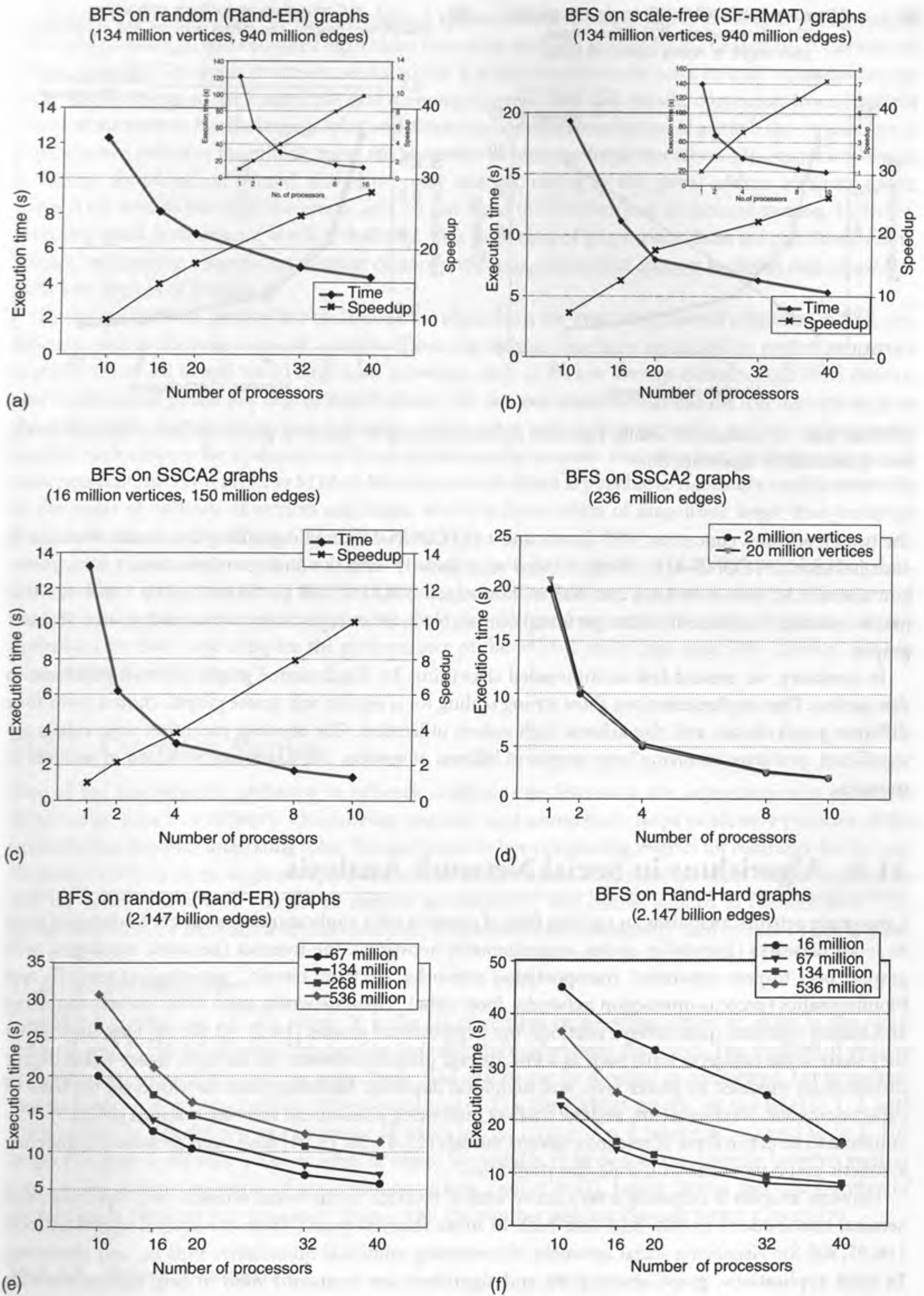

FIGURE 31.5 Breadth First Search performance results (a) Execution time and speedup for random graphs: 1–10 processors (inset), and 10–40 processors (b) Execution time and speedup for scale-free (SF-RMAT) graphs: 1–10 processors (inset), and 10–40 processors (c) Execution time and speedup for SSCA2 graphs (d) Execution time variation as a function of average degree for SSCA2 graphs (e) Execution time variation as a function of average degree for Rand-ER graphs (f) Execution time variation as a function of average degree for Rand-Hard graphs.

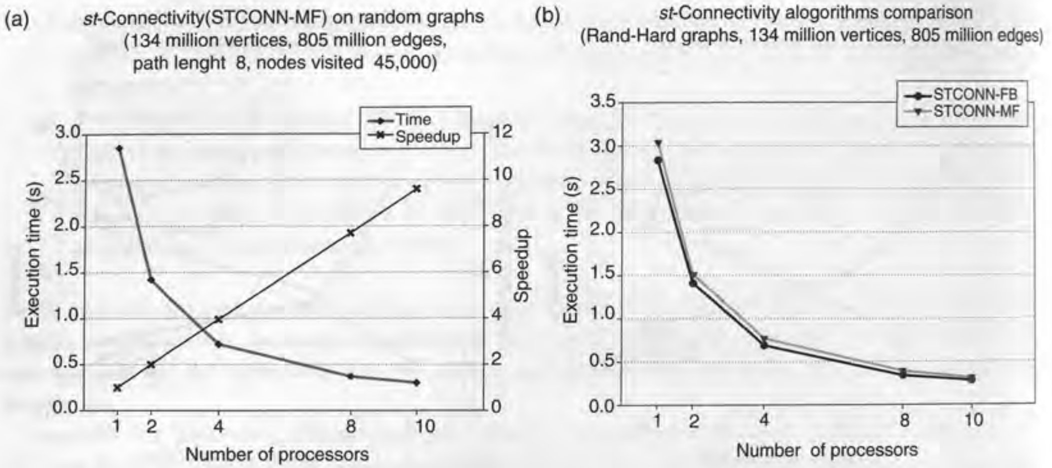

FIGURE 31.6 *st*-connectivity results: Execution time and speedup for Rand-ER graphs (a) and comparison of the two *st*-connectivity algorithms (b).

the two algorithms, concurrent BFS from *s* and *t* (STCONN-FB), and expanding the smaller frontier in each iteration (STCONN-MF). Both of them scale linearly with the number of processors for a problem size of 134 million vertices and 805 million edges. STCONN-FB performs slightly better for this graph instance. They were found to perform comparably in other experiments with random and SSCA#2 graphs.

In summary, we present fast multithreaded algorithms for fundamental graph traversal problems in this section. Our implementations show strong scaling for irregular and sparse graphs chosen from four different graph classes, and also achieve high system utilization. The absolute execution time values are significant; problems involving large graphs of billions of vertices and edges can be solved in seconds to minutes.

31.6 Algorithms in Social Network Analysis

Large-scale network analysis is an exciting field of research with applications in a variety of domains such as social networks (friendship circles, organizational networks), the Internet (network topologies, web graphs, peer-to-peer networks), transportation networks, electrical circuits, genealogical research and bioinformatics (protein-interaction networks, food webs). These networks seem to be entirely unrelated and indeed represent quite diverse relations, but experimental studies [14, 21, 42, 68, 69] have shown that they share some common traits such as a low average distance between the vertices, heavy-tailed degree distributions modeled by power laws, and high local densities. Modeling these networks on the basis of experiments and measurements, and the study of interesting phenomena and observations [24, 29, 75, 93], continue to be active areas of research. Several models [25, 47, 70, 73, 91] have been proposed to generate synthetic graph instances with scale-free properties.

Network analysis is currently a very active area of research in the social sciences [44, 79, 82, 90], and seminal contibutions to this field date back to more than 60 years. There are several analytical tools [16, 57, 80] for visualizing social networks, determining empirical quantitative indices, and clustering. In most applications, graph abstractions and algorithms are frequently used to help capture the salient features. Thus, network analysis from a graph theoretic perspective is about extracting interesting information, given a large graph constructed from a real-world dataset.

Network analysis and modeling have received considerable attention in recent times, but algorithms are relatively less studied. Real-world networks are often very large in size, ranging from several hundreds

of thousands to billions of vertices and edges. A space-efficient memory representation of such graphs is itself a big challenge, and dedicated algorithms have to be designed exploiting the unique characteristics of these networks. On single processor workstations, it is not possible to do exact in-core computations on large graphs owing to the limited physical memory. Current high-end parallel computers have sufficient physical memory to handle large graphs, and a naïve in-core implementation of a graph theory problem is typically two orders of magnitude faster the best external memory implementation [2]. Algorithm design is further simplified on parallel shared memory systems; owing to the global address memory space, there is no need to partition the graph, and we can avoid the the overhead of message passing. However, attaining good performance is still a challenge, as a large class of graph algorithms are combinatorial in nature, and involve a significant number of noncontiguous, concurrent accesses to global data structures with low degrees of locality.

In this section, we present fast multithreaded algorithms for evaluating several centrality indices frequently used in complex network analysis. These algorithms have been optimized to exploit properties typically observed in real-world large scale networks, such as the low average distance, high local density, and heavy-tailed power law degree distributions. We test our implementations on real datasets such as the web graph, protein-interaction networks, movie-actor and citation networks, and report impressive parallel performance for evaluation of the computationally intensive centrality metrics (betweenness and closeness centrality) on the Cray MTA-2. We demonstrate that it is possible to rigorously analyze networks in the order of millions of vertices and edges, which is three orders of magnitude larger than instances that can be handled by existing SNA software packages.

We give a brief overview of the various centrality metrics in Section 31.6.1, followed by the parallel algorithms in Section 31.6.3 and performance results in Section 31.6.4 on a variety of real-world datasets and synthetic scale-free graphs. We have also designed algorithms for these centrality metrics optimized for SMP and compare the performance of the MTA-2 and a high-end SMP (IBM p5 570) in Section 31.6.4.

31.6.1 Centrality Metrics

One of the fundamental problems in network analysis is to determine the *importance* of a particular vertex or an edge in a network. Quantifying *centrality* and *connectivity* helps us identify portions of the network that may play interesting roles. Researchers have been proposing metrics for centrality for the past 50 years, and there is no single accepted definition. The metric of choice is dependent on the application and the network topology. Almost all metrics are empirical, and can be applied to element-level [20], group-level [40], or network-level [88] analyses. We present a few commonly used indices in this section.

31.6.1.1 Preliminaries

Consider a graph $G = (V, E)$, where V is the set of vertices representing *actors* or *nodes* in the social network, and E, the set of edges representing the relationships between the actors. The number of vertices and edges are denoted by n and m, respectively. The graphs can be directed or undirected. Let us assume that each edge $e \in E$ has a positive integer weight $w(e)$. For unweighted graphs, we use $w(e) = 1$. A *path* from vertex s to t is defined as a sequence of edges $\langle u_i, u_{i+1} \rangle$, $0 \leq i \leq l$, where $u_0 = s$ and $u_l = t$. The *length* of a path is the sum of the weights of edges. We use $d(s, t)$ to denote the distance between vertices s and t (the minimum length of any path connecting s and t in G). Let us denote the total number of shortest paths between vertices s and t by σ_{st}, and the number passing through vertex v by $\sigma_{st}(v)$.

31.6.1.2 Degree Centrality

The degree centrality DC of a vertex v is simply the degree $deg(v)$ for undirected graphs. For directed graphs, we can define two variants: in-degree centrality and out-degree centrality. This is a simple local measure, based on the notion of neighborhood. This index is useful in case of static graphs, for situations when we are interested in finding vertices that have the most direct connections to other vertices.

31.6.1.3 Closeness Centrality

This index measures the closeness, in terms of *distance*, of an actor to all other actors in the network. Vertices with a smaller total distance are considered more important. Several closeness-based metrics [17,71,81] have been developed by the SNA community. A commonly used definition is the reciprocal of the total distance from a particular vertex to all other vertices:

$$CC(v) = \frac{1}{\sum_{u \in V} d(v,u)}$$

Unlike degree centrality, this is a global metric. To calculate the closeness centrality of a vertex v, we may apply BFS (for unweighted graphs) or a single-source shortest paths (SSSP, for weighted graphs) algorithm from v.

31.6.1.4 Stress Centrality

Stress centrality is a metric based on shortest paths counts, first presented in [84]. It is defined as

$$SC(v) = \sum_{s \neq v \neq t \in V} \sigma_{st}(v)$$

Intuitively, this metric deals with the *work* done by each vertex in a communications network. The number of shortest paths that contain an element v will give an estimate of the amount of stress a vertex v is under, assuming communication will be carried out through shortest paths all the time. This index can be calculated using a variant of the all-pairs shortest-paths algorithm, that calculates and stores all shortest paths between any pair of vertices.

31.6.1.5 Betweenness Centrality

Betweenness centrality is another shortest paths enumeration-based metric, introduced by Freeman in [43]. Let $\delta_{st}(v)$ denote the *pairwise dependency*, or the fraction of shortest paths between s and t that pass through v:

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Betweenness centrality of a vertex v is defined as

$$BC(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v)$$

This metric can be thought of as *normalized* stress centrality. Betweenness centrality of a vertex measures the control a vertex has over communication in the network, and can be used to identify key actors in the network. High centrality indices indicate that a vertex can reach other vertices on relatively short paths, or that a vertex lies on a considerable fraction of shortest paths connecting pairs of other vertices. We discuss algorithms to compute this metric in detail in the next section.

This index has been extensively used in recent years for analysis of social as well as other large-scale complex networks. Some applications include biological networks [37,54,76], study of sexual networks and AIDS [61], identifying key actors in terrorist networks [28,58], organizational behavior [23], supply chain management [27], and transportation networks [48].

There are a number of commercial and research software packages for SNA (e.g., Pajek [16], InFlow [57], UCINET [3]), which can also be used to determine these centrality metrics. However, they can only be used to study comparatively small networks (in most cases, sparse graphs with less than 40,000 vertices). Our goal is to develop fast, high-performance implementations of these metrics so that we can analyze large-scale real-world graphs of millions to billions of vertices.

31.6.2 Algorithms for Computing Betweenness Centrality

A straightforward way of computing betweenness centrality for each vertex would be as follows:

1. Compute the length and number of shortest paths between all pairs (s, t)
2. For each vertex v, calculate every possible pair-dependency $\delta_{st}(v)$ and sum them up

The complexity is dominated by Step 2, which requires $\Theta(n^3)$ time summation and $\Theta(n^2)$ storage of pair-dependencies. Popular SNA tools like UCINET use an adjacency matrix to store and update the pair-dependencies. This yields an $\Theta(n^3)$ algorithm for betweenness by augmenting the Floyd–Warshall algorithm for the all-pairs shortest-paths problem with path counting [18].

Alternately, we can modify Dijkstra's single-source shortest paths algorithm to compute the pair-wise dependencies. Observe that a vertex $v \in V$ is on the shortest path between two vertices $s, t \in V$, *iff* $d(s, t) = d(s, v) + d(v, t)$. Define a set of *predecessors* of a vertex v on shortest paths from s as $pred(s, v)$. Now each time an edge $\langle u, v\rangle$ is scanned for which $d(s, v) = d(s, u) + d(u, v)$, that vertex is added to the predecessor set $pred(s, v)$. Then, the following relation would hold:

$$\sigma_{sv} = \sum_{u \in pred(s,v)} \sigma_{su}$$

Setting the initial condition of $pred(s, v) = s$ for all neighbors v of s, we can proceed to compute the number of shortest paths between s and all other vertices. The computation of $pred(s, v)$ can be easily integrated into Dijkstra's SSSP algorithm for weighted graphs, or BFS Search for unweighted graphs. But even in this case, determining the fraction of shortest paths using v, or the pair-wise dependencies $\delta_{st}(v)$, proves to be the dominant cost. The number of shortest $s - t$ paths using v is given by $\sigma_{st}(v) = \sigma_{sv} \cdot \sigma_{vt}$. Thus computing $BC(v)$ requires $O(n^2)$ time per vertex v, and $O(n^3)$ time in all. This algorithm is the most commonly used one for evaluating betweenness centrality.

To exploit the sparse nature of typical real-world graphs, Brandes [18] came up with an algorithm that computes the betweenness centrality score for all vertices in the graph in $O(mn + n^2 \log n)$ time for weighted graphs, and $O(mn)$ time for unweighted graphs. The main idea is as follows. We define the *dependency* of a source vertex $s \in V$ on a vertex $v \in V$ as

$$\delta_s(v) = \sum_{t \in V} \delta_{st}(v)$$

The betweenness centrality of a vertex v can be then expressed as $BC(v) = \sum_{s \neq v \in V} \delta_s(v)$. The dependency $\delta_s(v)$ satisfies the following recursive relation:

$$\delta_s(v) = \sum_{w: v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_s(w))$$

The algorithm is now stated as follows. First, n SSSP computations are done, one for each $s \in V$. The predecessor sets $pred(s, v)$ are maintained during these computations. Next, for every $s \in V$, using the information from the shortest paths tree and predecessor sets along the paths, compute the dependencies $\delta_s(v)$ for all other $v \in V$. To compute the centrality value of a vertex v, we finally compute the sum of all dependency values. The $O(n^2)$ space requirements can be reduced to $O(m + n)$ by maintaining a *running centrality score.*

31.6.3 Parallel Algorithms

In this section, we present our new parallel algorithms and implementation details of the centrality metrics on the Cray MTA-2. Since there is abundant coarse-grained parallelism to be exploited in the centrality metrics algorithms, we have also designed algorithms for SMPs, and compare the performance of the

MTA-2 and the IBM p5 570 SMP system for large graph instances. We also evaluated the performance of connected components on SMPs and the MTA-2 in one of our previous works [8].

31.6.3.1 Degree Centrality

We use a cache-friendly adjacency array representation [74] for internally storing the graph, storing both the in- and out-degree of each vertex in contiguous arrays. This makes the computation of degree centrality straight-forward, with a constant time look-up on the MTA-2 and the p5 570. As noted earlier, degree centrality is a useful metric for determining the graph structure, and for a first pass at identifying important vertices.

31.6.3.2 Closeness Centrality

Recall the definition of closeness centrality

$$CC(v) = \frac{1}{\sum_{u \in V} d(v, u)}$$

We need to calculate the distance from v to all other vertices, and then sum over all distances. One possible solution to this problem would be to precompute a *distance matrix* using the $O(n^3)$ Floyd–Warshall All-Pairs Shortest Paths algorithm. Evaluating a specific closeness centrality value then costs $O(n)$ on a single processor ($O(n/p + \log p)$ using p processors) to sum up an entire row of distance values. However, since real-world graphs are typically sparse, we have $m \ll n^2$ and this algorithm would be very inefficient in terms of actual running time and memory utilization. Instead, we can just compute n shortest path trees, one for each vertex $v \in V$, with v as the source vertex for BFS or Dijkstra's algorithm. On p processors, this would yield $T_C = O\left(\frac{nm+n^2}{p}\right)$ and $T_M = nm/p$ for unweighted graphs. For weighted graphs, using a naïve queue-based representation for the expanded frontier, we can compute all the centrality metrics in $T_C = O\left(\frac{nm+n^3}{p}\right)$ and $T_M = 2nm/p$. The bounds can be further improved with the use of efficient priority queue representations.

Since the evaluation of closeness centrality is computationally intensive, it is valuable to investigate approximate algorithms. Using a random sampling technique, Eppstein and Wang [41] show that the closeness centrality of all vertices in a weighted, undirected graph can be approximated with high probability in $O\left(\frac{\log n}{\epsilon^2}(n \log n + m)\right)$ time, and an additive error of at most $\epsilon\Delta_G$ (ϵ is a fixed constant, and Δ_G is the diameter of the graph). The algorithm proceeds as follows. Let k be the number of iterations needed to obtain the desired error bound. In iteration i, pick vertex v_i uniformly at random from V and solve the SSSP problem with v_i as the source. The estimated centrality is given by

$$CC_a(v) = \frac{k}{n \sum_{i=1}^{k} d(v_i, u)}.$$

The error bounds follow from a result by Hoeffding [51] on probability bounds for sums of independent random variables. We parallelize this algorithm as follows. Each processor can run SSSP computations from k/p vertices and store the evaluated distance values. The cost of this step is given by $T_C = O\left(\frac{k(m+n)}{p}\right)$ and $T_M = km/p$ for unweighted graphs. For real-world graphs, the number of sample vertices k can be set to $\Theta\left(\frac{\log n}{\epsilon^2}\right)$ to obtain the error bounds given above. The approximate closeness centrality value of each vertex can then be calculated in $O(k) = O\left(\frac{\log n}{\epsilon^2}\right)$ time, and the summation for all n vertices would require $T_C = O\left(\frac{n \log n}{p\epsilon^2}\right)$ and constant T_M.

31.6.3.3 Stress and Betweenness Centrality

These two metrics require shortest paths enumeration and we design our parallel algorithms based on Brandes' [18] sequential algorithm for sparse graphs. Algorithm 6 outlines the general approach for the

Input: $G(V, E)$
Output: Array $BC[1..n]$, where $BC[v]$ gives the centrality metric for vertex v

```
1  for all v ∈ V in parallel do
2      BC[v] ← 0;
   for all s ∈ V in parallel do
3      S ← empty stack;
4      P[w] ← empty list, w ∈ V;
5      σ[t] ← 0, t ∈ V; σ[s] ← 1;
6      d[t] ← −1, t ∈ V; d[s] ← 0;
7      Q → empty queue;
8      enqueue s ← Q;
9      while Q not empty do
10         dequeue v ← Q;
11         push v → S;
12         for each neighbor w of v in parallel do
13             if d[w] < 0 then
14                 enqueue w → Q;
15                 d[w] ← d[v] + 1;
16             if d[w] = d[v] + 1 then
17                 σ[w] ← σ[w] + σ[v];
18                 append v → P[w];
19     δ[v] ← 0, v ∈ V;
20     while S not empty do
21         pop w ← S;
22         for v ∈ P[w] do
23             δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w]);
24         if w ≠ s then
25             BC[w] ← BC[w] + δ[w];
```

Algorithm 6: Parallel betweenness centrality for unweighted graphs.

case of unweighted graphs. On each BFS computation from s, the queue Q stores the current set of vertices to be visited, S contains all the vertices reachable from s, and $P(v)$ is the predecessor set associated with each vertex $v \in V$. The arrays d and σ store the distance from s, and shortest path counts, respectively. The centrality values are computed in Steps 22–25, by summing the dependencies $\delta(v)$, $v \in V$. The final scores need to be divided by two if the graph is undirected, as all shortest paths are counted twice.

We observe that parallelism can be exploited at two levels:

1. The BFS/SSSP computations from each vertex can be done concurrently, provided the centrality running sums are updated atomically.
2. The actual BFS/SSSP can be also be parallelized. When visiting the neighbors of a vertex, edge relaxation can be done concurrently.

It is theoretically possible to do all the SSSP computations concurrently (Steps 3–25 in Algorithm 6). However, the memory requirements scale as $O\big(p(m+n)\big)$, and we only have a modest number of processors on current SMP systems. So for the SMP implementation, we do a coarse-grained partitioning of work and assign each processor a fraction of the vertices from which to initiate SSSP computations. The loop iterations are scheduled dynamically so that work is distributed as evenly as possible. There are no synchronization costs involved, as a processor can compute its own partial sum of the centrality value for each vertex, and all the sums can be merged in the end using a efficient reduction operation.

Thus, the stack S, list of predecessors P and the BFS queue Q, are replicated on each processor. Even for a graph of 100 million edges, with a conservative estimate of 10 GB memory usage by each processor, we can employ all 16 processors on our target SMP system, the IBM p570. We further optimize our implementation for scale-free graphs. Observe that in Algorithm 6, Q is not needed as the BFS frontier vertices are also added to S. To make the implementation cache-friendly, we use dynamic adjacency arrays instead of linked lists for storing elements of the predecessor set S. Note that $|\Sigma_{v \in V} P_s(v)| = O(m)$, and in most cases the predecessor sets of the few high-degree vertices in the graph are of comparatively greater size. Memory allocation is done as a preprocessing step before the actual BFS computations. Also, the indices computation time and memory requirements can be significantly reduced by decomposing the undirected graph into its biconnected components.

However, for a multithreaded system like the MTA-2, this simple approach will not be efficient. We need to exploit parallelism at a much finer granularity to saturate all the hardware threads. So we parallelize the actual BFS computation, and also have a coarse grained partition on the outer loop. In the previous section, we show that our parallel BFS implementation attains scalable performance for up to 40 processors on low-diameter, scale-free graphs. We use the loop *future* parallelism so that the outer loop iterations are dynamically scheduled, thus exploiting parallelism at a coarser granularity.

For the case of weighted graphs, Algorithm 6 must be modified to consider edge weights. The relaxation condition changes, and using a Fibonacci heap or pairing heap priority queue representation, it is possible to compute betweenness centrality for all the n vertices in $T_C = O(mn + n^2 \log n/p)$. We can easily adapt our SMP implementation to consider weighted graphs also. We intend to work on a parallel multithreaded implementation of SSSP on the MTA-2 for scale-free graphs in future.

An approximate algorithm for betweenness centrality is detailed in [19], which is again derived from the Eppstein–Wang algorithm [41]. As in the case of closeness centrality, the sequential running time can be reduced to $O(\log n/\epsilon^2(n + m))$ by setting the value of k appropriately. The parallel algorithms can also be derived similarly, by computing only k dependencies ($\delta[v]$ in Algorithm 6) instead of v, and taking a normalized average.

31.6.4 Performance Results

This section summarizes the experimental results of our centrality implementations on the Cray MTA-2 and the IBM p5 570. We report results on a 40-processor MTA-2, with each processor having a clock speed of 220 MHz and 4 GB of RAM. The IBM p5 570 is a 16-way SMP with 16 1.9 GHz Power5 cores with simultaneous multithreading (SMT), and 256 GB shared memory.

We test our centrality metric implementations on a variety of real-world graphs, summarized in Table 31.2. Our implementations have been extended to read input files in both PAJEK and UCINET graph formats. We also use a synthetic graph generator [25] to generate graphs obeying small-world

TABLE 31.2 Test Dataset Characteristics

Dataset	Source	Network description
ND-actor	[67]	An undirected graph of 392,400 vertices (movie actors) and 31,788,592 edges. An edge corresponds to a link between two actors, if they have acted together in a movie. The dataset includes actor listings from 127,823 movies
ND-web	[67]	A directed network with 325,729 vertices and 1,497,135 arcs (27,455 loops). Each vertex represents a web page within the Univ. of Notredame *nd.edu* domain, and the arcs represent from → to links
ND-yeast	[67]	Undirected network with 2114 vertices and 2277 edges (74 loops). Vertices represent proteins, and the edges interactions between them in the yeast network
PAJ-patent	[72]	A network of about 3 million U.S. patents granted between January 1963 and December 1999, and 16 million citations made among them between 1975 and 1999
PAJ-cite	[72]	The *Lederberg* citation dataset, produced using HistCite, in PAJEK graph format with 8843 vertices and 41,609 edges

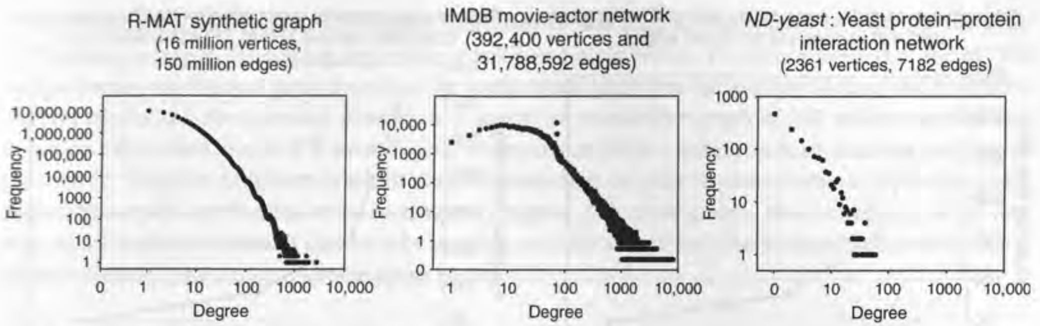

FIGURE 31.7 Degree distributions of some test graph instances.

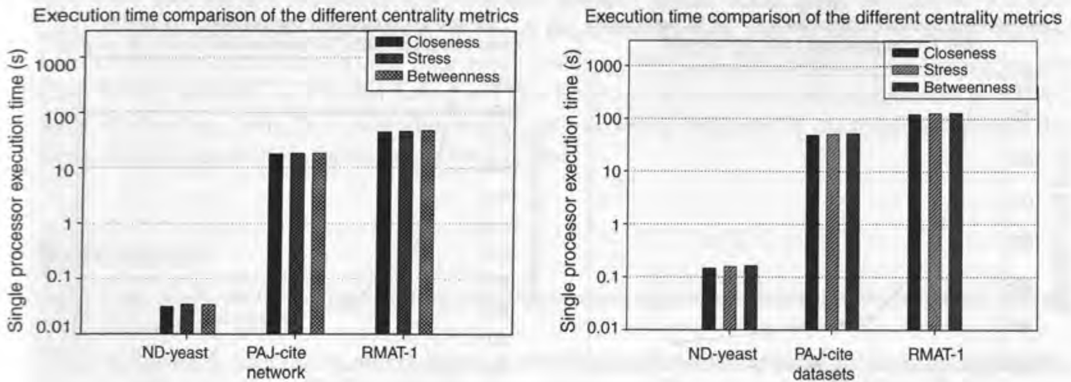

FIGURE 31.8 Single processor execution time comparison of the centrality metric implementations on the IBM p5 570 (left) and the Cray MTA-2 (right).

characteristics. The degree distributions of some test graph instances are shown in Figure 31.7. We can observe that on a log–log plot, the out-degree distribution is a straight line with a heavy tail, which is in agreement with prior experimental studies on scale-free graphs.

Figure 31.8 compares the single processor execution time of the three centrality metrics for three graph instances, on the MTA-2 and the p5 570. All three metrics are of the same computational complexity and show nearly similar running times in practice.

Figure 31.9 summarizes multiprocessor execution times for computing betweenness centrality, on the p5 570 and the MTA-2. Figure 31.9(a) gives the running time for the ND-actor graph on the p570. As expected, the execution time falls nearly linearly with the number of processors. It is possible to evaluate the centrality metric for the entire graph in around 42 minutes, on 16 processors. We observe similar performance for the patents citation graph. However, note that the execution time is highly dependent on the size of the largest nontrivial connected component in these real graphs. The patents network, for instance, is composed of several disconnected subgraphs, representing patents and citations in unrelated areas. However, it did not take significantly more time to compute the centrality indices for this graph compared to the ND-actor graph, even though this is a much bigger graph instance.

Figures 31.9(c) and (d) summarize the performance of the MTA-2 on ND-web, and a synthetic graph instance of the same size generated using the R-MAT algorithm. Again, note that the actual execution time is dependent on the graph structure; for the same problem size, the synthetic graph instance takes much longer than the ND-web graph. Compared to a four processor run, the execution time reduces significantly for 40 processors, but we do not attain performance comparable to our prior graph algorithm implementations [8,12]. We need to optimize our implementation to attain better system utilization, and

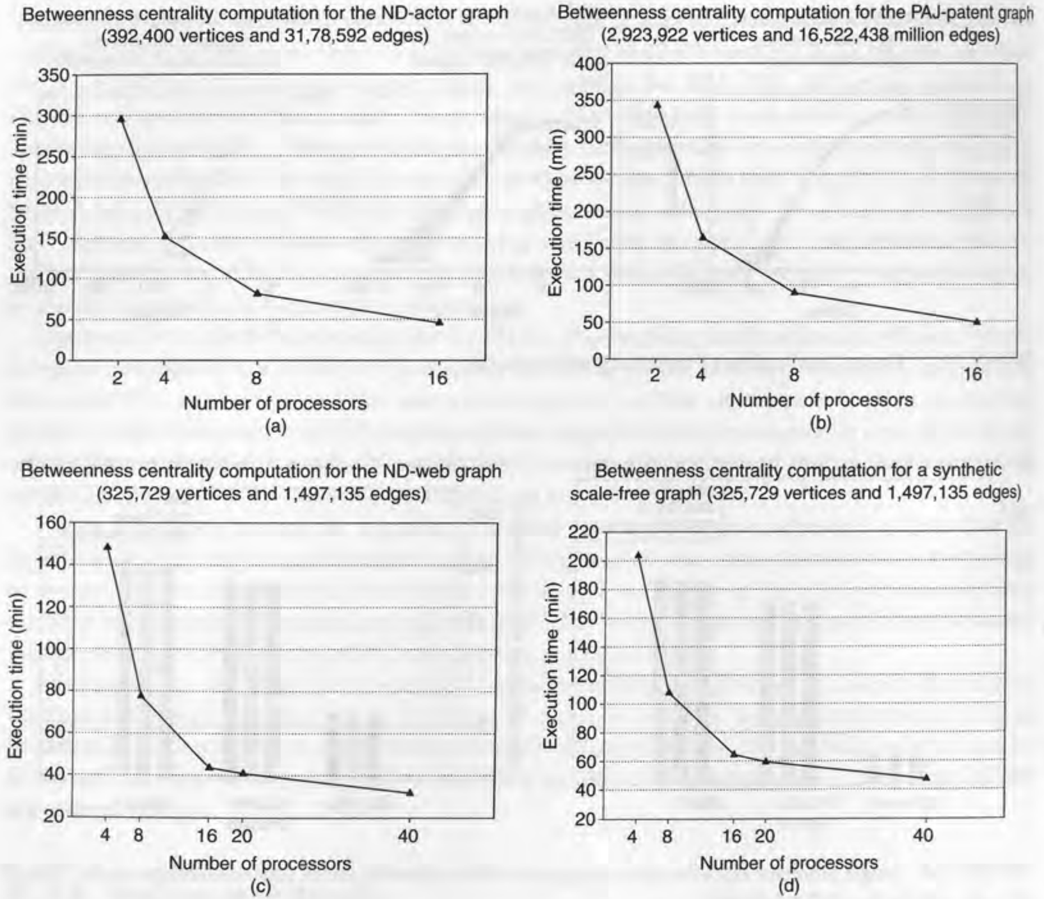

FIGURE 31.9 Multiprocessor performance of betweenness centrality for various graph instances on the IBM p5 570 [(a), (b)] and the Cray MTA-2 [(c), (d)].

the current bottleneck is due to automatic handling of nested parallelism. For better load balancing, we need to further preprocess the graph and manually assign teams of threads to the independent BFS computations. Our memory management routines for the MTA-2 implementation are not as optimized p570 routines, and this is another reason for drop in performance and scaling.

In summary, we present parallel algorithms for evaluating several network indices, including betweenness centrality, optimized for multithreaded architectures and SMP. To our knowledge, this is the first effort at parallelizing these widely used SNA tools. Our implementations are designed to handle problem sizes in the order of billions of edges in the network, and these are three orders of magnitude larger than instances that can be handled by current SNA tools. We are currently working on improving the betweenness centrality implementation on the MTA-2, and also extend it to efficiently compute scores for weighted graphs. In future, we plan to implement and analyze performance of approximate algorithms for closeness and betweenness centrality, and also apply betweenness centrality values to solve harder problems like graph clustering.

31.7 Conclusions

In this chapter, we discuss fast parallel algorithms for several fundamental combinatorial problems, optimized for multithreaded architectures. We implement the algorithms on the Cray MTA-2, a massively

multithreaded parallel system. One of the objectives of this chapter is to make the case for the multithreading paradigm for solving large-scale combinatorial problems. We demonstrate that the MTA-2 outperforms conventional parallel systems for applications where fine-grained parallelism and synchronization are critical. For instance, breadth-first search on a scale-free graph of 200 million vertices and 1 billion edges takes less than 5 seconds on a 40-processor MTA-2 system with an absolute speedup of close to 30. This is a significant result in parallel computing, as prior implementations of parallel graph algorithms report very limited or no speedup on irregular and sparse graphs, when compared to the best sequential implementation. It may now be possible to tackle several key PRAM algorithms [46,53,56,64] that have eluded practical implementations so far.

Acknowledgments

This work was supported in part by NSF Grants CAREER CCF-0611589, NSF DBI-0420513, ITR EF/BIO 03-31654, IBM Faculty Fellowship and Microsoft Research grants, NASA grant NP-2005-07-375-HQ, and DARPA Contract NBCH30390004. We thank Bruce Hendrickson and Jon Berry of Sandia National Laboratories for discussions on large-scale graph problems. We thank Richard Russell for sponsoring our Cray MTA-2 accounts, and Simon Kahan and Petr Konecny for their advice and suggestions on MTA-2 code optimization. John Feo (jofeo@microsoft.com), currently employed by Microsoft, performed this work while he was with his former employer Cray Inc.

References

[1] J. M. Abello and J. S. Vitter, editors. *External Memory Algorithms.* American Mathematical Society, Boston, MA, 1999.

[2] D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external-memory bfs algorithms. In *Proc. 17th Ann. Symp. Discrete Algorithms (SODA-06)*, pp. 601–610. ACM-SIAM, 2006.

[3] Analytic Technologies. UCINET 6 social network analysis software. http://www.analytictech.com/ucinet.htm

[4] B. Awerbuch and R. G. Gallager. A new distributed algorithm to find Breadth First search trees. *IEEE Trans. Inf. Theor.*, 33(3):315–322, 1987.

[5] B. Awerbuch, A. V. Goldberg, M. Luby, and S. A. Plotkin. Network decomposition and locality in distributed computation. In *IEEE Symp. on Foundations of Computer Science*, pp. 364–369, 1989.

[6] 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, April 2004.

[7] 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, April 2004.

[8] D.A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *Proc. 34th Int'l Conf. on Parallel Processing (ICPP)*, Oslo, Norway, June 2005.

[9] D.A. Bader, J. Feo, and *et al.* HPCS SSCA#2 Graph Analysis Benchmark Specifications v1.1, January 2006.

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

[11] D.A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *Proc. 12th Int'l Conf. on High Performance Computing*, Goa, India, Springer-Verlag, December 2005.

[12] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *Proc. 35th Int'l Conf. on Parallel Processing (ICPP)*, Columbus, OH, IEEE Computer Society, August 2006.

[13] D.A. 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.K. Prasanna, and U. Shukla, editors, *Proc. 9th Int'l Conf. on High Performance Computing (HiPC 2002)*, volume 2552 of Lecture Notes in Computer Science, pp. 63–75, Bangalore, India, Springer-Verlag, December 2002.

[14] A-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.

[15] G. Barnes and W. L. Ruzzo. Deterministic algorithms for undirected s-t connectivity using polynomial time and sublinear space. In *Proc. 23rd Annual ACM Symp. on Theory of Computing*, pp. 43–53, New York, ACM Press, 1991.

[16] V. Batagelj and A. Mrvar. Pajek—Program for large network analysis. *Connections*, 21(2):47–57, 1998.

[17] A. Bavelas. Communication patterns in task oriented groups. *J. Acous. Soc. Am.*, 22:271–282, 1950.

[18] U. Brandes. A faster algorithm for betweenness centrality. *J. Math. Soc.*, 25(2):163–177, 2001.

[19] U. Brandes and T. Erlebach, editors. *Network Analysis: Methodological Foundations*, volume 3418 of Lecture Notes in Computer Science. Springer-Verlag, 2005.

[20] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[21] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer Networks*, 33(1–6):309–320, 2000.

[22] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J.R. Westbrook. On external memory graph traversal. In *Proc. 11th Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 859–860, Philadelphia, PA, 2000.

[23] N. Buckley and M. van Alstyne. Does email make white collar workers more productive? Technical report, University of Michigan, 2004.

[24] D. S. Callaway, M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Network robustness and fragility: percolation on random graphs. *Physics Review Letters*, 85:5468–5471, 2000.

[25] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proc. 4th SIAM Intl. Conf. on Data Mining*, Florida, April 2004.

[26] B. V. Cherkassky, A.V. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Math. Programm.*, 73:129–174, 1996.

[27] D. Cisic, B. Kesic, and L. Jakomin. Research of the power in the supply chain. International Trade, Economics Working Paper Archive EconWPA, April 2000.

[28] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Comm. ACM*, 47(3):45–47, 2004.

[29] R. Cohen, K. Erez, D. Ben-Avraham, and S. Havlin. Breakdown of the internet under intentional attack. *Phys. Rev. Lett.*, 86:3682–3685, 2001.

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

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

[32] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Inc., Cambridge, MA, 1990.

[33] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *Proc. 23rd Int'l Symp. on Mathematical Foundations of Computer Science*, pp. 722–731, London, UK, Springer-Verlag, 1998.

[34] Cray Inc. The MTA-2 multithreaded architecture. http://www.cray.com/products/systems/mta

[35] DARPA Information Processing Technology Office. High productivity computing systems (HPCS) project, 2004. http://www.darpa.mil/ipto/programs/hpcs/

[36] F. Dehne, A. Ferreira, E. Cáceres, S. W. Song, and A. Roncato. Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. *Algorithmica*, 33:183–200, 2002.

[37] A. del Sol, H. Fujihashi, and P. O'Meara. Topology of small-world networks of protein–protein complex structures. *Bioinformatics*, 21(8):1311–1315, 2005.

[38] C. Demetrescu, A. Goldberg, and D. Johnson. 9th DIMACS implementation challenge—shortest paths. http://www.dis.uniroma1.it/~challenge9

[39] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271, 1959.

[40] P. Doreian and L.H. Albert. Partitioning political actor networks: some quantitative tools for analyzing qualitative networks. *Quant. Anthropol.*, 161:279–291, 1989.

[41] D. Eppstein and J. Wang. Fast approximation of centrality. In *Proc. 12th Ann. Symp. Discrete Algorithms (SODA-01)*, Washington, DC, 2001.

[42] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power–law relationships of the Internet topology. In *Proc. ACM SIGCOMM*, pp. 251–262, 1999.

[43] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[44] L. C. Freeman. *The Development of Social Network Analysis: A Study in the Sociology of Science.* Booksurge Pub., 2004.

[45] H. Gazit and G. L. Miller. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Inf. Process. Lett.*, 28(2):61–65, 1988.

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

[47] J-L. Guillaume and M. Latapy. Bipartite graphs as models of complex networks. In *Proc. 1st Int'l Workshop on Combinatorial and Algorithmic Aspects of Networking*, 2004.

[48] R. Guimera, S. Mossa, A. Turtschi, and L.A.N. Amaral. The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. *Proc. Natl. Acad. Sci. USA*, 102(22):7794–7799, 2005.

[49] 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, pp. 37–56, Baltimore, MD, Springer-Verlag, January 1999.

[50] D. R. Helman and J. JáJá. Prefix computations on symmetric multiprocessors. *J. Parallel Distrib. Comput.*, 61(2):265–278, 2001.

[51] W. Hoeffding. Probability inequalities for sums of bounded random variables. *J. Am. Stat. Assoc.*, 58:713–721, 1963.

[52] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. 4th European Conf. on Principles of Data Mining and Knowledge Discovery*, pp. 13–23, London, UK, 2000.

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

[54] H. Jeong, S. P. Mason, A.-L. Barabasi, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411:41, 2001.

[55] G. Karypis, E. Han, and V. Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.

[56] P. N. Klein and J. H. Reif. An efficient parallel algorithm for planarity. *J. Comp. Syst. Sci.*, 37(2):190–246, 1988.

[57] V. Krebs. InFlow 3.1—Social network mapping software, 2005. http://www.orgnet.com

[58] V. E. Krebs. Mapping networks of terrorist cells. *Connections*, 24(3):43–52, 2002.

[59] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley & Sons, Inc., New York, 1990.

[60] H. R. Lewis and C. H. Papadimitriou. Symmetric space-bounded computation (extended abstract). In *Proc. 7th Colloquium on Automata, Languages and Programming*, pp. 374–384, London, UK, 1980.

[61] F. Liljeros, C. R. Edling, L. A. N. Amaral, H. E. Stanley, and Y. Aberg. The web of human sexual contacts. *Nature*, 411:907, 2001.

[62] U. Meyer. External memory BFS on undirected graphs with bounded degree. In *Proc. 12th Annu. ACM-SIAM Symp. on Discrete Algorithms*, pp. 87–88, Philadelphia, PA, 2001.

[63] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[64] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proc. 26th Ann. IEEE Symp. Foundations of Computer Science*, pp. 478–489, Portland, OR, October 1985.

[65] B. M. E. Moret, D. A. Bader, and T. Warnow. High-performance algorithm engineering for computational phylogenetics. In *Proc. Int'l Conf. on Computational Science*, volume 2073–2074 of Lecture Notes in Computer Science, San Francisco, CA, Springer-Verlag, 2001.

[66] B. M. E. Moret, D. A. Bader, T. Warnow, S.K. Wyman, and M. Yan. GRAPPA: a high-performance computational tool for phylogeny reconstruction from gene-order data. In *Proc. Botany*, Albuquerque, NM, August 2001.

[67] Notredame CNet resources. http://www.nd.edu/~networks

[68] M. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[69] M. E. J. Newman. Scientific collaboration networks: shortest paths, weighted networks and centrality. *Phys. Rev. E*, 64, 2001.

[70] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graph models of social networks. *Proc. Natl. Acad. of Sci.*, USA, 99:2566–2572, 2002.

[71] U. J. Nieminen. On the centrality in a directed graph. *Soc. Sci. Res.*, 2:371–378, 1973.

[72] PAJEK datasets. http://www.vlado.fmf.uni-lj.si/pub/networks/data/

[73] C. R. Palmer and J. G. Steffan. Generating network topologies that obey power laws. In *Proc. ACM GLOBECOM*, November 2000.

[74] J. Park, M. Penner, and V. K. Prasanna. Optimizing graph algorithms for improved cache performance. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2002)*, Fort Lauderdale, FL, April 2002.

[75] R. Pastor-Satorras and A. Vespignani. Epidemic spreading in scale-free networks. *Phys. Rev. Lett.*, 86:3200–3203, 2001.

[76] J. W. Pinney, G. A. McConkey, and D. R. Westhead. Decomposition of biological networks using betweenness centrality. In *Proc. Poster Session of the 9th Ann. Int'l Conf. on Research in Computational Molecular Biology (RECOMB 2004)*, Cambridge, MA, May 2005.

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

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

[79] W. Richards. International network for social network analysis, 2005. http://www.insna.org

[80] W. Richards. Social network analysis software links, 2005. http://www.insna.org/INSNA/soft_inf.html

[81] G. Sabidussi. The centrality index of a graph. *Psychometrika*, 31:581–603, 1966.

[82] J. P. Scott. *Social Network Analysis: A Handbook*. SAGE Publications, 2000.

[83] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51(3):400–403, 1995.

[84] A. Shimbel. Structural parameters of communication networks. *Math. Biophy.*, 15:501–507, 1953.

[85] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.

[86] J. L. Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Comput.*, 21(9):1505–1532, 1995.

[87] J. Ullman and M. Yannakakis. High-probability parallel transitive closure algorithms. In *Proc. 2nd Ann. Symp. Parallel Algorithms and Architectures (SPAA-90)*, pp. 200–209, New York, 1990. ACM Press.

[88] University of Virginia. Oracle of Bacon. http://www.oracleofbacon.org

[89] A. Vazquez, A. Flammini, A. Maritan, and A. Vespignani. Global protein function prediction in protein–protein interaction networks. *Nat. Biotechnol.*, 21(6):697–700, June 2003.

[90] S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications.* Cambridge University Press, 1994.

[91] D. J. Watts and S. H. Strogatz. Collective dynamics of small world networks. *Nature*, 393:440–442, 1998.

[92] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and Ü. V. Çatalyürek. A scalable distributed parallel breadth-first search algorithm on Blue Gene/L. In *Proc. Supercomputing (SC 2005)*, Seattle, WA, November 2005.

[93] D. H. Zanette. Critical behavior of propagation on small-world networks. *Phys. Rev. E*, 64, 2001.

[94] Cray Inc. The Cray XMT platform. http://www.cray.com/products/xmt/

[95] J. Feo, D. Harper, S. Kahan, and P. Konecny. ELDORADO. In N. Bagherzadeh, M. Valero, and A. Ramírez, eds., *Proc. 2nd Conf. on Computing Frontiers*, pp. 28–34, Ischia, Italy, ACM Press, 2005.