Implementing a Portable Multi-threaded Graph Library: the MTGL on Qthreads

Brian W. Barrett, Jonathan W. Berry, and Richard C. Murphy
Sandia National Laboratories
Albuquerque, NM
{bwbarre,jberry,rcmurph}@sandia.gov

Kyle B. Wheeler
University of Notre Dame and
Sandia National LaborAroties
Notre Dame, IN
kwheeler@cse.nd.edu

1. Introduction

Graph-based Informatics applications challenge traditional high-performance computing (HPC) environments due to their unstructured communications and poor load-balancing. As a result, such applications have typically been relegated to either poor efficiency or specialized platforms, such as the Cray MTA/XMT series. The multi-threaded nature of the Cray MTA architecture[1] presents an ideal platform for graph-based informatics applications. As commodity processors adopt features to enable greater levels of multi-threaded programming and higher memory densities, the ability to run these multi-threaded algorithms on less expensive, more available hardware becomes attractive.

The Cray MTA architecture provides both an auto-threading compiler and a number of architectural features to assist the programmer in developing multi-threaded applications. Unfortunately, commodity processors have increased the amount of concurrency available by adding an ever-growing number of processor cores on a single socket, but have not added the fine-grained synchronization available on the Cray MTA architecture. Further, while auto-threading compilers are being discussed, none provide the feature set of the Cray offerings.

Although massively multi-threaded architectures have shown tremendous potential for graph algorithms, development poses unique challenges. Algorithms typically use light-weight synchronization primitives (Full/Empty bits, discussed in Section 3.1) for synchronization. Parallelism is not expressed explicitly, but instead compiler hints and careful code construction allow the compiler to parallelize a given code. Unlike the Parallel Boost Graph Library (PBGL) [8], which runs on the developers laptop as well as the largest supercomputers, applications developed for the MTA architecture only run on the MTA architecture. Experiments with the programming paradigm require access to the platform, which is obviously a constrained resource.

In this paper, we explore the possibility of using the Qthreads user-level threading library to increase the portability of scalable multi-threaded algorithms. The Multi-Threaded Graph Library (MTGL) [2], which provides generic programming access to the XMT, is our testbed for this work. We show the use of important algorithms from the MTGL on on emerging commodity multi-core and multi-threaded platforms, with only minor changes to the code base. Although performance is not at the same level as the same algorithm on a Cray XMT, the performance motivates our technique as a workable solution for developing multi-threaded codes for a variety of architectures.

1. Throughout this paper, we will use the phrase MTA architecture to refer to Cray's multi-threaded architecture, including both the Cray MTA-2 and Cray XMT platforms.

2. Background

Recent work [9], [12] has described the challenges in HPC graph processing. These challenges are fundamentally related to locality (both spatial and temporal), and the lack thereof when graph algorithms are applied to highly unstructured datasets. The PBGL [8] attempts to meet these challenges through storage techniques that reduce communication. These techniques have been shown to work well in certain contexts, though they introduce other challenges such as memory scalability. Even when they achieve run-time scalability, the processor utilization on commodity CPUs is considerably lower than that found in the MTA architecture.

2.1. Cray XMT

The Cray XMT is the successor to the Cray MTA-2 highly multi-threaded architecture. Unlike the MTA-2, in which all memory was equidistant from any processor on the network, the XMT uses a more traditional model in which memory is closer to a single processor than all others. The Cray XMT utilizes similar processors to the MTA-2, including the ability to sustain 128 simultaneous hardware threads, but with an improved 500 MHz clock rate. Rather than the custom network found on the MTA-2, the XMT utilizes the SeaStar based network found on the Cray XT massively parallel processor distributed memory platform.

2.2. Multi-core Architectures

As processor vendors have begun offering quad-core processors, as well as more commodity multi-threaded processors such as the Sun Niagara processors, it has become

U.S. Government work not protected by U.S. copyright

possible to write multi-threaded applications on more traditional platforms. Given the high cost of even a small XMT platform, the ability of modern workstations to support a growing number of threads makes them attractive for algorithm development and experimentation.

The Sun Niagara platform opens even greater multi-threaded opportunities, supporting 8 threads per core and 8 cores per socket, for a total of 64 threads per socket. Current generation Niagara processors support single, dual, and quad socket installations. Unlike the Cray XMT, the Sun Niagara uses a more traditional memory system, including L1 and shared L2 cache structures, and an unhashed memory system. The machines are also capable of running unmodified UltraSPARC executables.

2.3. Multi-threaded Programming

Our approach is to take algorithm codes that have already been carefully designed to perform on the MTA architecture, and run them without altering the core algorithm on commodity multi-core machines by simulating the threading hardware. In contrast, codes written using frameworks specifically designed for multi-core commodity machines (e.g. SWARM [13]) won't run on the MTA architecture.

Standard multi-core software designs, such as Intel's Thread Building Blocks [10], OpenMP [6], and Cilk [3], target current multicore systems, and their architecture reflects this. For example, they lack a means of associating threads with a locale. This becomes a significant issue as machines get larger and memory access becomes more non-uniform.

Another important consideration is the granularity and overhead of synchronization. Existing large scale multi-threaded hardware, such as the XMT, implement full/empty bits. This provides for blocking synchronization in a locality-efficient way. Existing multi-threaded software systems tend to use lock-based techniques, such as mutexes and spinlocks, or require tight control over memory layout. These methods are logically equivalent, but are not as efficient to implement. FEB's are memory efficient when implemented in hardware, and thus allow tight memory structures that can be safely operated upon without requiring locking structures to be inserted into them.

3. Qthreads

The Qthread API [16] is a library-based API for accessing lightweight threading and synchronization primitives similar to those provided on the MTA architecture. The API was designed to support large-scale lightweight threading and synchronization in a cross-platform library that can be readily implemented on both conventional and massively parallel architectures. On architectures where there is no hardware support for the features it provides, or where native threads are heavyweight, these features are emulated. There are several existing threading models that support lightweight threading and lightweight synchronization, but none that sufficiently closely emulate the MTA architecture semantics.

Equivalents for basic thread control, FEB-based read and write functions, as well as basic threaded loops (analogs for many of the pragma-defined compiler loop optimizations available on the MTA architecture) are all provided by the API. Even though the operations that do not have hardware support, such as FEB-based operations, are emulated, they retain usefulness as a means of intra-thread communication.

The API establishes convenient management of the basic memory requirements of threads as they are created. When insufficient resources are available, either thread creation fails or it waits for the resources to become available, depending on how the API is used.

Relatively speaking, locality of reference is not an important consideration to the MTA architecture, as the address space is hashed and divided among all processors at word boundaries. This is an unusual environment, and locality is an important consideration in most other large parallel machines. To address this, the Qthread API provides a generalized notion of locality, called a "shepherd", which identifies the location of a thread. A machine may be described to the library as a set of shepherds, which can refer to memory boundaries, CPUs, nodes, or whatever is a useful division. Threads are assigned to specific shepherds when they are created.

3.1. Implementation of MTA Intrinsics

The MTA architecture has several features that are intrinsic to the architecture, which the Qthread library emulates. These features include full/empty bits (FEBs), fast atomic increments, and conditionally created threads.

On the MTA architecture, a full/empty bit (FEB) is an extra hardware flag associated with every word in memory, marking that word either full or empty. Qthreads uses a centralized collection data structure to achieve the same effect: if an address is present in the collection, it is considered "empty", and if not, it is considered "full". Thus, all memory addresses are considered full until they are operated upon by one of the commands that will alter the memory word's contents and full/empty status. The synchronization protecting each word is pushed into the centralized data structure. Not all of the semantics of the MTA architecture can be fully emulated, however. For example, on the MTA architecture, all writes to memory implicitly mark the corresponding memory words as full. However, when pieces of memory are being used for synchronization purposes, even implicit operations are done purposefully by the programmer, and replacing implicit writes with explicit calls is trivial.

The MTA architecture also provides a hardware atomic increment intrinsic. Atomic increment functions have often

been considered useful, even on commodity architectures, and so hardware-based techniques for doing atomic increments are common. The Qthread API provides an atomic increment function that uses a hardware-based implementation on supported architectures, but which falls back to using emulated locks to achieve the same behavior on architectures without explicit hardware support in the library. This is an example of opportunistically using hardware features while providing a standardized interface; a key feature of the Qthread API.

3.2. Qthreads implementation of thread virtualization

Conditionally created threads are called "futures" in MTA architecture terminology, and are used to indicate that threads need not be created now, but merely whenever there are resources available for them. This can be crucial on the MTA, as each processor can handle at most 128 threads, and extremely parallel algorithms may generate significantly more. The Qthread API provides an analogous feature by providing alternate thread creation semantics that allow the programmer to specify the permissible number of threads that may exist concurrently, and which will stall thread creation until the number of threads is less than the number of permissible threads.

A key application of this is in loops. While a given loop may have a large number of entirely independent iterations, it is typically unwise to spawn all of the iterations as threads, because each thread has a context and eventually the machine will run out of memory to hold all the thread contexts. Limiting the number of concurrently extant threads limits the amount of overhead that will be used by the threads. In a loop, the option to stall the thread creation while the maximum number of threads still exist provides the ability to specify a threaded loop without the risk of using an excessive amount memory for thread contexts. The limit on the number of threads is a per-shepherd limit, which helps with load balancing.

4. The Multi-Threaded Graph Library

The Multi-Threaded Graph Library is a graph library designed in the spirit of the Boost Graph Library (BGL) and Parallel Boost Graph Library. The library utilizes the generic component features of the C++ language to allow flexibility in graph structures, without changes to a given algorithm. Unlike the distributed memory, message passing based PBGL, the MTGL was designed specifically for the shared-memory multi-threaded MTA architecture. The MTGL includes a number of common graph algorithms, including the breadth-first search, connected components, and PageRank algorithms discussed in this paper.

To facilitate writing new algorithms, the MTGL provides a small number of basic intrinsics upon which graph algorithms can be implemented. The intrinsics hide much of the complexity of multi-threaded race conditions and load-balancing from algorithm developers and users. Parallel Search (PSearch), a recursive parallel variant of depth-first search (which is not truly depth-first in order to achieve parallelism), combined with an extensive vertex and edge visitor interface, provides powerful parallelism for a number of algorithms.

MTA architecture-specific features used by the MTGL are either compiler hints specified via the `#pragma` mechanism or are encapsulated into a limited number of templated functions, which are easily re-implementable for a new architecture. An example is the `mt_readfe` call, which translates to `readfe` on the MTA architecture, a simple read for serial builds on commodity architectures, and `qthread_readfe` on commodity architectures using the Qthreads library.

The combination of an internal interface for explicit parallelism and the set of core intrinsics upon which much of the MTGL is based provides an ideal platform for extension to new platforms. While auto-threading compilers like those found on the MTA architecture are not available for other platforms, the small number of intrinsics can be hand-parallelized with a reasonable amount of effort.

5. Qthreads and the MTGL

Making the MTGL into a cross-platform library required overcoming significant development challenges. The MTA architecture programming environment has a large number of intrinsic semantics, and its cacheless hashed memory architecture has unusual performance characteristics. The MTA compiler also recognizes common programming patterns, such as reductions, and optimizes them transparently. For these reasons, the MTA developer is encouraged to develop "close to the compiler".

The size of stack necessary, for example, presents a challenge. Some MTGL routines are highly recursive, and the MTA transparently handles expanding the stack for each thread as-needed. The Qthread library, however, has a fixed stack size. Iterative solutions, combined with using larger stacks was required to address the issue.

Both the MTA architecture and commodity processors are susceptible to the problem of hot spotting, performance degradation due to repeated access to the same memory location. The MTA architecture suffers from both read and write hot spotting, due to constraints in traffic across the platform's network. Commodity processors, however, provide cache structures to improve performance and benefit from read hot spotting. Commodity architectures also have a larger granularity of memory sharing: a cache line, which can be as large as 64 bytes. Concurrent writes within a cache

line create a hot spot, even if the writes affect independent addresses. The cache was a consideration for atomic operations as well, as they typically cause a cache flush to memory. Avoiding atomic operations where possible, such as in reductions, is important for performance.

6. Multi-platform Graph Algorithms

We consider three graph kernel algorithms: a search, a component finding algorithm, and an algebraic algorithm. There are myriad other graph algorithms, but we use these three as primitive representatives on which other algorithms can be built.

6.1. BFS

Breadth-first search (BFS) is, perhaps, the most fundamental of graph algorithms. Given a vertex v, find the neighbors of v, then the neighbors of those neighbors, etc. Furthermore BFS is well-suited for parallelization. Pseudocode for BFS from [5] is included in Figure 1.

```
BFS(G,s)
1  for each vertex u ∈ V[G] − {s}
2      do color[u] ← WHITE
3         d[u] ← inf
4  color[s] ← GRAY
5  d[s] ← 0
6  Q ← ∅
7  while Q ≠ 0
8      do u ← DEQUEUE(Q)
9          for each vertex v ∈ Adj[u]
10             do if color[v] ← WHITE
11                then color[v] ← GRAY
12                     d[v] ← d[u] + 1
13                     ENQUEUE(Q,v)
14        color[v] ← BLACK
```

Figure 1. The basic BFS algorithm

There are two inherent problems with using this basic algorithm in a multithreaded environment. The first is that a parallel version of the *for* loop beginning on Line 9 will make many synchronized writes to the *color* array. This is a problem on machines like the Niagara regardless of the data characteristics. It is also a problem on the XMT if there is a vertex v of high in-degree (since many vertices u would test v's color simultaneously, making it a hot spot).

The second problem is even more basic: the ENQUEUE operation of Line 13 typically involves incrementing a tail pointer. As all threads will increment this same location, it is an obvious hot spot.

We avoid these problems by chunking and sorting: suppose that the next BFS level contains k vertices, whose adjacency lists have combined length l. We divide the work of processing these adjacencies into $\lceil l/C \rceil$ chunks, each of size C (except for the last one). Then $\lceil l/C \rceil$ threads process the chunks individually, saving newly discovered vertices to local stores. Each thread can then increment the Q tail pointer only once, mitigating that hot spot. However, in order to handle the *color* hot spot, we do not write the local stores directly into the Q. Rather, we concatenate them into a buffer, sort that buffer with a thread-safe sorting routine (qsort in Qthreads, or a counting sort on the XMT), then have a single thread put the unique elements of this array into the Q. This thread does linear work in serial, but the "hot spot" is now used to advantage in cache-based multicore architectures.

A better BFS algorithm is known for the XMT. Although we currently do not have an implementation of this algorithm, it would be a straightforward exercise to incorporate it into the MTGL so that the same program could run efficiently on either type of platform.

6.2. Connected Components

A connected component of a graph G is a set S of vertices with the property that any pair of vertices $u, v \in S$ are connected by a path. Finding connected components is a prerequisite for dividing many graph problems into smaller parts. The canonical algorithm for finding connected components in parallel is the Shiloach-Vishkin algorithm (SV) [15], and the MTGL has an implementation of this algorithm that roughly follows [1].

Unfortunately, a key property of many real-world datasets will limit the performance of SV in practice. Specifically, it is known both theoretically [7] (for random graphs), and in practice (for interaction networks such, the World-Wide Web, and many social networks) that the majority of the vertices tend to be grouped into one "giant component" (GCC). Algorithms like SV work by assigning a representative to each vertex. Toward the end of these algorithms, all vertices in the GCC are pointing at the same representative, making it a severe hot spot.

We adopt a simple alternative to SV, which we call GCC-SV. It is overwhelmingly likely (though we do not not provide any formal analysis here) that the vertex of highest degree is in the GCC. Given this assumption, we BFS from that vertex using the method of Section 6.1 (or psearch on the XMT), then collect all *orphaned edges* that do not link vertices discovered during this search. Running SV on the subgraph induced by the orphaned edges we find the remaining components. This subproblem is likely to be small enough so that even if the largest component of the induced subgraph is a GCC of that graph (which is likely), the running time is dwarfed by that of the original BFS. If there is no GCC in the original graph, then the original SV would perform well.

6.3. PageRank

```
#pragma mta assert nodep
for (int i=0; i<n; i++) {
    double total=0.0;
    int begin = g[i];
    int end = g[i+1];
    for (int j=begin; j<end; j++) {
        int src = rev_end_points[j];
        double r = rinfo[src].rank;
        double incr = (r/rinfo[src].degree);
        total += incr;
    }
    rinfo[i].acc = total;
}
```

Figure 2. The MTGL code for PageRank's inner loop on the XMT

PageRank, the algorithm made famous by Google for ranking web pages [14], is a linear algebraic technique for modeling the propagation of *votes* through a directed graph, where each page contributes a fraction of its vote to each of its out-neighbors. Ranks continue propagating until convergence. A thorough mathematical explanation of PageRank is beyond the scope of this paper. However, at an abstract level PageRank is a sequence of matrix-vector multiplications, each followed by a normalization step. In graph terms, the most computationally expensive portion of the algorithm is simply traversing all of the adjacencies in the graph in order to accumulate votes.

Figure 2 shows the vote accumulation loops of PageRank used by the MTGL on the XMT. The structure of these loops enables the XMT compiler to merge them into one, and to remove the reduction of votes into the variable `total` from the final line of the inner loop. The result is excellent performance. We simulate this in a Qthread-enabled version of this code in the MTGL in order to achieve good scaling on multi-core machines.

6.4. R-MAT graphs

R-MAT [4] is a parameterized generator of graphs that can mimic real-world datasets. The term stands for "Recursive-MATrix," derived from the generation procedure, which is a simulation of repeated *Kronecker products* [11] of the adjacency matrix by itself. Intuitively, the R-MAT procedure can be thought of as repeatedly dropping marbles through a series of plastic trays. The topmost one typically is divided into 4 quadrants, the second one into 16, etc. The bottom tray is the adjacency matrix. At each level, a marble will pass through one of 4 holes with probability given by 4 input parameters; a, b, c, d. Multiple edges are not allowed, so if a marble ends up on top of another marble in the adjacency matrix, it is discarded and we try again.

Varying the parameters a, b, c, d determines much about the structure of the resulting graph. For example, using $a = 0.25, b = 0.25, c = 0.25, d = 0.25$ would generate an Erdös-Rényi random graph. Putting more weight on one of the quadrants tends to generate an inverse power-law degree distribution, which is found in many real datasets.

In our experiments we generate two different classes of R-MAT graphs:

- *nice* graphs have $a = 0.45, b = 0.15, c = 0.15, d = 0.25$. These graphs feature two natural communities at each of many levels of recursion (quadrants a and d). However, even in graphs a quarter of a billion edges, the maximum vertex degree is only roughly a thousand.
- *nasty* graphs have $a = 0.57, b = 0.19, c = 0.19, d = 0.05$. These feature a much steeper degree distribution, with a maxmimum degree of roughly 200,000 in our quarter-billion edge example. Load balancing would naturally be more challenging in this case.

Furthermore, we label our graphs with the exponent of the number of vertices and hold the average degree at a constant 16, since this is relatively close to (though an over-estimate of) the average degree of a page in the WWW. For example, graph "R-MAT 21 Nasty" has 2^{21} vertices, 2^{24} undirected edges, and R-MAT parameters as given above.

7. Multiplatform Experiments

We compare performance of the three graph kernel algorithms described in Section 6—breadth-first search, connected components, and PageRank—on three platforms capable of executing multiple threads simultaneously: the Cray XMT, the Sun Niagara T2, and a traditional multi-socket, multi-core platform.

The Cray XMT used in testing contains 64 500 MHz ThreadStorm processors, each capable of sustaining 128 simultaneous hardware threads and 500 GB of shared memory. The SeaStar based network is a 3-d torus in a $8x4x2$ configuration. The system was running version 6.2.1 of the XMT operating system.

A Sun SPARC Enterprise T5240 server, with two 1.2 GHz UltraSPARC T2 processors, each capable of sustaining 64 simultaneous hardware threads, was also used in testing. The system contains 128 GB of memory and was running Sun Solaris 10, 5/08 Release. The Sun CoolThreads version of GCC was used to compile all tests.

Finally, a quad-socket, quad-core Opteron system, clocked at 2.2 GHz, provides a traditional multi-core environment. The system provides 32 GB of memory and is running Red Hat EL 5.1. GCC 4.1.2 was used to compile all tests.

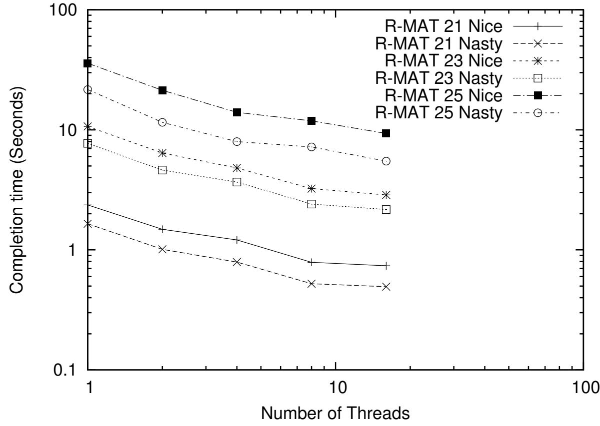

Figure 3. Opteron Breadth-First Search

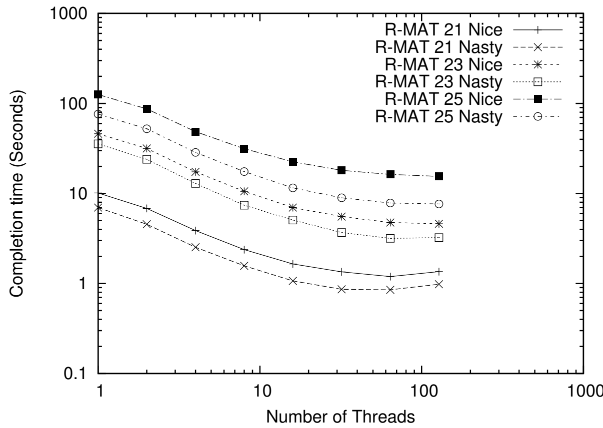

Figure 4. Niagara T2 Breadth-First Search

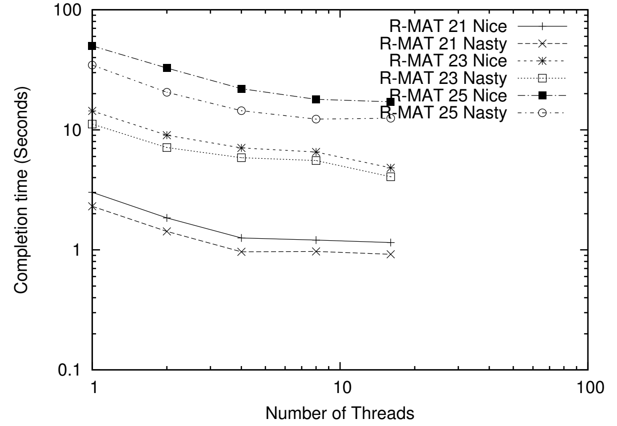

Figure 5. Opteron Connected Components GCC-SV

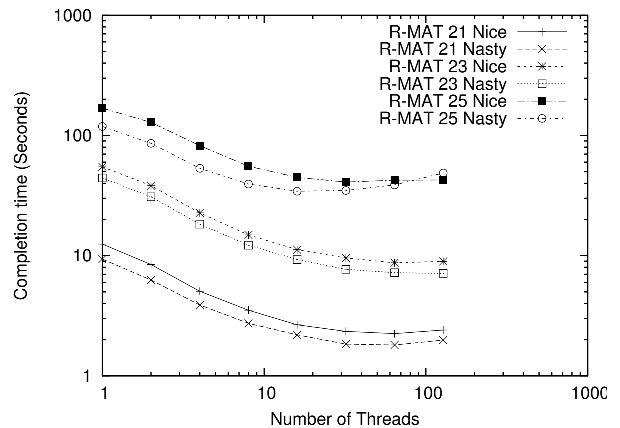

Figure 6. Niagara T2 Connected Components GCC-SV

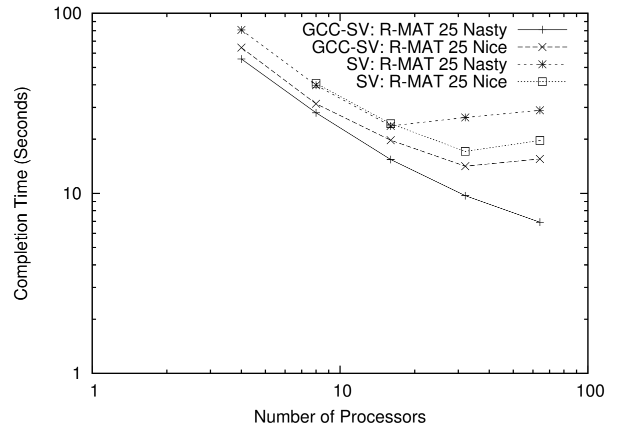

Figure 7. Cray XMT Connected Components - GCC-SV and SV

7.1. Breadth-First Search

We find that our method of avoiding hot spots in BFS enables scaling beyond what would be achievable by a naive algorithm. At the time of this writing, our implementation runs on the XMT, but does not perform as well as native XMT BFS implementations have done in the past. However, our method does leverage the multi-core and Niagara platforms effectively. As implied before, MTGL programmers will run BFS by associating a visitor object with the kernel algorithm, then running the latter. Underlying differences in the kernel implementation, such as that likely in the XMT implementation of BFS, will be hidden from the programmer.

7.2. Connected Components

Our connected components codes demonstrate strong scaling on multi-core and Niagara, as the GCC-SV algorithm is dominated by a single run of BFS on the realistic datasets we address. Furthermore, we are able to demonstrate strong scaling on the XMT as well by replacing the BFS by the recursive psearch. Note the effect of data on algorithm

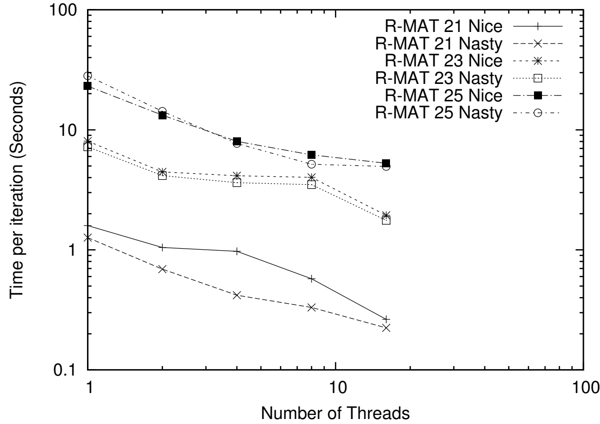

Figure 8. Opteron PageRank

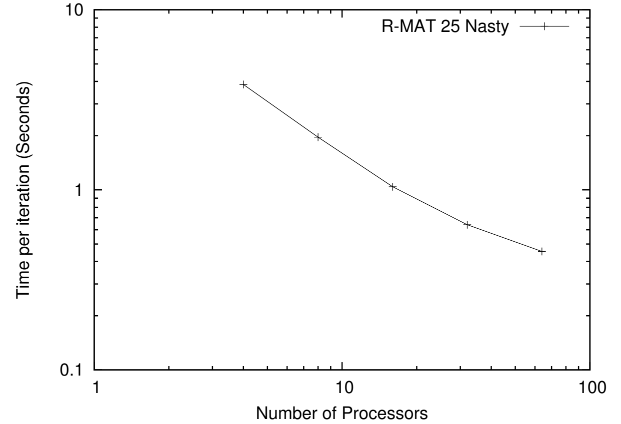

Figure 10. Cray XMT PageRank

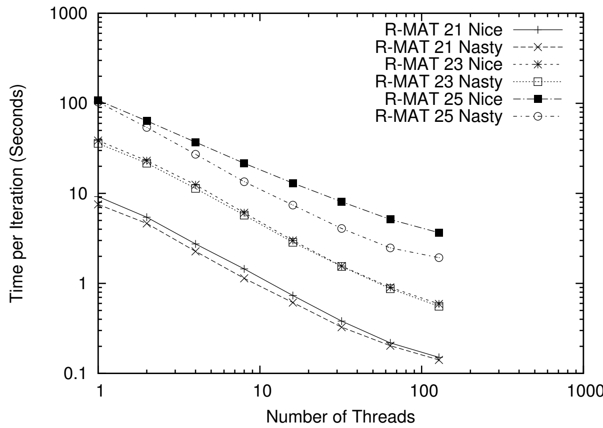

Figure 9. Niagara T2 PageRank

performance in Figure 7. Ironically, the "nasty" datasets are most friendly to the algorithm, as the vast majority of all vertices fall into the GCC in this case. As we consider the "nice" datasets, this GCC membership becomes less pathological (and less realistic). Therefore, the inherently hot spotting SV algorithm has more work to do once the GCC has been processed.

7.3. PageRank

As we saw in Figure 2, PageRank can be written to leverage the auto-parallelizing compiler of the XMT quite effectively. We cannot match the XMT's performance in emulation without work to reconstruct the compiler's optimization. However, a straightforward parallelization of the outer loop using qthreads still provides significant benefit, as we see in Figures 8 and 9.

8. Conclusions and future work

Developing multi-threaded graph algorithms, even when using the MTGL infrastructure, provides a number of challenges, including discovering appropriate levels of parallelism, preventing memory hot spotting, and eliminating accidental synchronization. In this paper, we have demonstrated that using the combination of Qthreads and MTGL with commodity processors enables the development and testing of algorithms without the expense and complexity of a Cray XMT. While achievable performance is lower for both the Opteron and Niagara platform, performance issues are similar.

While we believe it is possible to port Qthreads to the Cray XMT, this work is still on-going. Therefore, porting work still must be done to move algorithm implementations between commodity processors and the XMT. Although it is likely that the Qthreads-version of an algorithm will not be as optimized as a natively implemented version of the algorithm, such a performance impact may be an acceptable trade-off for ease of implementation.

9. Acknowledgments

The BFS used in this paper leverages a load balancing algorithm to divide adjacencies into chunks. Cynthia Phillips and Jon Berry developed an preliminary version of this algorithm in 2008.

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] BADER, D., CONG, G., AND FEO, J. On the architectural requirements of efficient execution of graph algorithms. In

The 33rd International Conference on Parallel Processing (ICPP) (2005), pp. 547–556.

[2] BERRY, J. W., HENDRICKSON, B. A., KAHAN, S., AND KONECNY, P. Software and algorithms for graph queries on multithreaded architectures. In *Proceedings of the International Parallel & Distributed Processing Symposium* (2007), IEEE.

[3] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not. 30*, 8 (1995), 207–216.

[4] CHAKRABARTI, D., ZHAN, Y., AND FALOUTSOS, C. R-mat: A recursive model for graph mining. In *In SDM* (2004).

[5] CORMAN, T., LEISERSON, C., RIVEST, R., AND STEIN, C. *Introduction to Algorithms*. MIT Press, 2001.

[6] DAGUM, L., AND MENON, R. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering 05*, 1 (1998), 46–55.

[7] ERDÖS, P., AND RÉNYI, A. On random graphs I. *Publicationes Mathematicae*, 6 (1959), 290–297.

[8] GREGOR, D., AND LUMSDAINE, A. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)* (July 2005).

[9] HENDRICKSON, B., AND BERRY, J. Graph analysis with high-performance computing. *Computers in Science and Engineering 10*, 2 (2008), 14–19.

[10] INTEL CORPORATION. *Intel®Thread Building Blocks*, 1.6 ed., 2007.

[11] LESKOVEC, J., AND FALOUTSOS, C. Scalable modeling of real graphs using kronecker multiplication. In *In Proceedings of the 24th International Conference on Machine Learning* (2007).

[12] LESKOVEC, J., LANG, K. J., DASGUPTA, A., AND MAHONEY, M. W. Statistical properties of community structure in large social and information networks. In *WWW* (2008), pp. 695–704.

[13] MINAR, N., BURKHART, R., LANGTON, C., AND ASKENAZI, M. The Swarm simulation system: A toolkit for building multi-agent simulations. Working Paper 96-06-042, Santa Fe Institute, 1996.

[14] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford Digital Library Technologies Project, 1998.

[15] SHILOACH, Y., AND VISHKIN, U. An o(n log n) parallel connectivity algorithm. *J. Algorithms 3*, 7 (1982), 57–67.

[16] WHEELER, K., MURPHY, R., AND THAIN, D. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium* (April 2008), MTAAP '08, IEEE Computer Society Press, pp. 1–8.