Early Experiences with Large-Scale Cray XMT Systems

David Mizell and Kristyn Maschhoff

Cray Inc.

Abstract

Several 64-processor XMT systems have now been shipped to customers and there have been 128-processor, 256-processor and 512-processor systems tested in Cray's development lab. We describe some techniques we have used for tuning performance in hopes that applications continued to scale on these larger systems. We discuss how the programmer must work with the XMT compiler to extract maximum parallelism and performance, especially from multiply nested loops, and how the performance tools provide vital information about whether or how the compiler has parallelized loops and where performance bottlenecks may be occurring. We also show data that indicate that the maximum performance of a given application on a given size XMT system is limited by memory or network bandwidth, in a way that is somewhat independent of the number of processors used.

Introduction

A few 64-processor Cray XMT systems [8] are now in the hands of Department of Defense and Department of Energy users. As of this writing, there is a 512-processor system and a 128-processor system under software testing in Cray's lab, and a 256-processor system has also been tested there. The authors have, by now, experimented with several applications and application kernels on the 128-processor and 256-processor prototypes.

The XMT is a direct successor of the Cray/Tera MTA-2 [1]. The processor has the same multithreaded architecture and instruction set, and the C/C++ compiler, debugger,

performance tools, and operating system are the next generations of those on the MTA-2 [4]. There are several differences between the MTA-2 and the XMT, however:

- The XMT processor is faster, 500 MHz as opposed to the MTA-2's 220 MHz.
- Rather than having the MTA-2's custom network, the XMT is based on the Cray XT3 circuit board and 3-D torus network infrastructure, for economic reasons.
- Rather than having a custom memory system like the MTA-2, the XMT uses commodity DDR1 memories and memory controllers, also for economic reasons.

For the above three reasons, the balance between processor throughput and memory or network bandwidth is significantly different on the XMT from what it was on the MTA-2.

Another significant difference is that larger XMT systems have been built. The largest MTA-2 system ever built had 44 processors. As mentioned above, XMT systems as large as 512 processors have been prototyped, 64-processor systems are in the hands of customers, and larger ones will be shipping in the near future. A larger number of processors implies a scaling challenge to the performance-minded programmer. Inefficiencies in the code or in the parallelization approach, that may not have been significant at lower numbers of processors, can hamper the application's ability to scale well to larger numbers of processors. The purpose of this paper is to present several examples of code modifications we had to make in order to scale XMT programs to numbers of processors larger than 64, and what feedback from the compiler and performance tools led us to these modifications.

Characteristics of the XMT System

As in the MTA-2 processor architecture, the XMT processor provides storage for the contexts of 128 threads. The processor hardware that holds the context of one thread is referred to as a "stream" in the MTA/XMT vernacular. In addition, many more threads can be defined in software and stored in memory, with the XMT runtime swapping them into and out of the processor. The processor can, in effect, perform a thread context switch on every instruction cycle, choosing at each cycle the next instruction from one of the threads that is ready to issue. When executing an application with a high degree of parallelism, there are enough threads ready to issue at any given time that XMT processors rarely stall waiting for results to arrive from memory or the network. The effect of this is that XMT processors tend to keep issuing network and/or memory requests. In modern parallel computer systems, memory and network are more often the performance bottleneck than is processor throughput. A massively multithreaded system like the XMT can usually keep the bottlenecked resource saturated, which is the best one can expect to do on any architecture. This performance characteristic of

the XMT shows up particularly well on codes dominated by remote references. We have seen applications designed to perform analysis of extremely large graphs perform one or two orders of magnitude faster on the XMT than on contemporary distributed memory systems, because those systems have relatively high latency and overhead when fetching or sending data from remote memories, and their commodity microprocessors stall after a small number of outstanding memory requests. Most of the examples in this paper entail operations performed on huge graphs that would be difficult or impossible to partition across the nodes of a distributed memory system in such a way that most references were local.

Characteristics of the XMT software stack are equally important to the programmer:

- The XMT's C/C++ compiler [4] is one of the most powerful, sophisticated optimizing/parallelizing compilers extant. In addition to parallelizing loops in any of several different ways, the compiler restructures, interchanges, or collapses loops, as well as performing a wide variety of serial code optimizations. Furthermore, the XMT programmer is practically dependent upon the XMT compiler. There is no way provided to application software to bypass the compiler or the thread management runtime and control parallelism directly. There is no analog of MPI or OpenMP, in the sense that those notations give the programmer much more explicit control over processes and threads. What the programmer does have is a diverse set of pragmas to insert into the source code, for communicating to the compiler that a loop can be parallelized without semantic error, for example, or what parallelization approach should be used for a given loop.
- Because the compiler does so much more optimizing and parallelizing than the user can do by hand, Cray provides CANAL [9], a software tool that annotates the source code line by line and loop by loop with information regarding how the code was optimized and parallelized. The performance-tuning XMT programmer simply will not succeed without understanding and frequently using CANAL.
- TRACEVIEW is also a valuable performance-tuning tool. TRACEVIEW and CANAL are both provided within Cray's performance-tuning environment Apprentice 2. TRACEVIEW provides detailed events tracing, tied to designated points in the source code. It can also provide a graphical plot over time of overall processor utilization, helping the programmer identify places in the code with low degrees of parallelism – which may be caused by a load imbalance or by a serial section of code.
- DASHBOARD is a graphical tool that the programmer can watch as the program executes, showing plots in real time of processor and memory utilization.

On distributed memory systems using MPI, it is fairly common for a performance-tuning programmer's only software tool to be the timer function. That is, the programmer's performance tuning routine is to iterate between modifying the source code, recompiling, and measuring the new execution time. This approach is ineffective for the XMT. The appropriate performance-tuning paradigm for the XMT is to modify the source code or the pragmas, compile, examine the CANAL output to understand whether and how the compiler parallelized loops, either start over or make a performance-measurement run and look at the TRACEVIEW output; continue.

Example 1: Breadth-First Search Loop Parallelization

This is a simple example aimed at illustrating how critical the CANAL tool is in discovering how the compiler treated a loop. The kernel performed a breadth-first search on a large graph, starting from the vertex specified in the call. The defining data of the graph was held in a top-level C "struct", specified as follows:

```
typedef struct {
  int N;
  int *Marked;
  Neighbor *Neighbors;
  int *numNeighbors;
} graph;
```

The integer array Marked represents whether or not a given vertex has yet been visited in the search. It is initialized to -1, then given the ID of the vertex's immediate predecessor once a thread has visited it. The first version of the initialization loop looked as follows:

```
int BFS(int root, graph *A)
{ int i, …
   …
  //initialize Marked array
  for( i=0; i<A->N; i++) A->Marked[i] = -1;
   …
```

Performance of the BFS was dismal. Examination of the CANAL output determined that this simple initialization loop was the culprit.

```
                |
                |   //initialize Marked array
```

```
X   |   for( i=0; i<A->N; i++) A->Marked[i] = -1;
    |
```

The X indicates that the loop was not parallelized. The compiler could not be sure whether or not there was some dependency between loop iterations hidden by the A pointer.

The cure was to manually dereference the A pointer:

```
int BFS(int root, graph *A)
{
  int *Marked = A->Marked;
  int N = A->N;
  for(int i=0;i<N;i++) Marked[i] = -1;
  …
```

Again, the CANAL output shows that the problem was solved by this change:

```
    |   int *Marked = A->Marked;
    |   int N = A->N;
P   |   for(int i=0;i<N;i++) Marked[i] = -1;
```

The P indicates that the compiler parallelized this loop. The improvement in performance was a factor of 50, for 16 processors and a random graph with 10M vertices.

Example 2: Breadth-First Search Queue Management

In our original implementation of breadth-first search, written by our colleague John Feo, the conceptually recursive loop in which a thread starts at some vertex, visits each of its neighbors, and while visiting a neighboring vertex creates the work for visiting each of that vertex's neighbors, was implemented as a doubly nested loop. The outer loop picks the ID of a vertex from a queue and visits it. If it hasn't already been visited, the thread goes into the inner loop, which places that vertex's neighbors into the queue. Here's the relevant code snippet of the inner loop.

```
/* Mark each unmarked neighbor node */
  for (i = firstNode; i < lastNode; i++) {
      int neighbor = Neighbors[i];
      if (Marked[neighbor] < 0) {
         int mark = readfe(Marked + neighbor);
```

```
            if (mark < 0) {
               mark = node;
               int k = int_fetch_add(N,1);
               Q[k] = neighbor;
            }
            writeef(Marked + neighbor, mark);
         }
      }
```

Note that this code snippet illustrates three of the XMT's synchronization primitives. XMT memory words have 64 bits for data and an additional bit that holds the word's "full/empty" state. The processor issues a readfe() ("read when full, set empty") as if it were a normal memory read, but the read only succeeds if the word's full/empty bit is set to full. Otherwise, the read has to wait until the condition is true. The hardware is designed to retry the read several times, and then trap to the runtime, which will probably swap the thread out of its stream, assuming that it will have to wait for a while longer. Once the read succeeds, the contents of the word are fetched to the processor and the word's full/empty bit is set to empty. Symmetrically, the writeef() ("write when empty, set full") only succeeds if the word is marked empty. Once this condition is met, the data contents are stored into the word and its full/empty bit is marked full. In this example, the thread is essentially locking the Marked array entry it is currently checking and possibly modifying, to prevent any other thread from touching this entry at the same time. The writeef() at the end unlocks the entry so that it is once again accessible to other threads.

Between those synchronized read and write calls, one can also see an int_fetch_add() call. This invokes an atomic fetch-and-add on the referenced memory word. In the code snippet above, the integer pointed to by the pointer variable N will have a 1 atomically added to its value.

This code scaled well, but only up to 32 processors. The problem was that the queue index variable, represented by the variable N in the snippet above, became a hot spot. All the threads doing neighbor visits were trying to increment the same variable in order to place their next vertex into the queue. The atomic int_fetch_add() operation helped the index increment work correctly, but no more quickly.

A way to confirm that a hot spot like this one is occurring is to instrument the code with the XMT intrinsic functions that access the system's hardware counters. The code snippet below illustrates how reads of the hardware counters are wrapped around the call to the BFS function just as the timer calls are:

```
issues = mta_get_task_counter(RT_ISSUES);
memrefs= mta_get_task_counter(RT_MEMREFS);
concur= mta_get_task_counter(RT_CONCURRENCY);
streams= mta_get_task_counter(RT_STREAMS);
traps = mta_get_task_counter(RT_TRAP);
retries = mta_get_task_counter(RT_MEM_RETRY);

time1 = timer();

int maxDist = BFS(0, A);

time2 = timer();

issues = mta_get_task_counter(RT_ISSUES) - issues;
memrefs= mta_get_task_counter(RT_MEMREFS) - memrefs;
concur= mta_get_task_counter(RT_CONCURRENCY) - concur;
streams= mta_get_task_counter(RT_STREAMS) - streams;
traps = mta_get_task_counter(RT_TRAP) - traps;
retries = mta_get_task_counter(RT_MEM_RETRY) - retries;
```

Because an unsuccessful attempt to read or update a shared variable will retry many times and then trap to the runtime, recording the number of retries and traps can indicate a hot-spotting problem. The table below shows the number of traps and how they increased for higher numbers of processors running this version of BFS.

Processors	traps
8	143
16	292
32	3377696
64	12716939

Feo tried to relieve the hot-spotting problem by splitting the inner loop into two loops. The first loop would count how many unvisited neighbors the current vertex had. Only then would it increment the shared queue variable with the total number of unmarked neighbors this thread was about to add to the queue. That would effectively "reserve" enough entries in the queue for this thread to use without contention from any other thread. In the second loop, the thread would walk back through the neighbors, pick out the unmarked ones, and add them to the queue in the reserved slots.

This approach was twice as slow as the original for small numbers of processors, namely because it walked through the set of neighbors of a vertex twice in the innermost loop. It did relieve the hot-spotting problem somewhat, though, and scaled to around 48 processors. Collection of hardware counter statistics, however, indicated that hot-spotting on the queue variables was still a problem.

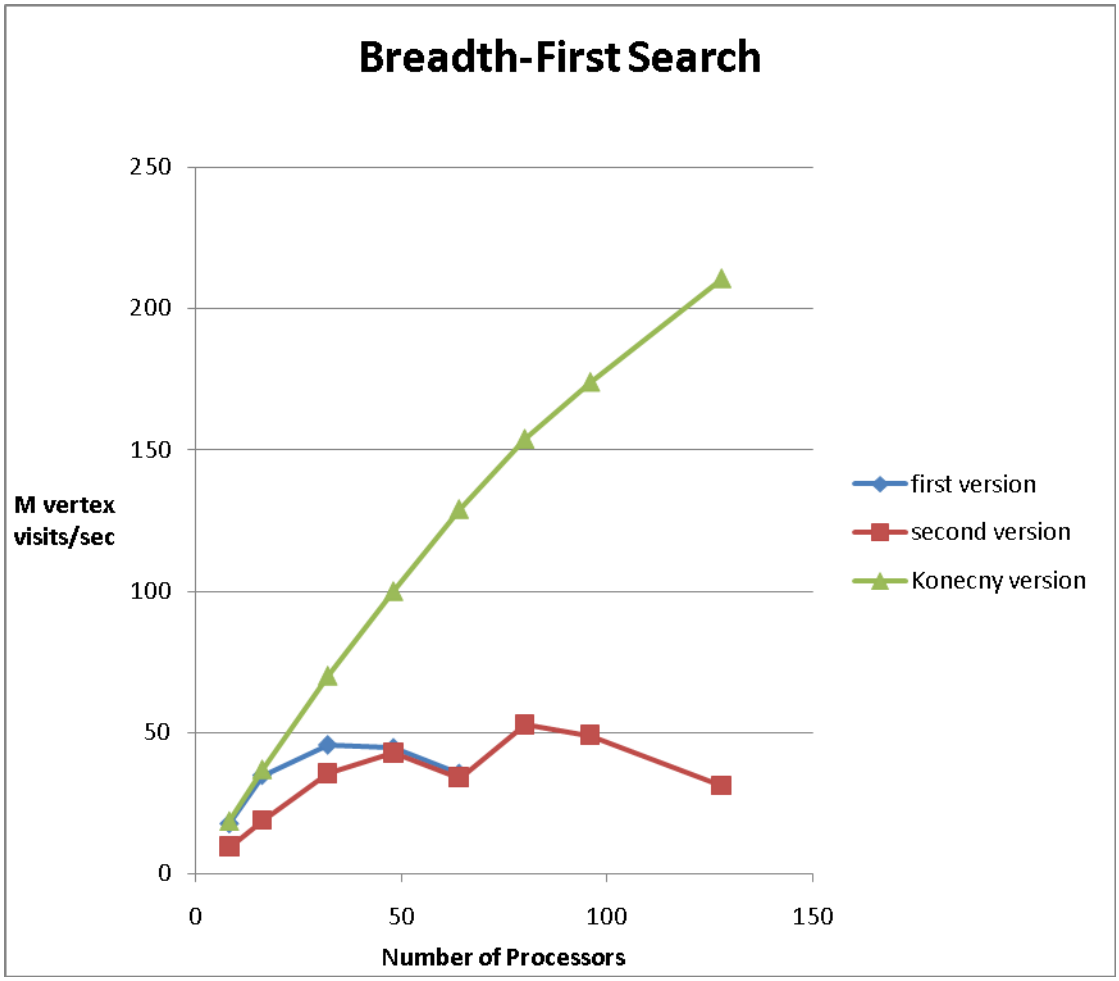
Our colleague Petr Konecny chose an approach that drastically reduced the hot-spotting and allowed BFS to scale to much larger numbers of processors. His approach was to explicitly take into account the number of threads that were going to be searching the graph, and allocate them each a section of the queue that they could use privately. A relevant snippet of the code is below. Note the call to the XMT runtime function MTA_NUM_STREAMS(), which returns the overall number of streams this job has available. The idea is that each active thread gets its own chunk of the queue, of size INBLOCK, that it can insert vertex IDs into without having to synchronize after it receives its allocation.

```
#pragma mta assert parallel
#pragma mta use 100 streams
  for(int th=0; th<MTA_NUM_STREAMS(); th++) {
    unsigned outhead = 0, outtail = 0;
    for(;;) {
      // grab INBLOCK nodes (& stubs) from the input
      unsigned inhead = int_fetch_add(&newhead, INBLOCK);
      // avoid overrun
      unsigned intail = min(inhead + INBLOCK, oldtail);
      if (inhead>=intail) break;   //stop if no work left

      for(int i=inhead; i<intail; i++) {
        int u = Q[i%qcap];
```

```
        if (u>=0) {
          int begin = numNeighbors[u];
          int end = numNeighbors[u+1];
          for(int j=begin;j<end;j++) {
            int v = neighbors[j];
            if (Marked[v]<0) {
              int mark = readfe(&Marked[v]);
              int newmark = mark<0 ? u : mark;
              Marked[v] = newmark;
              if (mark<0) {
          // we have set v's parent to u
          // reserve space for enqueueing OUTBLOCK nodes
                if (outhead>=outtail) {
                  outhead = int_fetch_add(&tail, OUTBLOCK);
                  outtail = outhead+OUTBLOCK;
                  // check for overflow
                  assert(outtail-oldhead<qcap);
                }
                Q[(outhead++)%qcap] = v;
      }  }  } }
```

Using this approach also enabled him to reduce the inner loop back to one pass through the vertex's neighbors. This is an example of how reducing the number of trips to memory in the inner loops can significantly improve performance on the XMT. A comparison of the three BFS versions is shown in the graph below. These runs were all done on a randomly generated graph with vertices having a uniformly distributed out-degree. The vertical axis represents millions of vertices visited per second for each run. One can see that John Feo's first version showed good performance on a small number of processors, but didn't scale. His second version was slower on a small number of processors, but scaled somewhat better. Petr Konecny's version shows much better scaling, at least for this type of random graph.



One might argue that this version is less elegant than the original, but sometimes such code redesign is necessary in order to scale to larger numbers of processors.

Example 3: Betweenness Centrality Loop Parallelization

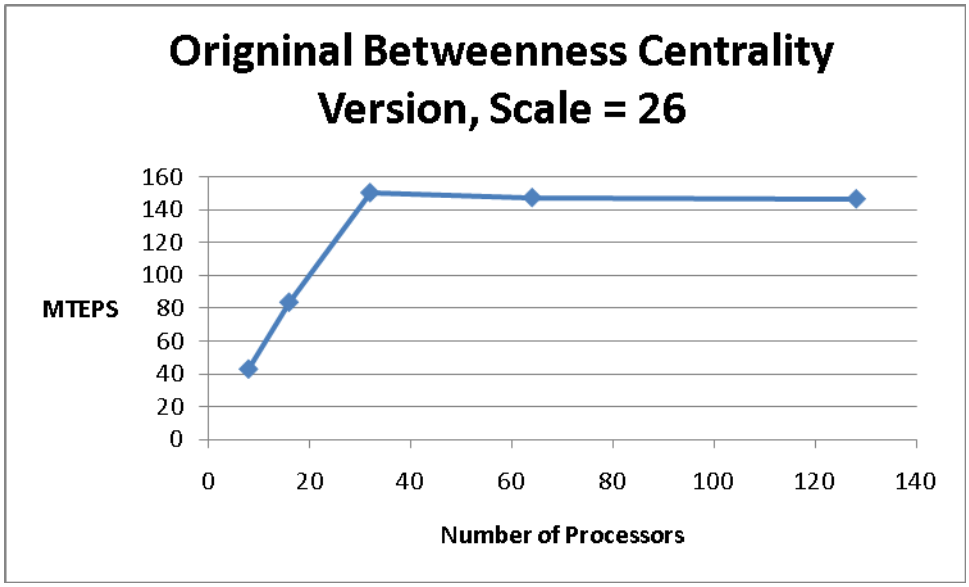
Betweenness centrality [7] is a fairly complex graph analytic algorithm that originated in the social network analysis community, and is now seeing use across a large set of directed graph-oriented applications, such as Internet social networking, biology and electrical power grids. Informally, the betweenness centrality metric of a vertex is the ratio of shortest paths that pass through that vertex versus all shortest paths.

Our XMT implementation is based on that of Bader and Madduri [3], which entails repeated breadth-first searches with some numerical "scorekeeping" added, with each search starting at a different vertex of the graph. Our first implementation had a sequential outer loop, each iteration of which launched a breadth-first search, which was implemented with doubly-nested

loops similar to those described in previous examples. Code snippets illustrating the loop structure are as follows:

```
/* Use |Vs| nodes to compute centrality values */
  for (s = 0; (s < NV) && (Vs > 0); s ++) {
  …
#pragma mta assert no dependence
      for (j = QHead[nQ - 1]; j < QHead[nQ]; j++) {
          …
          int myStart = start[v];
          int myEnd   = start[v + 1];
#pragma mta assert no dependence
          for (k = myStart; k < myEnd; k++) {
              …
```

While each of the iterations of the betweenness centrality outer loop is independent of the others except for the summation of centrality scores at the end, our implementation uses several arrays of a size equal to the number of vertices in the graph. Because we were working with graphs with vertex counts in the billions on the XMT, we felt that there probably wouldn't be room to replicate all those arrays and run several breadth-first searches in parallel. A speedup curve for this version of betweenness centrality is shown below. All of our betweenness centrality experiments were done in the context of SSCA2, the graph algorithms benchmark defined by David Bader and Kamesh Madduri [2]. Using the nomenclature of SSCA2, SCALE was set to 26 in these experiments, implying that the input graph had 2^{26} vertices and 2^{29} edges. The "K4approx" variable was set to 8, which means that 256 breadth-first searches were performed by the outermost loop. The vertical axis of the graph below is in millions of "TEPS" (traversed edges per second), Bader and Madduri's proposed performance estimator for betweenness centrality. TEPS serves as a sort of analog of a speedup curve, in that it consists of a value for the amount of work done divided by the execution time.



One can see that the original version of betweenness only scaled well up to 32 processors.

With the availability of larger XMT systems, considerations of memory limitations changed. We calculated that it was feasible to run a few iterations of the outer loop together in parallel on this size problem. The trick was to limit the number of iterations. Our first attempt at parallelizing the outermost loop looked as follows:

```
#define BFS_THREADS  16
…
#pragma mta assert parallel
  for(num_threads=0; num_threads < BFS_THREADS; num_threads
++){
   for(;;) {
          start_vertex = int_fetch_add(&Vs_ptr,1);
          if (start_vertex > Vs -1) break;
          …

#pragma mta assert no dependence
          for (j = QHead[nQ - 1]; j < QHead[nQ]; j++) {
             …
```

The purpose of the outer loop was to define a parallel loop of 16 iterations, the number of breadth-first searches we calculated that we could afford to have running in parallel. The loop

after that was intended to be the work each of the 16 threads did: grab the next vertex from the list and do a breadth-first search starting there.

Performance was disastrous. The key clue was in the CANAL listing, where it annotated the innermost loop:

```
7 pXX     |       for (j = QHead[nQ - 1]; j < QHead[nQ]; j++) {
```

The lower-case p indicates that the outermost loop was parallelized by the compiler, only because the user asserted it was parallel (see the pragma in the code snippet). The two Xs indicate that the next two levels of loop were not parallelized. This is a common choice made by the compiler, assuming that there is sufficient parallelism in the outer loop and it would be more efficient, because of parallelism control overheads, to leave inner loops sequential. Similarly, an attempt to apply the "loop future" pragma, directing the compiler to use this particularly dynamic parallel loop scheduling approach on the outermost loop, as seen below,

```
#pragma mta assert parallel
#pragma mta loop future
for(num_threads=0; num_threads < BFS_THREADS; num_threads
++){
…
```

had the side effect of restricting lower-level parallelism to within a processor. Since we had deliberately kept the number of outermost loop iterations small, this resulted in the computation being confined to BFS_THREADS number of processors. CANAL output again made us aware of this compiler decision, in the notes section that supplies more detail about each loop the compiler processed:

```
Parallel region   7 in cenTrality in loop 6
       Single processor implementation
```

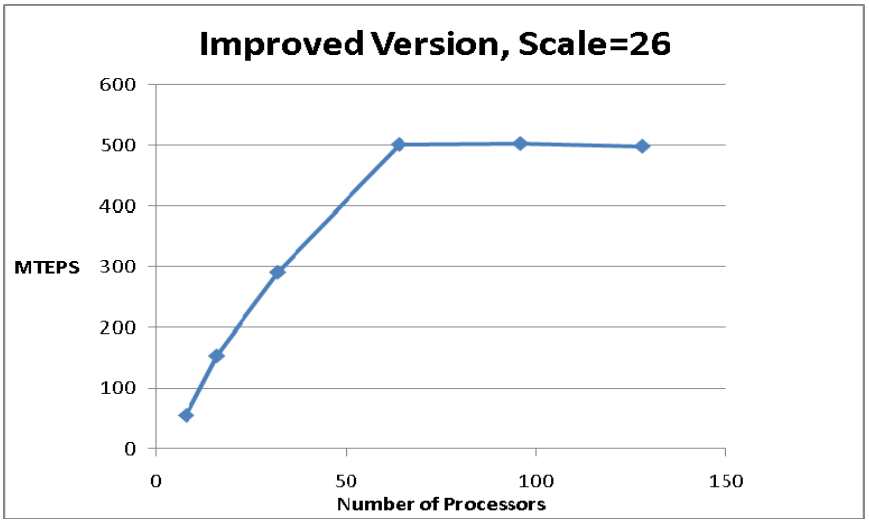
The notes section at the end of the CANAL listing tells us (somewhat indirectly; the "parallel regions" are sometimes hard to identify in the source code) that this loop will not be parallelized beyond a single processor. This again reflects a compiler policy that assumes most of the parallelism will be in the outermost loop, and a lower-overhead parallelism approach would be more appropriate for lower levels.

A more successful approach employed XMT's explicit "future" variables. Future variables support a form of lazy evaluation. A thread can be assigned to compute the variable's value,

and the spawning thread can find out later whether the computation has completed by trying to fetch the future variable. We assigned a future variable to every iteration of the outermost loop we felt we could afford to run in parallel, and launched them all, as shown below:

```
#define BFS_THREADS 16
  future int thread_id[BFS_THREADS];
  …
// Spawn off futures to run independent BFS processes
  for (num_threads=0; num_threads < BFS_THREADS;
num_threads++) {
     future thread_id[num_threads](num_threads, G, BC, Vs,
&Vs_ptr, permV, bfs_counter) {
         Process_cenTrality(G, BC, Vs, &Vs_ptr, …
         …
  for (num_threads=0; num_threads < BFS_THREADS;
num_threads++) {
     touch (&thread_id[num_threads]);
  }
```

The first loop above launches all of the futures, but launching a future doesn't guarantee that it is executed right away. Here, the programmer has forced that in the second loop, by calling the "touch" function on each future variable. That forces the immediate computation of the future value. We put the inner loops inside the Process_cenTrality() function, because the compiler only expects a future variable to be computed via a function call, which is executed by the spawned thread. Below is the speedup curve we achieved with this implementation.



This future-variable version of the code scaled to twice as many processors as the original, but flattened out after that. However, it achieved around twice the performance in terms of the MTEPS metric than the original code.

Example 3 serves to illustrate how experience with the XMT system, its compiler and its tools are integral to achieving scalability. It took us multiple tries to find a way to get the compiler to do what we wanted, namely provide a small amount of parallelism for the outermost loops and a large amount for the inner loops.

Example 4: Modifying the "stream_limit" Parameter

While the XMT at least holds its own in absolute performance compared to the MTA-2 and utterly destroys it on cost-performance, it is fair to say that the XMT is not as well-balanced an architecture as the MTA-2 was. Processor speed is higher, but memory and network bandwidth were not increased correspondingly. Thus, as applications are scaled upwards, memory or network bandwidth tends to saturate, rather than processor throughput. We have never seen processor utilization much above 50% in the applications we have tested so far. Most well-tuned applications peak out at around 30% processor utilization. As seen in the BFS example, minimizing trips to memory in the innermost loops is a good approach to improving scalability, because it helps avoid saturating the memory bandwidth. The notes section at the end of the CANAL listing is very helpful in this regard. It provides information about the number of instructions and the number of memory accesses in the inner loops.

We have seen the memory bandwidth saturate or the network congest when we try to scale a parallel computation upwards. Too many threads trying to perform remote references can saturate the memory bandwidth, eliminate any advantage available from the memory caches, queue up in the network interfaces or clog the 3-D torus network. We have relieved this in some cases by limiting the number of streams that are active on each processor. The XMT software system provides a configuration parameter, MTA_PARAMS, with which a user can control the number of active streams per processor. Using the Linux bash shell, the command looks as follows:

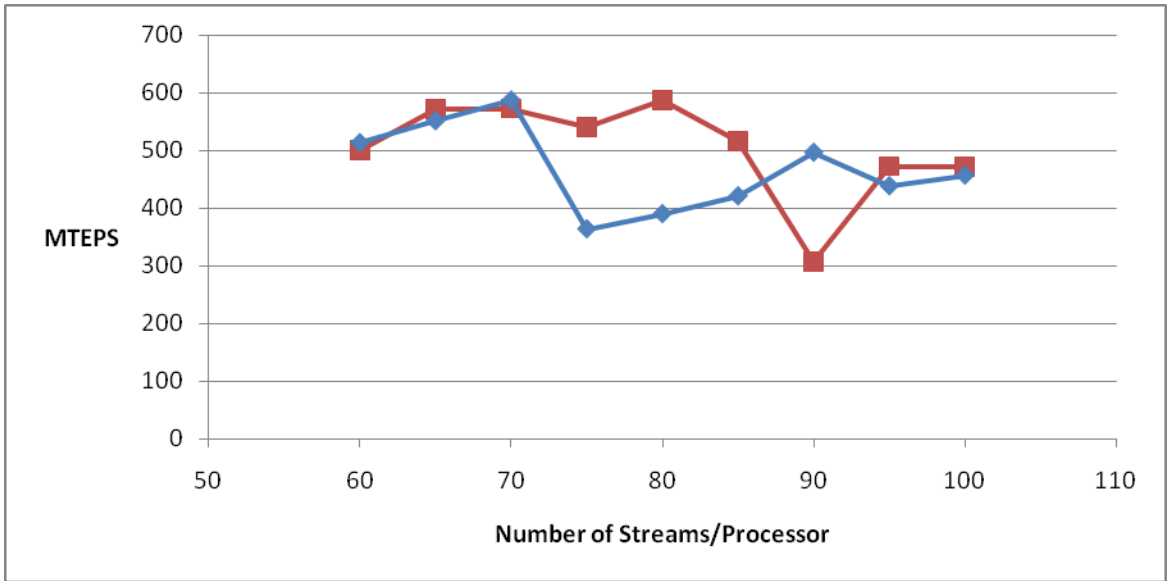
```
export MTA_PARAMS="stream_limit 85"
```

Of the 128 hardware streams available on each processor, the runtime reserves 24 or more for its own use, so the maximum number of streams available for application use is effectively around 100. In the bash command above, the user has restricted the number of available streams to 85 per processor. There is also an runtime function that can set the stream limit

from within a program. The call to accomplish the same objective as the bash command above looks like the following:

```
mta_set_stream_limit(85);
```

Once we had improved the scalability of betweenness centrality with the (slightly) parallel outer loop described in Example 3, we tested the code at various stream limit settings to see the effect on performance. In every case, 128 processors were used, on a power law graph generated using the R-MAT algorithm [5]. Our implementation set the number of parallel iterations of the outer loop to 16, so each thread executing the outer loop did roughly 16 of the 256 breadth-first searches. The graph below shows the results of two scaling runs with the above SSCA2 parameters. The independent variable was the number of streams made active in each processor. The result variable is in terms of millions of "TEPS", as before.



The two sets of experiments were consistent up to a stream limit of 70 per processor. They also both show an expected upward scaling, as the amount of parallelism is increased. Above 70, however, our performance runs were inconsistent between runs. We saw this in other experiments, as well. Apparently, when there are enough streams active to begin to cause queueing and congestion in the network, overall throughput becomes unpredictable.

These results prompted a further round of experiments, in which we varied both the number of processors running the improved betweenness centrality code, and varied the stream limit value – but tried to keep the product of the two constant. In other words, we were trying to keep the total number of active streams the same, for varying numbers of processors. Some results from these experiments are shown in the table below. The SSCA2 parameters of

SCALE=27 and K4approx=8 were used in these runs. The processor/stream combinations were chosen to be close to 8960 in the first set of runs, because the 128/70 case had achieved the highest TEPS performance we have yet seen on the XMT. The 7680 product was chosen because it has a lot of divisors. An R-MAT-generated graph with 2^{27} vertices and 2^{30} edges was used in each case.

processors	streams	product	time	MTEPS
128	70	8960	409.1	587.9
90	100	9000	447.8	537.1
95	94	8930	450	534.5
112	80	8960	425.4	565.4
128	60	7680	481	500
80	96	7680	435.7	552.1
96	80	7680	470.1	511.6
120	64	7680	470.6	511
110	70	7700	415.7	578.6
102	75	7650	448.2	536.6
113	68	7684	449.2	535.4
110	70	7700	417.8	575.7

One can see that there is some degree of consistency between runs with the same processor-stream product. On a 128-processor system, running betweenness centrality, we saw diminishing returns above 9000 streams. We suspect that this signifies that the memory bandwidth has become saturated. This is probably not the whole story, however. The 80/96 and 110/70 data points seem to indicate that more processors and fewer streams are advantageous. This may have something to do with network topology; it may be better to spread the same amount of computation across more processors, and thereby employ a larger number of network links.

This issue clearly needs further, better-instrumented experimentation before we understand it completely.

A confounding issue is that systems in everyday use may not show the same performance characteristics. We have performed all of these experiments on a dedicated machine. The XMT hashes memory addresses across the entire memory space no matter how many processors a user is running on. Thus our experimental runs using a small number of processors had available to them all the memory bandwidth of the whole system (other than negligible amounts consumed by the OS and runtime). The same computation on the same number of

processors may run more slowly if it has to compete for memory and network bandwidth with several other user jobs. The fact remains that users need to be aware that too many active streams can choke out either the memory or the network bandwidth.

Example 5: Hashing Words from Text Documents

During 2008, we interacted with a potential XMT customer who was interested in processing large numbers of text documents, extracting key words from each, and then using graph data structures to analyze a variety of topical similarities between the documents. We ran some performance experiments with a set of 2000 text documents that the customer provided us. Together, they contained about 70,000 words. Our code first read all the documents into memory, inserting the words from each document into an array of all the words associated with that document. Then, using a hash table, we inverted the relationship, creating an array of words, with each word appearing uniquely in the array and linked to a list of all the documents it appeared in.

This application only scaled up to about 16 processors. Attempts to optimize the hash table code did not make much difference. The insight came when we added code that gathered statistics on the data. In particular, we generated a histogram of the number of words that appeared in a given number of documents. The histogram's highest value was at 1; almost 24,000 words appeared in only one document each. It decreased more or less monotonically and reached zero at about 500, except that there were tiny blips on the histogram every 50 or so values following that. One word appeared in 1310 documents, another word appeared in 1508 documents, and another appeared in 1844 documents. There was a load imbalance, and it was inherent in the input data. Realizing that a few words would have very long document lists attached to their entry in the hash table, we changed the code so that new document IDs were inserted into the list in sorted order. This reduced the document list search by half on average, and enabled one more scaling factor of two.

This example illustrates that although using the XMT's performance tools is essential, the tools don't provide every possible insight into tuning performance. We found that this practice of inserting into the application code a little additional code that gathered statistics about the data was frequently useful in performance tuning and even occasionally in debugging.

Conclusions

We are excited about the Cray XMT. It has demonstrated markedly superior performance on graph algorithms for graphs not conducive to partitioning across distributed memory architectures. Our improved betweenness centrality implementation recently showed 350x

faster running time on 64 processors than an MPI implementation on 64 processors of an Opteron cluster, using the same set of SSCA2 defining parameters for the graph [6]. Furthermore, the XMT is one of the easiest parallel systems to program, in the sense that code written in the most straightforward way usually runs reasonably well on a moderate number of processors. What we have tried to illustrate in this paper is that success in squeezing out maximum efficiency for the sake of scaling up to larger numbers of processors is highly dependent on the programmer's understanding of the XMT architecture and the XMT compiler – and on quotidian use of CANAL and the other performance tools.

Acknowledgments

This work was supported by the Department of Defense via the Center for Adaptive Supercomputing Software – Multithreaded Architectures project at Pacific Northwest National Laboratories.

The authors wish to thank Sung-Eun Choi, Mike Ringenburg, Christian Hansen, and the other members of the Cray XMT software development team, and their manager Gail Alverson, for their support and the benefit of their expertise. Andy Kopser of the Cray XMT hardware team also provided valuable proofreading.

References

[1] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 188–197, New York, NY, USA, 1992. ACM.

[2] David Bader and Kamesh Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *Proc. 12th International Conference on High Performance Computing (HiPC 2005)*, 2005.

[3] David Bader and Kamesh Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *Proc. The 35th International Conference on Parallel Processing (ICPP)*, August 2006.

[4] David Callahan and Burton Smith. A future-based parallel language for a general-purpose highly-parallel computer. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 95–113, 1990.

[5] Deepayan Chakrabarti and Christos Faloutsos. Graph patterns and the R-MAT generator, in Cook and Holder, ed., *Mining Graph Data,* chapter 4, pages 65–96. Wiley-Interscience, 2007.

[6] Nick Edmonds, University of Indiana. private communication. January 2009.

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

[8] Simon Kahan, Petr Konecny, John Feo, David Harper. Eldorado. In *Proceedings of the 2nd Conference on Computing Frontiers*, page 46, November 2003.

[9] Allan Snavely, Larry Carter, John Feo. Performance and programming experience on the Tera MTA. In *SIAM Conference on Parallel Processing*, 1999.