The Graph 500 Benchmark on a Medium-Size Distributed-Memory Cluster with High-Performance Interconnect

Jordan B. Angel[a], Amy M. Flores[b], Justine S. Heritage[c], Nathan C. Wardrip[d], Andrew M. Raim[e], Matthias K. Gobbert[e,*], Richard C. Murphy[f], David J. Mountain[g]

[a]*Department of Mathematics and Statistics, East Tennessee State University*
[b]*Department of Mathematics and Statistics, Grinnell College*
[c]*Department of Mathematics and Computer Science, Dickinson College*
[d]*Department of Mechanical Engineering, Saint Cloud State University*
[e]*Department of Mathematics and Statistics, University of Maryland, Baltimore County, 1000 Hilltop Circle, Baltimore, MD, 21250, U.S.A.*
[f]*Micron Technology, Inc.*
[g]*Advanced Computing Systems Research Program*

Abstract

While traditional performance benchmarks for high-performance computers measure the speed of arithmetic operations, memory access time is a more useful performance gauge for many large problems today. The Graph 500 benchmark has been developed to measure a computer's performance in memory retrieval. The Graph 500 implementation considers large, randomly generated graphs, which may be spread across many nodes on a distributed-memory cluster. The benchmark conducts breadth-first searches on these graphs, and measures performance in billions of traversed edges per second (GTEPS). We present our experience implementing and running the Graph 500 benchmark on the medium-size distributed-memory cluster tara in the UMBC High Performance Computing Facility (www.umbc.edu/hpcf). The cluster tara has 82 compute nodes, each with two quad-core Intel Nehalem X5550 CPUs and 24 GB of memory, connected by a high-performance quad-data rate InfiniBand interconnect. Results are explained in detail in terms of the machine architecture, which demonstrates that the Graph 500 benchmark indeed provides a measure of memory access as the chief bottleneck for many applications. Our best run to date was of scale 31 using 64 nodes and achieved a GTEPS rate that placed tara at rank 98 on the November 2012 Graph 500 list.

Keywords: Graph 500 benchmark, Parallel computing, Distributed computing, High-performance interconnect, Breadth-first search.

1. Introduction

An increasing number of modern computational problems involves the analysis of large data. These data-centric problems tax a computer's memory and stress its capability to read and write to memory quickly. The Graph 500 benchmark [1] aims to measure these capabilities, and ranks top performing machines on the Graph 500 website (www.graph500.org). This is analogous to the well-known LINPACK benchmark used to rank computers for the TOP500 list (www.top500.org) that measures a computer's floating operations per second (FLOPS) as it solves a large system of linear equations. This paper studies the performance of the Graph 500 benchmark on a medium-size distributed-memory cluster with high-performance interconnect that achieved the rank of 98 in the November 2012 Graph 500 list, as shown in Figure 1.1. The cluster is a general-purpose cluster and has 82 compute nodes, each with two quad-core Intel Nehalem X5550 CPUs and 24 GB of memory, connected by a high-performance quad-data rate InfiniBand interconnect.

A high-performance computer's ability to access memory is becoming increasingly crucial as the number and magnitude of large data problems continues to grow. An illustrative example of this type of problem is a social

*Corresponding author. Email address: gobbert@umbc.edu. Phone: (410) 455–2404. Fax: (410) 455–1066.

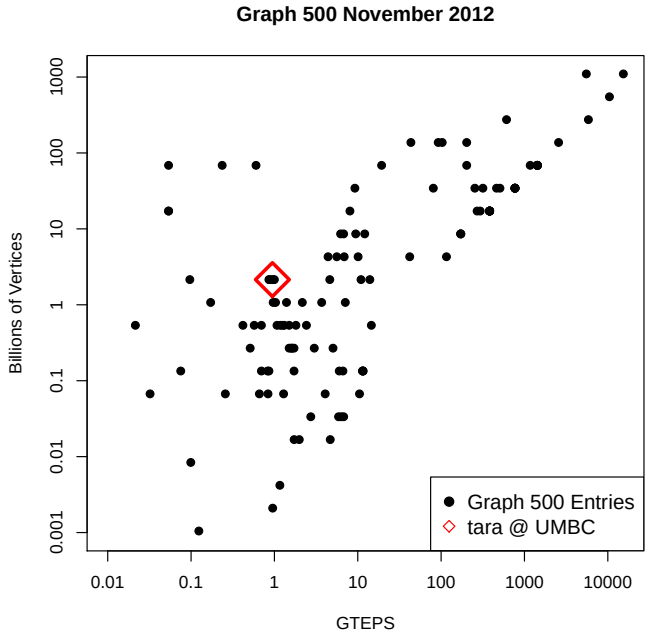

Figure 1.1: Placement of computers on the November 2012 Graph 500 list by GTEPS and billions of vertices.

network represented by a graph. As an example to understand the scale of the graph involved, consider that the social networking site Facebook reports "one billion monthly active users as of October 2012"[1]. If each user is represented as a vertex and each friendship as an edge, relationships between all users can be modeled as a massive graph. Facebook could benefit from efficient traversal of graphs as a means of analyzing data about its users and their interactions. As databases like Facebook's continue to expand, identifying machines that are capable of managing such extensive data is becoming increasingly important. The ranking based on the Graph 500 benchmark attempts to draw the focus of the HPC community to these problems [1].

Benchmarks are intended to stress a machine and identify the limits of its capabilities along specific dimensions of performance. Having a reliable tool to measure hardware performance is valuable to both manufacturers and consumers in understanding a machine's ability to address applications of interest. The well-known LINPACK benchmark used in the TOP500 list measures a computer's floating operations per second (FLOPS) as it solves a large system of linear equations. Therefore, TOP500 emphasizes performance of numerical calculations in its rankings. Graph 500 complements the LINPACK benchmark by recognizing data-intensive applications, focusing instead on performance of memory access. We will demonstrate here that the Graph 500 benchmark indeed provides a measure of memory access as the chief bottleneck for many applications. We share our experience implementing and running the benchmark, with many details on implementation choices and a reflection of the causes for certain aspects of observed behavior in terms of the hardware of the medium-size cluster tara.

Figure 1.2 provides a visual summary of the simulations on our medium-size distributed-memory cluster tara, which is a visualization of the data in Table 4.4 in Section 4. The missing bars in the front of the plot indicate that small numbers of nodes could not pool enough memory to accommodate larger graphs; for instance, 1 node's memory was only able to accommodate graphs up to scale 25, meaning with $2^{25} \approx 34$ million vertices — much smaller than the Facebook graph. As more nodes pool their memory, 64 nodes eventually accommodate a graph up to scale 31, meaning with $2^{31} \approx 2.1$ billion vertices — twice the scale of the Facebook graph. The ranking in the Graph 500 list is determined by the speed of traversing the edges of the graph in a breadth-first search, expressed in units of billions of

[1] Key Facts, http://newsroom.fb.com/content/default.aspx?NewsAreaId=22, accessed November 23, 2012.

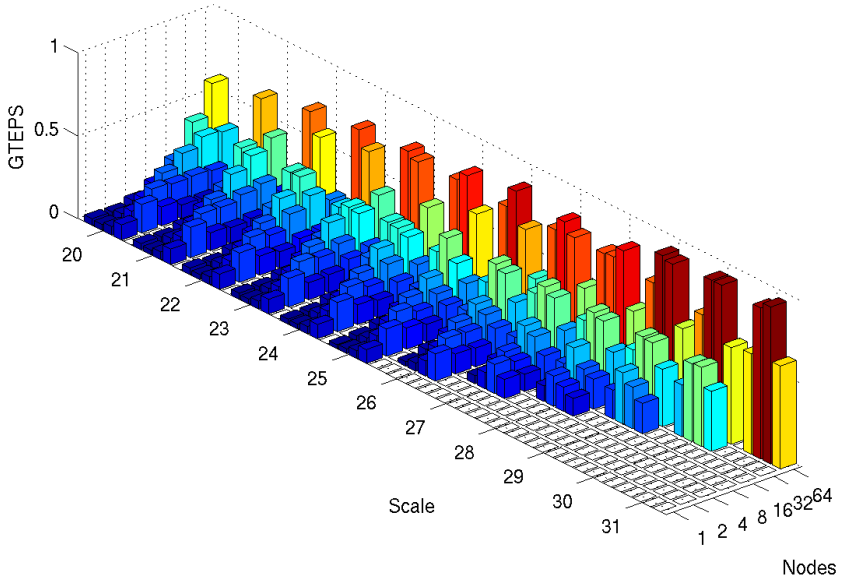

Figure 1.2: Three-dimensional bar plot of GTEPS of the Graph 500 benchmark runs on tara for scales 20, . . . , 31 with 1, 2, 4, 8, 16, 32, 64 nodes, within each of these using 1, 2, 4, 8 processes per node. Best result is 0.946 GTEPS for scale 31 with 64 nodes using 4 processes per node.

traversed edges per second (GTEPS). Generally, it is beneficial for one's ranking to solve the largest possible problem, which Figure 1.2 makes immediately clear with the highest bars towards the right of the plot. But Figure 1.1 shows that there are higher-ranked computers in the list (i.e., with higher GTEPS) that in fact solve smaller problems than we did. Many of these computers are 1-node machines with extremely large memory that also have a tighter connection to their memory inside their single node, which enables them to have extremely fast memory access and thus higher speed measured by the GTEPS. However, it is apparent that one benefit of a distributed-memory cluster is its ability to pool memory, eventually with more memory than can be available in any single node machine. The summary plot in Figure 1.2 actually shows four bars for each pair of scale and nodes; these correspond to the four choices of running 1, 2, 4, or 8 parallel processes per node on the available 8 computational cores of the two quad-core CPUs, with 1 core used being in the top-left and 8 cores in the bottom-right of each group of four. Notice that only the number of nodes used (not the number of processes per node) determines the scale of the graph possible, since the memory on a cluster of this type is shared by the CPU cores. On 1 node and other small number of nodes, there is little 'penalty' from the communications among nodes through the MPI commands in the code, and hence we see that it is beneficial to use as many processes as possible per node. When now the number of nodes is increased, while the graph remains quite small in scale, the situation reverses and it is in fact most beneficial to use only 1 core per node. But for larger numbers of nodes and when more of each node's memory is used to accommodate graphs with large scales, we see the interesting phenomenon of an intermediate optimum: It is then best to run either 2 or 4 processes per node, as exhibited by the middle two of the four bars for each pair of scale and nodes being highest. It turns out that the best result of 0.946 GTEPS for scale 31 with 64 nodes used 4 processes per node. This behavior is explained by the compromise between efficient access to memory on each node for the particular architecture of our nodes and the performance available in the high-performance InfiniBand interconnect that connects the nodes. The remainder of the paper is dedicated to analyzing and explaining these observations in more detail and connecting them to the particular hardware of our cluster, which is a general-purpose cluster with a typical compromise between local memory and interconnect quality.

In Section 2 we outline the technical details of the cluster tara. Section 3 provides a description of the Graph 500 specifications as well as details code modifications of the reference code of the Graph 500 benchmark. Section 4 presents the results of the benchmark on tara and discusses in detail the implications of the hardware of our cluster on the benchmark performance. Section 5 concludes the paper.

2. Computational Environment

The distributed-memory cluster tara in the UMBC High Performance Computing Facility (www.umbc.edu/hpcf) has 82 IBM (System x3650 M2) compute nodes, each with two quad-core Intel Nehalem X5550 CPUs (2.66 GHz, 8 MB cache) and 24 GB of memory.

Nodal architecture: The nodal architecture is outlined in Figure 2.1. The key advance of the Intel Nehalem processors is their direct connection to portions of the node's memory through three memory channels from each CPU. This implies that both CPUs can access memory simultaneously, provided each only requires data in memory connected through its channels. In the case that one CPU needs to access data in the memory connected to the other CPU, it needs to go through the QPI to the second CPU, before accessing that node's memory. This situation would arise in particular if a serial job on a node (that is, a job that only uses one computational core in one CPU) uses more than 50% of the node's memory of 24 GB, since a portion of this memory is necessarily connected to the second CPU.

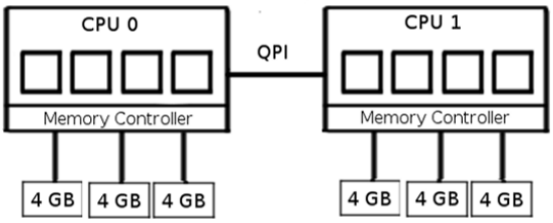

Figure 2.1: The Intel Nehalem CPUs on a compute node have three memory channels, each with one DIMM of 4 GB. The two CPUs are connected to each other via Intel's ultra-fast QuickPath Interconnect (QPI).

Interconnect architecture: All nodes contain an InfiniBand network card (4x QDR, 8x PCI-e 2.0). The nodes are connected to each other and 160 TB of central storage by a QLogic 12800 Ultra-HP InfiniBand switch (25.92 Tbps switching capacity). There are several paths that memory can be accessed through: the memory channel, QPI, PCI-e 2.0 bus, and InfiniBand network. The bandwidths of these paths are listed in Table 2.1. Memory that is stored on another node is accessed by first by going from a CPU through the PCI-e 2.0 bus to the InfiniBand network card and then out through the InfiniBand switch to the second node's InfiniBand network card, the second node's PCI-e 2.0 data bus, followed by the second node's CPU and its connected memory channel.

Table 2.1: Bandwidth of tara's architecture.

Device/Bus	Bandwidth (GB/s)
QPI	12.8
Memory Channel	10.8
8x PCI-e 2.0	3.9
InfiniBand 4x QDR	3.5

3. Graph 500 Specifications

3.1. The Benchmark Specifications

For the purposes of the benchmark, we will consider an unweighted, undirected graph to be $G = (V, E)$, where G is a collection of a set of vertices V and a corresponding set of edges E. The Graph 500 benchmark utilizes a particular graph structure, defining the scale S of a graph and requiring that the number of vertices be a power of two; that is, $|V| = 2^S$. Furthermore, there are in total M edges per vertex, so that the number of edges is $|E| = M\ |V|$. The benchmark currently prescribes M to be 16.

The Graph 500 benchmark requires the execution of the following steps:

- **Generate an edge list**
  For a given scale S, a random edge list is created for the purpose of constructing the graph. The output edge list contains pairs of vertices (v_1, v_2) which represents an edge between the two vertices having indices v_1 and v_2.

  The edge list is generated with a Kronecker generator or a similar generator that produces an equivalent edge list. For detailed information on the Kronecker generator, see [2].

- **Kernel 1: Construct a graph**
  This step accepts the edge list and creates a valid graph. This requires the removal of self-loops (instances where the start and destination vertex are the same). The constructed graph is returned in a convenient data structure, such as an adjacency matrix or adjacency list.

- **Randomly generate 64 search keys**
  The specification requires that a different start vertex be used in each of 64 executions of the search. The search keys indicate the root vertex for a given iteration. The start vertex should produce a non-trivial search, meaning it must have at least one edge to another vertex.

- **Kernel 2: Perform a breadth-first search starting at each search key**
  Kernel 2 is the step of the benchmark whose performance determines the ranking in the list. Using each of the 64 different start vertices in turn, a breadth-first search (BFS) uncovers the connected component of the graph. A connected component is defined as the set of vertices and edges that are reachable from the starting vertex. Figure 3.1 provides a visual explanation of the search. The general algorithm for a breadth-first search is as follows:

  1. Add the root vertex to a queue.
  2. Dequeue the root vertex and mark it as visited, add each of its neighboring vertices to the queue.
  3. Dequeue each vertex individually, marking the vertex as visited and adding its unvisited neighbors to the queue.
  4. Repeat the previous step until the queue is empty, indicating that all connected vertices have been visited.

- **Validate that search results are correct**
  This step checks for errors in the discovered graph by performing the following tests:

  1. The BFS tree does not contain cycles.
  2. Each edge connects vertices whose BFS levels differ by exactly one.
  3. Every edge in the input list has vertices with levels that differ by at most one or that both are not in the BFS tree.
  4. The BFS tree spans the entire connected component.
  5. A vertex and its parent are joined by an edge in the original graph.

- **Output**
  The output must include the following information: scale, number of edges, number of breadth-first searches, graph construction time, search timings (min, max, median, mean, standard deviation), number of edges traversed (min, max, median, mean, standard deviation), and traversed edges per second (min, max, median, mean, standard deviation, harmonic mean). An example of this output, taken from our highest GTEPS run, is shown in Figure 3.2.

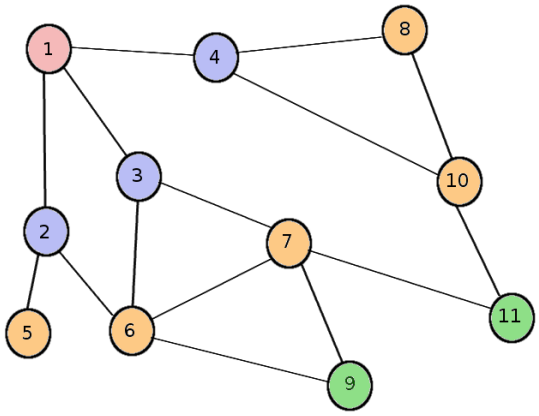

Figure 3.1: Illustration of a breadth-first search starting from root vertex 1. Vertices {2,3,4} are discovered in the first step, vertices {5,6,7,8,10} are discovered in the second step, and finally vertices {9,11} are discovered in the third step.

The Graph 500 uses the harmonic mean of the traversed edges per second (TEPS) to determine a supercomputer's ranking on its list. The harmonic mean is often used when considering an average of rates. Here, the final TEPS result is calculated as

$$\left(\frac{1}{TEPS_1} + \frac{1}{TEPS_2} + \cdots + \frac{1}{TEPS_{64}}\right)^{-1},$$

where $TEPS_i$, $i = 1, \ldots, 64$, is the traversed edges per second for the i^{th} execution of the search. By examining the average rate at which a machine can traverse large graphs, the Graph 500 benchmark is able to provide information about the computer's aptitude for solving data-intensive problems.

3.2. Reference Code

We downloaded version 2.1.4 of the Graph 500 reference code from the benchmark's website. This code, which is written in the programming language C with the standard parallel communications library MPI, was compiled on tara using the `gcc` compiler with OpenMPI 1.3.3 (on tara, this corresponds to setting the `switcher` program to use the `gcc-openmpi-1.3.3-p1` configuration). Since OpenMPI 1.3.3 is not MPI 2.2 compliant we are forced to remove an error check in the header file `/graph500-2.1.4/mpi/mpi_workarounds.h` so that the code may define the MPI datatypes it needs. Compilation produces five executables. For this study we use the executable `/graph500-2.1.4/mpi/graph500_mpi_simple`, which distributes vertices across processes. The breadth-first search is executed using nonblocking point-to-point communication commands `MPI_Isend` and `MPI_Irecv` to update visited status and parents on the owner process. This information is also used to update local queues. The command `MPI_Allreduce` is used to check if queues are globally empty, indicating the search is complete. The code for this procedure can be found in the `run_bfs` function written in the file `/graph500-2.1.4/mpi/bfs_simple.c`.

The validation step sometimes produced segmentation faults. To avoid this we removed the validation step from `/graph500-2.1.4/mpi/main.c`, but this portion of code is also responsible for counting the number of traversed edges. To get a TEPS rating we instead divide the total number of edges by the search time. Due to the benchmark specification of 16 edges connected to each vertex, the graph is almost entirely connected, so this has almost no effect on the results. Logging was inserted to record overall run time of the benchmark. For debugging purposes, the reference code was also modified to output edge list, search keys, and timings if the appropriate flags are set on execution.

The following is a complete list of all code changes which were made to the original reference code, for the purposes of running the benchmark on tara and writing this paper.

```
SCALE:                          31
edgefactor:                     16
NBFS:                           64
graph_generation:               114.382
num_mpi_processes:              256
construction_time:              135.525
min_time:                       36.0322
firstquartile_time:             36.2045
median_time:                    36.296
thirdquartile_time:             36.3707
max_time:                       37.5305
mean_time:                      36.3153
stddev_time:                    0.201449
min_nedge:                      34359738368
firstquartile_nedge:            34359738368
median_nedge:                   34359738368
thirdquartile_nedge:            34359738368
max_nedge:                      34359738368
mean_nedge:                     34359738368
stddev_nedge:                   0
min_TEPS:                       9.15514e+08
firstquartile_TEPS:             9.4471e+08
median_TEPS:                    9.46654e+08
thirdquartile_TEPS:             9.49047e+08
max_TEPS:                       9.53585e+08
harmonic_mean_TEPS:             9.46149e+08
harmonic_stddev_TEPS:           661248
min_validate:                   2.35621e-310
firstquartile_validate:         0
median_validate:                0
thirdquartile_validate:         0
max_validate:                   0
mean_validate:                  0
stddev_validate:                0
total_time:                     2575.74 seconds
```

Figure 3.2: Output from our highest GTEPS run. Note that some statistics are no longer meaningful due to changes from the reference implementation, which are discussed in detail in Section 3.2. Statistics with suffix `_validate` have been left at their default value of zero, due to the validation step being bypassed. Statistics with suffix `_nedge` are set to $2^{31} \cdot 16$ because the total number of edges has been used in place of the number of traversed edges in computing GTEPS. The `total_time` statistic has been added by us and reports observed wall clock time in seconds.

Source file: `graph500-2.1.4/mpi/main.c`

- In `main()`, recorded start and end time of the whole program
- In `main()`, collected total memory usage summed over all processes. When calling this, the maximum and minimum memory usage over all processes is printed to stdout
- In `main()`, added support for an additional (optional) command line argument `debug`. When this is present, code prints out the edgelist, search keys, and parent list
- In `main()`, added logging of search keys if `debug` flag is set
- In `main()`, added logging of edgelist if `debug` flag is set
- In `main()`, computed "traversed edges" as total edges in graph (workaround)
- In `main()`, printed out parent list (result of BFS) if `debug` flag is set
- In `main()`, commented out validation block
- In `main()`, printed out total memory usage `VmRSS` and `VmSize`
- In `main()`, printed elapsed time of entire program

Source file: `graph500-2.1.4/mpi/Makefile`

- Added compilation of new source files `memory_parallel.c` and `memory.c`

Source file: `graph500-2.1.4/prng.c`. This code does not get used in the executable `graph500_mpi_simple` that is used in our runs.

Table 3.1: Graph size in number of vertices $|V|$, number of edges $|E|$, predicted memory usage, observed memory usage, and minimum number of nodes required on tara, as the scale S is increased.

Scale	$\|V\| = 2^S$ (millions)	$\|E\| = 16\|V\|$ (millions)	Predicted memory (GB)	Memory used (GB)	Minimum nodes required
20	1.04	16.77	0.25	3.67	1
21	2.09	33.55	0.5	4.20	1
22	4.19	67.10	1	5.25	1
23	8.38	134.21	2	7.35	1
24	16.77	268.43	4	11.53	1
25	33.55	536.87	8	19.90	1
26	67.10	1,073.74	16	36.62	2
27	134.21	2,147.48	32	70.01	4
28	268.43	4,294.96	64	136.79	8
29	536.87	8,589.93	128	270.28	16
30	1,073.74	17,179.86	256	537.17	32
31	2,147.48	34,356.73	512	1,096.53	64

- In `make_mrg_seed()` and `init_random()`, added logging of seed
- In `make_mrg_seed()` and `init_random()`, added an undefined constant to prevent compilation

New source files:

- `memory.c`
- `memory.h`
- `memory_parallel.c`
- `memory_parallel.h`

3.3. Scale Size and Memory Requirements

Performance benchmarks are intended to stress the system, therefore intuition suggests it is beneficial to push the scale S of the problem as large as possible. This increases the memory required by the code, and the available memory of the system will constrain the size of the graph size eventually. The benchmark specifications require that the number of MPI processes be a power of 2. Since our cluster tara has 82 computer nodes with two quad-core CPUs, the possible numbers of nodes are 1, 2, 4, 8, 16, 32, and 64, with 1, 2, 4, or 8 processes per node.

Table 3.1 states the number of vertices $|V|$, the number of edges $|E|$, memory usage predicted and observed, and the minimum number of compute nodes required for running problems with some selected scale sizes on tara. The Graph 500 specification provides the formula $256 \cdot 2^S$ to predict the number of bytes required by the reference code. However, we found that the predicted memory formula represents approximately half of the actual memory usage observed in our runs. The last column of the table displays the minimum number of nodes required to hold a graph of the given scale size. This number considers the actual memory usage, that each node has 24 GB of memory, and that the number of nodes must be a power of two. Since 64 is the largest such number of nodes which tara can accommodate, $S = 31$ is the largest scale possible to run on tara.

Since memory is the limiting factor for the largest possible problem size, we first addressed the actual memory used on tara in the execution of the code. We added a check for memory usage toward the end of the benchmark program, after the breadth first search and just before memory is deallocated. In Unix, memory usage and other system information is stored in the `proc` filesystem. We record the memory usage on each process by reading the field `VmSize` from the file `/proc/self/status`, which represents both physical memory and swap space in use by a process.

4. Results

The Graph 500 reference code was executed a total of 252 times on tara. Runs used scale sizes from 20 to 31, number of nodes (1, 2, 4, 8, 16, 32, 64), and number of processes per node (1, 2, 4, 8). The numbers of processes per

node are abbreviated as 1 ppn, 2 ppn, 4 ppn, 8 ppn in the following tables. In all tables, the long dashes "—" indicate cases where more memory is required than available on the given number of nodes, in accordance with the minimum number of nodes required for each scale in Table 3.1. Note that for a given scale size, the same problem is generated for all configurations of nodes and ppn via a seed in the random graph generator. This means that the performance results within each setting of S are directly comparable. Table 4.1 shows the maximum observed memory usage per process and Table 4.2 expresses this maximum as a percentage of a node's 24 GB memory.

Table 4.3 displays the total wall clock time for kernel 2 in each run; that is, the sum of wall times of the 64 breadth-first searches. In general, when a large number of processes are in use to handle a small scale problem (upper right of the table), it is clear that communication overhead is dominating useful work. Consider fixing a ppn and scale, and looking across the rows as number of nodes increase. For 1 ppn, run time roughly halves as nodes used doubles for medium to large scales. There are some exceptions, such as scale $S = 23$ and $S = 24$, where run times increase from 1 to 2 nodes. For 2 ppn, the scaling improves, and is much closer to halving as number of nodes doubles. For 4 ppn, performance seems to suffer when increasing from 4 to 8 nodes. There is mild improvement from 8 to 16 nodes, and more improvement beyond 16 nodes, especially for larger S. For 8 ppn, we see a similar pattern to 4 ppn.

Now let us fix the number of nodes and the scale S, and look down the columns of Table 4.3 as ppn increases. For a single node, times are often increasing when moving from 1 ppn to 2 ppn. When increasing from 2 ppn to 4 ppn and 8 ppn, run times are closer to halving. The speedup is better for two nodes than one node, and run times are closer to halving as ppn is doubled. Using four nodes, run times are halving from 1 to 2 ppn and from 2 to 4 ppn, but are increasing from 4 to 8 ppn (to roughly the level of 2 ppn). When eight nodes are in use, performance is best when 2 ppn are in use. For 8 ppn the performance is roughly at the level of 1 ppn. The same trends are present for 16, 32, and 64 nodes. When 64 nodes are used, there is improvement in going from 1 to 2 ppn for large S, and also some improvement from 2 to 4 ppn.

Now consider Table 4.4, which shows billions of traversed edges per second (GTEPS) for each run; this is the data that is visualized in the summary plot in Figure 1.2 in Section 1. These GTEPS results are related to the times in Table 4.3, where a higher GTEPS rate for a given scale and node combination means lower overall runtime. For each scale size S, it is clear that increasing the number of nodes improves GTEPS. Increasing ppn to the maximum generally provides a small improvement if a small number of nodes are in use, and a reduction in performance with a larger number of nodes. Consider only the rightmost column where 64 nodes are used. For small to medium S, the best GTEPS rating is obtained on 1 or 2 ppn. For the largest S, the best GTEPS is obtained for 2 or 4 ppn.

Let us summarize the trends in performance. There appears to be a sudden drop in performance when increasing from 4 to 8 nodes. When more nodes are in use, increasing ppn becomes less effective. Finally, use of more nodes provides the highest GTEPS, and using 2 or 4 ppn rather than 1 or 8 ppn provides a significant improvement for the largest S. The best overall result of 0.946 GTEPS was observed using 64 nodes with 4 processes per node for scale 31. This placed tara at rank 98 on the November 2012 Graph 500 list[2], as shown in Figure 1.1 in Section 1.

We now connect the observed performance results to the memory architecture discussed in Section 2.

- On each node there are six 4 GB DIMMs. Each Nehalem processor has three memory channels connecting it directly to three of the six DIMMs. A running process therefore has direct access to those three DIMMs. When the other half of memory needs to be accessed, communication must take place from the host Nehalem processor to its peer, and then on to the its connected memory. The latency is close to 75% higher when accessing memory from one processor through the second [3]. Consider again the column in Table 4.3 when only one node is in use. The run time of 1 ppn is significantly faster than double that of 2 ppn. This may indicate that the memory architecture is becoming a bottleneck from many interprocess communications between two or more processes involved in the breadth-first search.

- Consider the network connecting the compute nodes for another potential bottleneck. Recall that when multiple nodes are in use, the randomly generated graph is distributed across the nodes' pooled memory. A breadth-first search being conducted on 64 nodes has roughly a 1 in 64 chance (or 1.6%) that a vertex stored on a given process has one of its neighbors also stored locally. The other 98.4% of the time the neighbor will be located on a different node and communication will need to take place over the InfiniBand network to conduct the search.

[2] http://www.graph500.org/results_nov_2012.

Table 4.1: Maximum memory used per process (GB). Insufficient memory denoted by "—".

		1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
Scale	1ppn	0.73	0.48	0.41	0.38	0.37	0.36	0.37
= 20	2ppn	0.52	0.47	0.44	0.43	0.43	0.43	0.45
	4ppn	0.45	0.44	0.43	0.42	0.43	0.45	0.49
	8ppn	0.42	0.43	0.43	0.44	0.45	0.49	0.57
Scale	1ppn	1.26	0.75	0.48	0.42	0.39	0.37	0.37
= 21	2ppn	0.79	0.55	0.48	0.45	0.43	0.44	0.45
	4ppn	0.52	0.48	0.45	0.44	0.44	0.46	0.49
	8ppn	0.45	0.45	0.44	0.44	0.47	0.49	0.57
Scale	1ppn	2.32	1.28	0.75	0.49	0.42	0.39	0.38
= 22	2ppn	1.32	0.81	0.55	0.48	0.46	0.45	0.48
	4ppn	0.79	0.55	0.48	0.46	0.45	0.47	0.49
	8ppn	0.53	0.48	0.46	0.45	0.48	0.49	0.57
Scale	1ppn	4.45	2.34	1.28	0.76	0.50	0.43	0.40
= 23	2ppn	2.38	1.34	0.82	0.56	0.49	0.47	0.49
	4ppn	1.32	0.82	0.55	0.49	0.47	0.48	0.51
	8ppn	0.79	0.55	0.50	0.47	0.49	0.51	0.58
Scale	1ppn	8.70	4.47	2.35	1.29	0.76	0.50	0.44
= 24	2ppn	4.51	2.41	1.35	0.82	0.57	0.51	0.50
	4ppn	2.39	1.35	0.83	0.57	0.51	0.50	0.52
	8ppn	1.33	0.82	0.57	0.51	0.50	0.52	0.59
Scale	1ppn	17.20	8.73	4.49	2.35	1.30	0.77	0.51
= 25	2ppn	8.77	4.55	2.42	1.36	0.83	0.59	0.54
	4ppn	4.52	2.41	1.36	0.84	0.58	0.54	0.56
	8ppn	2.39	1.36	0.84	0.58	0.54	0.56	0.59
Scale	1ppn	—	17.23	8.74	4.49	2.37	1.31	0.80
= 26	2ppn	—	8.80	4.55	2.43	1.37	0.86	0.62
	4ppn	—	4.55	2.43	1.37	0.86	0.61	0.57
	8ppn	—	2.43	1.38	0.86	0.61	0.59	0.63
Scale	1ppn	—	—	17.23	8.75	4.50	2.39	1.32
= 27	2ppn	—	—	8.81	4.56	2.45	1.39	0.87
	4ppn	—	—	4.56	2.45	1.39	0.88	0.65
	8ppn	—	—	2.46	1.39	0.89	0.65	0.74
Scale	1ppn	—	—	—	17.26	8.75	4.52	2.43
= 28	2ppn	—	—	—	8.81	4.59	2.49	1.44
	4ppn	—	—	—	4.59	2.49	1.43	0.95
	8ppn	—	—	—	2.49	1.43	0.95	0.76
Scale	1ppn	—	—	—	—	17.30	8.78	4.55
= 29	2ppn	—	—	—	—	8.84	4.61	2.51
	4ppn	—	—	—	—	4.61	2.51	1.48
	8ppn	—	—	—	—	2.51	1.48	1.03
Scale	1ppn	—	—	—	—	—	17.28	8.82
= 30	2ppn	—	—	—	—	—	8.88	4.64
	4ppn	—	—	—	—	—	4.64	2.55
	8ppn	—	—	—	—	—	2.55	1.58
Scale	1ppn	—	—	—	—	—	—	17.37
= 31	2ppn	—	—	—	—	—	—	8.95
	4ppn	—	—	—	—	—	—	4.76
	8ppn	—	—	—	—	—	—	2.71

Table 4.2: Percent of node's memory used per process. Insufficient memory denoted by "—".

		1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
Scale	1ppn	3.03	2.01	1.72	1.58	1.53	1.51	1.54
= 20	2ppn	2.18	1.98	1.84	1.80	1.77	1.81	1.88
	4ppn	1.89	1.83	1.78	1.77	1.80	1.87	2.05
	8ppn	1.74	1.78	1.78	1.82	1.87	2.04	2.38
Scale	1ppn	5.24	3.12	2.02	1.74	1.61	1.56	1.56
= 21	2ppn	3.30	2.27	1.99	1.86	1.81	1.84	1.89
	4ppn	2.19	1.99	1.87	1.82	1.85	1.91	2.05
	8ppn	1.89	1.85	1.83	1.85	1.94	2.04	2.39
Scale	1ppn	9.67	5.34	3.14	2.04	1.76	1.64	1.60
= 22	2ppn	5.51	3.39	2.29	2.02	1.90	1.88	2.00
	4ppn	3.30	2.28	2.01	1.90	1.88	1.98	2.06
	8ppn	2.19	2.01	1.90	1.89	1.99	2.06	2.39
Scale	1ppn	18.52	9.76	5.34	3.15	2.07	1.79	1.69
= 23	2ppn	9.94	5.60	3.40	2.31	2.06	1.96	2.03
	4ppn	5.51	3.40	2.31	2.06	1.96	2.02	2.11
	8ppn	3.30	2.31	2.08	1.97	2.02	2.11	2.42
Scale	1ppn	36.24	18.63	9.79	5.37	3.18	2.08	1.83
= 24	2ppn	18.80	10.05	5.63	3.43	2.36	2.13	2.10
	4ppn	9.96	5.62	3.44	2.36	2.12	2.09	2.15
	8ppn	5.53	3.43	2.36	2.12	2.09	2.17	2.44
Scale	1ppn	71.67	36.36	18.69	9.81	5.40	3.21	2.14
= 25	2ppn	36.53	18.94	10.07	5.66	3.48	2.44	2.26
	4ppn	18.85	10.06	5.65	3.49	2.40	2.25	2.33
	8ppn	9.97	5.66	3.50	2.41	2.27	2.34	2.47
Scale	1ppn	—	71.77	36.40	18.70	9.89	5.45	3.32
= 26	2ppn	—	36.66	18.95	10.15	5.71	3.58	2.56
	4ppn	—	18.95	10.14	5.73	3.58	2.54	2.39
	8ppn	—	10.14	5.73	3.57	2.56	2.47	2.62
Scale	1ppn	—	—	71.80	36.45	18.74	9.95	5.52
= 27	2ppn	—	—	36.73	19.00	10.21	5.78	3.65
	4ppn	—	—	18.99	10.21	5.77	3.65	2.70
	8ppn	—	—	10.24	5.80	3.69	2.71	3.07
Scale	1ppn	—	—	—	71.93	36.46	18.85	10.11
= 28	2ppn	—	—	—	36.72	19.11	10.37	5.98
	4ppn	—	—	—	19.12	10.37	5.98	3.97
	8ppn	—	—	—	10.37	5.98	3.97	3.16
Scale	1ppn	—	—	—	—	72.09	36.59	18.95
= 29	2ppn	—	—	—	—	36.85	19.21	10.46
	4ppn	—	—	—	—	19.21	10.45	6.17
	8ppn	—	—	—	—	10.45	6.17	4.29
Scale	1ppn	—	—	—	—	—	72.00	36.73
= 30	2ppn	—	—	—	—	—	36.99	19.33
	4ppn	—	—	—	—	—	19.33	10.64
	8ppn	—	—	—	—	—	10.65	6.58
Scale	1ppn	—	—	—	—	—	—	72.36
= 31	2ppn	—	—	—	—	—	—	37.31
	4ppn	—	—	—	—	—	—	19.82
	8ppn	—	—	—	—	—	—	11.28

Table 4.3: Total kernel 2 runtime in HH:MM:SS. Insufficient memory denoted by "—".

		1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
Scale	1 ppn	00:00:27	00:00:29	00:00:16	00:00:08	00:00:05	00:00:02	00:00:02
= 20	2 ppn	00:00:31	00:00:19	00:00:09	00:00:05	00:00:03	00:00:03	00:00:03
	4 ppn	00:00:21	00:00:10	00:00:05	00:00:06	00:00:05	00:00:06	00:00:11
	8 ppn	00:00:11	00:00:06	00:00:09	00:00:12	00:00:13	00:00:19	00:00:51
Scale	1 ppn	00:00:55	00:01:00	00:00:33	00:00:17	00:00:09	00:00:05	00:00:03
= 21	2 ppn	00:01:10	00:00:39	00:00:19	00:00:09	00:00:06	00:00:05	00:00:04
	4 ppn	00:00:46	00:00:22	00:00:11	00:00:12	00:00:10	00:00:09	00:00:12
	8 ppn	00:00:22	00:00:11	00:00:17	00:00:22	00:00:20	00:00:23	00:00:39
Scale	1 ppn	00:01:47	00:02:05	00:01:10	00:00:36	00:00:20	00:00:10	00:00:05
= 22	2 ppn	00:02:26	00:01:20	00:00:38	00:00:20	00:00:13	00:00:09	00:00:07
	4 ppn	00:01:32	00:00:42	00:00:22	00:00:22	00:00:19	00:00:15	00:00:15
	8 ppn	00:00:47	00:00:24	00:00:38	00:00:40	00:00:34	00:00:31	00:00:45
Scale	1 ppn	00:04:05	00:04:24	00:02:17	00:01:13	00:00:40	00:00:21	00:00:11
= 23	2 ppn	00:04:55	00:02:46	00:01:20	00:00:39	00:00:28	00:00:19	00:00:13
	4 ppn	00:02:55	00:01:30	00:00:46	00:00:49	00:00:35	00:00:25	00:00:21
	8 ppn	00:01:34	00:00:48	00:01:19	00:01:19	00:01:00	00:00:47	00:00:54
Scale	1 ppn	00:07:51	00:08:23	00:04:46	00:02:39	00:01:17	00:00:41	00:00:22
= 24	2 ppn	00:09:25	00:05:28	00:02:43	00:01:21	00:00:55	00:00:35	00:00:23
	4 ppn	00:06:17	00:03:07	00:01:34	00:01:38	00:01:15	00:00:48	00:00:34
	8 ppn	00:03:17	00:01:41	00:03:03	00:02:41	00:01:39	00:01:17	00:01:15
Scale	1 ppn	00:23:03	00:16:06	00:09:27	00:05:14	00:02:45	00:01:26	00:00:43
= 25	2 ppn	00:20:53	00:11:05	00:06:33	00:02:53	00:01:46	00:01:09	00:00:42
	4 ppn	00:12:19	00:06:03	00:03:02	00:03:39	00:02:25	00:01:29	00:00:57
	8 ppn	00:06:33	00:03:18	00:04:51	00:05:02	00:03:37	00:02:11	00:02:24
Scale	1 ppn	—	00:43:06	00:19:29	00:10:13	00:05:45	00:02:53	00:01:29
= 26	2 ppn	—	00:22:26	00:11:05	00:06:12	00:03:46	00:02:29	00:01:34
	4 ppn	—	00:12:31	00:06:07	00:06:34	00:05:22	00:02:33	00:01:36
	8 ppn	—	00:07:03	00:09:37	00:10:19	00:06:40	00:04:19	00:02:46
Scale	1 ppn	—	—	00:48:55	00:21:45	00:10:53	00:05:40	00:03:00
= 27	2 ppn	—	—	00:23:11	00:12:44	00:07:26	00:04:51	00:02:31
	4 ppn	—	—	00:12:34	00:14:24	00:10:53	00:05:10	00:02:54
	8 ppn	—	—	00:20:51	00:21:00	00:12:05	00:07:53	00:04:46
Scale	1 ppn	—	—	—	00:50:49	00:22:26	00:12:06	00:05:59
= 28	2 ppn	—	—	—	00:24:44	00:13:40	00:09:57	00:05:37
	4 ppn	—	—	—	00:28:31	00:22:10	00:10:06	00:05:30
	8 ppn	—	—	—	00:45:05	00:27:28	00:14:16	00:09:09
Scale	1 ppn	—	—	—	—	00:52:58	00:25:49	00:12:09
= 29	2 ppn	—	—	—	—	00:28:14	00:19:39	00:09:55
	4 ppn	—	—	—	—	00:40:55	00:19:48	00:10:54
	8 ppn	—	—	—	—	00:48:00	00:28:44	00:17:12
Scale	1 ppn	—	—	—	—	—	01:00:30	00:25:45
= 30	2 ppn	—	—	—	—	—	00:39:53	00:19:24
	4 ppn	—	—	—	—	—	00:38:19	00:19:35
	8 ppn	—	—	—	—	—	00:54:40	00:33:25
Scale	1 ppn	—	—	—	—	—	—	01:00:06
= 31	2 ppn	—	—	—	—	—	—	00:39:32
	4 ppn	—	—	—	—	—	—	00:39:00
	8 ppn	—	—	—	—	—	—	01:04:45

Table 4.4: Billions of traversed edges per second (GTEPS). Insufficient memory denoted by "—".

		1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
Scale	1ppn	0.039	0.036	0.065	0.124	0.219	0.406	0.597
= 20	2ppn	0.034	0.056	0.111	0.214	0.291	0.348	0.334
	4ppn	0.052	0.102	0.191	0.179	0.184	0.174	0.096
	8ppn	0.092	0.171	0.114	0.082	0.078	0.053	0.029
Scale	1ppn	0.039	0.036	0.062	0.122	0.203	0.399	0.654
= 21	2ppn	0.029	0.058	0.107	0.203	0.313	0.382	0.447
	4ppn	0.048	0.095	0.190	0.187	0.219	0.243	0.153
	8ppn	0.094	0.182	0.107	0.095	0.108	0.091	0.055
Scale	1ppn	0.040	0.034	0.063	0.123	0.228	0.402	0.725
= 22	2ppn	0.029	0.056	0.108	0.208	0.316	0.403	0.598
	4ppn	0.047	0.099	0.196	0.167	0.252	0.320	0.251
	8ppn	0.092	0.177	0.115	0.098	0.135	0.142	0.093
Scale	1ppn	0.037	0.034	0.061	0.119	0.199	0.340	0.765
= 23	2ppn	0.031	0.052	0.106	0.215	0.318	0.370	0.662
	4ppn	0.048	0.092	0.198	0.181	0.223	0.361	0.429
	8ppn	0.089	0.172	0.103	0.109	0.159	0.191	0.153
Scale	1ppn	0.029	0.032	0.061	0.101	0.225	0.398	0.783
= 24	2ppn	0.028	0.053	0.103	0.204	0.310	0.392	0.748
	4ppn	0.045	0.094	0.195	0.155	0.248	0.367	0.503
	8ppn	0.086	0.176	0.119	0.103	0.170	0.232	0.220
Scale	1ppn	0.024	0.034	0.060	0.116	0.191	0.370	0.762
= 25	2ppn	0.029	0.051	0.105	0.207	0.279	0.481	0.815
	4ppn	0.047	0.094	0.196	0.155	0.217	0.358	0.613
	8ppn	0.084	0.166	0.128	0.107	0.178	0.266	0.116
Scale	1ppn	—	0.027	0.059	0.111	0.207	0.335	0.750
= 26	2ppn	—	0.049	0.100	0.205	0.274	0.473	0.870
	4ppn	—	0.092	0.183	0.177	0.240	0.445	0.669
	8ppn	—	0.164	0.120	0.113	0.187	0.301	0.400
Scale	1ppn	—	—	0.049	0.102	0.195	0.416	0.758
= 27	2ppn	—	—	0.099	0.179	0.292	0.484	0.835
	4ppn	—	—	0.176	0.150	0.256	0.447	0.769
	8ppn	—	—	0.116	0.104	0.172	0.281	0.493
Scale	1ppn	—	—	—	0.093	0.185	0.402	0.762
= 28	2ppn	—	—	—	0.182	0.309	0.478	0.773
	4ppn	—	—	—	0.145	0.211	0.457	0.839
	8ppn	—	—	—	0.107	0.180	0.299	0.503
Scale	1ppn	—	—	—	—	0.174	0.373	0.735
= 29	2ppn	—	—	—	—	0.308	0.482	0.929
	4ppn	—	—	—	—	0.251	0.476	0.907
	8ppn	—	—	—	—	0.183	0.346	0.547
Scale	1ppn	—	—	—	—	—	0.311	0.691
= 30	2ppn	—	—	—	—	—	0.468	0.929
	4ppn	—	—	—	—	—	0.477	0.929
	8ppn	—	—	—	—	—	0.362	0.583
Scale	1ppn	—	—	—	—	—	—	0.605
= 31	2ppn	—	—	—	—	—	—	0.911
	4ppn	—	—	—	—	—	—	0.946
	8ppn	—	—	—	—	—	—	0.621

Table 4.5: Processes per node with highest GTEPS rate for given scale size and number of nodes. Insufficient memory denoted by "—".

Scale	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
20	8	8	4	2	2	1	1
21	8	8	4	2	2	1	1
22	8	8	4	2	2	2	1
23	8	8	4	2	2	2	1
24	8	8	4	2	2	1	1
25	8	8	4	2	2	2	2
26	—	8	4	2	2	2	2
27	—	—	4	2	2	2	2
28	—	—	—	2	2	2	4
29	—	—	—	—	2	2	2
30	—	—	—	—	—	4	4
31	—	—	—	—	—	—	4

For 8 processes per node and scale 31, the InfiniBand network becomes saturated and acts as a bottleneck due to the high amount of traffic and relatively lower bandwidth. Therefore, twice as many processes competing for network resources is another reason that 8 ppn is often slower than 4 ppn. Table 4.5 shows which ppn has the highest GTEPS rate for each combination of scale size and nodes used. As the number of nodes increases, the interplay between an increasing number of processes communicating and the potential bandwidth increase can be seen.

- Another possible insight comes from the the architecture of a network switch. A switch is capable of giving full bandwidth communication between pairs of nodes. For example, when a 64 node case is run, if each node were to be paired up with another node there would be 32 full speed network connections each transmitting data between the pairs at 3.5 GB/s. However if 63 of the nodes all tried to communicate with the remaining node they would all need to share the 3.5 GB/s connection to that node. Recall that point-to-point MPI operations are utilized in the reference code, so that all pairs of processes may be communicating. As the number of nodes increases, the possible number of parallel connections through the network switch increases and more efficient load balancing is able to take place. Table 4.3 indicates that the switch may be a bottleneck for as little as four nodes. Focusing on the four nodes column of the table, we suspect that the slowdown seen at 8 ppn may be caused by a large amount of mutual communication between processes among the four nodes.

5. Conclusions

We have discussed the Graph 500 as a benchmark to measure a computer's ability to efficiently access memory. The benchmark prescribes a series of random graph generations and breadth-first searches of these graphs, using traversed edges per second (TEPS) as a performance measurement. Starting with the Graph 500 reference implementation, we have made some customizations to run the benchmark on our cluster tara in order to carry out a performance study. We found that the fastest GTEPS rate obtained was 0.946, on 64 nodes using 4 processes per node, and a scale size of 31. A careful look at the memory and network architecture of the nodes helped to explain the changes in GTEPS as nodes and processes per node were varied. In general, the GTEPS rating did not benefit from using the maximum processes per node. This is also true for more CPU-intensive workloads such as a classical iterative solver for a sparse system of linear equations in [4], but to a much lesser extent, as more processes per node give less efficient scaling, but still benefits overall run time and allow for use of more processing cores on fewer nodes. Our performance study illustrates the subtle ways that the Graph 500 benchmark stresses memory access across a distributed system. The GTEPS rate of 0.946 placed tara at rank 98 on the November 2012 Graph 500 list, out of 123 machines ranked so far. This ranking was announced at the Supercomputing 2012 conference in November 2012 in the Birds-of-a-Feather session "Fifth Graph500 List." To get an impression of the change in the list over six months, note that tara would have ranked 58 on the June 2012 list, as discussed in the tech. report [5] from August 2012.

Acknowledgments

The results and experiences summarized in this paper were obtained as part of the REU Site: Interdisciplinary Program in High Performance Computing (www.umbc.edu/hpcreu) in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County (UMBC) in Summer 2012. The program combines an intensive training in scientific, statistical, and parallel computing with team work on consulting projects posed by outside clients. Three of the four undergraduate student team members, Jordan Angel, Nathan Wardrip, and Amy Flores, were able to travel to Supercomputing 2012 in Salt Lake City to receive the certificate for the ranking. They met with faculty mentor Dr. Matthias Gobbert and client David Mountain (Advanced Computing Systems Research Program); see Figure 5.1. Not able to join them at the conference were team member Justine Heritage, graduate assistant and

Figure 5.1: Pictured are (left to right) the client David Mountain, mentor Dr. Matthias Gobbert, and students Jordan Angel, Nathan Wardrip, and Amy Flores, holding the official certificate for the Graph 500 ranking.

HPCF RA Andrew Raim, and client Richard Murphy (now with Micron Technology, Inc., who was affiliated with Sandia National Laboratories during the project). The REU Site: Interdisciplinary Program in High Performance Computing is funded jointly by the National Science Foundation and the National Security Agency (NSF grant no. DMS–1156976), with additional support from UMBC, the Department of Mathematics and Statistics, the Center for Interdisciplinary Research and Consulting (CIRC), and the UMBC High Performance Computing Facility (HPCF). We sincerely appreciate the additional funding for the students to travel to Supercomputing 2012.

References

[1] R. C. Murphy, K. B. Wheeler, B. W. Barrett, J. A. Ang, Introducing the Graph 500, Cray User Group 2010 Proceedings (2010).

[2] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication, in: A. Jorge, L. Torgo, P. Brazdil, R. Camacho, J. Gama (Eds.), Knowledge Discovery in Databases: PKDD 2005, Vol. 3721 of Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 133–145.

[3] G. Balakrishnan, R. M. Begun, Optimizing the performance of IBM System x and BladeCenter servers using Intel Xeon 5500 series processors, IBM White Paper (March 2009).

[4] A. M. Raim, M. K. Gobbert, Parallel performance studies for an elliptic test problem on the cluster tara, Tech. Rep. HPCF–2010–2, UMBC High Performance Computing Facility, University of Maryland, Baltimore County (2010).
URL www.umbc.edu/hpcf

[5] J. B. Angel, A. Flores, J. Heritage, N. Wardrip, A. M. Raim, M. K. Gobbert, R. C. Murphy, D. J. Mountain, Graph 500 performance on a distributed-memory cluster, Tech. Rep. HPCF–2012–11, UMBC High Performance Computing Facility, University of Maryland, Baltimore County (2012).
URL www.umbc.edu/hpcf