Profiling Non-numeric OpenSHMEM Applications with the TAU Performance System

John Linford[2], Tyler A. Simon[1,2], Sameer Shende[2,3], and Allen D. Malony[2,3]

[1] University of Maryland Baltimore County
[2] ParaTools Inc.
[3] University of Oregon

Abstract. The recent development of a unified SHMEM framework, OpenSHMEM, has enabled further study in the porting and scaling of applications that can benefit from the SHMEM programming model. This paper focuses on non-numerical graph algorithms, which typically have a low FLOPS/byte ratio. An overview of the space and time complexity of Kruskal's and Prim's algorithms for generating a minimum spanning tree (MST) is presented, along with an implementation of Kruskal's algorithm that uses OpenSHEM to generate the MST in parallel without intermediate communication. Additionally, a procedure for applying the TAU Performance System to OpenSHMEM applications to produce in-depth performance profiles showing time spent in code regions, memory access patterns, and network load is presented. Performance evaluations from the Cray XK7 "Titan" system at Oak Ridge National Laboratory and a 48 core shared memory system at University of Maryland, Baltimore County are provided.

1 Introduction

Non-numerical algorithms (NNA) are characterized by a low FLOPS/byte ratio and can be defined as those which spend most of their computational time doing either a search or sort. Memory locality, not computational load, is the primary performance factor. This class of algorithms is particularly challenging for HPC systems in the Top500 [11] as these systems are optimized for compute-intensive codes. NNAs often involve searching or traversing graphs, which are defined by a collection of *vertices* connected by *edges*. Graphs are used to model problems defined in terms of relationships or connections between objects. The mathematical properties of graphs facilitate the modeling of many useful computational problems, e.g. problems related to connectivity and routing optimization in networks.

Many graph algorithms begin by finding a *minimum spanning tree* (MST). Given a connected graph, a spanning tree is a subgraph that connects all the vertices and is a tree. For weighted graphs, the sum of the edge weights in a spanning tree is the weight of the spanning tree. A minimum spanning tree is a spanning tree with weight less than or equal to the weight of every other spanning tree. MSTs have many practical applications in communication networks,

S. Poole, O. Hernandez, and P. Shamis (Eds.): OpenSHMEM 2014, LNCS 8356, pp. 105–119, 2014.

network design and layout of highway systems. They provide a reasonable way to cluster points in space into natural groups and can be used to approximate solutions to hard problems, like the Traveling Salesman Problem [13].

Two common algorithms for finding an MST are Kruskal's [10] and Prim's [15]. The primary difference between these algorithms is the order of graph traversal (breadth first vs. depth first). We provide an overview of the space and time complexity of these algorithms and present an OpenSHMEM [3,7] implementation of Kruskal's algorithm. The implementation uses the symmetric hierarchical memory to generate the MST in parallel without intermediate communication, thereby minimizing network load. We use the TAU Performance System [16] to quantify the performance of our OpenSHMEM MST algorithm.

2 Background

2.1 The TAU Performance System

The challenge of developing parallel scientific and engineering applications for scalable, high-performance computer systems routinely involves the use of parallel performance tools to gain a deeper understanding of the code's execution characteristics and to guide optimizations. The evolving hardware technology of parallel computing platforms and the sophistication of software technologies to program them gives rise to complex performance interactions that requires careful measurement and analysis to understand. TAU is a robust, powerful, state-of-the-art toolset for performance investigation that has been applied across many scalable parallel computing paradigms and environments. TAU works efficiently on hundreds of thousands of threads and processes with codes written in Fortran, C/C++, Python, UPC, and Chapel, utilizing MPI, SHMEM, and DMAPP for communication, and pthreads and OpenMP for multi-threading.

Shown in Figure 1, TAU consists of three layers: instrumentation, measurement, and analysis. Each layer uses multiple modules that may be configured in a flexible manner under user control. TAU implements a multi-level instrumentation framework that includes source, runtime, and compiler-based instrumentation to expand the scope of performance instrumentation. This design makes it possible for TAU to easily provide alternative instrumentation techniques that target a common measurement API. The role of the instrumentation layer is to insert code (a.k.a. probes) to make performance events visible to the measurement layer. Performance events can be defined and instrumentation inserted in a program at several levels of the program transformation process. A complete performance view may require contribution of event information across code levels. To support this, TAU's instrumentation mechanisms are based on the code type and transformation level: source (manual, preprocessor), object (compiler-based instrumentation), library interposition (pre-wrapped libraries), static linker (redirecting and substituting calls to an alternate routine), runtime linker (runtime interposition of libraries), binary (pre-execution at runtime), re-writing (dynamic instrumentation), interpreter (language runtime), virtual

Listing 1.1. OpenSHMEM Minimum Spanning Tree

```
start_pes(0);
if (shmem_my_pe() == 0) {
  // Read graph size (number of nodes) from file
  // Broadcast graph size to all other PEs
}

// All PEs receive graph size
shmem_barrier_all();

// Calculate work division
nNodes = graphSize / n_pes;

// Allocate shmem for graph
graph = (int*)shmalloc(graphSize*nNodes*sizeof(int));
span = (int*)shmalloc(graphSize*nNodes*sizeof(int));
memset(span, 0, graphSize*nNodes*sizeof(int));

int * buffer = NULL;
if (my_pe == 0) {
  buffer = malloc(graphSize*nNodes*sizeof(int));
  for (i=0; i<n_pes; ++i) {
    for (j=0; j<graphSize*nNodes; ++j) {
      // Read edge weight from file into buffer[j]
    }
    shmem_int_put(graph, buffer, nNodes*graphSize, i);
  }
}

// All PEs receive graph
shmem_barrier_all();
free(buffer);

// All PEs pick their lowest edges (Kruskal's algorithm)
for (j=0; j<nNodes; ++j) {
  minWeight = INT_MAX;
  minNode = 0;
  for (i=0; i<graphSize; ++i) {
    weight = graph[j*graphSize+i];
    if (weight && (weight < minWeight)) {
      minWeight = weight;
      minNode = i;
    }
  }
  span[j*graphSize+minNode] = minWeight;
}

shmem_barrier_all();
// span now contains the minimum spanning tree
```

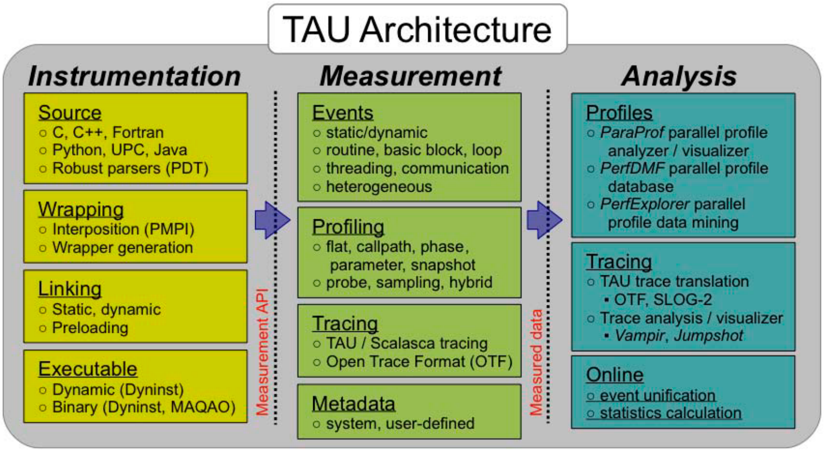


Fig. 1. The TAU framework architecture

machine (byte-code instrumentation), and operating system (kernel-level instrumentation). This broad coverage across instrumentation levels provides flexibility in exploring performance artifacts at any level of program transformation. TAU is open source and all instrumentation, measurement, and analysis tools are distributed under a BSD-like license.

The core of TAU is its scalable measurement infrastructure which provides rich parallel profiling and tracing support. Performance information containing execution time and hardware counter data can be captured for instrumented events for all threads of execution. Communication events additionally record message-related data that track process interactions. TAU can also associate metadata with performance experiments.

SHMEM application analysis is achieved with TAU via source code instrumentation, library interposition, compiler instrumentation, and sampling. TAU can track memory allocations – both in a PE's local memory and in the symmetric heap – and network I/O caused by transfers to and from symmetric memory. In statically linked binaries (e.g. for Cray systems), memory tracking is achieved by wrapping all allocation and deallocation routines at link time. Dynamically linked applications may use *tau_exec* to preload a memory tracking runtime library.

To track OpenSHMEM API calls and monitor communication between PEs, TAU provides a single wrapper library applicable to all OpenSHMEM implementations. In past, SHMEM wrapper library development was determined by the specific SHMEM library used by the system. This necessitated the production and maintenance of several library variants and complicated the goal of portable performance observation for the SHMEM runtime layer. However, the portability of the OpenSHMEM standard enables TAU to maintain only a single implementation of a single OpenSHMEM wrapper library. This approach provides performance measurements at the language and the runtime level to achieve complete coverage of the OpenSHMEM application source and runtime libraries.

2.2 Minimum Spanning Tree Algorithm

We use the symmetric hierarchical memory defined by OpenSHMEM to calculate the MST as shown in Listing 1.1. The basic idea is to divide the graph among the OpenSHMEM processing elements (PEs) by writing the edge weights to symmetric memory (the *graph* variable in Listing 1.1). Each PE examines the nodes it has been sent and writes the minimum edge weight to another region of symmetric memory (*span* in Listing 1.1). This effectively distributes Kruskal's greedy algorithm across all PEs. By selecting the minimum edge connected to all nodes, we ensure the resultant graph will be the minimum spanning tree.

3 Related Work

OpenSHMEM [3,7] has emerged as an effort to join the disparate SHMEM implementations into a common, portable and high performance standard. Bader and Cong have demonstrated fast data structures for distributed shared memory [1]. Pophale et. al defined the performance bounds of OpenSHMEM [14].

NNA benchmarks are emerging to evaluate HPC system at scale. The Graph 500 is one such benchmark [12]. The problem of finding a minimum spanning tree (MST) can be formally stated as: given an undirected, weighted graph $G = (V, E)$, a minimum spanning tree is the set of edges T in E that connect all vertices in V at a minimum cost. Figure 2 illustrates the MST for a fully connected graph with random edge weights and eight vertices. Both Prim's and Kruskal's algorithms are considered "greedy" since they make the locally optimal choice at each iteration, i.e. they choose an edge with minimum weight that does not create a cycle [15,10].

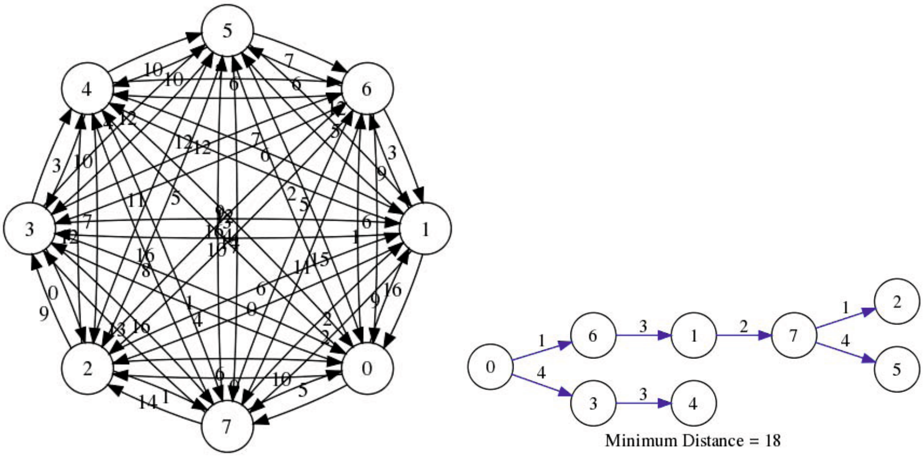


Fig. 2. Fully connected eight node graph with minimum spanning tree

3.1 Performance Analysis

Both profiling and tracing are relevant to better understanding the performance characteristics of an application. While profiling shows summary statistics, tracing can reveal the temporal variation in application performance. Among tools that use the direct measurement approach, the VampirTrace [9] package provides a wrapper interposition library that can capture the traces of I/O operations using the library preloading scheme used in `tau_exec`. Scalasca [4] is a portable and scalable profiling and tracing system that can automate the detection of performance bottlenecks in message passing and shared memory programs. Like many other tools, it uses library wrapping for MPI. TAU may be configured to use Scalasca or VampirTrace internally. TAU, VampirTrace, and Scalasca use the PAPI [2] library to access hardware performance counters present on most modern processors. However, only the `tau_exec` scheme provides the level of integration of all sources of performance information – MPI, I/O, and memory – of interest to us, with the rich context provided by TAU. With this support, we can utilize the VampirServer [8] parallel trace visualization system to show the performance data through scalable timeline displays. Profile performance data can also be easily stored in the PerfDMF database [6]. TAU's profile browser, ParaProf, and its cross-experiment analysis and data-mining tool PerfExplorer [5] can interface with the performance database to help evaluate the scalability of an application.

4 Analysis of Prim's and Kruskal's MST Algorithms

This section provides time and space complexity analysis for a serial implementation of two MST algorithms in preparation for the discussion of a Kruskal's algorithm implementation that uses OpenSHMEM.

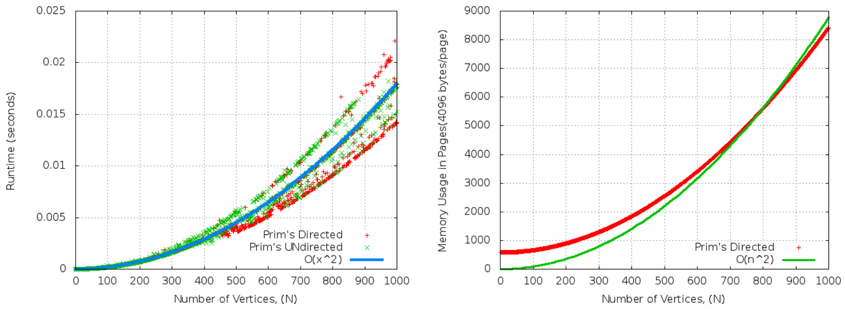
4.1 Prim's Algorithm

Prim's algorithm works by picking any vertex X to be the root of the MST. While the tree does not contain all vertices in the graph, find a minimally weighted edge leaving the tree and add it to the tree.

1. Choose any vertex X. Let S(set of vertices)=$\{X\}$ and A(set of edges)=$\emptyset$.
2. Find a minimally weighted edge such that one endpoint is in S and the other endpoint is in $(V - S)$. Add this edge to A and the other endpoint to S.
3. If $(V - S) = \emptyset$, then stop. $\{S, A\}$ is the Minimum Spanning Tree.
4. Else go to Step 1.

We implemented Prim's algorithm in C and generated a MST using random edge weights for 1 to 1000 vertices. Figure 3(a) shows the runtime of the serial Prim's implementation using both directed and undirected graphs. Only the number of vertices vary by case. These were run on an eight processor Intel Nehalem node that was not dedicated, thus others were using the system. We expect this is the cause of the fluctuation in times between the directed and undirected curves. The $O(n^2)$ time complexity of the algorithm is apparent, as shown by the blue line which was fit to the undirected data. We see that the directed graphs had the same $O(n^2)$ trend but ran slower overall by a constant.

The space complexity of Prim's Algorithm is shown in Figure 3(b). The data was fit to an $O(n^2)$ curve and we can see the actual space usage is between linear and quadratic, but not quite as low as $O(n \times log_2(n))$. The results for space usage for the directed and undirected graph experiments were identical so only the directed results are presented here.



(a) Time

(b) Space

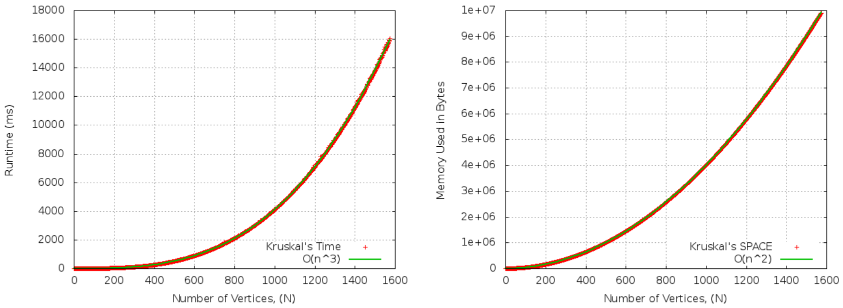
Fig. 3. Complexity of Prim's algorithm

4.2 Kruskal's Algorithm

Kruskal's algorithm is similar to Prim's but with a different search order:

1. Place every vertex in its own set.
2. Select edges in order of increasing weight.
3. As each edge is selected, determine if the vertices connected by that edge are in different sets. If so then insert the edge into the set that is the MST and take the union of sets containing each vertex.
4. Repeat steps 1-3 until all edges have been explored.

Figure 4(a) shows the Kruskal's algorithm runtime for an undirected graph under experimental conditions similar to those in the Prim's experiments. The graph representation is an adjacency matrix, so edge weights can be compared in constant time. At each iteration, the minimum edge can be located in $O(log_2(E))$ time, which is $O(log_2(V))$, since the graph is simple. The total running time is $O((V + E) \times log_2(V))$, which is $O(E \times log_2(V))$ since the graph is simple and connected. The data show a very tight $O(n \times log_2(n))$ fit, which was the theoretical expectation. The space complexity of Kruskal's algorithm is shown in Figure 4(b). The data fits almost perfectly to the theoretical $O(n^2)$ expectation.



(a) Time

(b) Space

Fig. 4. Complexity of Kruskal's algorithm

5 OpenSHMEM Performance Analysis

We executed our OpenSHMEM experiments on 256 nodes of the "Titan" Cray XK7 supercomputer and on a 48 core shared memory system at University of Maryland Baltimore County (UMBC), consisting of 8, 6 core AMD Opteron processors with 530GB of globally accessible shared memory. The application code was an OpenSHMEM implementation of Kruskal's algorithm as in Listing 1.1. In each experiment, we used TAU to gather data for time spent in code regions of interest (e.g. `broadcast_graph` and `calc_spanning_tree`), heap memory allocated and deallocated by routines like `malloc` and `free`, total bytes of memory used in all routines, network I/O events initiated by routines like `shmem_int_put`), and time spent waiting in barrier routines. We also enabled callpath profiling in TAU so that the full execution path to all events was recorded. We used source based instrumentation and the TAU compiler wrapper scripts to automatically introduce measurement hooks into the OpenSHMEM application code at compile time.

TAU uses the PSHMEM interface to support measurement of OpenSHMEM routines. For every routine in the OpenSHMEM standard, PSHMEM provides an analogous routine with a slightly different name. This allows profiling tools to intercept and measure OpenSHMEM calls made by a user's application by defining routines with the same function signatures as OpenSHMEM routines – "wrapper" functions – which call the appropriate PSHMEM routines. TAU provides an OpenSHMEM wrapper library which can be linked to any OpenSHMEM application to acquire runtime measurements of OpenSHMEM routines. The library can be used statically- or dynamically-linked applications. In our experiments the application was statically linked since statically linked executables are preferred on Titan.

Our fully instrumented application was approximately 4% slower than the uninstrumented application. If only time in code regions of interest were measured (no memory or I/O measurements are taken) then overhead was approximately 1.5%. Regardless of which events are recorded, TAU's overhead is approximately O(1) in the number of application processes, i.e. as the number of PEs increases the overhead incurred by TAU remains relatively constant. Large applications can benefit from reducing instrumentation to only regions of interest by use of a TAU select file. We used a TAU select file to insert timers at specific source lines in the OpenSHMEM application rather than fully instrument every routine.

Figure 5 shows exclusive time spent by PE 0 in regions of the code. The vast majority of the time (64.987 seconds) is spent in reading the graph from file and broadcasting the edge weights to the other PEs. The time spent calculating the minimum spanning tree is minimal, only 0.008 seconds. Figure 6 is the counterpart to Figure 5. It shows exclusive time spent by PE 1 in regions of the code. The profile on all PEs except for PE 0 is nearly identical. The majority of time is spent receiving data from PE 0, followed by a brief calculation to construct the MST.

If we exclude the file I/O data we see only those routines directly involved in the MST calculation. Figure 7 shows inclusive mean time spent by all PEs in routines that do not perform file I/O or wait for PE 0 to complete its file I/O operations. This figure shows that the initial broadcast of the graph weights via `shmem_int_put` is the most expensive step in the MST calculation.

Figures 5-7 show data taken from *interval events*, which have a defined start and stop point. Interval events are used to measure time spent in a code region. Events such as memory allocations are recorded as `atomic events`, which record a quantity (e.g. bytes allocated) when they are triggered. Figure 8 shows the mean of atomic events gathered during the 512 PE experiment on Titan. Memory allocation events and network sends and receives are visible. For example, across all 512 PEs there was an average of thirteen heap memory allocations in the `shmem_int_put` call in the `broadcast_graph` section of the application. TAU has also flagged two potential memory leaks caused by not explicitly deallocating memory before the program exists. These leaks are of no concern since the operating system will deallocate all heap memory on program exit. However, if this application were converted to a library then these leaks would need to be addressed.

Communication atomic events record the number of bytes sent or received between PEs. TAU can display this information as a communication heat map as in Figure 9. The communications matrix of the Titan supercomputer and the 48-node shared memory appliance are markedly different. In both cases, there is virtually no communication between PEs after PE 0 has distributed the graph data. On both systems, peak communication volume is visible in red in the zeroth row of the array and we see that no nonzero PE communicates with itself. However, each PE on the 48-core appliance sends eight bytes to every other PE while on Titan only PE 0 communicates with other PEs. From our callpath data we determined that these sends on the 48-core appliance were initiated by the `shmem_barrier_all` routine. This demonstrates TAU's ability to highlight implementation differences between libraries and explain unexpected communication patterns.

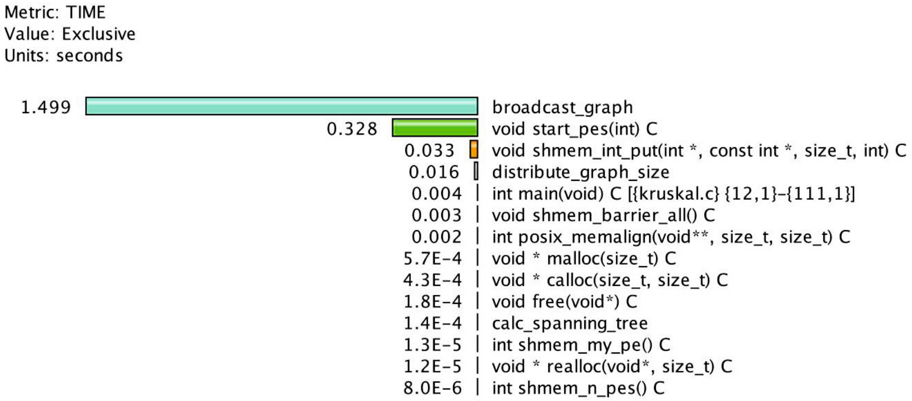


Fig. 5. Exclusive time on PE 0 of 512 on Titan, showing time spent reading the graph from file and broadcasting it to the other PEs

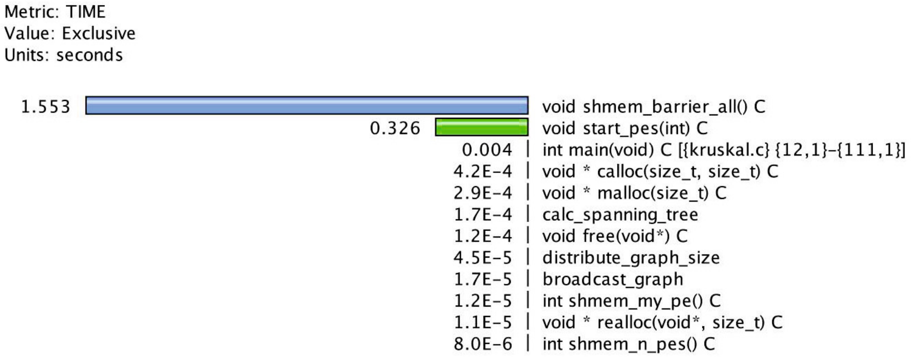


Fig. 6. Exclusive time on PE 1 of 512 on Titan, showing time spent waiting to receive the graph data from PE 0 and then calculating the MST

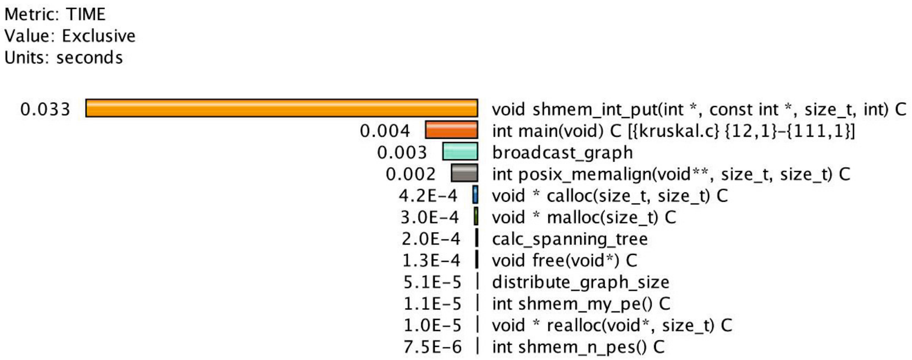
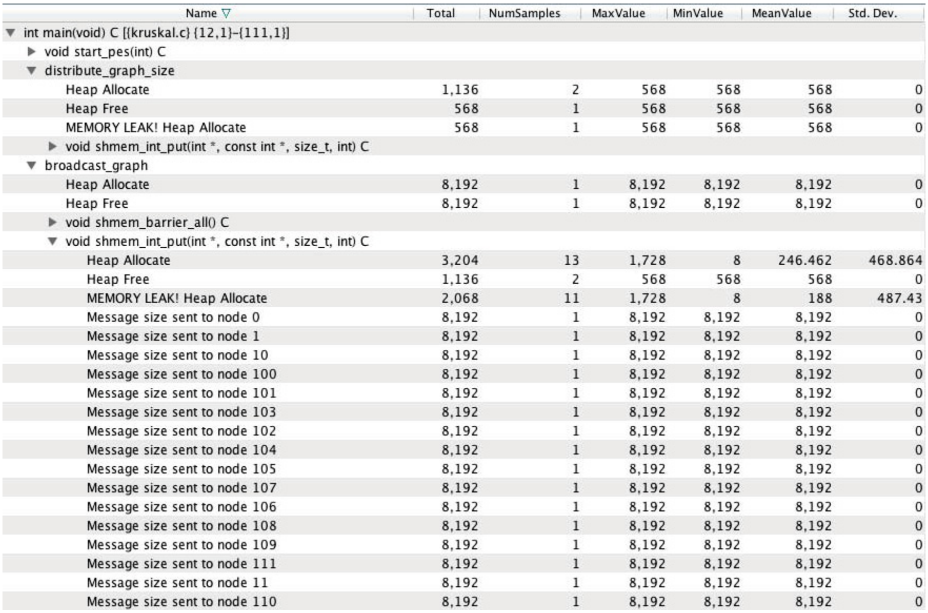


Fig. 7. Exclusive mean time on all 512 PEs on Titan in routines called while calculating the MST after the graph has been broadcast



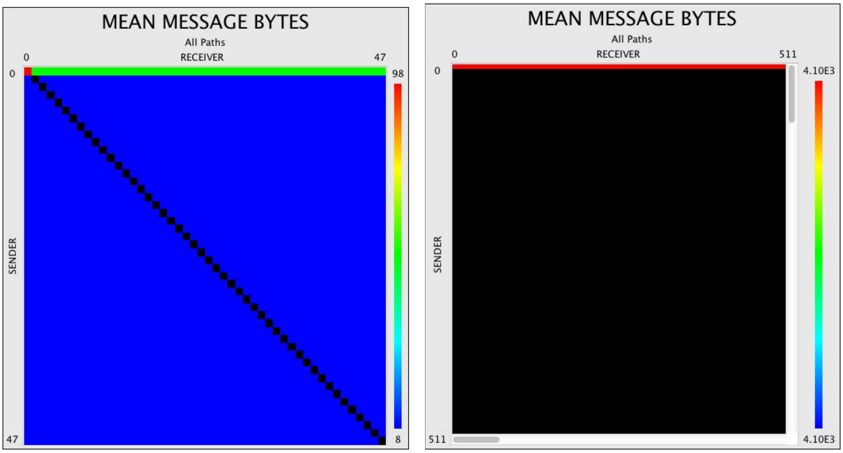
Name ▽	Total	NumSamples	MaxValue	MinValue	MeanValue	Std. Dev.
▼ int main(void) C [{kruskal.c} {12,1}-{111,1}]						
▶ void start_pes(int) C						
▼ distribute_graph_size						
Heap Allocate	1,136	2	568	568	568	0
Heap Free	568	1	568	568	568	0
MEMORY LEAK! Heap Allocate	568	1	568	568	568	0
▶ void shmem_int_put(int *, const int *, size_t, int) C						
▼ broadcast_graph						
Heap Allocate	8,192	1	8,192	8,192	8,192	0
Heap Free	8,192	1	8,192	8,192	8,192	0
▶ void shmem_barrier_all() C						
▼ void shmem_int_put(int *, const int *, size_t, int) C						
Heap Allocate	3,204	13	1,728	8	246.462	468.864
Heap Free	1,136	2	568	568	568	0
MEMORY LEAK! Heap Allocate	2,068	11	1,728	8	188	487.43
Message size sent to node 0	8,192	1	8,192	8,192	8,192	0
Message size sent to node 1	8,192	1	8,192	8,192	8,192	0
Message size sent to node 10	8,192	1	8,192	8,192	8,192	0
Message size sent to node 100	8,192	1	8,192	8,192	8,192	0
Message size sent to node 101	8,192	1	8,192	8,192	8,192	0
Message size sent to node 103	8,192	1	8,192	8,192	8,192	0
Message size sent to node 102	8,192	1	8,192	8,192	8,192	0
Message size sent to node 104	8,192	1	8,192	8,192	8,192	0
Message size sent to node 105	8,192	1	8,192	8,192	8,192	0
Message size sent to node 107	8,192	1	8,192	8,192	8,192	0
Message size sent to node 106	8,192	1	8,192	8,192	8,192	0
Message size sent to node 108	8,192	1	8,192	8,192	8,192	0
Message size sent to node 109	8,192	1	8,192	8,192	8,192	0
Message size sent to node 111	8,192	1	8,192	8,192	8,192	0
Message size sent to node 11	8,192	1	8,192	8,192	8,192	0
Message size sent to node 110	8,192	1	8,192	8,192	8,192	0

Fig. 8. The mean of context events observed in the OpenSHMEM implementation of Kruskal's algorithm on 512 PEs on Titan

TAU can construct a complete application callgraph from the callpath data as shown in Figure 10. Boxes are colored according to their exclusive time: more time is spent in red boxes than blue boxes. We note that three different memory allocation routines were used in this application, though heap memory allocation was only explicitly performed via `malloc` and `shmalloc`.

TAU can also perform application scaling studies. To demonstrate this on Titan, we varied the number of PEs as a power of two ranging from four to 512. Afer each run, we used TAU's profile browser (ParaProf) to package the application data as a packed profile file and then imported the packed profile into a TAUdb database [6]. We then used TAU's cross-experiment analysis and data-mining tool PerfExplorer [5] to explore the application scalability.

Figure 11 shows the runtime breakdown for varying PE counts on Titan. `shmem_barrier_all` is the most expensive routine, accounting for approximately 80% of the application runtime in all cases. The relative cost of broadcasting the graph decreases as the number of cores increases. The cost of computing the MST is small in all cases and is included in the "other" category accounting for approximately 3% of the application runtime.



(a) 48-core Shared Memory Appliance

(b) Titan

Fig. 9. Communication heat maps showing differences in the OpenSHMEM implementations on the 48-core appliance and Titan

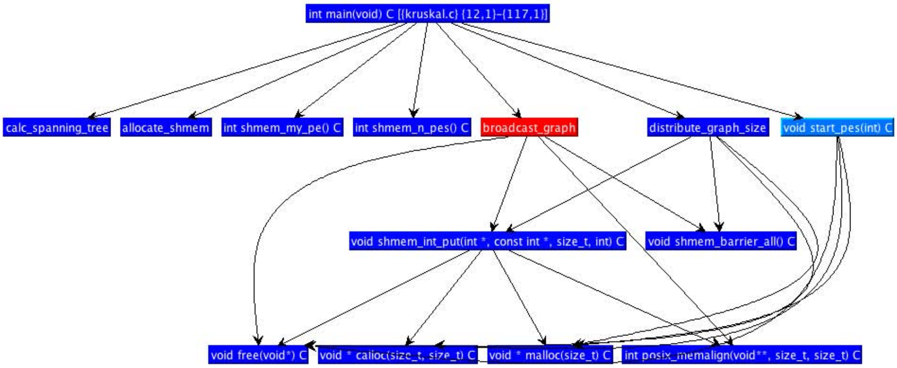

int main(void) C [{kruskal.c} {12,1}-{117,1}]
calc_spanning_tree
allocate_shmem
int shmem_my_pe() C
int shmem_n_pes() C
broadcast_graph
distribute_graph_size
void start_pes(int) C
void shmem_int_put(int *, const int *, size_t, int) C
void shmem_barrier_all() C
void free(void*) C
void * calloc(size_t, size_t) C
void * malloc(size_t) C
int posix_memalign(void**, size_t, size_t) C

Fig. 10. Kruskal's OpenSHMEM callgraph. Box color corresponds to exclusive time. Blue boxes are close to minimum and red boxes are close to maximum.

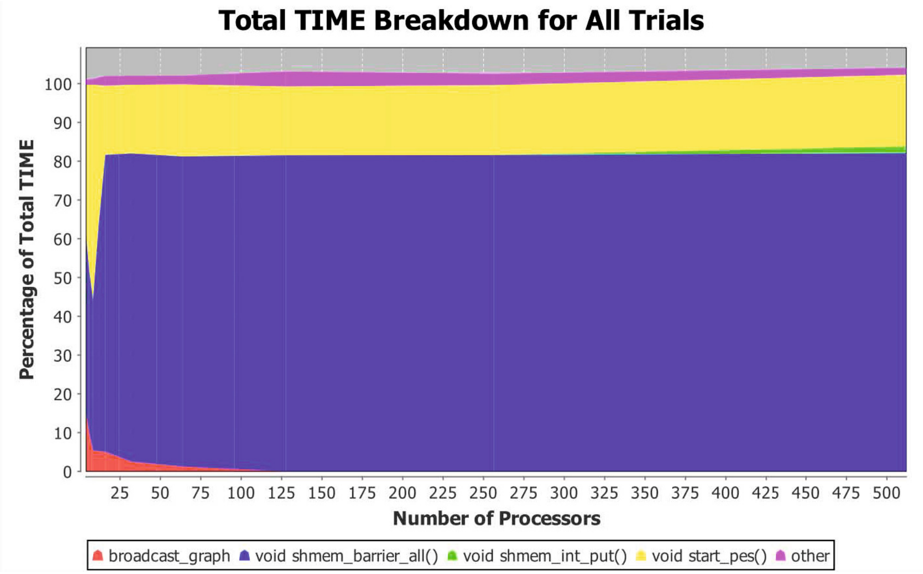


Fig. 11. Kruskal's OpenSHMEM callgraph. Box color corresponds to exclusive time. Blue boxes are close to minimum and red boxes are close to maximum.

6 Conclusions and Future Work

We have provided a performance analysis of both serial and parallel implementations of the standard minimum spanning tree (MST) algorithms from Prim and Kruskal. We have developed an efficient OpenSHMEM implementation of Kruskal's MST algorithm and provided a profile of that implementation on two HPC systems. We have demonstrated the portability of applications that use OpenSHMEM and the portability of the profiling features of the TAU Performance System. Our results suggest that OpenSHMEM is a flexible and powerful API for PGAS programming that can be applied effectively to non-numeric algorithms with a low FLOPS/byte ratio.

Profiling tools like TAU would benefit from further standardization and support for the PSHMEM interface. Unlike the PMPI interface which is fairly mature, complete, and widely available, the PSHMEM interface has not been completely established and in some implementations of OpenSHMEM is only partially implemented. This necessitates special checks when TAU compiles its OpenSHMEM wrapper library. TAU's maintainers will continue to improve TAU's resilience to variations in the PSHMEM interface until the interface is finalized.

TAU could also benefit from an interface which exposes synchronization of the symmetric heap. At present, TAU intercepts the underlying system allocation and deallocation calls and OpenSHMEM library calls to mark operations on the symmetric heap. However, it is difficult to observe in a trace when an update to the symmetric heap becomes visible to other PEs. TAU could make use of a mechanism for notifying a performance measurement system of symmetric heap updates when they occur to improve the quality of the application performance data.

Acknowledgments. Authors would like to thank The University of Oregon NeuroInformatics Center and the NSF Center for Hybrid Multicore Productivity Research at UMBC. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

1. Bader, D.A., Cong, G.: Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. J. Par. Distrib. Comp. 66(11), 1366–1378 (2006), http://dx.doi.org/10.1016/j.jpdc.2006.06.001
2. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. International Journal of High Performance Computing Applications 3(14), 189–204 (2000)
3. Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., Smith, L.: Introducing OpenSHMEM: SHMEM for the PGAS community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS 2010, pp. 2:1–2:3. ACM, New York (2010), http://doi.acm.org/10.1145/2020373.2020375
4. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: Scalable parallel trace-based performance analysis. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 303–312. Springer, Heidelberg (2006)
5. Huck, K., Malony, A.: PerfExplorer: A performance data mining framework for large-scale parallel computing. In: Proceedings of the ACM/IEEE Conference on Supercomputing, SC 2005 (2005)
6. Huck, K., Malony, A., Bell, R., Li, L., Morris, A.: PerfDMF: Design and implementation of a parallel performance data management framework. In: Proceedings of the International Conference on Parallel Processing. IEEE (2005)
7. Jose, J., Kandalla, K., Luo, M., Panda, D.: Supporting hybrid MPI and OpenSHMEM over InfiniBand: Design and performance evaluation. In: The 41st International Conference on Parallel Processing (ICPP), pp. 219–228 (2012)
8. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.E.: Introducing the open trace format (OTF). In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3992, pp. 526–533. Springer, Heidelberg (2006)
9. Knupfer, A., Brunst, H., Nagel, W.: High performance event trace visualization. In: Proceedings of Parallel and Distributed Processing (PDP). IEEE (2005)
10. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society 7 (1956)
11. Meuer, H., Strohmaier, E., Dongara, J., Simon, H.: TOP 500 Supercomputer Sites (2013), http://www.top500.org

12. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the Graph 500 (May 2010)
13. Papadimitriou, C.H.: The Euclidean traveling salesman problem is NP-complete. Theoretical Computer Science 4(3), 237–244 (1977)
14. Pophale, S., Nanjegowda, R., Curtis, T., Chapman, B., Jin, H., Poole, S., Kuehn, J.: OpenSHMEM performance and potential: A NPB experimental study. In: The 6th Conference on Partitioned Global Address Space Programming Models, PGAS 2012 (2012)
15. Prim, R.C.: Shortest connection networks and some generalizations. Bell System Technical Journal 36, 1389–1401 (1957)
16. Shende, S.S., Malony, A.D.: The TAU Parallel Performance System. Int. J. High Perform. Comput. Appl. 20(2), 287–311 (2006), http://dx.doi.org/10.1177/1094342006064482