Profiling Non-Numeric OpenSHMEM Applications with the TAU Performance System

John C. Linford, Tyler A. Simon, Sameer Shende, Allen D. Malony
{jlinford, tasimon, sameer, malony}@paratools.com
ParaTools, Inc.

Wednesday 5 March 2014

OpenSHMEM’13/14
Annapolis, MD

Slides Available At:
http://www.paratools.com/openshmem14
Summary

1. Non-Numerical Algorithms
2. Kruskal’s Algorithm for Minimum Spanning Tree
3. OpenSHMEM Implementation of Kruskal’s Algorithm
4. TAU Performance System support for OpenSHMEM
5. Performance Analysis of OpenSHMEM Kruskal’s
6. Conclusion and Acknowledgements
Non-Numerical Algorithms (NNA)

OpenSHMEM and Kruskals’s Algorithm for MST
Non-Numerical Algorithms (NNA)

- Low FLOPS/byte ratio.
- Majority of time spent in search or sort.
- Performance depends on **memory locality**.
- Challenging for Top500 HPC systems.

http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/
Searching or traversing graphs

- Connectivity and routing optimization,
- Network design,
- Highway system layout,
- Cluster points in space into natural groups,
- Approximate solutions to hard problems, e.g. TSP

Motif occurrence in yeast ➔

⇐ Network connectivity


Mazurie et al. Genome Biology 2005 6:R35
doi:10.1186/gb-2005-6-4-r35
Minimum Spanning Trees

• A subgraph that connects all the vertices and is a tree.
• **Kruskal’s Algorithm**: choose globally optimal edge (BFS)
• Prim’s Algorithm: choose locally optimal edge (DFS)
Kruskal’s Algorithm

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.

In Short:
Pick least-weighted edges until the MST contains all nodes.
Complexity of Kruskal’s Algorithm

Time Complexity $O(n \lg n)$

Space Complexity $O(n^2)$
OpenSHMEM Kruskal’s Algorithm

// Allocate symmetric heap for graph
graph = (int*)shmalloc(graphSize*nNodes*sizeof(int));
span = (int*)shmalloc(graphSize*nNodes*sizeof(int));

// Load and distribute graph from PE_0
... shmem_barrier_all(); ...

// All PEs pick their lowest edges and store to span[]
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[] contains the minimum spanning tree
Performance Analysis of Kruskal’s Algorithm
TAU Performance System®
http://tau.uoregon.edu/

• Tuning and Analysis Utilities (18+ year project)
• Comprehensive performance profiling and tracing
  • Integrated, scalable, flexible, portable
  • Targets all parallel programming/execution paradigms

• Integrated performance toolkit
  • Instrumentation, measurement, analysis, visualization
  • Widely-ported performance profiling / tracing system
  • Performance data management and data mining
  • Open source (BSD-style license)

• Supports OpenSHMEM (and other SHMEM implementations)
TAU Support for OpenSHMEM

- Source- and compiler-based instrumentation, library interposition, and sampling.
- Track memory allocations in the symmetric heap and in PE local memory.
- Track network I/O caused by transfers to and from symmetric memory.
- Track OpenSHMEM API calls.

Thanks to the OpenSHMEM standard, TAU maintains a single implementation of the SHMEM API wrapper library.
Performance on ORNL’s Titan

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

1.499

0.328

broadcast_graph

void start_pes(int) C

0.033

void shmem_int_put(int *, const int *, size_t, int) C

0.016

distribute_graph_size

0.004

int main(void) C {{kruskal.c} \{12,1\}--\{111,1\}}

0.003

void shmemp_barrier_all() C

0.002

int posix_memalign(void**, size_t, size_t) C

5.7E-4

void * malloc(size_t) C

4.3E-4

void * calloc(size_t, size_t) C

1.8E-4

void free(void*) C

1.4E-4

calc_spanning_tree

1.3E-5

int shmemp_my_pe() C

1.2E-5

void * realloc(void*, size_t) C

8.0E-6

int shmemp_n_pes() C

1.5 seconds are spent reading the graph from file and broadcasting the edge weights to the other PEs. The time spent calculating the minimum spanning tree is only 140 microseconds.
Performance on ORNL’s Titan

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

```
1.553

0.326

void shmem_barrier_all() C
void start_pes(int) C
0.004 | int main(void) C [{kruskal.c} {12,1}–{111,1}]
4.2E-4 | void * calloc(size_t, size_t) C
2.9E-4 | void * malloc(size_t) C
1.7E-4 | calc_spanning_tree
1.2E-4 | void free(void*) C
4.5E-5 | distribute_graph_size
1.7E-5 | broadcast_graph
1.2E-5 | int shmem_my_pe() C
1.1E-5 | void * realloc(void*, size_t) C
8.0E-6 | int shmem_n_pes() C
```
Performance on ORNL’s Titan

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

0.033 | void shmem_int_put(int *, const int *, size_t, int) C
0.004 | int main(void) C [{kruskal.c} {12,1}--{111,1}]
0.003 | broadcast_graph
0.002 | int posix_memalign(void**, size_t, size_t) C
4.2E-4 | void * calloc(size_t, size_t) C
3.0E-4 | void * malloc(size_t) C
2.0E-4 | calc_spanning_tree
1.3E-4 | void free(void*) C
5.1E-5 | distribute_graph_size
1.1E-5 | int shmem_my_pe() C
1.0E-5 | void * realloc(void*, size_t) C
7.5E-6 | int shmem_n_pes() C
Performance on ORNL’s Titan

All non-zero PEs have virtually identical performance profiles.
Performance on ORNL’s Titan

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Total</th>
<th>NumSamples</th>
<th>MaxValue</th>
<th>MinValue</th>
<th>MeanValue</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>int main(void) C [[kruskal.c] [12,1]--[111,1]]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>void start_pes(int C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>distribute_graph_size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Allocate</td>
<td>1,136</td>
<td>2</td>
<td>568</td>
<td>568</td>
<td>568</td>
<td>0</td>
</tr>
<tr>
<td>Heap Free</td>
<td>568</td>
<td>1</td>
<td>568</td>
<td>568</td>
<td>568</td>
<td>0</td>
</tr>
<tr>
<td>MEMORY LEAK! Heap Allocate</td>
<td>568</td>
<td>1</td>
<td>568</td>
<td>568</td>
<td>568</td>
<td>0</td>
</tr>
<tr>
<td>void shmem_int_put(int *, const int *, size_t, int) C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>broadcast_graph</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Allocate</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Heap Free</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>void shmem_barrier_all() C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>void shmem_int_put(int *, const int *, size_t, int) C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Allocate</td>
<td>3,204</td>
<td>13</td>
<td>1,728</td>
<td>8</td>
<td>246,462</td>
<td>468,864</td>
</tr>
<tr>
<td>Heap Free</td>
<td>1,136</td>
<td>2</td>
<td>568</td>
<td>568</td>
<td>568</td>
<td>0</td>
</tr>
<tr>
<td>MEMORY LEAK! Heap Allocate</td>
<td>2,068</td>
<td>11</td>
<td>1,728</td>
<td>8</td>
<td>188</td>
<td>487,43</td>
</tr>
<tr>
<td>Message size sent to node 0</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 1</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 10</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 100</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 101</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 103</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 102</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 104</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 105</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 107</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 106</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 108</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 109</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 111</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 11</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
<tr>
<td>Message size sent to node 110</td>
<td>8,192</td>
<td>1</td>
<td>8,192</td>
<td>8,192</td>
<td>8,192</td>
<td>0</td>
</tr>
</tbody>
</table>
PerfExplorer: Scalability of Kruskal’s

Use **PerfExplorer** (not ParaProf) for multi-profile analysis.

Value Chart: **TIME**

- **GRAPH_SIZE**
  - 0.0
  - 0.5
  - 1.0
  - 1.5
  - 2.0
  - 2.5
  - 3.0
  - 3.5
  - 4.0
  - 4.5

- **Mean TIME - milliseconds**
  - 0.0
  - 0.5
  - 1.0
  - 1.5
  - 2.0
  - 2.5
  - 3.0
  - 3.5
  - 4.0

**calc_spanning_tree**
With $O(n \log n)$

OpenSHMEM Kruskal's

Graph Size - nodes

Mean TIME - milliseconds

$O(n \log n)$

TAU
PerfExplorer: Scalability of OpenSHMEM

Use PerfExplorer (not ParaProf) for multi-profile analysis.
Portability: Performance on UMBC’s 48-Core Shared Memory Appliance

Profiles were essentially the same as on Titan except for the comm. matrix
Performance Analysis Overhead

- 4% fully instrumented, a.k.a “pulling out all stops.”
- 1.5% if only time in code regions of interest were measured.
  - No memory or I/O measurements.
  - No callpath information in profile.
- TAU’s overhead is $O(1)$ in the number of PEs.
- Large applications can benefit from reducing instrumentation to only regions of interest by use of a TAU select file.
Conclusion

- OpenSHMEM’s symmetric heap facilitates an effective implementation of Kruskal’s Algorithm for MST.
- The TAU Performance System can be used to explore the performance of any OpenSHMEM application with low overhead (1.5-4%)
- OpenSHMEM facilitates portability: our code ran well without modification on two radically different systems.
- Tools like TAU could benefit from further standardization and support for the PSHMEM interface.
Acknowledgements

The University of Oregon NeuroInformatics Center (NIC).

The National Science Foundation Center for Hybrid Multicore Productivity Research at UMBC (CHMPR).

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.