Comparison of Performance Analysis Tools for Parallel Programs Applied to CombBLAS

REU Site: Interdisciplinary Program in High Performance Computing

Wesley Collins[1], Daniel T. Martinez[1], Michael Monaghan[2], Alexey A. Munishkin[3],
Graduate assistants: Ari Rapkin Blenkhorn[1], Jonathan S. Graf[4], Samuel Khuvis[4],
Faculty mentor: Matthias K. Gobbert[4], Client: John C. Linford[5]

[1]Department of Computer Science and Electrical Engineering, UMBC,
[2]College of Earth and Mineral Sciences, The Pennsylvania State University,
[3]Jack Baskin School of Engineering, Department of Computer Engineering, UCSC,
[4]Department of Mathematics and Statistics, UMBC, [5]ParaTools, Inc.

Technical Report HPCF–2015–28, hpcf.umbc.edu > Publications

Abstract

Performance analysis tools are powerful tools for high performance computing. By breaking down a program into how long the CPUs are taking on each process (profiling) or showing when events take place on a timeline over the course of running a program (tracing), a performance analysis tool can tell the programmer exactly, where the computer is running slowly. With this information, the programmer can focus on these performance "hotspots," and the code can be optimized to run faster. We compared the performance analysis tools TAU, ParaTools ThreadSpotter, Intel VTune, Scalasca, HPCToolkit, and Score-P to the example code CombBLAS (combinatorial BLAS) which is a C++ implemenation of the GraphBLAS, a set of graph algorithms using BLAS (Basic Linear Algebra Subroutines). Using these performance analysis tools on CombBLAS, we found three major "hotspots" and attempted to improve the code. We were unsuccessful in improving these "hotspots" due to a time limitation but still gave suggestions on improving the OpenMP calls in the CombBLAS code.

Key words TAU, ThreadSpotter, Intel VTune, Scalasca, HPCToolkit, Score-P.

AMS subject classifications (2010) 65-04, 65Y05, 65Y10, 65Y15.

1 Introduction

Performance analysis tools, or PATs for short, can be powerful tools for improving high performance computing code. High performance computing applications vary from simulating fast and accurate simulations of climate patterns to interpreting large data sets relating to terrorist activities which must be solved in a limited amount of time. These performance analysis tools help the programmer to focus on the parts of the code that cause the overall program to run slowly and fix them.

When a program is running slowly it may be difficult to tell where the bottleneck is located in the code, in particular if the code is written in parallel. These performance analysis tools help by breaking the program's code down to how long each CPU takes to run each process (profiling) or show when events take place on a timeline over the course of running

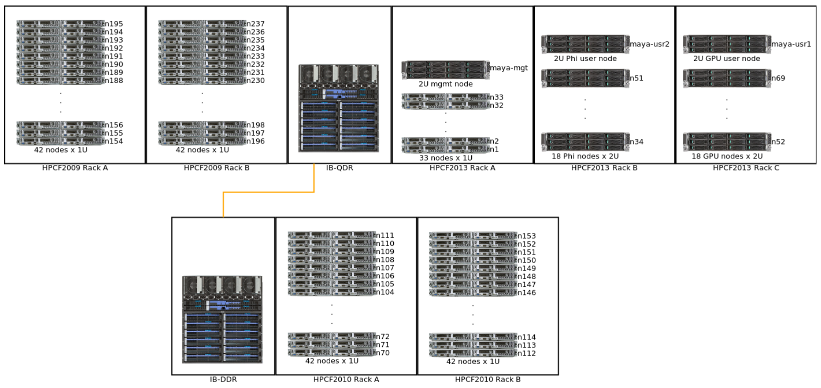


Figure 1.1: Schematics of maya.

a program (tracing). A performance analysis tool can tell the programmer exactly where the computer is running slowly and possibly give suggestions to the programmer to what possible improvements can be made to the code. With this information, the programmer can thus focus on these performance “hotspots” and make the code to be optimized so as to make the program run faster.

In our study we compare the performance analysis tools TAU, ParaTools ThreadSpotter, Intel VTune, Scalasca, HPCToolkit, and Score-P. We then applied each PAT to our test case: CombBLAS (Combinatorial BLAS), a C++ algorithm in the GraphBLAS set of graph algorithms using BLAS (Basic Linear Algebra Subroutines). Using the results given from each PAT, we attempted to improve the implementation of CombBLAS and show performance data obtained on the distributed-memory cluster maya in the UMBC High Performance Computing Facility.

The parallel code was run on the 2013 portion of the cluster maya (see Figure 1.1). There are 72 nodes of which 67 are used for computing. Figure 1.2 shows a schematic of one of the compute nodes that contains two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs. The nodes in maya 2013 are connected by a quad-data rate InfiniBand interconnect.

The remainder of this report is organized as follows: Section 2 introduces the test case that we use with the various performance analysis tools. It describes the high performance project code and what the application of the project is, i.e., what the code is intended to solve. Section 3 presents our experiences with using the various performance analysis tools, with a subsection for each PAT. Using the results from each PAT, we improved the performance of the test code which is reported in Section 4. Finally, Section 5 summarizes the results of using the various PATs and compares them based on our experience using these PATs, such as describing which PATs are the best in terms of giving accurate suggestions for improving the performance of the high performance code CombBLAS, how simple it is to install the PAT, and how easy it is to read and understand the results given from each PAT.

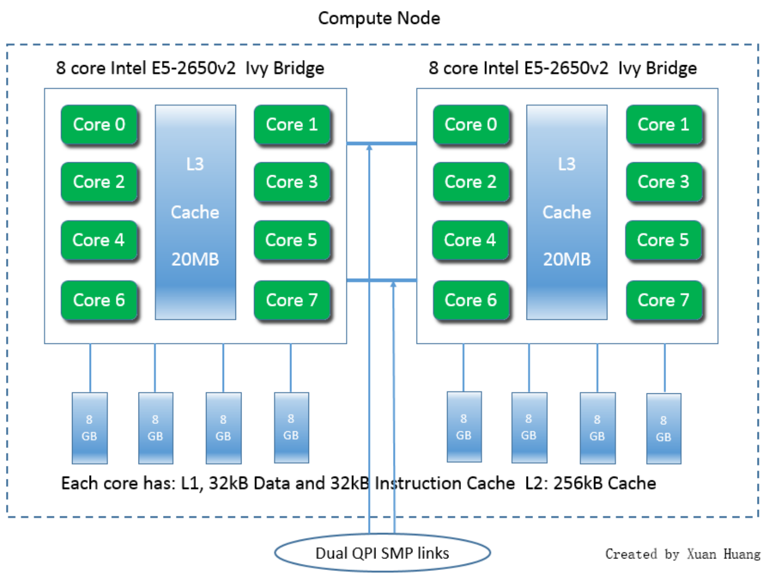


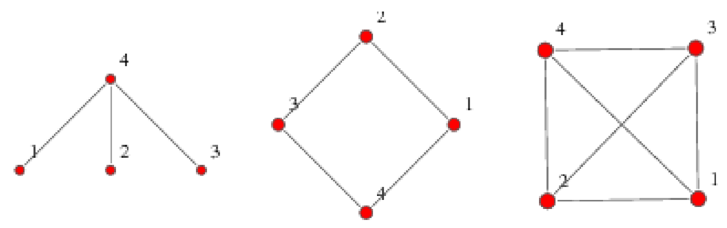
Figure 1.2: Schematic of a maya 2013 node.

2 Test Case

2.1 Graphs and Breadth First Search Algorithm

A graph is a representation of a set of objects (vertices) where these objects can be connected by links (edges), which are either directed or undirected.

A graph can be represented as a two-dimensional array or matrix. This matrix is called an adjacency matrix, meaning that each element in the matrix represents a link from going from row vertex to column vertex, i.e., if there is a 1 in the element associated with row 2 and column 3 then there is a link between vertex 2 to vertex 3. Figure 2.1 displays three examples of undirected graphs and below are their respective adjacency matrices.



$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Figure 2.1: Examples of basic graphs and their respective adjacency matrices [5].

The Breadth First Search (BFS) Algorithm is commonly used for finding the shortest path between any two vertices. The algorithm starts with one target vertex, and then moves outward to the other vertices that share an edge with it. This process continues with the previously visited vertices ignored when attempting to look at the vertex that shares an edge with the current target vertex. A counter is used to keep track of how many different vertices had to be traversed before the target vertex was found. The shortest path between any two vertices is the smallest amount of transverses or movements across an edge starting at one vertex and ending at another vertex. The algorithm is pictorially displayed in Figure 2.2. The BFS algorithm is run on the example graph starting at vertex s and ends when all vertices have been visited.

Figure 2.3 contains the full pseudocode description of the common BFS algorithm. The algorithm is as follows: pick any vertex in the graph G, call it s. Then for the rest of the vertices in the graph G, you initialize them to default values of having not been visited, `color=WHITE`, setting distance from s as unknown, `d=`∞, and setting parent, i.e., from what vertex did the algorithm go from to reach the current vertex, to no parent, π`=0`. Then set s to being currently processed, `color=GRAY`, distance to s is clearly 0, and there is no parent since s was the first vertex picked in the graph G. Then add s to a queue, a data-structure to hold the current vertices being processed. Then remove from the queue and call the removed vertex u. Then go though each neighbor of u, call the current one being processed v, and check if it already being processed or finished processed by the algorithm, `color=BLACK`. If the vertex v is not being processed or not finished processed then change the vertex v to being processed, `color=GRAY`, increment its distance, and set its parent to u. Finally add v to the queue after completing the update described above, and repeat with the next neighbor of u until all neighbors have be finished, and the queue becomes empty, i.e., no more vertices to process.

2.2 GraphBLAS: The Algorithms

GraphBLAS is a set of graph algorithms written in terms of mathematical linear algebra problems. This means that it is a document explaining how to break down graph algorithms into linear algebra equations. It is written this way to maximize the performance of an implementation of GraphBLAS such as our test case, CombBLAS.

2.3 CombBLAS: The Implementation in C++

CombBLAS is the implementation of GraphBLAS, which is written in the C++ programming language and includes parallel code that is implemented using OpenMP and MPI. The full documentation of the project is found at [2], and the design paper which describes the implementation of CombBLAS is found at [1].

We will be running an application example that uses CombBLAS. The example is a Filtered Breath First Search applied to a Twitter-like database, meaning that the vertices or objects are people and links or edges between various people are their connections: similar likes and favorites.

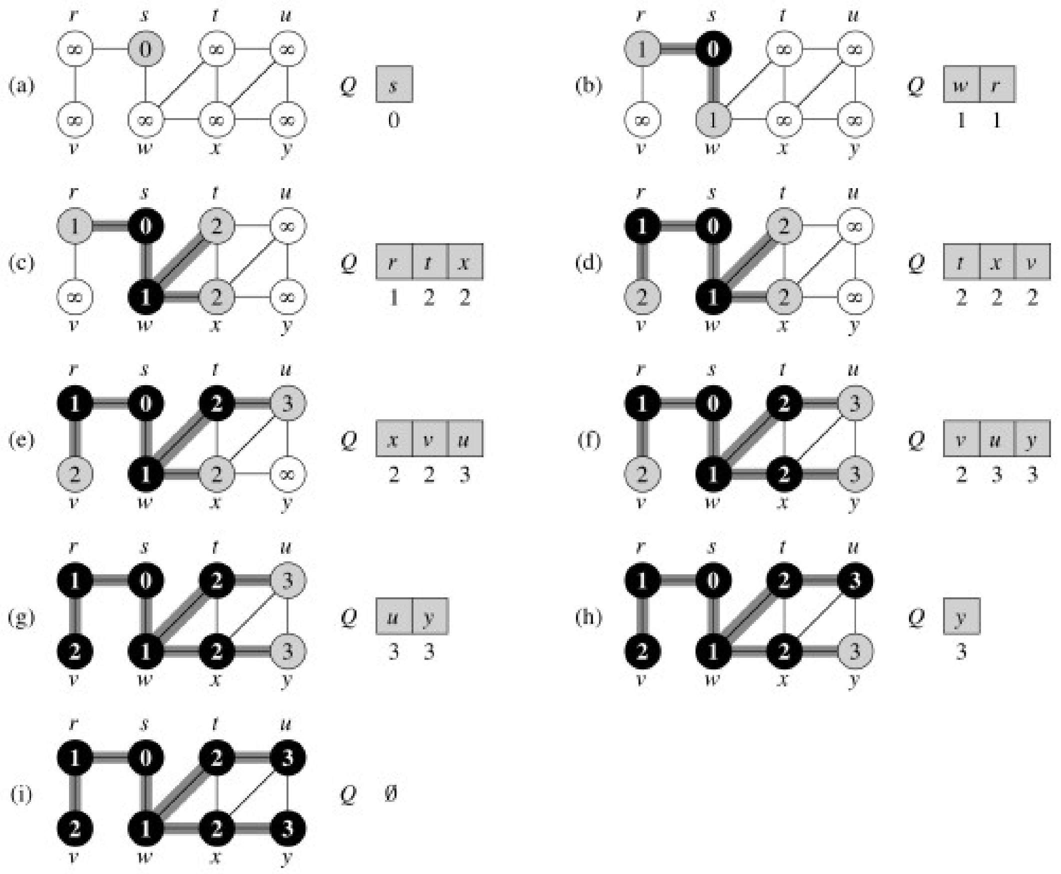
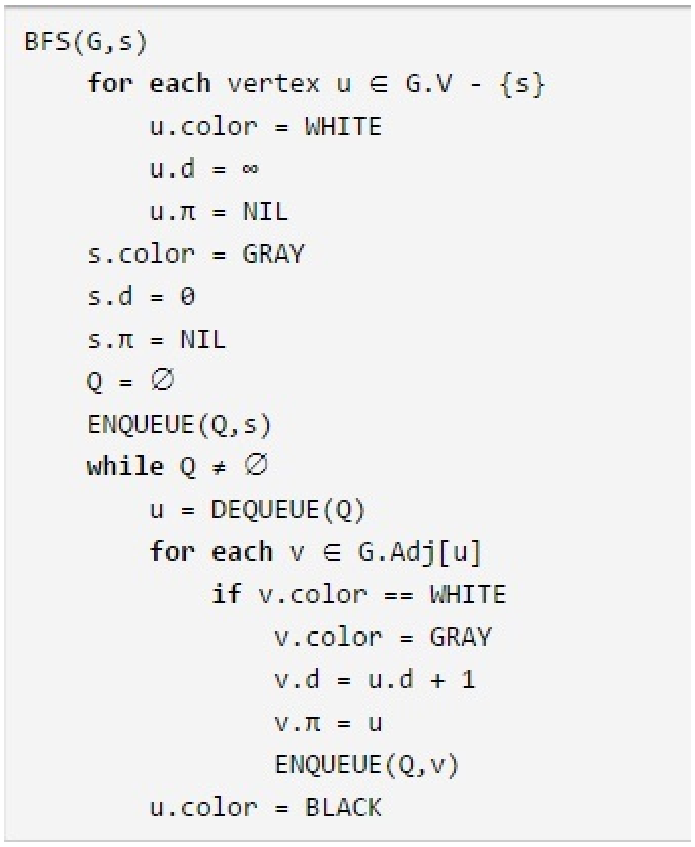


Figure 2.2: Pictorial example of BFS algorithm [3].

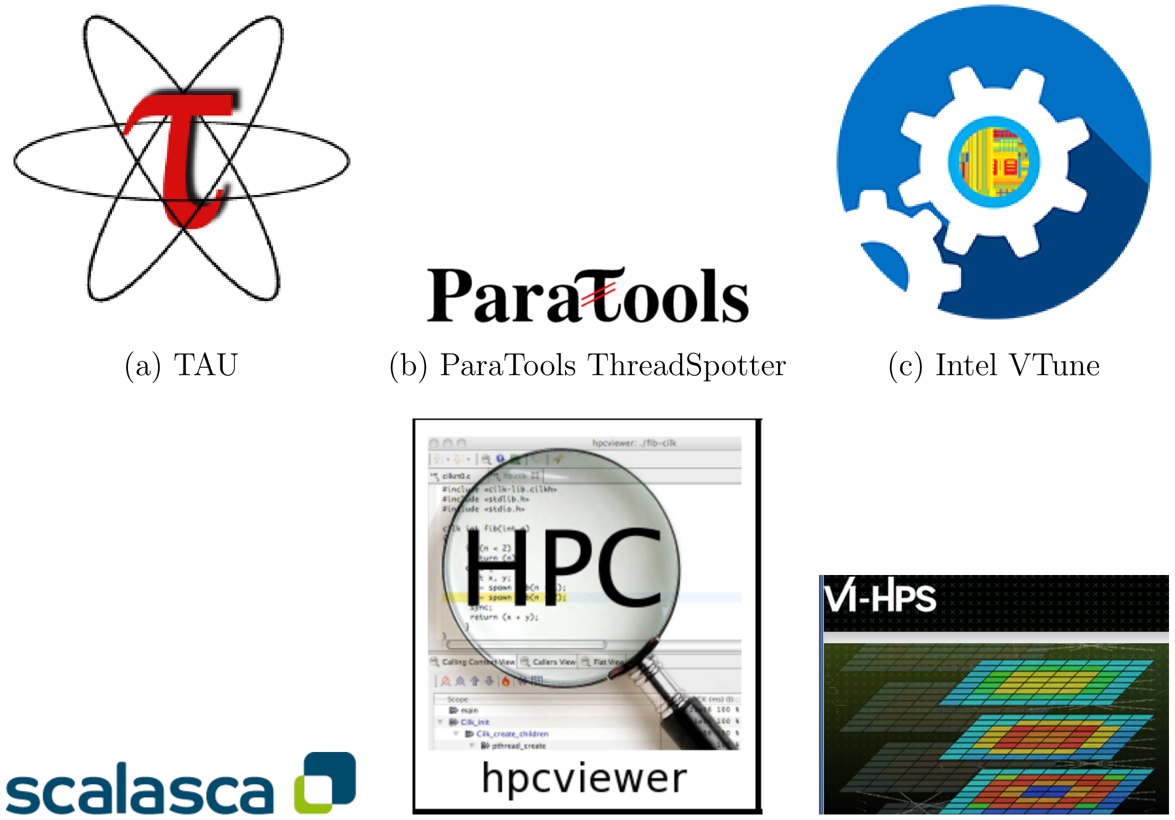


```
BFS(G,s)
    for each vertex u ∈ G.V - {s}
        u.color = WHITE
        u.d = ∞
        u.π = NIL
    s.color = GRAY
    s.d = 0
    s.π = NIL
    Q = ∅
    ENQUEUE(Q,s)
    while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ G.Adj[u]
            if v.color == WHITE
                v.color = GRAY
                v.d = u.d + 1
                v.π = u
                ENQUEUE(Q,v)
        u.color = BLACK
```

Figure 2.3: Pseudocode of BFS algorithm [3].

3 High Performance Analysis Tools

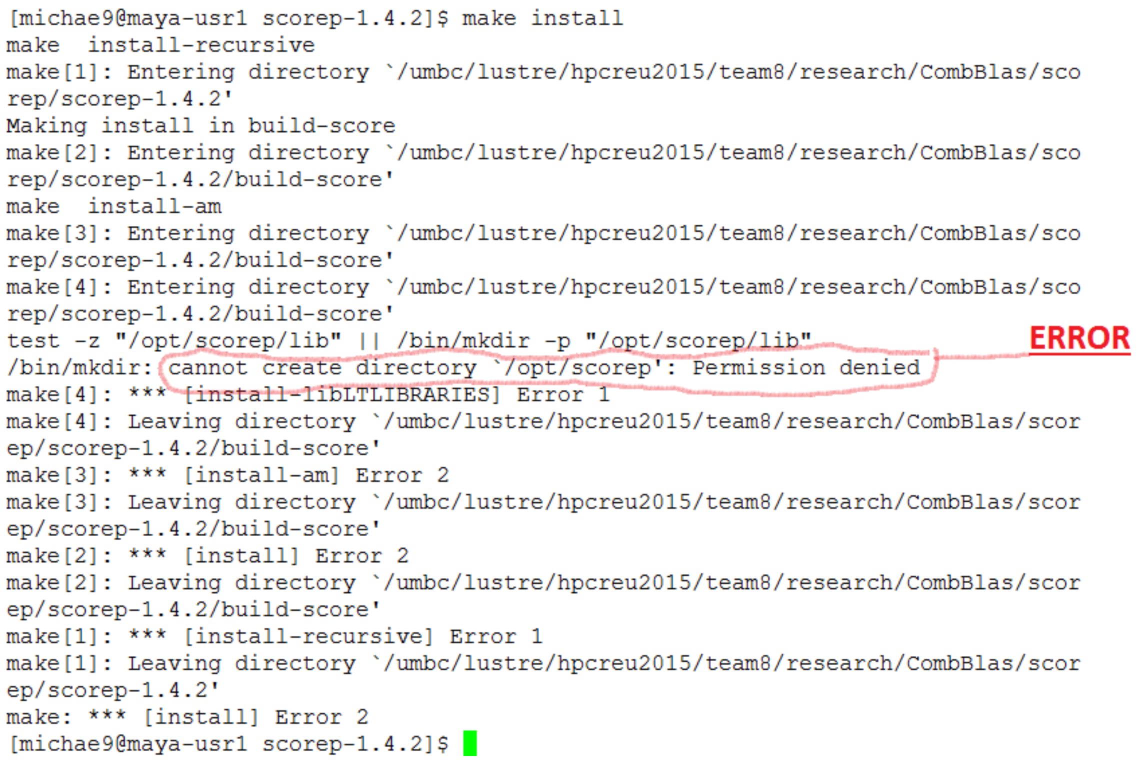
High performance analyzers are used to improve the quality of a computer program. They do this by identifying "hotspots" where the computer program takes a long time to run and compute necessary computations such as matrix-vector and matrix-matrix products. The first type of performance measurement is profiling where how long each part of the code is shown and how long they take. The other is tracing capability, which shows the order of events on a timeline. Also, it is possible to perform data measurements directly or indirectly. Directly would be performed with probes and includes calls into the code. This is done to get an exact measurement. The other method which is indirectly is performed using sampling and would not include any sort of code modification. The high performance analyzer tools (PATs) that we will be testing are TAU, ParaTools ThreadSpotter, Intel VTune, Scalasca, HPCToolkit, and Score-P. The icons for these tools are displayed in Figure 3.1. These icon images are displayed to alert the reader to the identity of these high performance analysis tools when searching these PATs on the internet for further information.



(a) TAU (b) ParaTools ThreadSpotter (c) Intel VTune

(d) Scalasca (e) HPCToolkit (f) Score-P

Figure 3.1: Overview of the six different PATs.



```
[michae9@maya-usr1 scorep-1.4.2]$ make install
make  install-recursive
make[1]: Entering directory `/umbc/lustre/hpcreu2015/team8/research/CombBlas/sco
rep/scorep-1.4.2'
Making install in build-score
make[2]: Entering directory `/umbc/lustre/hpcreu2015/team8/research/CombBlas/sco
rep/scorep-1.4.2/build-score'
make  install-am
make[3]: Entering directory `/umbc/lustre/hpcreu2015/team8/research/CombBlas/sco
rep/scorep-1.4.2/build-score'
make[4]: Entering directory `/umbc/lustre/hpcreu2015/team8/research/CombBlas/sco
rep/scorep-1.4.2/build-score'
test -z "/opt/scorep/lib" || /bin/mkdir -p "/opt/scorep/lib"
/bin/mkdir: cannot create directory `/opt/scorep': Permission denied
make[4]: *** [install-libLTLIBRARIES] Error 1
make[4]: Leaving directory `/umbc/lustre/hpcreu2015/team8/research/CombBlas/scor
ep/scorep-1.4.2/build-score'
make[3]: *** [install-am] Error 2
make[3]: Leaving directory `/umbc/lustre/hpcreu2015/team8/research/CombBlas/scor
ep/scorep-1.4.2/build-score'
make[2]: *** [install] Error 2
make[2]: Leaving directory `/umbc/lustre/hpcreu2015/team8/research/CombBlas/scor
ep/scorep-1.4.2/build-score'
make[1]: *** [install-recursive] Error 1
make[1]: Leaving directory `/umbc/lustre/hpcreu2015/team8/research/CombBlas/scor
ep/scorep-1.4.2'
make: *** [install] Error 2
[michae9@maya-usr1 scorep-1.4.2]$
```

Figure 3.2: Score-P install error.

3.1 Score-P

Score-P is a tool that was developed by Virtual Institute High Productivity Supercomputing. It is a project developed from the German BMBF project SILC and the US DOE project PRIMA [4].

Score-P is an open source software package provided under the BSD 3-Clause License and is provided free of charge. There are support plans available for this tool, which is located at support@score-p.org.

The installation went very smoothly as they have a tutorial of how to perform the installation:

```
>wget http://www.vi-hps.org/upload/packages/scorep/scorep-1.4.2.tar.gz
```

And then after downloading it, we tried to install it:

```
>tar xvf scorep-1.4.2.tar.gz
>cd scorep-1.4.2
>./configure --prefix=$PWD/../../Opt/score-p
>make
>make install
```

Despite the simplicity of the installation we were unable to install the tool in our team directory. Figure 3.2 displays the error we got when we were in the process of installing the tool, which was due to issues with the Lustre file system on maya. This resulted in the installation of the Score-P to be done on another part of maya. This is another reason for the failure of installation of Scalasca, since it depends on a working installation of Score-P. This is because during the installation process an error occurs that claims the user does not have permission, even though it is a free software. See Section 3.2 for more information about the errors in installation.

3.2 Scalasca

Scalasca is a German performance analysis tool used to evaluate the performance of programs run in parallel. It is a collaborative effort between the German Research School for Simulation Sciences GmbH, RWTH Aachen University, the University of Tennessee, and Technische Universitat Dresden. Scalasca is run in conjunction with Score-P allowing for more in depth analysis of a computer's performance while running a program and another program called cube to allow for the visualization.

Installation of the PAT was not difficult, but instrumentation of Scalasca was far less simple than that of the other tools. The installation itself was simple, beginning with untarring the file with the following command. Then running the configure script and compiling completed the installation.

```
>tar xzf scalasca-2.2.2.tar.gz
>cd scalasca-2.2.2
>./configure --prefix=$PWD/../../Opt/scalasca
>make
>make install
```

The problem comes with the instrumentation of Scalasca. The tool is dependent on Score-P even though the installation guide states the additional software as "unnecessary but highly recommended". However, when one tries to run a Scalasca command, an error is produced because the software cannot find Score-P on the path. Then more problems arise when the user tries to instrument Score-P in the programs that one is trying to execute. The command that is given by the user guide for automatic instrumentation is:

```
>scorep make
```

This command never worked, stating there were no input files for instrumentation. All instrumentation on the maya cluster had to be manually done, which calls for the editing of the makefile in the directory of the executable. For simple makefiles this is not a difficult task as the user can go about changing the makefile one of three ways:

```
One simply changes
CC := mpiicc
    or
CXX := mpiicpc
    to
CC := scorep mpiicc
    or
CXX := scorep mpiicpc
```

Or the second method is to add a variable before the compiler:

```
CC := mpiicc
   or
CXX := mpiicpc
   to
CC := $(PREP) mpiicc
   or
CXX := $(PREP) mpiicpc

Then exiting the code for the program and entering the command:
>make PREP=scorep
```

The third method is if the user already knows what the makefile is using as a compiler then they can use the command:

```
>make CC="scorep mpiicc"
      or
>make CXX="scorep mpiicpc"
```

Similar to the second method above. When the user does this they can then run the Scalasca analyzer with the following command and parameters:

```
>scalasca -analyze [options] [<launch_cmd> [<launch_flags>]] <target>\
 [target args]
```

So for example if someone wanted to run the executable "power" on two nodes, they would use the command:

```
>scalasca -analyze srun -n 2 power
```

The other arguments for the code then follow the name of the program. When the user knows the compiler, the instrumentation of Score-P is not elementary, but it is not difficult either. The difficulty arises when the user cannot locate the compiler that the makefile is using or if the compiler is in multiple directories. For example, when one uses cmake to create the make files for CombBLAS, it creates a very complicated makefile. One would expect the above processes to work, however, the makefiles that are created do not have "cc", "CC", "CXX", "mpiicc", or "mpiicpc". Those commands are indicators of a compiler, yet they are not located anywhere in the CombBLAS makefiles; the comments in the makefile do not point to any compiler. The user guide does not give any guidance on this specific issue, and even other methods found on the internet rely on there being some indication of a compiler in the makefile.

When the compiler is relatively easy to find in the makefile, and the above commands are followed, the software will create a Score-P log, configuration, and a cube file, which brings up another problem with Scalasca. Without installing cube on the maya cluster or on one's own laptop, it is impossible to visualize the code. The alternative is to get an output with a plethora of numbers that the Scalasca user guide does not explain because it assumes the user installed the Cube software. If the user did install cube, the interface is very simple, it asks what file you would want evaluated, and ends in extension .cubex. However, after two weeks of trying to get the Score-P instrumentation to work with cmake in the CombBLAS code, we abandoned Scalasca.

3.3 ParaTools ThreadSpotter

ThreadSpotter is a performance analysis tool originally created by RogueWave. It was opensourced in 2014, and ParaTools is now the primary developer and custodian of the project. ThreadSpotter is used to evaluate the performance of programs that are to be run with multiple threads. It is a free software and is fairly easy to install.

We downloaded two files from the ParaTools website, one that contained the source code for ThreadSpotter (ThreadSpotter-1.0), the other that contained the required third-party libraries (ext_library). We then placed the ext_library file inside of the ThreadSpotter-1.0 file. Then we used the following commands:

```
>./configure --prefix=$PWD/../../Opt/threadspotter
>make
>make install
```

When the make commands were used, we originally encountered an error in our team directory due to an issue with the Lustre file system that we were using. We were able to fix this problem by installing ThreadSpotter in a location that used a different file system. Running ThreadSpotter is also very easy. To run ThreadSpotter, we created a slurm script to be run on the cluster maya.

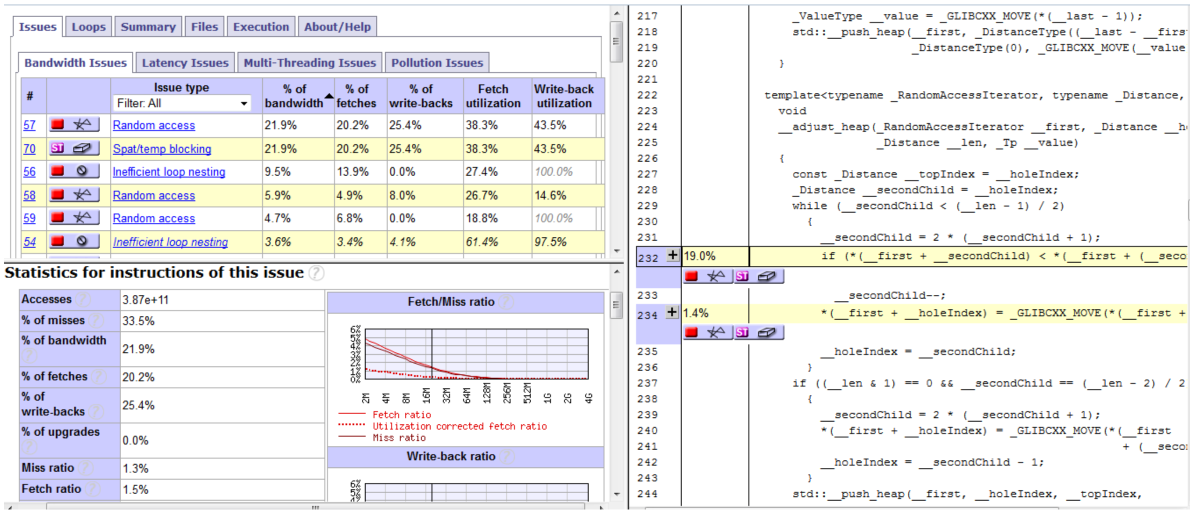
```
#!/bin/bash
#SBATCH --job-name=PTHRD
#SBATCH --output=slurm.out
#SBATCH --error=slurm.err
#SBATCH --partition=batch
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4
#SBATCH --constraint=hpcf2013
#SBATCH --exclusive

export OMP_NUM_THREADS=4
srun sample -r ../fbfs 22
```

We then submitted the job to be run on maya.

```
>sbatch run-pthrd.slurm
```

We then used the following commands to create a report and to create an HTML document for viewing information that was generated through ThreadSpotter.

```
>report -i sample.smp
>view-static -i report.tsr
```



#		Issue type	% of bandwidth	% of fetches	% of write-backs	Fetch utilization	Write-back utilization
57		Random access	21.9%	20.2%	25.4%	38.3%	43.5%
70		Spat/temp blocking	21.9%	20.2%	25.4%	38.3%	43.5%
56		Inefficient loop nesting	9.5%	13.9%	0.0%	27.4%	100.0%
58		Random access	5.9%	4.9%	8.0%	26.7%	14.6%
59		Random access	4.7%	6.8%	0.0%	18.8%	100.0%
54		*Inefficient loop nesting*	*3.6%*	*3.4%*	*4.1%*	*61.4%*	*97.5%*

Statistics for instructions of this issue

Accesses	3.87e+11
% of misses	33.5%
% of bandwidth	21.9%
% of fetches	20.2%
% of write-backs	25.4%
% of upgrades	0.0%
Miss ratio	1.3%
Fetch ratio	1.5%

```
              _ValueType __value = _GLIBCXX_MOVE(*(__last - 1));
              std::__push_heap(__first, _DistanceType((__last - __firs
                                    _DistanceType(0), _GLIBCXX_MOVE(__value
            }

          template<typename _RandomAccessIterator, typename _Distance,
            void
            __adjust_heap(_RandomAccessIterator __first, _Distance __h
                          _Distance __len, _Tp __value)
            {
              const _Distance __topIndex = __holeIndex;
              _Distance __secondChild = __holeIndex;
              while (__secondChild < (__len - 1) / 2)
                {
                  __secondChild = 2 * (__secondChild + 1);
19.0%             if (*(__first + __secondChild) < *(__first + (__seco
                    __secondChild--;
1.4%              *(__first + __holeIndex) = _GLIBCXX_MOVE(*(__first +
                  __holeIndex = __secondChild;
                }
              if ((__len & 1) == 0 && __secondChild == (__len - 2) / 2
                {
                  __secondChild = 2 * (__secondChild + 1);
                  *(__first + __holeIndex) = _GLIBCXX_MOVE(*(__first
                                                             + (__seco
                  __holeIndex = __secondChild - 1;
                }
              std::__push_heap(__first, __holeIndex, __topIndex,
```

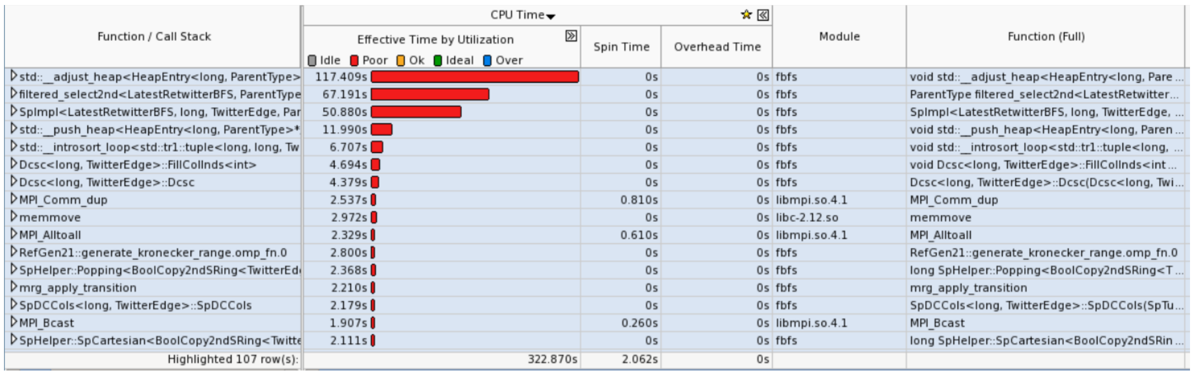
Figure 3.3: Output with 1 node, 1 process per node, and 4 threads per process with scale=22.

The HTML document requires the use of the web browser Mozilla Firefox to display correctly. The output of the tool is very detailed. In the top left of Figure 3.3 are the possible problems, under that section of the figure are possible statistics on the issue, and on the right side, the tool shows where the problem is located in the code. In these are terms and data that only a more advanced computer scientist would understand. Also the information given with this PAT seems to be more hardware based as opposed to software based, with the information provided focusing on memory allocation and cache misses of certain functions as opposed to time taken to perform said functions. ThreadSpotter also does not restrict it's optimization suggestion to the target executable, but also has suggestions in the Redhat/Linux files themselves. If we were able to fully understand the information given by the output, we would be able to use the tool to its fullest extent with our level of understanding.

The output provides suggestions for improving specific portions of the code, something that is very beneficial to easily fixing it if one understands the language being used. It provides suggestions to fixing the code when it is serial, as well as suggestions on how to optimize thread interactions when it is run with multiple threads.

3.4 Intel VTune

Intel VTune is a performance analysis tool created by Intel Corporation to evaluate the performance of programs that are to be run in parallel. Intel VTune has a cost ranging from \$899 to \$3999. They also have a 30 day trial that you may use for free to determine if you like it or not. Once downloaded, it is very easy to install as it uses an installation wizard to guide the user along. Once installed, we began to test out some of the samples using Intel VTune. The instrumentation is relatively straightforward as there are two ways to perform the hotspot analysis. The first is very simple as Intel VTune provides a sample code and walks the user through how to use VTune. The guide does assume that the user has prior programming experience. The second way is by performing parallel computing and this is not explained how to be performed. The first step is the user would use the following



Function / Call Stack	CPU Time: Effective Time by Utilization (Idle / Poor / Ok / Ideal / Over)	Spin Time	Overhead Time	Module	Function (Full)
std::__adjust_heap<HeapEntry<long, ParentType>	117.409s	0s	0s	fbfs	void std::__adjust_heap<HeapEntry<long, Pare ...
filtered_select2nd<LatestRetwitterBFS, ParentType	67.191s	0s	0s	fbfs	ParentType filtered_select2nd<LatestRetwitter...
SpImpl<LatestRetwitterBFS, long, TwitterEdge, Par	50.880s	0s	0s	fbfs	SpImpl<LatestRetwitterBFS, long, TwitterEdge, ...
std::__push_heap<HeapEntry<long, ParentType>*	11.990s	0s	0s	fbfs	void std::__push_heap<HeapEntry<long, Paren ...
std::__introsort_loop<std::tr1::tuple<long, long, Tw	6.707s	0s	0s	fbfs	void std::__introsort_loop<std::tr1::tuple<long, ...
Dcsc<long, TwitterEdge>::FillColInds<int>	4.694s	0s	0s	fbfs	void Dcsc<long, TwitterEdge>::FillColInds<int ...
Dcsc<long, TwitterEdge>::Dcsc	4.379s	0s	0s	fbfs	Dcsc<long, TwitterEdge>::Dcsc(Dcsc<long, Twi...
MPI_Comm_dup	2.537s	0.810s	0s	libmpi.so.4.1	MPI_Comm_dup
memmove	2.972s	0s	0s	libc-2.12.so	memmove
MPI_Alltoall	2.329s	0.610s	0s	libmpi.so.4.1	MPI_Alltoall
RefGen21::generate_kronecker_range.omp_fn.0	2.800s	0s	0s	fbfs	RefGen21::generate_kronecker_range.omp_fn.0
SpHelper::Popping<BoolCopy2ndSRing<TwitterEd	2.368s	0s	0s	fbfs	long SpHelper::Popping<BoolCopy2ndSRing<T ...
mrg_apply_transition	2.210s	0s	0s	fbfs	mrg_apply_transition
SpDCCols<long, TwitterEdge>::SpDCCols	2.179s	0s	0s	fbfs	SpDCCols<long, TwitterEdge>::SpDCCols(SpTu...
MPI_Bcast	1.907s	0.260s	0s	libmpi.so.4.1	MPI_Bcast
SpHelper::SpCartesian<BoolCopy2ndSRing<Twitte	2.111s	0s	0s	fbfs	long SpHelper::SpCartesian<BoolCopy2ndSRin ...
Highlighted 107 row(s):	322.870s	2.062s	0s		

Figure 3.4: Output with 1 node, 4 processes per node, and 4 threads per process with scale=22.

command.

```
>srun --nodes=1 --ntasks-per-node=4 --constraint=hpcf2013 amplxe-cl -c hotspots
 -r results_n01p04t04 -- ./fbfs 22
```

This will generate a directory titled results_n01p04t04. The user would then enter this directory and see one of the files being this: results_n01p04t04.0.amplxe

The user would then perform the command

```
>amplxe-gui results_n01p04t04.0.amplxe
```

This command will launch VTune, Figure 3.4 displays the typical GUI display the user will see once entering the above command in the command prompt of a Linux or Unix terminal, and it will automatically perform a Basic Hotspot Analysis. The figure displays output for CombBLAS run with 1 node using 4 processes and 4 threads per process with scale=22. This figure shows a list of functions that took the longest amount of time for the CPU to run. This is shown in the second column while the respective functions can be found in the first column. We observe from this figure that the top hotspots in our code are:

```
(1) {$std::__adjust_heap<HeapEntry<long, ParentType>*, long, HeapEntry<long,
                  ParentType>>.}
(2) {$filtered_select2nd<LatestRetwitterBFS, ParentType>.clone.}
(3) {$SpImpl<LatestRetwitterBFS, long, TwitterEdge, ParentType,
                  ParentType>::SpMXSpV}.
```

Figure 3.5 displays the results of how each thread is running and performing work for our test case, CombBLAS, on 1 node using 4 processes and 4 threads per process with scale=22. The figure shows how much work each thread is doing over the course of running the program. The name of the thread is shown in the far left column. The green means that the thread is idle while the brown means the thread is working. We observe from this figure that most of the code is run on one thread while the remaining threads are idling.

In conclusion VTune is a very good PAT, but the user should have prior programming experience to be able to troubleshoot and understand how to operate Intel VTune in a Linux or Unix environment.

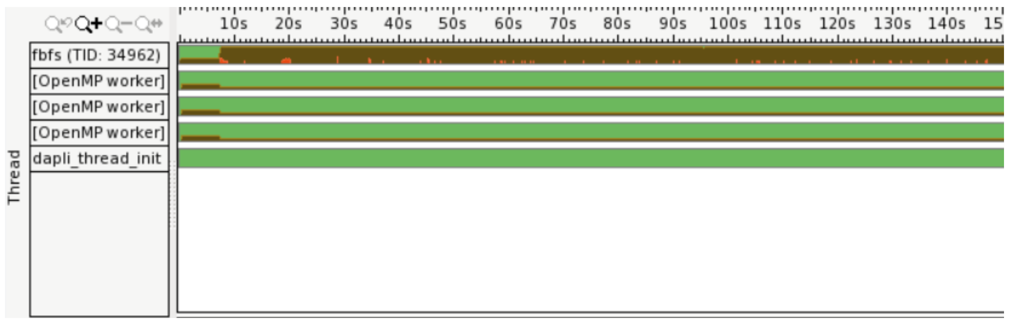


Figure 3.5: Thread Diagnostic with 1 node, 4 processes per node, and 4 threads per process with scale=22.

3.5 TAU

The TAU Performance System is a performance analysis tool that is used to analyze the performance of parallel programs in various programming languages. It is licensed under a BSD style license, and the TAU Performance System ®trademark is licensed by ParaTools, Inc., which has worked substantially with TAU and provides support for TAU. TAU is a shared project between the University of Oregon Performance Research Lab, the LANL Advanced Computing Laboratory, and the Research Centre Jülich at ZAM, Germany. It is a free of charge and can be downloaded from the University of Oregon's website. Annual support for TAU can be purchased from ParaTools for $89,995 per year.

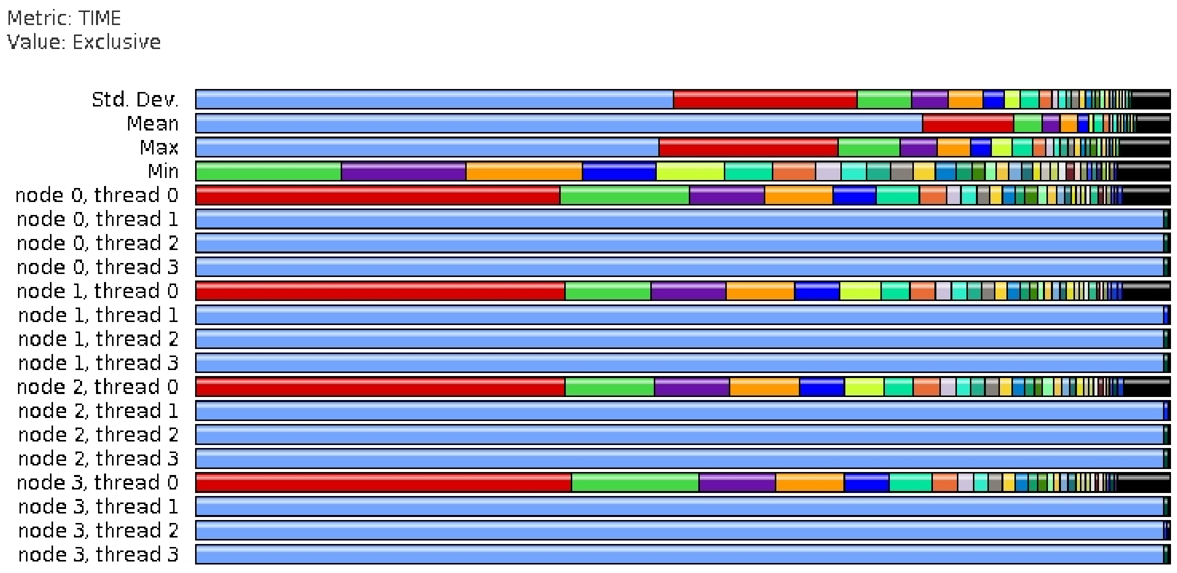


Figure 3.6: Output with 1 node, 4 processes per node, and 4 threads per process with scale=22.

After downloading TAU, the process to install it was fairly simple and straightforward. We first downloaded the source code for TAU, PDT, and PAPI. PDT, or Program Database

Toolkit, is used to analyze source code in various programming languages as well as to make program information available to developers. PAPI is open source software from the University of Tennessee, Knoxville and is used to provide a consistent programming interface for the performance counter hardware found in microprocessors. We then installed PDT:

```
>cd pdtoolkit-3.20
>./configure -GNU
>make install
```

We also set up PAPI for the Intel compiler:

```
>cd papi-5.4.1/src
>make clean
>./configure --prefix=/umbc/lustre/hpcreu2015/team8/research/workspace/papi-5.4.1
 /intel-15.0 CC=icc CXX=icpc FC=ifort F77=ifort
>make
>make install
```

After this was done, we then configured TAU for serial runs:

```
>cd tau-2.24.1/
>./configure -tag=intel \-cc=icc -c++=icpc -fortran=intel -bfd=download \
 -unwind=download \
 -pdt=/umbc/lustre/hpcreu2015/team8/research/workspace/pdtoolkit-3.20 \
 -pdt_c++=g++ \
 -papi=/umbc/lustre/hpcreu2015/team8/research/workspace/papi-5.4.1/intel-15.0
>make install
```

We configured TAU for OpenMP:

```
>./configure -tag=intel -cc=icc -c++=icpc -fortran=intel -bfd=download \
 -unwind=download \
 -pdt=/umbc/lustre/hpcreu2015/team8/research/workspace/pdtoolkit-3.20 \
 -pdt_c++=g++ \
 -papi=/umbc/lustre/hpcreu2015/team8/research/workspace/papi-5.4.1/intel-15.0 \
 -openmp -ompt=download
>make install
```

We configured TAU for MPI:

```
>./configure -tag=intel -cc=icc -c++=icpc -fortran=intel -bfd=download \
 -unwind=download \
 -pdt=/umbc/lustre/hpcreu2015/team8/research/workspace/pdtoolkit-3.20 \
 -pdt_c++=g++ \
 -papi=/umbc/lustre/hpcreu2015/team8/research/workspace/papi-5.4.1/intel-15.0
 -mpi -mpiinc=/cm/shared/apps/intel/mpi/current/intel64/include \
 -mpilib=/cm/shared/apps/intel/mpi/current/intel64/lib
>make install
```

Lastly, we configured TAU for OpenMP and MPI:

```
>./configure -tag=intel -cc=icc -c++=icpc -fortran=intel -bfd=download \
 -unwind=download \
 -pdt=/umbc/lustre/hpcreu2015/team8/research/workspace/pdtoolkit-3.20
 -pdt_c++=g++ \
 -papi=/umbc/lustre/hpcreu2015/team8/research/workspace/papi-5.4.1/intel-15.0
 -mpi \
 -mpiinc=/cm/shared/apps/intel/mpi/current/intel64/include \
 -mpilib=/cm/shared/apps/intel/mpi/current/intel64/lib -openmp -ompt=download
>make install
```

Various export commands were then used to set the file paths correctly:

```
>export TAUROOT=/umbc/lustre/hpcreu2015/team8/research/workspace/tau-2.24.1/x\
 86_64/lib
>export PATH=/umbc/lustre/hpcreu2015/team8/research/workspace/tau-2.24.1/x86_\
 64/bin:$PATH
>export MANPATH=/umbc/lustre/hpcreu2015/team8/research/workspace/tau-2.24.1/m\
 an:$MANPATH
>export LD_LIBRARY_PATH=/umbc/lustre/hpcreu2015/team8/research/workspace/tau-\
 2.24.1/x86_64/lib:$LD_LIBRARY_PATH
```

Setting up TAU to use with the filtered breadth-first search code was easy to accomplish. After using a few export commands to both set the path of the Makefile, as well as change some options with TAU, we used cmake and make commands to compile the code to be compatible with TAU.

```
>export TAU_MAKEFILE=$TAUROOT/Makefile.tau-intel-icpc-papi-ompt-mpi-pdt-openmp
>export TAU_OPTIONS="-optPDTInst"
>mkdir tau_build
>cd tau_build
>cmake -DCMAKE_C_COMPILER=tau_cc.sh -DCMAKE_CXX_COMPILER=tau_cxx.sh ..
>make
```

After the code was compatible with TAU, we created a slurm script to run the code through maya. Below is an example of such a script.

```
#!/bin/bash
#SBATCH --job-name=TAU
#SBATCH --output=slurm.out
#SBATCH --error=slurm.err
#SBATCH --partition=batch
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4
#SBATCH --constraint=hpcf2013
#SBATCH --exclusive

export OMP_NUM_THREADS=4
```

```
srun Applications/fbfs 22
```

We ran the code on maya using the following command.

```
>sbatch run-tau.slurm
```

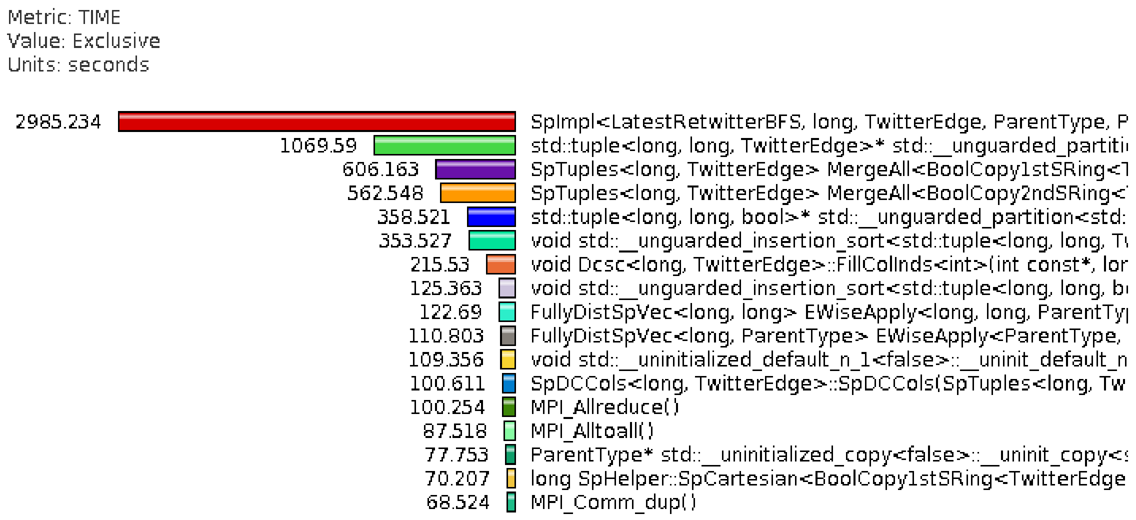
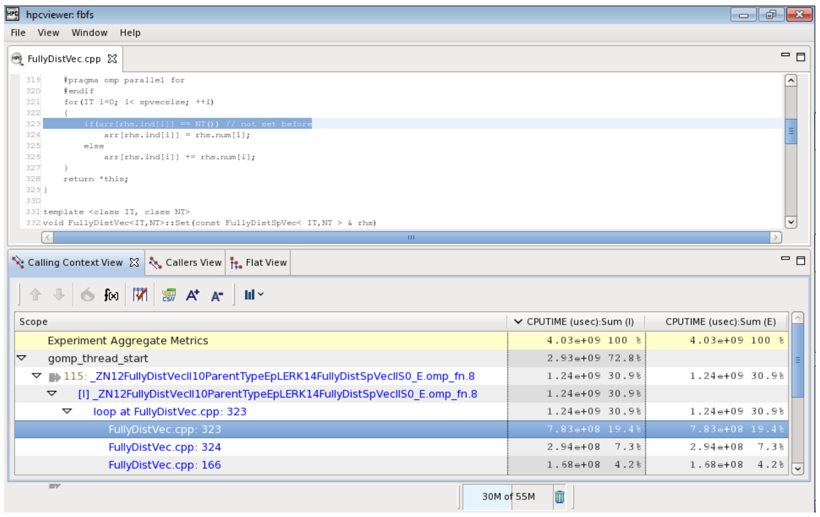


Figure 3.7: Output with 1 node, 4 processes per node, and 4 threads per process with scale=22.

The user interface for TAU is, at first glance, confusing. After further inspection, taking several minutes to experiment with the different options, it became very easy to use. As shown in Figure 3.6, the GUI displays the performance of each process, along with each individual thread. The functions in the program are represented by several different colored bars, while the lengths of the bars represent the length of time that each function took to run. Hovering over each bar reveals the exact time, along with the name of the function. The light blue colored bars, which are present in threads 1 through 3 for each node, represent the threads being idle. If a specific thread is clicked on, a more detailed display appears which has information on every individual function, as shown in Figure 3.7. The time is displayed at a default of a thousandth of a second, with settings available to see a more precise time. Unlike the previous figure, all functions are listed, no matter how long they take to run. There is a way to selectively profile an application, recording the time of specific functions or excluding others, however we were not able to get this feature to work correctly. There are many different settings that can be experimented with to analyze the performance of the code.

3.6 HPCToolkit

HPCToolkit is a tool that was developed by Rice University. Rice University provides the software free of charge. The installation went very smoothly, as they have a tutorial of how to perform the installation for Linux.



```
319     #pragma omp parallel for
320     #endif
321     for(IT i=0; i< spvecsize; ++i)
322     {
323         if(arr[rhs.ind[i]] == NT()) // not set before
324             arr[rhs.ind[i]] = rhs.num[i];
325         else
326             arr[rhs.ind[i]] += rhs.num[i];
327     }
328     return *this;
329 }
330
331 template <class IT, class NT>
332 void FullyDistVec<IT,NT>::Set(const FullyDistSpVec< IT,NT > & rhs)
```

Scope	CPUTIME (usec):Sum (I)	CPUTIME (usec):Sum (E)
Experiment Aggregate Metrics	4.03e+09 100 %	4.03e+09 100 %
gomp_thread_start	2.93e+09 72.8%	
115: _ZN12FullyDistVecIl10ParentTypeEpLERK14FullyDistSpVecIlS0_E.omp_fn.8	1.24e+09 30.9%	1.24e+09 30.9%
[I] _ZN12FullyDistVecIl10ParentTypeEpLERK14FullyDistSpVecIlS0_E.omp_fn.8	1.24e+09 30.9%	
loop at FullyDistVec.cpp: 323	1.24e+09 30.9%	1.24e+09 30.9%
FullyDistVec.cpp: 323	7.83e+08 19.4%	7.83e+08 19.4%
FullyDistVec.cpp: 324	2.94e+08 7.3%	2.94e+08 7.3%
FullyDistVec.cpp: 166	1.68e+08 4.2%	1.68e+08 4.2%

Figure 3.8: Output with 1 node, 4 processes per node, and 4 threads per process with scale=22.

Before downloading the source code for installation of the tool, we went to the directory where we wanted to place the source code:

```
>cd /umbc/lustre/hpcreu2015/team8/research/workspace
```

Then we ran the code below to fetch the code from the website:

```
>svn co http://hpctoolkit.googlecode.com/svn/trunk hpctoolkit
>svn co http://hpctoolkit.googlecode.com/svn/externals hpctoolkit-externals
```

Then for the maya cluster, we had to configure the installation for the linux redhat x86-x64 bit architecture, which meant installing the external packages first:

```
>module load cmake
>cd hpctoolkit-externals
>mkdir BUILD && cd BUILD
>../configure CC=icc CXX=icpc \
 --prefix=$PWD/../../../Opt/hpctoolkit-externals
>make install
>make clean
```

And now for installing hpctoolkit itself:

```
>cd hpctoolkit
>mkdir BUILD && cd BUILD
>../configure CC=icc CXX=icpc \
 --prefix=$PWD/../../../Opt/hpctoolkit \
 --with-externals=$PWD/../../../Opt/hpctoolkit-externals \
>make install
```

The commands above install the tools needed to create the databases for analyzing the test code's high performance C++ source code, and thus installation of "hpcviewer" and "hpctraceviewer" is needed for viewing the data from those databases. The installation procedure for "hpctraceviewer" is as follows:

```
>wget http://hpctoolkit.org/download/hpcviewer
 /hpctraceviewer-5.3.2-r1840-linux.gtk.x86_64.tgz
>tar xvf hpctraceviewer-5.3.2-r1840-linux.gtk.x86_64.tgz
>cd hpctraceviewer
>./install $PWD/../Opt/hpctraceviewer
```

The installation procedure for "hpcviewer" is as follows:

```
>wget http://hpctoolkit.org/download/hpcviewer
 /hpcviewer-5.3.2-r1779-linux.gtk.x86_64.tgz
>tar xvf hpcviewer-5.3.2-r1779-linux.gtk.x86_64.tgz
>cd hpcviewer
>./install $PWD/../Opt/hpcviewer
```

With the installation complete, learning to use HPCToolkit is not straightforward at first. However, thankfully Rice University provided a user's manual PDF that describes in detail how to use the tools. For starters, we used this set of commands to create the database and view it in "hpcviewer":

```
>module load hpctoolkit
>sbatch run-hpctoolkit.slurm
>hpcstruct ./fbfs
>hpcprof-mpi -S fbfs.struct -I ... \
 hpctoolkit-fbfs-measurementsXXXXXXX
>hpcviewer hpctoolkit-database
```

where $XXXXXXX$ is a number generated by running "hpcrun" the run-hpctoolkit.slurm file. The `-I ...` is the include path to where the source code of the program is located at (we didn't include the path in the example above). Below is an example of such a file:

```
#!/bin/bash
#SBATCH --job-name=HPCToolkit
#SBATCH --output=slurm.out
#SBATCH --error=slurm.err
#SBATCH --partition=batch
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4
#SBATCH --constraint=hpcf2013

export OMP_NUM_THREADS=4
srun hpcrun ./fbfs 22
```

Figure 3.8 shows the GUI interface that appears when the user displays the data from

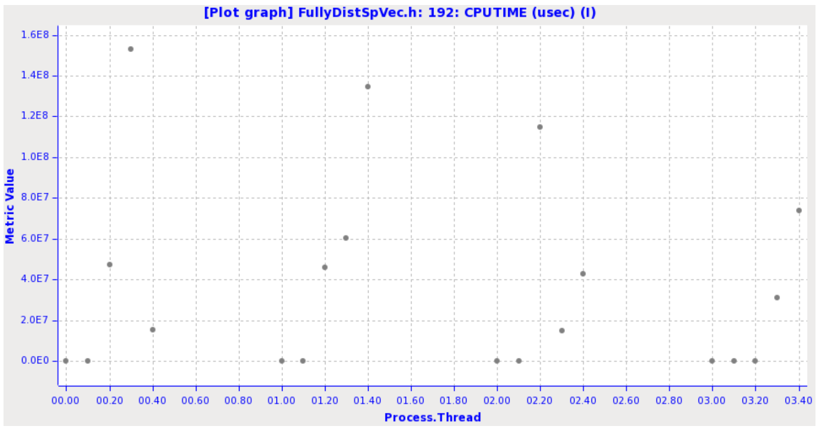


Figure 3.9: Thread Diagnostic with 1 node, 4 processes per node, and four threads per process with scale=22.

a parallel code application such as our "fbfs" in CombBLAS. Displayed output in the GUI shows the source code, and the tool suggests improvements to the code at specific points. The bottom portion contains a red "flame" button that the user can click on to enable the tool to find a "hotspot" and displays how much time (both in units of time and percentage of how much in total run time did this function take while the program was running) in the left side of the bottom of the screen. Just as in Intel VTune we find from Figure 3.8 that the hotspots for this code are:

```
(1) {$std::__adjust_heap<HeapEntry<long, ParentType>*, long, HeapEntry<long,
                  ParentType>>.}
(2) {$filtered_select2nd<LatestRetwitterBFS, ParentType>.clone.}
(3) {$SpImpl<LatestRetwitterBFS, long, TwitterEdge, ParentType,
                  ParentType>::SpMXSpV}.
```

Figure 3.9 shows runs for "fbfs" after running hpcrun in a slurm script. Here, the output shows the process and thread number across the vertical axis and the total work in time done per each thread on a certain OpenMP code segment. We observe again from this figure that one thread does most of the work while the remaining threads idle.

4 Analyzing CombBLAS using PATs

Using four out of the six PATs with which we where able to successfully use on our test case, CombBLAS, has given us the ability to determine several suggestions on which to improve the CombBLAS code. The four PATs are TAU, ParaTools ThreadSpotter, Intel VTune, and HPCToolkit. Before using the tools, we had to compile the code with an additional flag option: `-g`. That flag option allows the PATs to direct the programmer to the particular location where the PAT says the hotspot is or what particular function is causing the program to run slowly.

Originally, the CombBLAS code would always run on just one thread regardless of the number of threads that we tried to set using the command:

```
>EXPORT OMP_NUM_THREADS=#
```

Where # is a number in the set $\{1, 2, 4, 8, 16\}$.

In order to fix the above problem, we determined that we needed to include an additional library `<omp.h>`, and compiler flag `-fopenmp`.

Once we did this, we noticed that when we attempted to run the code on a higher number of threads to improve the run-time, we determined that the run-time stays roughly the same regardless of us varying the number of threads from 1 to 16 in the set $\{1, 2, 4, 8, 16\}$ and keeping the same amount of nodes, and processes per node. This is due to the fact that when the program enters these OpenMP `#pragma`'s for performing multi-threading there is little work to be done in them, even though when the program enters such sections it does indeed split up the work between the threads as seen in the HPCToolkit run (Figure 3.9) and Intel VTune run (Figure 3.5).

The second thing we determined from using the PATs is that they agreed on the top hotspots in the code, which are listed from the highest, i.e., (1), to the lowest, (3). The functions are listed below:

```
(1) {$std::__adjust_heap<HeapEntry<long, ParentType>*, long, HeapEntry<long,
                 ParentType>>.}
(2) {$filtered_select2nd<LatestRetwitterBFS, ParentType>.clone.}
(3) {$SpImpl<LatestRetwitterBFS, long, TwitterEdge, ParentType,
                 ParentType>::SpMXSpV}
```

We examined the three functions in detail, but we were unable to find a way to improve them, i.e., reduce the run-time of those function calls; however, in the time available we could not find a specific spot to add OpenMP parallelization, i.e., inserting `#pragma` calls, due to the complexity of the CombBLAS library.

5 Conclusions

Since we evaluated six PATs, we are able to identify several differences between them. One difference among them is that several of the PATs are indirect, which means that it uses sample code to generate suggestions, and others are direct, where hooks are used to create blocks of code that will be analyzed. Intel VTune, HPCToolkit, ThreadSpotter and Scalasca are all indirect while TAU and Score-P were direct.

After evaluating the four PATs that we could successfully use on our application program CombBLAS, we then moved on to trying to improve CombBLAS. In our attempt to improve CombBLAS, we determined three major “hotspots” in the code, but in the time available we could not reduce the run-time of these “hotspots”, which would have improved the speedup of the CombBLAS code significantly, due to the level of difficulty of the code.

Overall, each PAT (performance analysis tool) identifies the same hotspots, but uses different techniques to obtain the result. Another difference among the PATs is how the output is displayed. Scalasca, for example, does not provide a GUI (graphical user interface) unless you download the software Cube. TAU, Intel VTune, HPCToolkit, and ThreadSpotter displayed their output via GUI, while Score-P provides its output through the terminal. Intel VTune and HPCToolkit are event capturing profilers that show which function takes the longest to run and displays the results in a event capturing routine: the hotspot is expanded and displays either a calltree or caller view of how the hotspot was reached. TAU is an in depth profiler that gives the total running time of each function call, but can also be used to give a trace view.

Acknowledgments

These results were obtained as part of the REU Site: Interdisciplinary Program in High Performance Computing (hpcreu.umbc.edu) in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County (UMBC) in Summer 2015. This program is funded by the National Science Foundation (NSF), the National Security Agency (NSA), and the Department of Defense (DOD), 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). HPCF is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from UMBC. Co-authors Wesley Collins and Daniel Martinez were supported, in part, by the UMBC National Security Agency (NSA) Scholars Program through a contract with the NSA. Graduate assistants Ari Rapkin Blenkhorn, Jonathan S. Graf, and Samuel Khuvis were supported during Summer 2015 by UMBC.

References

[1] Aydın Buluç and John R. Gilbert. The Combinatorial BLAS: design, implementation, and applications. *International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[2] Aydın Buluç, John R. Gilbert, Adam Lugowski, and Scott Beamer. Combinatorial BLAS Library (MPI reference implementation). http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/, accessed on July 6, 2015.

[3] Jay Doshi, Chanchal Khemani, and Juhi Duseja. Breadth First Search. http://codersmaze.com/data-structure-explanations/graphs-data-structure/breadth-first-search-traversal/, accessed on July 7, 2015.

[4] Score-P Support. Score-P. http://www.vi-hps.org/projects/score-p/, accessed on July 16, 2015.

[5] Eric W. Weisstein. Adjacency matrix. http://mathworld.wolfram.com/AdjacencyMatrix.html, accessed on July 24, 2015.