Profiling and Tracing SUMMA Matrix Multiplication with TAU

Brae Troutman
troupb18@wfu.edu
Wake Forest University

Abstract

Analyzing the provenance and performance of data and computational resource usage is a fundamental step in learning about, implementing, and analyzing parallel and distributed algorithms using the Message Passing Interface (hereafter referred to as MPI). Their are numerous ways to go about this analysis, varying in the types of metrics they can produce and how much work they require from the end user in utilizing them. I present a C++ implementation of the SUMMA matrix multiplication algorithm and associated analysis using the TAU profiling/tracing system, framed in the context of parallel algorithms pedagogy– that is, the implementation and subsequent analysis of parallel algorithms in a classroom setting. All of this is with the goal of demonstrating the advantages and shortcomings of TAU as a means of analysis in the classroom, in comparison with direct code instrumentation on the part of the programmer.

Keywords: profiling, tracing, distributed, parallel

ACM Reference Format:
Brae Troutman. 2022. Profiling and Tracing SUMMA Matrix Multiplication with TAU. In *Proceedings of (Parallel Algorithms Final Projects).* ACM, New York, NY, USA, 7 pages.

1 Introduction

SUMMA is a well documented parallel and distributed parallel matrix multiplication algorithm[5] that utilized the MPI interface to distribute data and workload across multiple computational contexts. When working in a distributed environment like this however, analyzing performance and following the movement of data becomes harder to manage the more processors, nodes, or analogous resources are applied to the problem. In the classroom context, such work is often done by hand, with students instrumenting their own code with time checks and logging output to keep track of performance and analyze the correctness of their implementations: but these same methods do not carry over effectively when working in highly parallel and distributed contexts. This paper and the associated project implements the SUMMA algorithm in C++ as it might be in a parallel algorithms course, and uses the TAU profiling and tracing tool to analyze performance. In this way, we show how such a profiling and debugging tool can be used in a classroom context to lift the student away from the work of collecting performance data for their parallel algorithms and allow them to focus on the algorithms themselves, as well as showing the advantages and shortcomings of the TAU tool and its associated sister tools in facilitating parallel algorithms pedagogy.

1.1 Technical Context

Before diving into implementation and analysis, it is important to lay out the parameters and conditions under which the following project was implemented, seeing as replicating the experiment with different software, architectures, etc. may lead to different or erroneous results. This also provides context for architectural limitations with running parallel applications. All tools and software used are free and open-source.

- Architecture
  - OS: Pop!_OS 22.04 LTS (Debian-based Ubuntu Derived Linux Distro)
  - 12 12th Gen Intel Cores (i5-1235U)
  - 8 GiB RAM
- Tools + Versions Used
  - TAU 2.32
  - PDToolkit 3.25.1
  - slog2rte (Jumpshot-4) 1.2.6
  - OpenMPI 4.1.2

1.2 Tool Installation and Setup

Frankly, the bulk of the work for this project came from actually downloading and setting up the environment to work with MPI and the TAU profiler on my personal Linux computer. As such, I feel that the installation and environment construction process warrants an explanation so that interested students or teachers can replicate the same process on their personal computers.

If more interested in the results of the project than how to replicate it yourself, you can skip ahead to section 2

OpenMPI. Definitely the simplest setup of the tools used here, OpenMPI simply uses the operating system's native

Parallel Algorithms Final Projects, December, 2022, Winston Salem, NC

package manager to install. For a Debian derivative, this means using the apt-get tool for installation:

```
$ sudo apt-get install openmpi-bin \
  openmpi-common libopenmpi-dev libgtk2.0-dev
```

For Fedora derivatives install with the dnf native package manager– after which the mpi module must be loaded anytime the relevent include files, libraries, and binaries are required.

```
$ sudo dnf install openmpi openmpi-devel
$ module load mpi/openmpi-x86_64
```

For Mac and Windows users will have to install from source, but Windows users may also use Windows Subsystem for Linux and an X-Windowing client to copy Debian or Fedora installation.

PDToolkit + slog2rte(Jumpshot). PDToolkit is a necessary dependency of TAU that allows for automatic instrumentation of C[++] and Fortran code, rather than requiring the user to make explicit procedure calls within their code to profile their program. [4]

To install, download the zipped project archive and unzip it. From here, cd into the unzipped package and run:

```
$ ./configure
$ make
$ make install
```

All the appropriate binaries will be generated, and all that left is to add the bin directory generated to the PATH environment variable. Jumpshot can be downloaded from here and unzipped, no building necessary.

TAU. Finally, TAU installation follows a similar process to the installation of the PDT tool: configuration is a bit more involved, however, as the compiler needs to be directed toward appropriate MPI and PDT includes and libraries in the system.[4]

Once again, download and unpack the package. In order to use PDT to automatically instrument compiled code and to run MPI based programs, TAU must be configured to generate the correct Makefile. Change into the unzipped TAU directory and run:

```
$ ./configure -pdt=path/to/pdt/bin \
  -mpi -mpiinc=path/to/openmpi/include \
        -mpilib=path/to/openmpi/lib \
  -PROFILE -TRACE
$ make
$ make install
```

This will generate the appropriate Makefile to allow for compiling, profiling, and tracing with OpenMPI and PDT. Then the TAU bin directory must be added to the PATH environment variable, and TAU will be ready to compile, profile, and visualize MPI programs.

2 SUMMA Algorithm and Implementation

Matrix-matrix multiplication is a fundamental computation throughout scientific computing disciplines, and thus is a well-defined and well studied problem in terms of its efficiency and possible parallel implementations. The SUMMA matrix multiplication algorithm utilizes MPI functions to distribute two matrices across a number of processes and evenly divide computation between each process.

In this section, I will outline the SUMMA algorithm in pseudo-code, explain how that translates over to C++ code, and describe the choices/limitations of my own approach to implementing the algorithm.

2.1 The Algorithm

SUMMA can be represented with the following MATLAB-like pseudocode, taken from the MPI Model slides and adapted to fit into C-style zero-indexing: [5]

```
FUNCTION C = SUMMA(C,A,B,b,pi)
    (ranki, rankj) = MyProcessID(pi)
    for i = 0 up to sqrt(p)
        for j = 0 up to n/(b*sqrt(p))
             pi(ranki,i) broadcasts the jth b
                columns of its local block of
                multiplicand A to every process
                in their row. Store in Atemp
             pi(i, rankj) broadcasts the jth b
                rows of its local block of
                multiplier B to every process
                in their column, Store in Btemp
            C(ranki, rankj) = C(ranki, rankj)
                              + Atemp * Btemp
        end for
    end for
end function
```

This algorithm assumes that ownership of the multiplicand matrix A, multiplier matrix B, and product matrix C is divided up between processes before execution, and execution of the above code is performed by all processes simultaneously (that is, an instance of this algorithm effectively runs on every node in the cluster, process on the computer, or whatever topology applies to the current MPI context).

My C implementation differs from the standard algorithm in that it imposes some limitations and guards on the input to make it simple to check the correctness of the algorithms and to allow for the simplification of the implementation process. SUMMA itself allows for the multiplication of non-square matrices distributed between an $r \times c = p$ processor grid, while my implementation requires that both the multiplicand and multiplier matrices be square and of the same height and width.

In addition, the multiplier B is set to be the identity matrix of $\mathbf{I}_n$, so that the correctness of each multiplication $AB = C$

can be validated by confirming that $A = C$ after execution, where A is the randomly generated multiplicand.

Finally, matrices A and C are stored in column-major order. This makes it simple to extract any b columns of A to transmit to all other processes in a row with a simple `memcpy` operation. On the same vein, blocks of matrix B are stored in row-major order on each processor, making it similarly simple to extract any b rows of B. While this simplifies the communication process, it sacrifices the spatial locality of local matrix-matrix multiplies on each processor.

The full C++ implementation and Makefiles for compilation can be found at BraeTroutman/summa.

2.2 Theoretical Performance

This algorithm makes use of only one MPI collective operation, Broadcast, which has a set latency of $\alpha \cdot O(\log p)$ and bandwidth of $\beta \cdot O(n)$.[3] Thus, deriving the time and communication complexity and thus theoretical performance of SUMMA simply requires determining the amount of data being communicated at each step in the algorithm and how often these communications are repeated. We can divide this analysis up in terms of computation, communication latency, and communication bandwidth.

Computation. In this algorithmic model, the input and output matrices A,B, and C are all of size $n \cdot n$ and are divided evenly between each processor: that is, each processor holds onto $\frac{1}{p}$th of matrix A and $\frac{1}{p}$th of matrix B. However, at each iteration of the inner loop, a multiplication of one $\frac{n}{\sqrt{p}} \times b$ matrix and another $b \times \frac{n}{\sqrt{p}}$ matrix occurs, each with a time complexity of $O\left(\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}} \times b\right) = O\left(\frac{bn^2}{p}\right)$. The total number of iterations combining the inner and outer loops is $\frac{n}{b}$, so this computation occurs that many times, meaning that the total computational complexity is $\frac{n}{b} \times O\left(\frac{n^2}{p}\right) = O\left(\frac{n^3}{p}\right)$

Bandwidth. Over the course of execution, each process in the grid must transmit all of its local block of matrix A to every other process in its row, and all of its local block of matrix B to every other process in its column– whether that happens all at once or over the course of several iterations depends on the tunable blocksize value b. Over all $\frac{n}{b}$ iterations of the outer for-loops, 2 broadcasts will occur each time, each of size $\frac{n}{\sqrt{p}} \times b$. Since the bandwidth cost of a broadcast is linear in the size of the data being transmitted, the total bandwidth cost is $\frac{n}{b} \times \frac{n}{\sqrt{p}} \times b = O(\frac{n^2}{\sqrt{p}})$

Latency. Finally, the latency cost of a broadcast in MPI is $O(\log p)$. However, in SUMMA our two broadcasts are executed $\frac{n}{b}$ times, meaning we must account for a total latency of $\frac{n}{b} \times O(\log p) = O(\frac{n}{b} \log p)$

Total Cost. So, tallying everything together, our total cost and guide for analyzing the performance of SUMMA is:

$$\gamma \cdot O\left(\frac{n^3}{p}\right) + \alpha \cdot O\left(\frac{n}{b} \log p\right) + \beta \cdot O\left(\frac{n^2}{\sqrt{p}}\right)$$

As can be seen here, all of the algorithmic costs of SUMMA in MPI are independent of the blocksize parameter b save for the latency: by increasing the size of b we minimize the latency of our communication to approach $O(\log p)$, but at the cost of requiring larger auxiliary memory to store temporary matrix blocks received on each processor, which might not be achievable for large matrices where the memory footprint of storing each block of the input and output matrices may already be on the order of gigabytes.[5]

3 TAU Experimental Analysis

Now that we have our theoretical analysis and expectations for SUMMA's performance, we can look at how it actually performs using TAU and associated tools. In this section, we will do exactly that, utilizing TAU to compile the `summa.cpp` source file, and use the data generated to do analysis with the `paraprof` and `pprof` tools, and extract this data to visualize scaling across test cases.

Before we do this however, it is important to understand the difference between profiling and tracing for the context of the information interacted with below.

Profiling. is about determining the resources used by a given step in a computation, and the amount of time it takes to execute

Tracing. on the other hand focuses on what is happening at any given moment in program's execution: instead of wondering how long it takes for something to happen, tracing is about knowing when it happens– either relative to other processes in the application or based on time passing in the real world. [1]

In this project I focus primarily on profiling because it more closely maps to performance, whereas tracing is useful for understanding how processes in an MPI application interact with one another.

3.1 Compilation and Execution Instructions

[4] Compiling for TAU is not that different from compiling for standard execution, and is actually quite similar to MPI's compilation method in that it makes use of a script that acts as a wrapper around the g++ compiler to compile executables. The only really different aspect of compilation is in setting the context for profiling versus tracing in the system environment: both compilations must be done separately and require different environment variables to be set. For that reason, this section is divided up into compiling for profiling and compiling for tracing.

Compiling for Profiling. [4] To create a profiling executable, we first must set the appropriate Makefile from TAU's source in the TAU_MAKEFILE environment variable. This will tell the TAU compiler what libraries and includes to access when instrumenting the C++ code.

```
$ export TAU_MAKEFILE=path/to/Makefile.tau\
  -mpi-pdt
$ tau_cxx.sh summa.cpp -o summaprof
```

This will produce a `summaprof` executable that we will be able to run later. The TAU_MAKEFILE will be in TAU's lib directory, which will be located in the unzipped and built TAU archive from earlier.

Compiling for Tracing. To create the tracing executable, every step is the same except for the Makefile that we point TAU to:

```
$ export TAU_MAKEFILE=path/to/Makefile.tau-\
  mpi-pdt-profile-trace
$ tau_cxx.sh summa.cpp -o summatrace
```

3.2 Profiling

After compiling the source code, all that's left is to actually run the executables and a take a look at the output. Running the executable compiled for profiling will produce a profile file for every single node/process/thread that executed the program, each containing a table made up of the functions/collective operations used in the executable and statistics about each: how long they took to execute, how many times they were called, how many subroutines they called, etc. From here, the data in these files can be loaded into the `paraprof` tool included with TAU to visualize the performance of each individual execution of the program. To analyze across multiple executions with varying numbers of processes or different input sizes we must do some preprocessing of the data collected by TAU to aggregate performance metrics from execution to execution.

Execution. Executing the program is just the same as executing any other MPI program: this particular executable takes two arguments, the number of row/columns in the input matrix, and the blocksize parameter b. Note that the number of processors must be a perfect square, as we expect a square process grid.

```
$ mpirun --oversubscribe -n 100 ./summaprof 400 1
```

This produces the output of the program to the console, and has the side-effect of producing 100 profile files– one for each process/node that executed the program. They will be named `profile.[0-99].0.0`. We oversubsribe the number of processors that MPI uses for the sake of producing more visible metrics for visualization later, because if our number of processes is too small then the time to execute broadcasts is so small that the cost can not even be seen when showed up against the global MPI costs of syncs, `MPI_init()`, and `MPI_finalize()`, which are incurred by every MPI program.

Paraprof + Pprof. The `paraprof` and `pprof` tools are both commandline tools provided in TAU's bin directory that allow for aggregation and analysis of the data in the `profile` files produced by instrumented MPI code. `paraprof` is a more robust tool that allows for graphical analysis of profiled data, while +pprof+ is a text-based tool that is more convenient for BASH scripting.

Paraprof. Using `paraprof` first, we will look at the performance of our 100-node execution of the SUMMA algorithm to multiply two 400 by 400 matrices. To launch `paraprof`, simply run the command `paraprof` in the project directory.

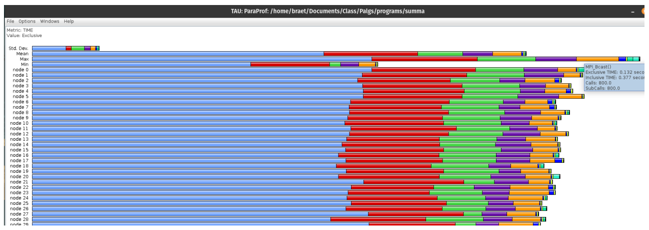


Figure 1. paraprof landing page: side by side of each node's execution

In Figure 1, we see `paraprof`'s landing page, which shows a side by side comparison of the running times of the MPI program on each of the 100 nodes, and a summary showing the longest, shortest, and mean execution times across all nodes. The colors correspond to each procedure/collective in the program, and the proportion they consume on the timeline corresponds to the proportion of execution time they contributed to the total runtime of the program. In this visualization, it seems that `MPI_Init()` in blue and `MPI Collective Sync` in red consume the most execution time, followed by `MPI_Finalize()`. All of these procedures are part of every MPI program, regardless of the collectives used within.[2] The only collective explicitly called in the program that is really visible on the timeline is `MPI_Bcast`, corresponding to our repeated broadcasts of matrix blocks from one processor to every other in their row or column group.

We can get a more granular view of this information by introspecting into the mean performance metric across all nodes, by double clicking on the "mean" row.

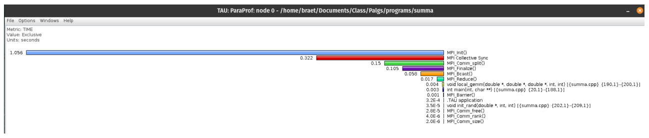
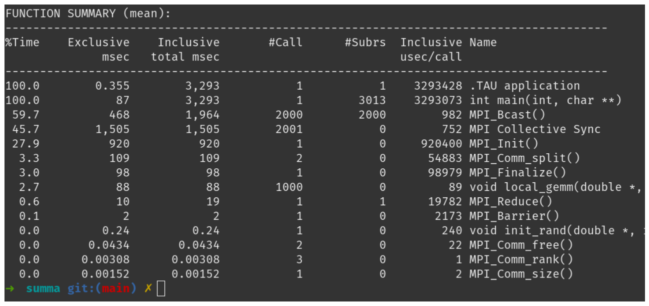

TAU: ParaProf: node 0 - /home/braet/Documents/Class/Palgs/programs/summa
File Options Windows Help
Metric: TIME
Value: Exclusive
Units: seconds
1.056 MPI_Init()
0.322 MPI Collective Sync
0.15 MPI_Comm_split()
0.105 MPI_Finalize()
0.058 MPI_Bcast()
0.017 MPI_Reduce()
0.004 void local_gemm(double *, double *, double *, int, int) [{summa.cpp} {190,1}-{200,1}]
0.003 int main(int, char **) [{summa.cpp} {20,1}-{188,1}]
0.001 MPI_Barrier()
3.2E-4 .TAU application
3.5E-5 void init_rand(double *, int, int) [{summa.cpp} {202,1}-{209,1}]
2.8E-5 MPI_Comm_free()
4.0E-6 MPI_Comm_rank()
2.0E-6 MPI_Comm_size()

Figure 2. paraprof 100 node SUMMA average performance

Zooming in on an individual row from Figure 1, we can see all of the functions and collectives that TAU profiled, and how much they contributed to the overall running time. Looking at the small barely visible yellow bar, it seems that the local matrix-matrix multiplication on each node took an average of 0.004 seconds, compared to the nearly 1 second taken by `MPI_Init()` alone. What this says about this specific instance of SUMMA is that with a matrix of such relatively small size, costs of initializing MPI outweigh the amount of time actually saved by using its collective communication. So while there is objectively a benefit to parallelizing the matrix multiplication for this matrix, we might save time for matrices around this size by using OpenMP or a similar, more lightweight parallel framework rather than OpenMPI. When scaling to matrices of size on the order of 1000 x 1000 and greater however, the cost of local computation and Broadcasting will quickly begin to outweigh the MPI initialization and cleanup routines.

Pprof. `paraprof` allowed for introspection into and analysis of single executions of the SUMMA algorithm, which is useful in its own right. However, when working with parallel algorithmic analysis it is important to be able to observe how performance scales as more processors are added to tackle a problem or the size of the problem is modified. Since `paraprof` is a graphical tool, extracting performance information from each execution of the algorithm as the problem scales would be a tedious hands-on process requiring the student to cyclically execute, record data from visualizing, and repeat. With `pprof` however, extracting data from TAU profiles is a simple matter of text processing through BASH wizardry. Calling `pprof` in the project directory full of `profile` files will print the execution data for every node, and also print the aggregated metrics across all nodes: i.e. average and total execution times.



```
FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.355        3,293           1           1    3293428 .TAU application
100.0           87        3,293           1        3013    3293073 int main(int, char **)
 59.7          468        1,964        2000        2000        982 MPI_Bcast()
 45.7        1,505        1,505        2001           0        752 MPI Collective Sync
 27.9          920          920           1           0     920400 MPI_Init()
  3.3          109          109           2           0      54883 MPI_Comm_split()
  3.0           98           98           1           0      98979 MPI_Finalize()
  2.7           88           88        1000           0         89 void local_gemm(double *,
  0.6           10           19           1           1      19782 MPI_Reduce()
  0.1            2            2           1           0       2173 MPI_Barrier()
  0.0         0.24         0.24           1           0        240 void init_rand(double *,
  0.0       0.0434       0.0434           2           0         22 MPI_Comm_free()
  0.0      0.00308      0.00308           3           0          1 MPI_Comm_rank()
  0.0      0.00152      0.00152           1           0          2 MPI_Comm_size()
→  summa git:(main) ✗
```

Figure 3. pprof 100 node SUMMA average performance

In Figure 3, we see that the same information shown in `paraprof` graphically can be extracted and displayed as text data by its text-based cousin `pprof`. Since we can easily get the average execution time across all nodes for a given execution, we can take this data and convert it into a csv record, which can be added to an accumulating csv file to show scaling over time.

Since this implementation of SUMMA requires perfect squares for the number of processors, and the machine being used for this project has 12 cores, analyzing scaling is not feasible because only two data points are possible: one with 1 core to run SUMMA, and one with 4. However, it can still be shown how TAU profile data can be extracted to csv format for large-scale automatic scaling analysis.

Looking at Figure 3, it would be useful to extract the "Inclusive total msec" metric column and turn it on its side, and intercalate commas in between each value to provide a csv record to be part of a larger csv file. This can be done easily by pipelining the output of `pprof` into further processing commands.

```
# find out how many functions are
# profiled by TAU: this will be
# the number of lines we need to
# pull from the end of pprof's
# output
numfuns=$(pprof -l | tail +2 | wc -l)

# convert the lines of functions
# into a comma-separated list
# of functions
funlist=$(pprof -l \
            | tail +2 \
            | tail -$numfuns \
            | sed -E 's/\(.*\)//g' \
            | paste -s -d,)

# convert the 3rd column of the
# pprof mean data into a comma-
# separated list of msec values
echo $(pprof -s \
        | tail -$numfuns \
        | awk -F ' ' '{print $3}' \
        | sed -E 's/,//g' \
        | paste -s -d,)
```

The BASH script above uses `pprof` to extract running time information from a particular execution of SUMMA. To explain the commands used:

- `pprof -l` prints a list of the functions profiled by TAU
- `tail 2+` returns all but the first line of its input (removing the header printed by `pprof`
- `wc -l` counts the number of lines in its input, here yielding the number of functions TAU profiled
- `sed -E 's/\(.*\)//g'` removes anything that occurs between parentheses in its input, removing the arguments in each function (this was important because the asterisks used for pointers in some function definitions were going through shell expansion)

- `paste -s -d,` stitches together the lines of its input with commas
- `awk -F ' ' '{print $3}'` extracts the third word of each line in its input and prints it

All of these tools can be piped together to extract profiling data in a format easily usable in programs. Executing this script repeatedly for different numbers of MPI nodes can quickly and programmatically record data for parallel scaling experiments.

For example, running this script on the profiling data from the mpirun in the above section yields:

```
$ bash tau2csv.sh
3293,3293,1964,1505,...
```

Which matches the data in column 3 of the output in Figure 3.

3.3 Tracing

Tracing is useful for debugging, which can be difficult when working with parallel applications where multiple processes are executing simultaneously. TAU can produce traces for the execution of a given MPI program, recording at what point in time relative to the beginning of the application's runtime a given function has control over the execution of the program. This yields a different viewpoint from profiling, because profiling keeps track of the total amount of a programs runtime that takes place in each function– this does not describe how control is passed around, or record information about one process halting execution while another continues, both of which tracing can keep track of.

Producing Traces. To produce traces for visualization, the TAU tracing instrumented executable must be run, and then the traces produced by the instrumented code must be aggregated for input to the visualization program, Jumpshot. Thankfully, TAU provides a script that can easily aggregate multiple trace files across nodes, and another tool to convert from the TAU trace format to slog2 format.

```
$ mpirun --oversubscribe -n 100 \
  ./summatrace 400 1
$ tau_merge tautrace* summa.trc
$ tau2slog2 summa.trc tau.edf \
  -o summa.slog2
```

After executing these commands all of the tracing and event data from the SUMMA execution will be aggregated in the file `summa.slog2`.

Visualizing Traces. Now the aggregated traces can be loaded into Jumpshot for analysis. Run the Jumpshot executable jarfile using:

```
$ java -jar path/to/slog2rte/lib/jumpshot
```

Then the `summa.slog2` file must be found and loaded using the built in file finder in the application.

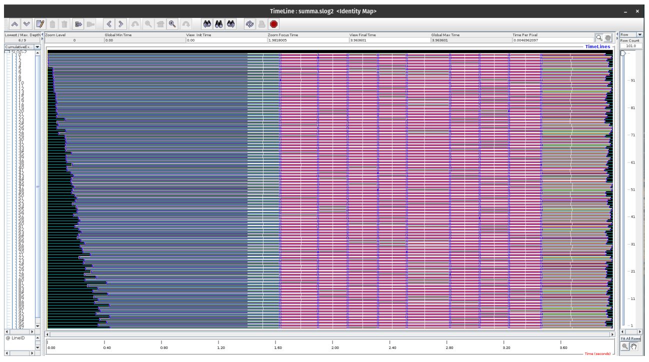


Figure 4. Jumpshot timeline overview

Figure 4 is what shows immediately after loading the trace file into Jumpshot. At a high level it seems that the first few nodes start execution roughly simultaneously, while the rest of the nodes begin progressively later– this MPI execution was oversubscribed to 100 nodes on a 12 core machine, of course. The dark blue blocks on the left represent the MPI initialization process, while the pink block in the middle is when all processes are roughly synchronized by the continuous broadcasting of sub-blocks of matrices from process to process. Finally, the block at the end shows the MPI finalization process.

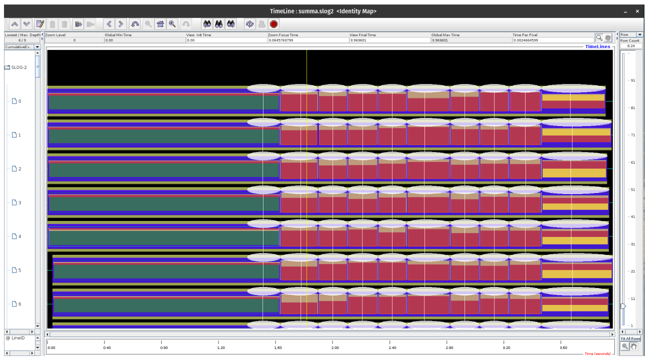


Figure 5. Jumpshot timeline zoomed in

Honing in on the first few processes, the view is more detailed in demonstrating the information that Jumpshot visualizes. Blocks embedded within one another represent functions that are called inside of another one: for example, the entire program takes place in .TAU Application (in chartreus), and all functions are called inside of main (in blue). White circles mark system events on during execution, but with the current descriptions in Jumpshot what exactly these events are remains unspecified: since they seem to occur simultaneously across all nodes they are likely signals involved in interprocess communication to coordinate execution.

Oversubscription is an obvious issue in the performance of this run of the SUMMA algorithm – but there are times when oversubscription might be warranted. For example, when working with data in real life, a perfect square number of processors is likely not available: to create a square processor grid, one would either have to over or under subscribe to the nearest square number of processors. Erring on the side of over subscribing would make use of all processors, but it seems from the trace here that the cost of context switching oversubscribed processes and global syncs across virtual "nodes" that are not all running simultaneously is not negligible, leading to the conclusion that it is definitely worthwhile to re-implement the SUMMA code to work with an arbitrary processor grid and allow for a wider range of processors for working with input data.

4 Conclusion

All in all, TAU is a useful tool for profiling and tracing parallel and distributed algorithms and their performance. In a classroom context, it is certainly feasible and even advantageous for students to have access to TAU for analyzing their parallel programs– although it does not necessarily provide any information that students working on small-scale distributed projects could not retrieve themselves with explicit instrumentation of their code.

That being said, the tool could be indispensable when working on large scale projects scaling across hundreds of nodes with huge input problems distributed across them: what was demonstrated in this project was only a portion of what TAU is capable of, with it also being able to track communication sizes and destinations across MPI processes and work with other parallel programming libraries, such as OpenMP. Visualization and automation is beneficial in the sense that it takes the onus of gathering and visualizing performance information off of the end user, and TAU does just that– with a rather steep learning curve in terms of installation. If TAU were delivered out-of-the-box, simply available to students to utilize on a compute cluster then I imagine it would greatly facilitate parallel algorithms pedagogy. As for installation on individual computers and small-scale projects, I would say TAU is less than ideal for the casual user.

Aside: if it seems like having a tool like TAU available on the DEAC cluster would be useful for future classes, I would be happy to coordinate with Cody to figure out how installing on the cluster would differ from installing on a personal computer.

References

[1] Thomas Ball and James R. Larus. 1992. Optimally Profiling and Tracing Programs. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, USA) *(POPL '92)*. Association for Computing Machinery, New York, NY, USA, 59–70. https://doi.org/10.1145/143165.143180

[2] Guodong Li, Michael Delisi, Ganesh Gopalakrishnan, and Robert Kirby. 2008. Formal specification of the MPI-2.0 standard in TLA+. 283–284. https://doi.org/10.1145/1345206.1345257

[3] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. 2007. Performance analysis of MPI collective operations. *Cluster Computing* 10, 2 (01 Jun 2007), 127–143. https://doi.org/10.1007/s10586-007-0012-0

[4] Zam Uo and Lanl. 2022. *TAU User Guide.* Retrieved December 7 2022 from https://www.cs.uoregon.edu/research/tau/docs/newguide/bk01.html

[5] R. A. Van De Geijn and J. Watts. 1997. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274. https://doi.org/10.1002/(SICI)1096-9128(199704)9:4<255::AID-CPE250>3.0.CO;2-2