Performance Analysis and Tuning with TAU

Kevin James Edwards

kjedward@ncsu.edu

CSC 548 – Parallel Systems

North Carolina State University

Dr. Frank Mueller

1. Introduction

TAU is a performance analysis tool that can be used to for assisting in creating more efficient code. One can use the tools provided with TAU to help determine where inefficient portions of code may lie so that they can be analyzed and strengthened. TAU also has the capability to analyze parallel applications that use OpenMP and message passing programs using MPI.

Since parallel applications must be especially focused on efficiency, TAU and similar tools can be extremely important for identifying portions of applications that are negatively affecting runtime. Parallel applications in particular can be difficult to assess and determine where weaknesses lie. So with the help of TAU and other tools, one can determine if particular techniques can be used to maximize efficiency.

2. Project Description

My goal for this project was to use TAU to identify inefficiencies in parallel benchmarks and take this information along with what I have learned in class to remedy these benchmarks. Ultimately the goal is to decrease runtime for a couple of benchmarks. I had planned on using these techniques on two different benchmarks, one written in OpenMP and the other using the message passing paradigm, MPI. I would need to first get information about the execution of these benchmarks and then find potential weak areas and attempt to cure them. The inefficient areas might be memory or cache related or they might be message passing related for the MPI benchmark. The techniques we have learned for improving efficiency we mostly associated with these two areas of the application.

3. Initial Approach

I will first need to install TAU and link it with some potential benchmarks to work out any kinks in getting the tool working correctly on our architecture. I must then focus on finding potential problem areas in benchmarks. Luckily some previous research has been done in this area and inefficiencies have already been found in some benchmarks. Therefore I must use TAU on these benchmarks to make sure that I am seeing these inefficiencies myself. Then the problem becomes alleviating some of these inefficiencies using techniques we have learned in class. Again, there has been some work done in this area so I will likely be able to use others' work to assist in this portion of the project.

4. Metrics

The most important metric that I will be optimizing for will be overall runtime. This will be the focus of my research, reducing runtime. However, decreasing runtime will be in all cases a function of some other metric. Many times it is items such as communication or cache usage that are the biggest factors in

runtime. TAU will be able to assist me in determining which of these factors is influencing runtime the most. So if for a particular benchmark, there exists a lot of cache misses, my goal will then turn to reducing these misses in order to increase efficiency and reducing runtime. It is also possible that a benchmark is using message techniques that are less efficient that other techniques. I this case I can introduce non-blocking messages to help increase message passing efficiency. The most important metrics used will be runtime however this metric will depend on other metrics which will be the focus of my research.

5. Tuning and Analysis Utilities

Tuning and Analysis Utilities (TAU) is a suite of programs that allows a programmer to analyze software written in many different languages and using many different paradigms. In order to use TAU you must also have installed Program Database Toolkit (PDT), an application used to analyze source code. With PDT installed you must install TAU and configure it for your particular usage. Items noted in the configuration process are the compilers you wish to use for C and C++, or any other language as well as the different paradigms you would like to use. Here you can specify that you would like to analyze MPI and OpenMP applications. Once you configure TAU it will create Makefiles that are used to compile any source code you wish to analyze. It will create Makefiles that are used for compiling programs using different features. Such as one that is for MPI programs, another that is for OpenMP programs and finally another one that is used for compiling OpenMP + MPI programs. Then you must install TAU which compiles itself using the compilers you configured earlier.

Now when you would like to analyze an MPI application you must modify your Makefile such that instead of using your normal C compiler you know use a shell script included with TAU. You must also set the shell variable such that the script knows which configured Makefile to use to compile your application. Once all is configured for a particular application you can proceed to compile using the Makefile as normal. If you have setup everything correctly, TAU will first try and compile the program using the PDT compiler, this will help in getting better statistics about the execution of your program. Otherwise it will compile the program normally using the compilers you configured for TAU. The compilation process completes by linking your program with the TAU library. Finally the output of the make process can be thought of as your standard MPI program, you run it the same way you would any MPI program. The difference is any execution of the program will automatically create profile files about the run. In the case of an MPI program, a profile file will be created for each node used in execution. These files hold the necessary information to analyze your application.

Profiling OpenMP applications is not as easy. Again you must modify your Makefile for the program to link in the TAU library however you must also modify your source code. There is not automatic profiling that occurs; you must specifically inject sections into your source code to tell TAU where to start a profile timer and where to stop it. This is obviously more work than the automatic profiling of MPI programs. Another problem also arises when you want to stop profiling your OpenMP program and compile it normally. You must first remove all of the profiling sections you injected for TAU to see because your usual compiler will not recognize these. When you do get your OpenMP program linked with the TAU library, the executable it creates can be run just like any other OpenMP application. However, after being executed, profiling files are created just as they were for MPI programs. Again, these files contain the raw data that can be analyzed later.

The profiling files can be used an input to the utility that comes with TAU, paraprof. This application reads the data in the files and creates color coded graphs to visualize the execution data. Paraprof creates graphs per node in MPI programs and gives you data showing the percentage of runtime associated with each MPI routine used. OpenMP statistics are different in that you have to specify sections of source code to be profiled. Therefore runtime is broken down into these sections defined. This is the main functionality of paraprof however you can do many more things such as more in-depth data per node and call graphs.

6. Actual Approach

My intentions for this project were to link TAU with many benchmarks and get analysis for all of them before determining which benchmark would be best to tackle. I wanted to keep my potential benchmark candidates for improvement large so I could chose the most appropriate. However, this did not occur because of the problems I had with TAU.

It was difficult for me to get TAU installed and configured on our cluster because the default release does not accept our C compiler, mpicc, as a valid compiler. Therefore I had to modify the configuration file so that it would accept mpicc; otherwise the linking phase of using TAU would not work because I couldn't compile any MPI programs. This was the first obstacle that was difficult for me to get over as I have never before had to install an application in a Linux environment like this before.

Once I was able to get past the installations and configuration of TAU I know had to deal with the building of individual MPI and OpenMP programs. The modifications needed for compiling MPI programs were relatively easy compared to that of the OpenMP programs. For MPI applications usually only one modification was needed for the Makefile as well as specifying the correct TAU Makefile. The OpenMP

modifications to the Makefile were more significant and usually you have to reconfigure TAU before using with OpenMP programs. The problem arises that TAU has been configured to use mpicc for C compilation but now must use icc for use with OpenMP programs. Finally, as stated before, you must add specific profiling sections to your source code for TAU to profile the execution. I had envisioned TAU to be as easy to use as another analysis library we used earlier in class, mpiP. Unfortunately this was not the case and TAU gave me plenty of headaches.

Due to the problems I was having with TAU, I turned a good portion of my time to research. Since I was unable to use TAU to profile any programs early in the project, let alone my intentions of many, I researched potential benchmarks instead. I had found some information on the SWEEP3D benchmark and intended to follow this path. However I had also been given the advice to tackle writing an OpenMP version of the Strassen algorithm. Since this was something I could begin before being able to profile programs with TAU, I had some knowledge of from previous classes and it fell in line with some previous work in this class; I decided to begin by tackling this problem.

7. Strassen OpenMP Implementation

Looking through some literature about the Strassen algorithm and implementations of it the resounding theme was that it was hard to implement due to the large overhead. The Strassen algorithm works by decreasing the number of multiplications, which are costly, and replacing them with many additions and subtractions. The problem is that you must also now introduce seven extra matrices, each a quarter the side of the original matrices. This almost doubles the amount of memory needed to complete the matrix multiplication not including additional temporary matrices that are also needed. Since the Strassen algorithm is also defined recursively the amount of space needed can become quite large and swamp the memory.

Therefore I decided to not perform any recursive calls to the Strassen algorithm and instead rely on normal matrix multiplication after the first Strassen level. Essentially I would determine the helper matrices, M1 – M7 and then determine the values of the result matrix using normal matrix multiplication when needed as defined by the algorithm. This reduces additional memory usage cause by recursively using the Strassen algorithm.

The way in which I broke down the algorithm between multiple threads was by giving each thread one or more of the helper matrices to compute. So each thread had a copy of the A and B matrices and was assigned to determine some of the helper matrices as determined by the OpenMP implementation. Specifically I used the OpenMP pragma sections to define a group of items to be divided among the

threads and the calculation of each M matrix was a particular section. This method was extended further for determining the result matrix. So there was a barrier after the calculation of the M matrices and then each quadrant of the result matrix was divided among the threads. Again I used the OpenMP sections pragma as before.

For the multiplication of sub matrices needed in determining the M matrices I used an OpenMP tiled matrix multiplication as used previously in other matrix multiplications this semester. This is the typical three loop matrix multiplication with tiling added and then split among the OpenMP threads. However this was the main problem I found in my implementation. Since I was already in an OpenMP parallel section when using this matrix multiply routine, there was no need to parallelize this routine. In fact, parallelization here created performance problems.

Even with this change, the OpenMP implementation of the Strassen algorithm was not performing very well. Runtimes were significantly larger than that of a straight OpenMP three-loop implementation. This was attributed to the large overheard needed for the Strassen algorithm. I moved all of the temporary matrices inside specific sections in order to alleviate some of this overhead however this did not help much either. Therefore I turned to an OpenMP + MPI implementation of the Strassen algorithm, hoping to spread the memory overhead among nodes and threads rather than just threads.

8. Strassen MPI + OpenMP Implementation

As previously stated, the OpenMP implementation of Strassen's algorithm was not performing well and I was unsure of how to improve upon it. So I turned to an MPI and OpenMP implementation using some of the same ideas that I used in the OpenMP only version.

The first obstacle was deciding how to break up the problem between nodes the most effectively. My first idea was the break the problem into four nodes, one handling each quadrant of the result matrix. I determined this idea would not make use of anymore than five nodes (one to disperse the data and four for all quadrants) and each node would be performing some of the same calculations. This didn't seem like a viable solution so I then thought of having each node take one of the helper matrices M1 – M7, calculate the value and return it to the master node. At which point the master would use these values to create the result matrix. This is the idea I stuck with for implementation. It does have some flaws, which I will discuss later, but did end up producing some decent results.

One of the biggest problems I saw with any MPI Strassen implementation was that any node would have to know about the entire problem (both the A and B matrices) in order to do any work. This creates a

large amount of large messages that seemed unavoidable. Therefore the first step in this implementation included broadcasting the values of the A and B matrices to each node in the group. Then the next portion of the algorithm is different for the root and slave nodes.

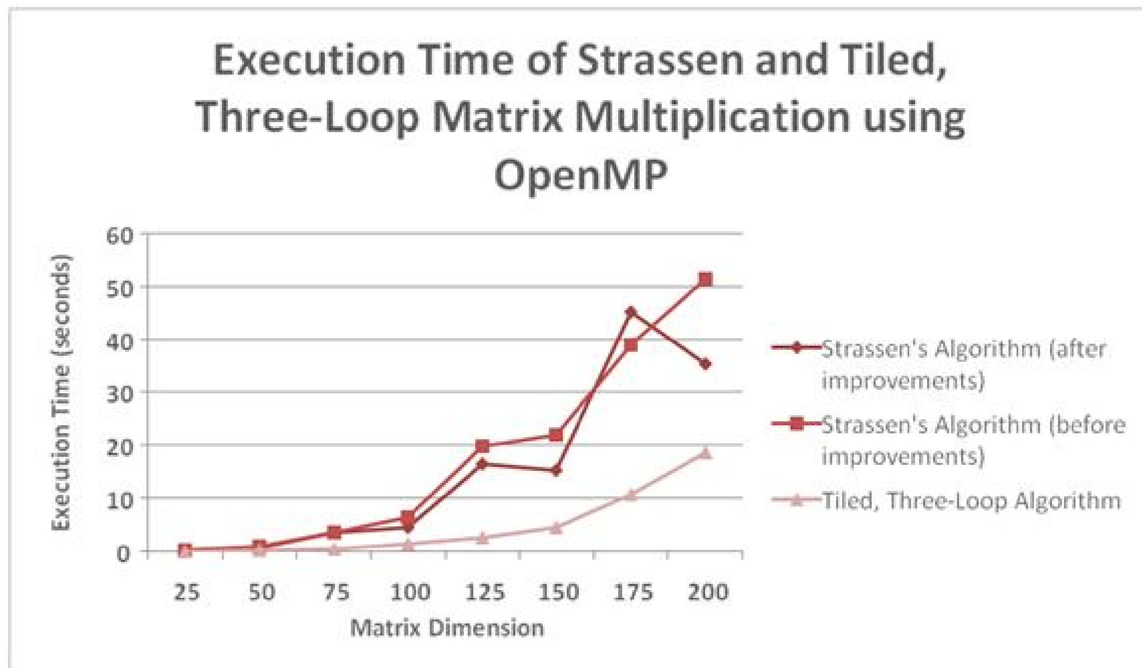
Slave nodes are using for the computation of M matrices. Each slave node determines which M node(s) to compute based upon its rank in the group and the size of the group. For example, for a group of 3 slave nodes, the 2nd node will calculate M2 and M5. There is no communication needed for the operation therefore this doesn't create extra overhead. During the calculation of each M value I again decided that too much overhead would be introduced if subsequent matrix multiplications would be calculated using the Strassen algorithm so I decided to use the OpenMP, tiled, three-loop algorithm. Once each node has finished calculating their M matrices they will send the value back to the root node.

For the root node as soon as the values of A and B have been broadcasted it must wait for the M matrices to be returned. My first idea for this communication was for the root to not care from where the M values would come, any slave node could send the value. The only problem is that each slave node would first have to tell the root which M matrix it is sending before it sends the actual value. This created an extra message for each data transfer and proved to not be the most efficient way. Instead I decided for the root to know exactly which nodes will send which values. Due to this change in technique I could now employ non-blocking receives and this proved to increase efficiency fairly well.

As soon as the root receives all of the M matrices it can now begin determining the final result of the matrix multiplication. For this portion of the implementation I decided to make use of the OpenMP ideas I had used in the previous implementation. I again used the OpenMP pragma sections to give each thread a different portion of the final result calculation. One idea I had for this calculation was to help OpenMP out a little. Since there will be two threads and four quadrants to calculate I thought it would make sense to combine these quadrants into halves. Now each thread could take one half of the calculation and I could group the half by the M matrices it used. I tried this tweak out and it actually proved to not help in execution at all. Therefore I reverted back to the original idea.

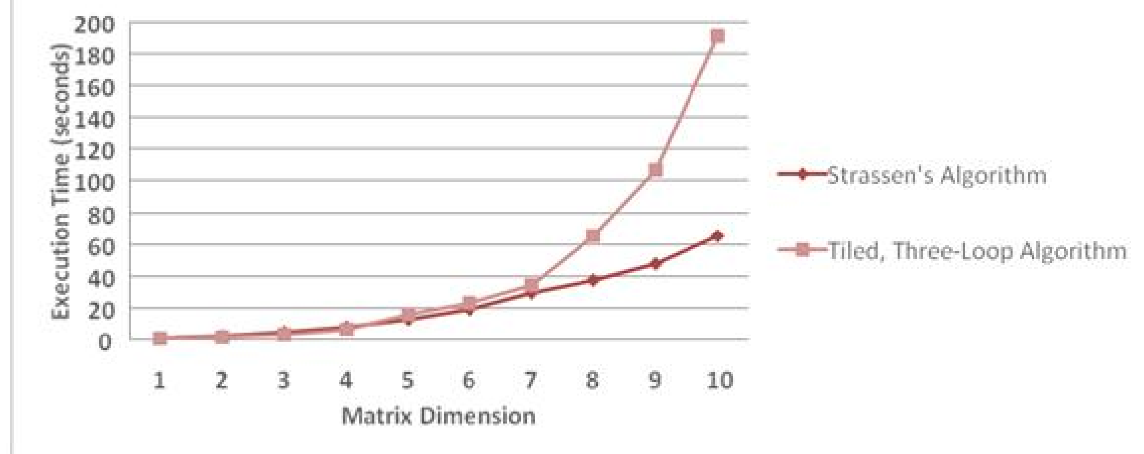
9. Results

As I stated earlier, the OpenMP implementation of the Strassen algorithm did not perform very well. I tried some tweaks, some of which were effective, however I could only improve it so much and therefore turned to an MPI + OpenMP implementation. The following are the results I achieved with the Strassen algorithm (before and after improvements) compared to the standard OpenMP three-loop algorithm.



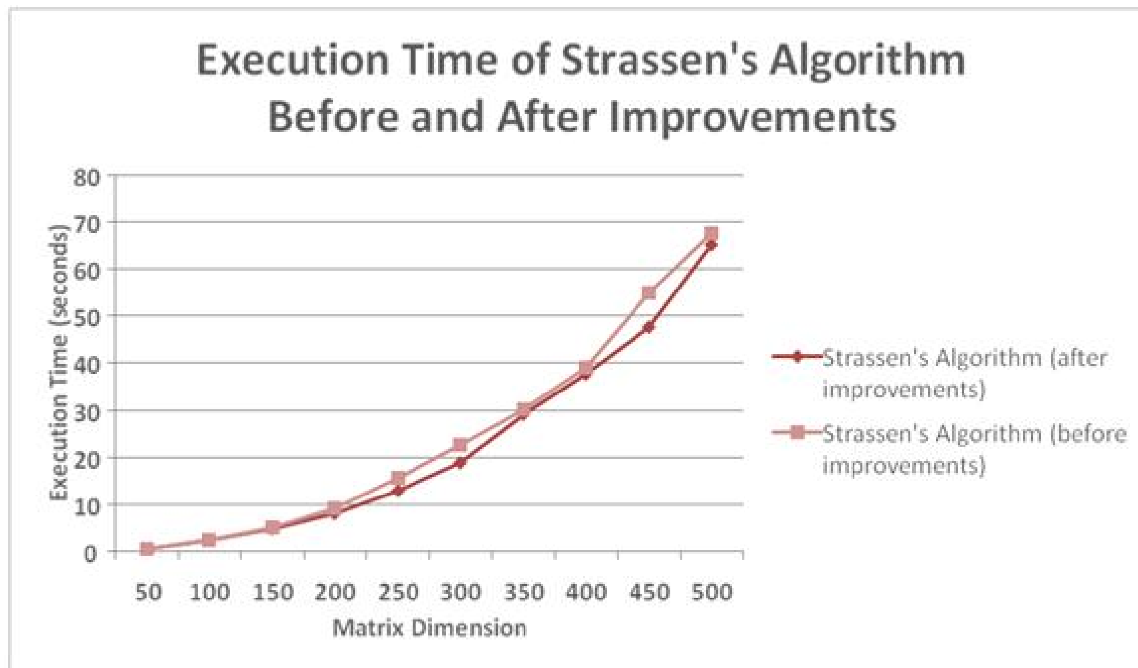
As the graphs shows, the improvements I was able to make did improve the execution time but the overhead created by the Strassen algorithm still kills this algorithm's execution time. The shortcomings induced by extra memory usage are magnified as the matrix dimension increases. Therefore the standard three loop tiled version using OpenMP would be a better choice in this instance.

For the MPI + OpenMP Strassen algorithm, the results prove to be quite different. My hunch that dispersing some of the calculations and the extra memory needed across nodes would reduce the bad effects of the Strassen algorithm proved valid. In this case I compared the MPI + OpenMP Strassen implementation with the fastest matrix multiplication algorithm I had previously written for a class assignment. This particular algorithm used the basic three-loop algorithm but split the work among both nodes and threads on each node, hence also using MPI and OpenMP.



As you can see here from the graph, both Strassen's and the typical three loop method have very similar runtimes with small matrices. Even with the smallest matrices the three-loop method slightly out performs the Strassen algorithm. However, as the matrix dimension increases, the Strassen implementation begins to show its superiority. This makes good sense because all of the literature I found about the Strassen algorithm insinuated that the overhead it induces is only overcome by large matrices.

The following graph shows the result of my improvements on the Strassen implementation. Unfortunately this does not include all of the improvements I made because I cannot recall all of them. However this does include the most significant one in which I modified the message passing protocol and reduced the number of messages sent. However the main item that hinders performance for this implementation is the need for all nodes in the system to have all the information about the input matrices, A and B. I originally used MPI_Bcast to broadcast the value of these matrices and then later tried using a system of non-blocking point-to-point sends and receives, hoping to gain performance. Instead I realized that MPI_Bcast was the most efficient way to send this data as I assume it has been tweaked especially for this case.



10. Future Work

Although this implementation did produce some good results for large matrices it does have some problems. First off is that it is not scalable. The current implementation uses a maximum of 8 nodes (1 as the root and 7 other for all of the M matrices) therefore introducing extra nodes will not help in any way. There is no easy fix for this downfall that I can think of other than completely changing the algorithm. One idea I have is to take a similar approach to my first idea. This idea was to break the result matrix into quadrants and assign each to a slave node. This idea could be extended in a recursive manner such that each node that is in charge of a quadrant of the main result node can now assign work to other free nodes in the system. These additional nodes could either be responsible for calculating M matrices or performing recursive Strassen calls. Either method might prove to render better results than my current limit.

Another pitfall of this implementation is the large amount of data that needs to be transferred. Specifically the need to broadcast all the input matrices to each node. This is another problem that would need to be handled via a major change in the algorithm. It is possible that the aforementioned possible algorithm described to help in scalability may also help in the large message sizes. Now instead of every node needing to know the entire input matrices only four need to. All other matrices can be limited to know only portions of the main problem.

This algorithm described seems to be a valid direction to take the Strassen matrix multiplication implementation. It could help with some of the known problems with my current implementation but further research would need to be done. This algorithm may help with scalability but end up adding more messaging than accounted for.

11. Conclusion

Although I had originally planned on attacking two different benchmarks, the biggest challenge for me was the use of TAU. As soon as I was able to get it working for one application I could not get another program to compile. Specifically I could never get TAU to link with programs written in FORTRAN. Therefore I placed most of the focus on the Strassen implementation, specifically using OpenMP and MPI. This was the easiest and most effective way for me to get information about execution using TAU. Strassen is also familiar to me from my algorithms class and proved to be a challenge during the implementation phase. It took me a while to figure out the best way to convert an algorithm defined recursively using divide and conquer method into a parallel application in which dependencies must be reduced. I ended up with what I think is the simplest solution as usually simplicity is superior.

In result I was able to implement an MPI + OpenMP implementation of the Strassen's algorithm that out performs the other matrix multiplication algorithms we have worked on in class for large matrices. I was able to overcome the challenges of mapping a recursive algorithm to a parallel one as well as the overhead associated with Strassen's algorithm.

12. References

ASCI Purple Benchmarks. http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html, 2006

Choi, Jaeyon. Dongarra, Jack J.. Walker, David W.. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. 1993.

Grayson, Brian and Shah, Ajay Pankaj. A High Performance Parallel Strassen Implementation. 1995.

Jaydeep Marathe, Anita Nagarajan, Frank Mueller. Detailed Cache Coherence Characterization for OpenMP Benchmarks. ACM. 2004.

Program Database Toolkit (PDT), http://www.cs.uoregon.edu/research/pdt/home.php, 2006

Strassen Algorithm, http://en.wikipedia.org/wiki/Strassen_algorithm, 2006

Samit Jain, Matthew Crocker, Nasim Mahmood and James C. Browne. Productivity and Performance Through Components: The ASCI Sweep3D Application

TAU supports OpenMP and OpenMPI (OpenMP+MPI) now. http://www.cs.uoregon.edu/research/paracomp/tau/tauprofile/images/openmpi/. 2006.

Tuning and Analysis Utilities (TAU), http://www.cs.uoregon.edu/research/tau/home.php, 2006