Project Report

SOFTENG 751

Software Engineering - University of Auckland

8. June 2017

Stud. no.: 4566036	Name: Waleed Al-saady
Stud. no.: 681307586	Name: Xiang Tianyu
Stud. no.: 924997367	Name: Kristoffer Sloth Gade

Contents

Table of Contribution

Name	Contribution
Waleed Al-saady	1/3
Xiang Tianyu	1/3
Kristoffer Sloth Gade	1/3

1 Introduction

Todays computers are facing high demands from the users, the computers always have to be faster. The performance of the computers have to efficiently use the computer power. A way to access high performance computing is by the use of parallel programming. When parallelizing tasks, the computer is able to compute more tasks in less time, this is a good and efficient way of using multi-core processors. To make the best out of parallel programming, analysis of the code is a good way to find out where to optimize the application. Analysis can be hard to do, hard to get a good overview of the flow throughout the application. This is where the use of parallel programming visualization comes in handy. By having a way to visualize the flow through the code, it makes it easier to get an overview and look for bottlenecks in the code. This is important because if we can eliminate bottlenecks in the code, then we will have a more responsive and faster application for the user.

This report will contain the documentation of the project process for group 20 in SOFTENG 751. The report will include some background for the subject, a description and the implementation of the application that is being analyzed, will be shown in chapter 4. A walkthrough of how to extract the trace file, used for visualizing the flow, will be in chapter 5. Chapter 6 will contain the visualizing and analysis of the application through the trace file, and at last in chapter 7, a conclusion on the project.

2 Background

Parallel computation is more complex than a sequential program. Textual presentation of data describing the execution of a parallel program can be very difficult. So, a graphical visualization tool is becoming useful and powerful to help us understanding complex tasks. The purpose of a visualization may be to debug a program, to evaluate and optimize performance, or to provide a form of electronic documentation by means of data structure display, algorithm animation, or other graphical display that conveys knowledge of the functioning of a program, or an algorithm [Kraemer and Stasko, 1993]. Generally, visualization of a program can be divided into two steps, collecting the data and display the data.

2.1 Data Collection

Any instrumentation of data collection will to some degree perturb the performance of the system. The extent of this perturbation call probe effect. The probe effect is determined by the volume of data collected, the monitoring hardware, system software, system architecture, and programming paradigm.

There are different ways to collect information from the program. Data may be collected either by tracing or by profiling. A profile provides an inventory of performance events and timings for the execution as a whole. This ignores the chronology of the events in an absolute sense. Nothing is timestamped and the resulting report does not say what events happened before other events in an absolute sense. A trace records the chronology, often with timestamps. It preserves temporal and spatial relationships between events. The amount of data in the trace increases with the runtime. So, the trace file can become very large. Either profiling or tracing is a post-mortem basis, which means we examine and analyze a profile or a trace file after the program execution. Most of performance analysis tools use this mechanism[Nagel and Arnold, 1994].

The instrumentation can be in different levels. The tracing of hardware-level events on a parallel machine may be either centralized or distributed. In a centralized scheme, a single monitoring system collects all data, whereas in a distributed scheme, events are captured locally at each processor. A hardware data collection system will generally have less effect on the program than a software one. However, not all systems have monitoring hardware in place, nor can all events of interest be easily captured at a hardware level.

Software instrumentation involves the placement of small piece of code in the operating system, the run-time system, or in the application program. Instrumentation of the operating system can be used to collect information such as message sends and receives, process creation, scheduling, and termination, context switching, memory access, and system calls. Instrumentation of the runtime environment can provide information regarding the state of runtime queues, the acquisition and release of locks, entry to and exit from parallel sections, arrival at and departure from barriers. Instrumentation at the application program level can provide the association between operating or runtime system events and particular portion of the source code. It can provide visualizations that are detailed enough to be used for debugging purpose, allowing the viewer to verify the correct execution of a portion of the program.

Insertion of interesting event code can be manually or automatically. There are various libraries that provide the parallel programming functions that can be used for manually produce trace data, but it needs large effort and difficult to manage. We can also use some tools to automatically modifies the program to produce trace data. Or some other tool doesn't need to modifies user's program, all instrumentation is automatically inserted at compile time. A number of these systems support instrumentation at several levels. They can integrate the data collection at both the user level and the operating or runtime system level.

2.2 Data Display

There are numbers of tools can be used to visualize the program since we obtained the traces. Different tools provide different visualization features at different levels. The followings are some generic displays that most of these tools can provide.

2.2.1 Program Graph

Program graph displays process histories as event streams on a time grid. Each row represents the sequential stream of events for a single process. Each column represents an interval of time, during which the events in different processes may occur concurrently. Figure 2.1 is an example of program graph[TUDresden, 2016].

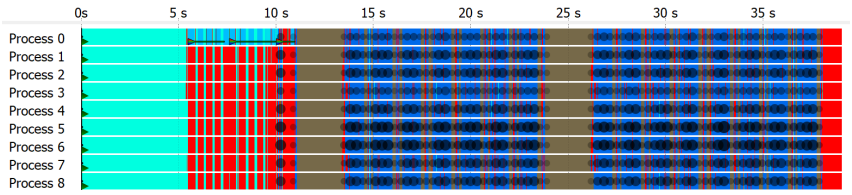

Figure 2.1: *Program Graph*

2.2.2 Communication Graph

Communication graph provides information about communication events. In the Figure 2.2, Messages exchanged between two different processes are depicted as black lines[TUDresden, 2016]. The leftmost (starting) point of a message line and its underlying process bar therefore identify the sender of the message, whereas the rightmost position of the same line represents the receiver of the message.

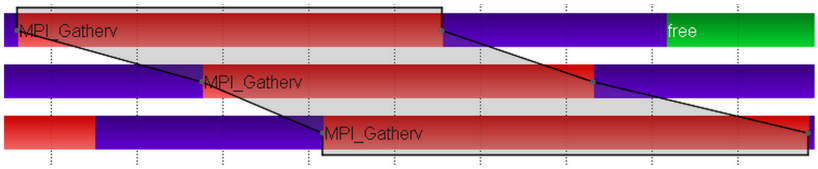

Figure 2.2: *Communication Graph*

2.2.3 Statistical Display

Many performance tools also provide displays of statistical information. Figure 2.3 is a statistical chart that shows the accumulated time consumption across different functions[TUDresden, 2016]. A comparison between the different functions can be made and dominant functions can be distinguished easily.

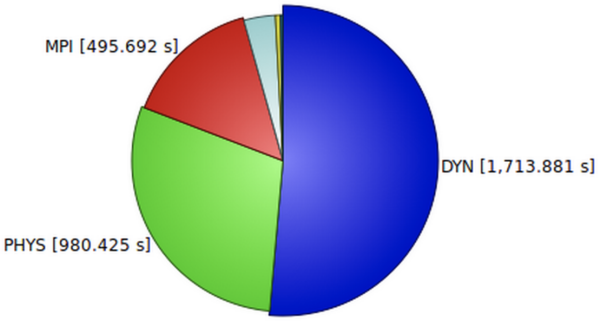

Figure 2.3: *Statistical Chart*

There are also many other application-specific displays. We can choose different tools according to our needs.

3 Different Visualization Tools

Vampir

Vampir provides a manageble framework for analysis, which enables developers to quickly display program behavior at any level of detail. Detailed performance data obtained from a parallel program execution can be analyzed with a collection of different performance views. Intuitive navigation and zooming are the key features of the tool.

VampirTrace is the code instrumentation and run-time measurement framework for Vampir. During a program run of an application, VampirTrace generates an OTF frace file, which can be analyzed and visualized by Vampir.

The OTF (Open Trace Formats) have been designed as well-defined trace formats with open, public domain libraries for writing and reading. This open specification of the trace information enables analysis and visualization tools like Vampir to operate efficiently at large scale.

Jumpshot

Jumpshot is a Java-based visualization tool for doing postmortem performance analysis. Jumpshot-4 is the latest visualization program for the trace file format SLOG-2, which provides a hierachical

structure to store a large number of drawable objects in a scalable and efficient way for visualization[Moore, 1999].Jumpshot is freely available and is packaged with the MPICH portable implementation of MPI.

The tool provides high-level abstraction of the details without reading in huge amount of data into the graphical display engine. It can present multiple views of trace file data. The primary view is a series of timelines, one for each process, showing with colored bars the state of each process at each time. This view can be zoomed and scrolled for close examination of specific times. Other views include histograms of state durations and a "mountain range" view showing the aggregate number of processes in each state at each time.

ParaGraph

ParaGraph is a graphical display tool for visualizing the behavior and performance of parallel programs that use MPI (Message-Passing Interface).The visual animation of a parallel program is based on execution trace information gathered during an run of the program on a message-passing parallel computer system[Heath and Finger, 2003].

ParaGraph is written in C. It combines the functionality of instrumentation and visualization. ParaGraph is an interactive, event-driven program. Its basic structure is that of an event loop and a large switch that selects actions based on the nature of each event. It provides diffrent kinds of displays to depict to parallel program. Utilization diplays are concerned promarily with processor utilization. Communication displays are concerned primarily with depicting interprocessor communication. Task displays are provided that use information supplied by user, with the help of MPICL, to decipt the portion of user's parallel program that is executing at any given time.

TAU

TAU Performance System is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, UPC, Java, Python. TAU (Tuning and Analysis Utilities) is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements as well as event-based sampling. TAU's profile visualization tool, paraprof, provides graphical displays of all the performance analysis results, in aggregate and single node/context/thread forms. In addition, TAU can generate event traces that can be displayed with the Vampir, Paraver or JumpShot trace visualization tools[of Oregon, 2016].

4 Application

For the application to analyse, it was chosen to implement the sorting algorithm mergesort, which can be parallelized by sorting different int arrays. The reason mergesort was chosen is because it is a relatively simple algorithm which will take some computing power and time to finish, and therefore can be parralelized to optimize. To implement the program a paratask project was created, and a new paratask class was made. Additionally we added the *PARCutil.jar* file and *ptruntime.jar* into the build path. The mergesort algorithm that was implemented was adapted from an online source [Vogel, 2009]. In our first program ParaTaskExample.ptjava, we decided to sort 10 arrays of a large size, initializing them with random numbers that are randomized within the full range of possible int values (negative to positive). These arrays were

then parallelized using ParaTask with the number of tasks set to 10, and the Parallel Iterator was used to allow the ParaTask to grab each int array from the ArrayList of int arrays, without any race conditions. This allowed the efficient storage of the int arrays into one ArrayList, and the ParaTask was able to serve each thread starting a new task a single uncontentested and guaranteed int array with the help of the Parallel Iterator.

The Parallel Iterator has inbuilt checking mechanisms to make sure that a thread is served one item from the ArrayList at a time. When the hasNext() function is called on the Parallel Iterator object, it will reserve one item (the int array) specifically for that thread and no other thread can access that item. The thread can then access it's assigned item (int array) with the next() method. Meanwhile when other threads call the hasNext() function another item will be reserved specifically for them. In this way when the hasNext() method is called ten times, each item is correctly assigned to the threads calling the method (and the threads can be 10 or 4 or any number).

ParaTask is a powerful tool created by the Parallel and Reconfigurable Computing Lab in the University of Auckland. It allows greatly streamlined and efficient parallelization of different task types including one off tasks, I/O (interactive) tasks and multiple tasks (specifying either a number or *). In our program we chose to use ParaTask for our sorting method with an argument of 10 to specify the number of arrays. In a slightly modified version ParaTaskExampleArraysParallelised.ptjava, we also parallised the array creation using ParaTask.

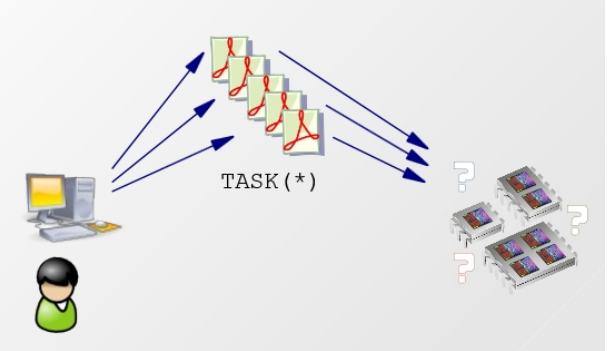

Figure 4.1: *ParaTask*

Next we decided to create a program with some dependencies. To achieve this we created 6 arrays which were to be sorted and then underwent some processing tasks whereby some tasks had a dependency on others before they could start. The array creation and sorting was done in parallel using ParaTask. After the first sorting (sortId0) was finished, it was then processed. The first array was processed by adding to each entry the next item like so: array[i] = array[i]/2 + array[i+1]/2. Then the next three processing tasks (processId1-processId3) had a single dependency on the first task (processId0), this was implemented by using ParaTask's dependsOn keyword shown in listing 4.1. The first four tasks all use the method paraTaskProcess.

Listing 4.1: *Three tasks with a single dependence*

```
1 TaskID<Void> processId1 = paraTaskProcess(intArrays.get(1),
2     intArrays.get(0), message) dependsOn(processId0);
3 ...
4 TaskID<Void> processId2 = paraTaskProcess(intArrays.get(2),
5
6     intArrays.get(0), message) dependsOn(processId0);
7 ...
8 TaskID<Void> processId3 = paraTaskProcess(intArrays.get(3),
9     intArrays.get(0), message) dependsOn(processId0);
```

The processing involves some calculations with the first array, hence it depends on the first array to finish its processing (in processId0). The last two processing tasks depend on two prior tasks, processId4 depends on processId1 and 2, and processId5 depends on processId2 and 3, shown in listing 4.2.

Listing 4.2: *Two tasks with two dependencies*

```
1 TaskID<Void> processId4 = paraTaskProcess2(intArrays.get(4),
2 intArrays.get(1), intArrays.get(2), message)
```

```
3  dependsOn(processId1,processId2);
4  ...
5  TaskID<Void> processId5 = paraTaskProcess2(intArrays.get(5),
6  intArrays.get(2), intArrays.get(3), message)
7  dependsOn(processId2,processId3);
```

The last two tasks that have double dependencies use the method paraTaskProcess2.

This is the Directed Acyclic Graph (DAG) for all of our Task dependencies.

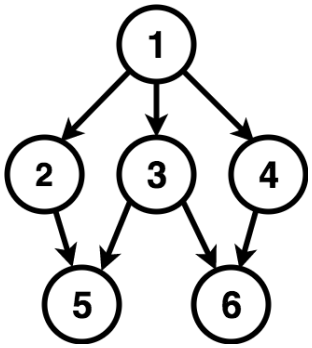

Figure 4.2: *DAG showing dependencies*

5 Creating the Trace File

To create the trace file we tried many different methods, most of which were unsuccessful. A previous student's ParaTask logging project was supplied to us. This provided some insight but unfortunately since it was slightly dated it could not be opened. We decided that we should focus on the exact method to create trace files since that is our main requirement for the project. We tried different approaches like JRastro which is a tracing library for java using the JVMTI interface[Schnorr, 2012]. JRastro produces trace files of the PAJE format and runs on linux. This proved unsuccessful. We also downloaded the OTF library which featured methods in order to write an OTF file[TUDresden, 2014]. This was only available in C and Python. An idea was to implement it in C and once it's working log ParaTask to get the required measurements and format and then feed that into the C program. This did not work. We then tried to use VampirTrace which produces OTF trace format files[TUDresden, 2013a]. It claims to be able to trace Java programs via the JVMTI and also requires linux. Unfortunately there was very little instructions for this, only one paragraph on page14 is provided for tracing Java in the whole manual [TUDresden, 2013b]. It took us considerable time to figure out how to run VampirTrace on linux to be able to instrumentalize our code. In the end we found a successful way which was as follows:

1. Firstly download the VampirTrace library [TUDresden, 2013a]
2. Run the following code in a terminal (note each will take some time):

Listing 5.1: *commands for VampirTrace*

```
1  ./configure --prefix=/where/to/install
2  make all install
```

3. Once it is installed you compile the java program as follows (the libs directory must contain the jar files for ParaTask, PARCUtils and ParIterator):

Listing 5.2: *Compiling the Java program*

```
1 javac -cp .:./libs/*  -Xlint:unchecked ParaTaskWithDependency.java
```

4. Now run the class file:

Listing 5.3: *Tracing the Java program*

```
1 vtjava -cp .:./libs/* -Xms512M -Xmx4096M ParaTaskWithDependency
```

This will run the program and generate the trace files in the same directory. Once the OTF file and its various other files are created, we can open it using the Vampir program which is a visualisation tool that requires the OTF trace format to visualize our program.

6 Visualization

The visualization is where the analysis and visual overview of the application is taking place. In this chapter it will be described how the visualization of the Mergesort application can help in understanding the flow of the application code. To visualize the flow, the tool Vampir is used. Vampir has three main sections in the GUI, *Master Timeline, Process Timeline* and *Function summary*. These sections are used to get an understanding of the application. Vampir is capable of more than is used, in the report the main functions will be described.

To visualize a parallel program the first thing that is needed is a tracefile from the application under analysis. How to get a tracefile is described in chapter 5

Master Timeline shows all tasks done during the lifetime of the application. As seen in figure 6.1 the different processes are shown and have different colors. The colors indicate the group membership of the method the method family [TUDresden, 2016]

Process Timeline can be used to show an individual process and it's methods. The process is divided into more levels, which is the different callstacks [TUDresden, 2016].

Function summary shows how much time spent in each function. Here it can be seen if the program is using more time within a function than expected. On figure 6.6 it can be seen that 197,3 seconds are in total spent in *workerTakeNextTask()*

We will now discuss the results of tracing and visualising our example program. The first two programs (ParaTaskExample.ptjava and ParaTaskArraysParallelised.ptjava) can be considered earlier versions of the program. Their trace files can be found in the project's Github repository if needed. We will focus on the trace results of our final program ParaTaskWithDependencies.ptjava. The master timeline from Vampir in Figure 6.1 below shows the full trace of the

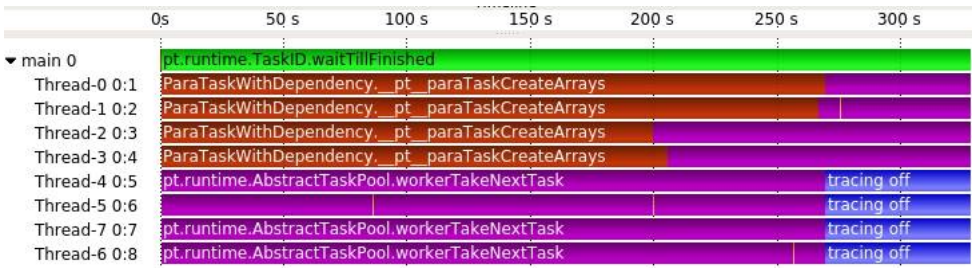

Figure 6.1: *Master timeline for Mergesort*

program. There are a total of 8 threads created. In threads 0 to 3 this is where the actual program starts. First we see calls to __pt__paraTaskCreateArrays - two calls in threads 0 and 1, and one call in threads 2 and 3; hence it is running in parallel as expected. As described in earlier chapters, this program creates six int arrays of size 40000000 with randomized numbers. This method is created through ParaTask (which appends __pt__) and corresponds to the original paraTaskCreateArrays method found in the ParaTaskWithDependencies.ptjava program code.

After the arrays are created in threads 0 to 3 there are dozens of methods calls to a ParaTask method called pt.runtime.AbstractTaskPool.workerTakeNextTask. This is also the case for most of threads 4 to 7. After analysing the threads in depth, we found that this method is called excessively and usually consecutively. This method belongs to the ParaTask and is used in its inner workings.

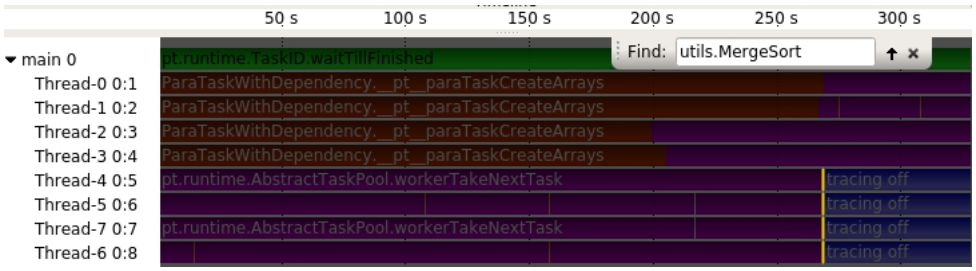

Figure 6.2: *search result for MergeSort class*

When searching for the sorting methods, we found that they are concentrated into one small window of time in threads 4 to 7 shown in Figure 6.2 as yellow bars (which is the search result for the whole MergeSort class). They start soon after array creation. As for their ending time, the sorting did not finish this quickly when we ran it, and the full duration will include the calls from the ParaTask library. Nevertheless, we can see that the sorting is proceeding in parallel.

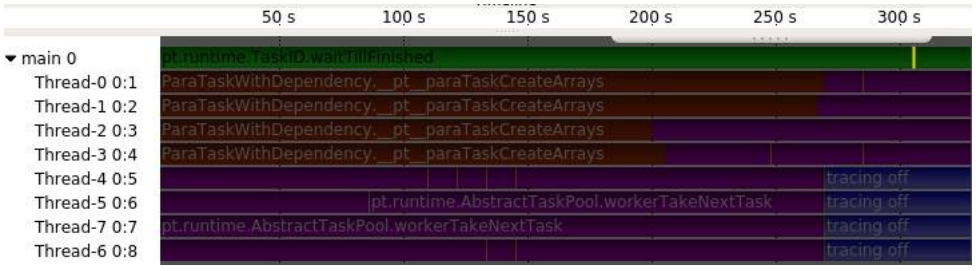

Figure 6.3: *search result for the method paraTaskProcess*

Next we analysed the two methods where processing takes place. First we search for the method paraTaskProcess, results are shown in Figure 6.3. The calls are made after the 300second mark. The calls are made in the main process and not in the threads. This may be related to the fact that we use the dependsOn keyword here. We can confirm from the program's output in Figure 6.4 that processing does occur in parallel.

```
Array with Task ID: 9 has been sorted by thread number: 5
Now processing Array with Task ID: 13
Array with Task ID: 13 has been processed by thread number: 5
Now processing Array with Task ID: 16, this task depends on Task 13 which has completed.
Now processing Array with Task ID: 14, this task depends on Task 13 which has completed.
Array with Task ID: 16 has been processed by thread number: 5
Now processing Array with Task ID: 15, this task depends on Task 13 which has completed.
Array with Task ID: 14 has been processed by thread number: 4
Array with Task ID: 15 has been processed by thread number: 5
```

Figure 6.4: *Program's output shows that processing is in parallel*

Figure 6.5 shows the search result for the second processing method paraTaskProcess2. Similarly it occurs in the main process and towards the end.

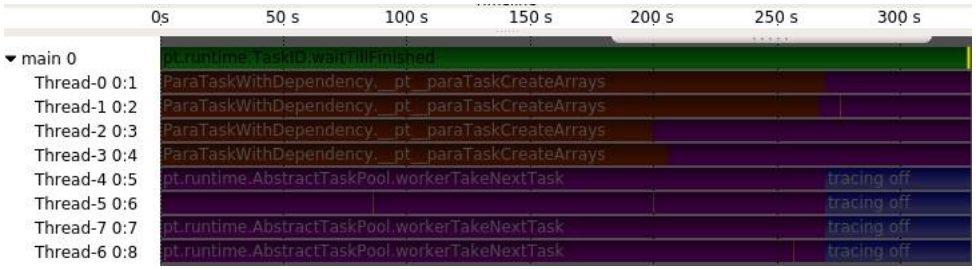

Figure 6.5: *search result for the method paraTaskProcess2*

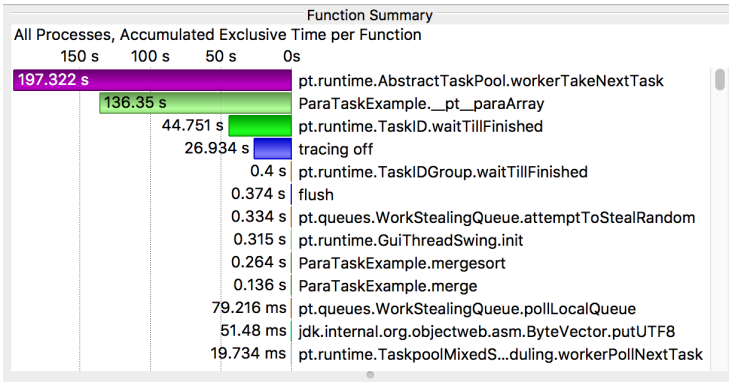

Figure 6.6: *Function summary for Mergesort*

Figure 6.6 shows the most called methods. We would like to mention the limitation that when running the program in linux, array creation is much slower than sorting and processing, which was not the case when running the program in Windows using Eclipse.

7 Conclusion

To keep up with the demand of fast computers, analysis of parallel applications is a great way to understand the flow through the code. It was discovered that by using visualization tools like VAMPIR, it was possible to get better overviews of the parallelism, and find out where an application could be improved, such as removing bottlenecks. By using VampirTrace and VAMPIR at was possible to track how much time the mergesort used in the different functions. By visualizing it is now easier to point out, where in the code an optimization has to be added. Time and effort spent on maintaining a system can be minimized, by debugging using visualization tools, in addition to traditional debugging.

We have discovered that VampirTrace is able to intelligently trace some methods like paraTaskCreateArray. However there were some ParaTask methods whose full duration was not revealed clearly, but their starting time was shown. This may be attributed to either limitations of VampirTrace's compiler tracing or to the inner workings of the parallel program. In any case we believe the tracing of parallel programs in this way still provides a lot of valued information. VampirTrace has filtering options which may be used to optimise the trace.

References

M. T. Heath and J. E. Finger. Paragraph: A performance visualization tool for mpi. 2003.

E Kraemer and J T Stasko. The visualization of parallel systems: An overview. 18(2):105, 1993. doi: http://dx.doi.org/10.1006/jpdc.1993.1050.

Ross Moore. Jumpshot-4 users guide, 1999. URL http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/jumpshot-4/node36.html.

W. E. Nagel and A. Arnold. Performance visualization of parallel programs - the parvis environment. *Intel Supercomputer Users Group Conference*, pages 24–31, 1994.

University of Oregon. Tau user guide, 2016. URL https://www.cs.uoregon.edu/research/tau/tau-usersguide.pdf.

Lucas Schnorr. Jrastro github and instructions, 2012. URL https://github.com/schnorr/jrastro.

TUDresden. Vampirtrace main page with download, 2013a. URL https://tu-dresden.de/zih/forschung/projekte/vampirtrace?set_language=en.

TUDresden. Vampirtrace manual, 2013b. URL https://tu-dresden.de/zih/forschung/ressourcen/dateien/laufende-projekte/vampirtrace/dateien/VT-UserManual-5.14.4.pdf.

TUDresden. Otf library and api, 2014. URL https://tu-dresden.de/zih/forschung/projekte/otf.

TUDresden. Vampir 9.2 manual, 2016. URL https://www.vampir.eu/tutorial/manual.

Lars Vogel. Mergesort in java - tutorial. http://www.vogella.com/tutorials/JavaAlgorithmsMergesort/article.html, 2009. Accessed: 2016-05-25.