

Programming of Supercomputers  
Group Assignment2 Report  
Parallel Debugging and Performance Analysis

Mihai-Gabriel Robescu (03709201)  
[mihai.robescu@tum.de](mailto:mihai.robescu@tum.de)

Yi JU (03691953)  
[yi.ju@tum.de](mailto:yi.ju@tum.de)

Hsieh Yi-Han (03708688)  
[kimihsieh.tw@tum.de](mailto:kimihsieh.tw@tum.de)  
Koushik Karmakar (03704659)  
[koushik.karmakar@tum.de](mailto:koushik.karmakar@tum.de)

27-11-2018

### 3.1 Task 1 (28 points total)

In this task, investigate and describe briefly in the report the following concepts:

#### 1. Race condition (1 point)

Ans. Race condition is a behavior that the output of a system or a process depends on the sequences or the timing of uncontrollable events appearing. This word can be dated back to that two signals try to compete with each other to influence the priority to output results.

#### 2. Deadlock (1 point)

Ans. In parallel computing, a **deadlock** is a state in which each member of a group is waiting for some other member to take action, such as sending a message or more commonly releasing a lock. Deadlock is a common problem in multiprocessing systems, parallel computing, and distributed systems, where software and hardware locks are used to arbitrate shared resources and implement process synchronization.

#### 3. Heisenbug (observer's effect) (1 point)

Ans. Heisenbug is a software bug that may change its behavior or even disappear when people attempt to study and or debug it.

#### 4. Cache coherency and false sharing (1 point)

Ans. **Cache coherence** is the uniformity of shared resource data that ends up stored in multiple local **caches**. When clients in a system maintain **caches** of a common memory resource, problems may arise with incoherent data, which is particularly the case with CPUs in a multiprocessing system.

**False sharing** is a performance-degrading usage pattern that can arise in systems with distributed, coherent caches at the size of the smallest resource block managed by the caching mechanism. When a system participant attempts to periodically access data that will never be altered by another party, but those data shares a cache block with data that *are* altered, the caching protocol may force the first participant to reload the whole unit despite a lack of logical necessity. The caching system is unaware of activity within this block and forces the first participant to bear the caching system overhead required by true shared access of a resource.

#### 5. Load imbalance (1 point)

Ans. Load imbalance means the imbalanced distribution of workloads across multiple computing resources, such as computers, a computer cluster, network links, central processing units, or disk drives.

#### 6. Amdahl's law (1 point)

**Ans.** In computer architecture, **Amdahl's law** is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved.

Amdahl's law can be formulated in the following way:

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

where  $S_{latency}$  is the theoretical speedup of the execution of the whole task;  $s$  is the speedup of the part of the task that benefits from improved system resources;  $p$  is the proportion of execution time that the part benefiting from improved resources originally occupied. Furthermore,

$$\begin{cases} S_{latency}(s) \leq \frac{1}{(1-p)} \\ \lim_{s \rightarrow \infty} S_{latency}(s) = \frac{1}{(1-p)} \end{cases}$$

shows that the theoretical speedup of the execution of the whole task increases with the improvement of the resources of the system and that regardless of the magnitude of the improvement, the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement.

#### 7. Parallelization overhead (1 point)

Ans. Parallelization overhead is any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task during parallelization.

#### 8. Floating-point arithmetic challenges (4 points)

##### (a) Comparisons

Due to rounding errors, most floating-point numbers end up being slightly imprecise. As long as this imprecision stays small, it can usually be ignored. However, it also means that numbers expected to be equal (e.g. when calculating the same result through different correct methods) often differ slightly, and a simple equality test fails.

##### (b) Definition of a zero and signed zeros

In computing, some number representations allow for the existence of two zeros, often denoted by  $-0$  (negative zero) and  $+0$  (positive zero), regarded as equal by the numerical comparison operations but with possible different behaviors in particular operations. The IEEE 754 standard for floating-point arithmetic (presently used by most computers and programming languages that support floating point numbers) requires both  $+0$  and  $-0$ . Real arithmetic with signed zeros can be considered a variant of the extended real number line such that:

$$\frac{1}{+0} = +\infty; \frac{1}{-0} = -\infty$$

##### (c) Cancellation or loss of significance

Cancellation or loss of significance is an undesirable effect in calculations using finite-precision arithmetic such as floating-point arithmetic. It occurs when an operation on two numbers increases relative error substantially more than it increases absolute error, for example in subtracting two nearly equal numbers

(known as catastrophic cancellation). The effect is that the number of significant digits in the result is reduced unacceptably. Ways to avoid this effect are studied in numerical analysis.

#### (d) Amplification and error propagation

In statistics, propagation of error is the effect of variables' errors on the error of a function based on them.

### 3.1.1 Questions

**1. Which of the concepts affect performance but not correctness? (2 points)**

Ans. Deadlock; Heisenbug; Load imbalance; Amdahl's law; Parallelization overhead

**2. Which of the concepts affect the correctness of the application? (2 points)**

Ans. Cache coherency and false sharing; Floating-point arithmetic challenges

**(a) Of these, which are exclusive to parallel programming? (1 point)**

Cache coherency and false sharing

**(b) Of these, which are not exclusive to parallel programming? (1 point)**

Floating-point arithmetic challenges

**3. Which of them can occur in OpenMP applications? (1 point)**

Ans. Race condition ; Deadlock ; Heisenbug; Amdahl's law; Parallelization overhead; Cache coherency and false sharing; Floating-point arithmetic challenges

**4. Which of them can occur in MPI applications? (1 point)**

Ans. Deadlock ; Heisenbug; Load imbalance; Amdahl's law; Parallelization overhead; Floating-point arithmetic challenges

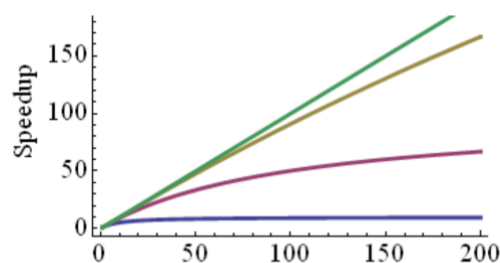
**5. Is cache coherency necessary on MPI applications with a single process and a single thread per rank? Explain. (3 points)**

Ans. No. Because cache coherence is the uniformity of shared resource data that stored in CPU caches. In other word, caches must be kept coherent in order to avoid different values of the same memory location being cached. When there is just one single process, there is no chance to cache different values of the same memory location. And because Message Passing Interface (MPI) is a standardized and portable message-passing standard and industry to function on a wide variety of parallel computing architectures. That means there are no Input and output device involving.

**6. Is Amdahl's law applicable to strong scaling applications? Explain. (2 points)**

Ans. Strong Scaling means that total problem size stays the same as the number of processors increases. The Amdahl's law would defeat strong scaling. For large N, the parallel speedup doesn't asymptote to N, but to a constant  $1/a$ , where  $a$  is the serial fraction of the work. The graph below compares perfect speedup (green) with maximum speedup of code that is 99.9%, 99% and 90% parallelizable

$$\begin{aligned} T(N) &= \text{total time} = p/N + s \\ p &= \text{parallel workload} \\ s &= \text{serial time} \\ S(N) &= \text{speedup} = T(1)/T(N) \\ &= (p + s) / (p/N + s) \\ \text{If } a &= s / (p + s), \text{ then} \\ S(N) &= N / [1 + (N-1)a] \\ &\rightarrow 1/a \text{ for large } N \end{aligned}$$



from [www.cac.cornell.edu](http://www.cac.cornell.edu)

**7. Is Amdahl's law applicable to weak scaling applications? Explain. (2 points)**

Ans. Weak Scaling means that the problem size increases at the same rate as the number of processors, keeping the amount of work per processor the same. In other words, the idea is to do more tasks of fixed size  $t$  in the same length of "wall" time, rather than a fixed workload in less time. Amdahl's Law

equation assumes that the computation problem size doesn't change (hence the parallel section, serial section ratio). So, usually the Gustafson's Law is more applicable to weak scaling applications.

## 8. Which of these limit the scalability of applications? (2 points)

Race condition; Deadlock; Heisenbug; Parallelization overhead.

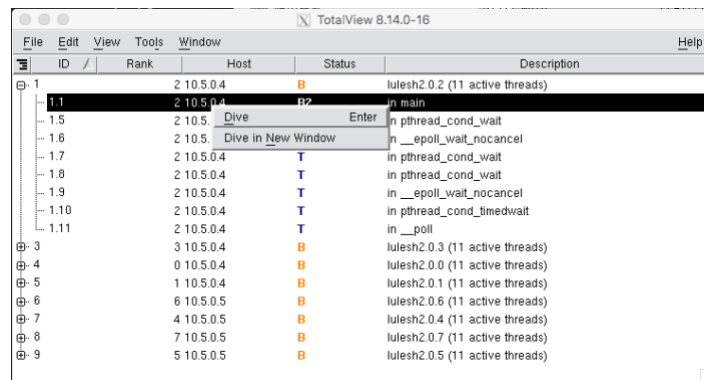
### 5.1 Task 2 (12 points total)

#### 1. Session Manager (2 points)

Ans. The Sessions Manager is a GUI interface to manage your debugging sessions. Use the manager to load a new program, to attach to a program, or to debug a core file. The manager keeps track of your debugging sessions, enabling you to save, edit or delete any previous session. You can also duplicate a session and then edit its configuration to test programs in a variety of ways.

#### 2. Root Window (2 points)

Ans. Root Window provide user with overall but simple information of all processes and Treads. This information in the columns include MPI rank, Hierarchy, host, status and short description. User not only can sort the objects in the list but dive into details by chose the objects which user is interested in.



#### 3. Process Window (2 points)

Compared to Root Window, Process Window provide more detailed information of processes and treads.

The Process Window contains various panes that display the code for the process or thread that we're

debugging, as well as other related information. The large scrolling list in the middle is the Source Pane.

It provides user with full source code, debugger setting like break point and can track deep in each Thread.

User can easily switch to specific process and thread which user is interested in on the window. Therefore,

most of the observations are in this window. Process Window mainly consists of four panes: Stack Trace pane, Stack Frame pane, Source Pane and combined Pane(Action Points, Processes and Threads Pane).

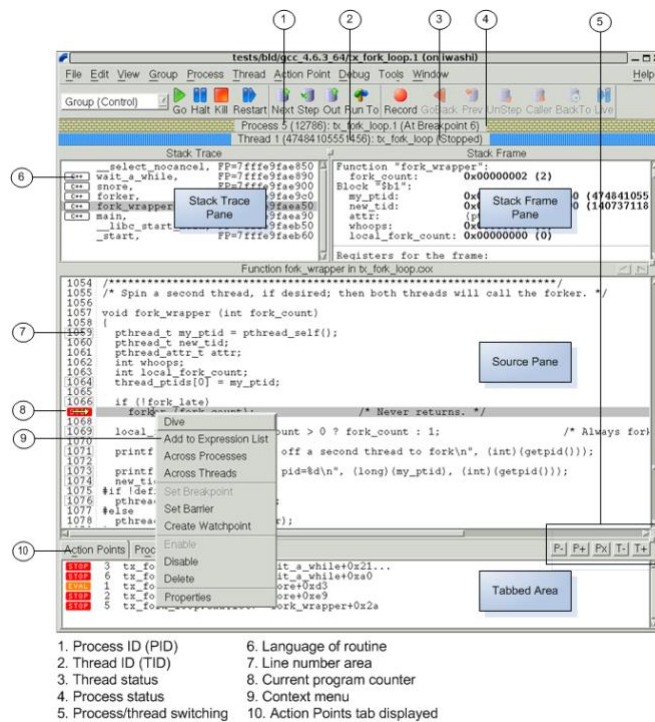


Image Source: <https://docs.roguewave.com/TotalView/8.14/>

#### • Stack Trace Pane (1 point)

Ans. Stack Trace Pane shows executing call stack of routines(function) in specific threads. There are three columns in the pane: Left one shows which programming language used in the routines. Middle one shows the name of routines. Right one shows the address of the routines of the routine's frame pointer. When user selected the row which user is interested in, the Source Pane and Stack Frame Pane will show corresponding information.

#### • Stack Frame Pane (1 point)

Ans. Stack Frame Pane shows the local variables, function parameters and registers corresponding to the selected routine name in Stack Trace Pane. By clicking the selected objects in Stack Frame Pane, user can dive into details in variable window.

#### • Source Pane (1 point)

Ans. Source Pane shows source code or assembler of selected routine(function) in Stack Trace Pane. It also shows PC (program counter) in left. User set debugging operators in this Pane. For example, put or delete breakpoint in left in selected line with boxed. By clicking the selected functions, user can dive into detailed or assembler and the Source Pane will show the information.

#### • Action Points, Processes, Threads Pane (1 point)

Ans. In Action Points, it shows the breakpoints, evaluation points, and watchpoints in the Routines of watching processes.

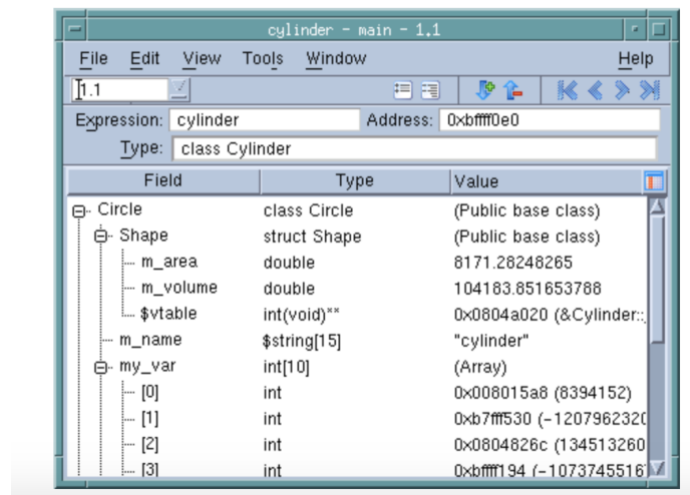
In Process, it shows all processes in the session and shows each status with color corresponding to different status like Blue(Stopped), Orange(Breakpoint). By clicking the selected processes, user switch the information of Source Pane and Stack Frame Pane to the first thread of processes.

In Treads Pane, it consists of two columns. The first column shows the thread ID and system ID. The second line shows the status corresponding to different characters like E(error) and H(Held).

#### 4. Variable Window (2 points)

Ans. Variable Window shows the information of selected variables (include array). User can display, edit, sort and filter the information like expression, type, address and value in the each filed. However, the address of register variables cannot be edited. Variable Window also shows status of variable for user to check error message caused by variable changed or other issue.

Figure 91 – Variable Window

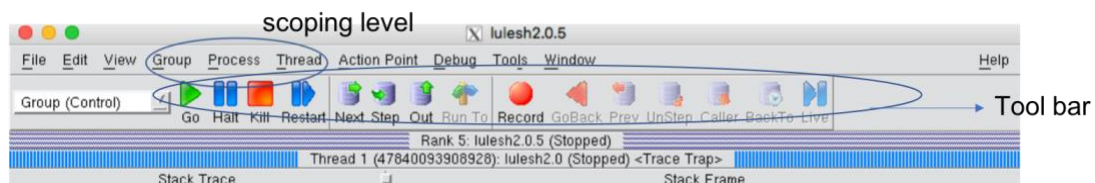


## 6.1 Task 3 (16 points total)

1. Describe in the report how the above operations are performed in TotalView. (2 points each, 8 total)

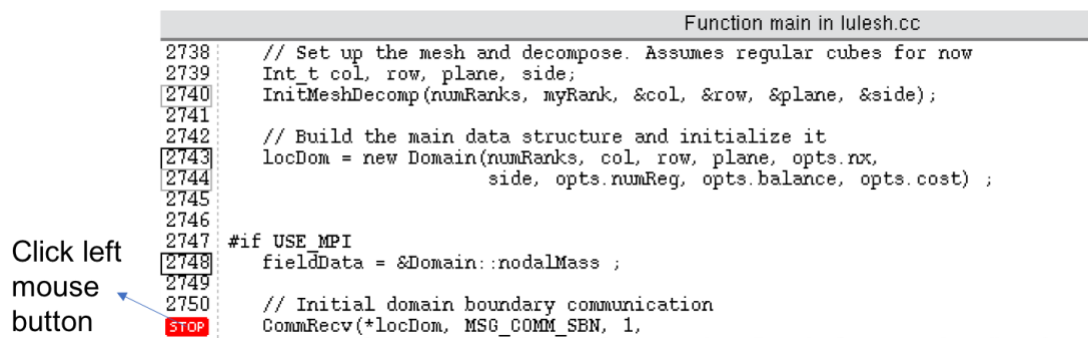
### Control execution:

Ans. When user start to run the debugging program, using the tool bar in the top of Process Window to execute every action. User can choose Group, Process or Thread scoping level and use the same command just like in the tool bar. Each command has different function like Go, Next, Step, etc.



### Setting breakpoints:

Ans. In the Source Pane of Process Window, choose the desired code line and then click left mouse button.



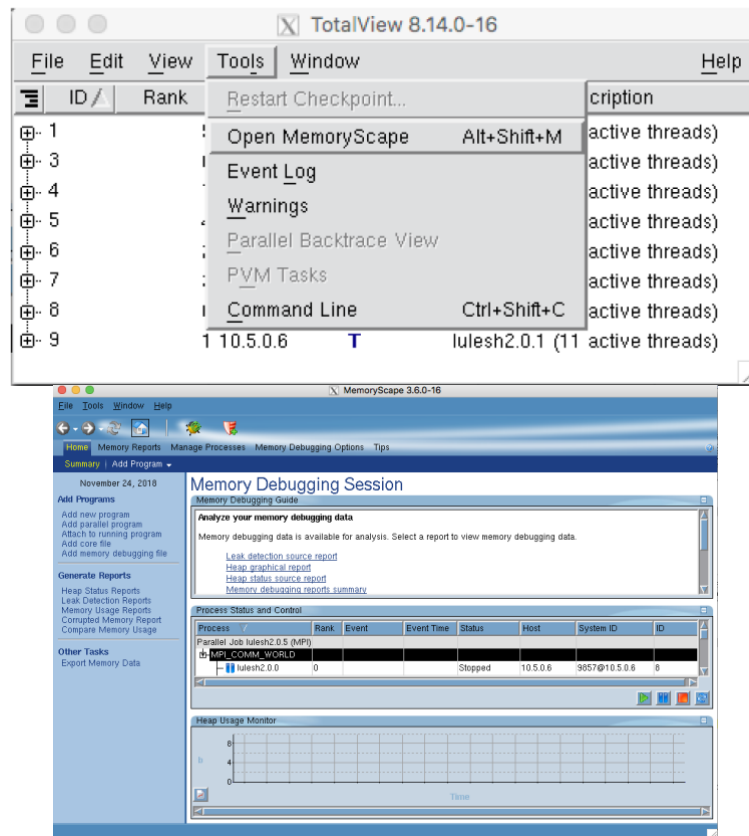
### Diving into functions:

Ans. In the Source Pane of Process Window, select the function which we are interested in and click right button of mouse, then choose “dive”.

### View memory:

Ans. User can select desired variable in Stack Frame Pane or Source Pane, and then double click it to open the Variable Window. In address column, user can see the memory address. If user want to check the leak or Heap status in the memory, user can open the MemoryScope.





MemoryScape

2. Give a short explanation on why these operations are important in a debugger. (2 points each, 8 total)

#### Control execution:

Ans. Most of debugging commands are related to Control execution like GO, Next and step. When user wants to operate on or organize desired scoping level like Group, Processes or Thread, user needs to use choose it before execution.

#### Setting breakpoints:

Ans. During parallel programming execution, there may be some error or the wrong value. By halting the execution at the breakpoint, user can check whether the function or variable is operating normally or not.

#### Diving into functions:

Ans. There are many routines or subroutines containing important variables and user may want to put a breakpoint in the function to observe the action of the whole function structure. Therefore, user needs to dive into it to get detailed information.

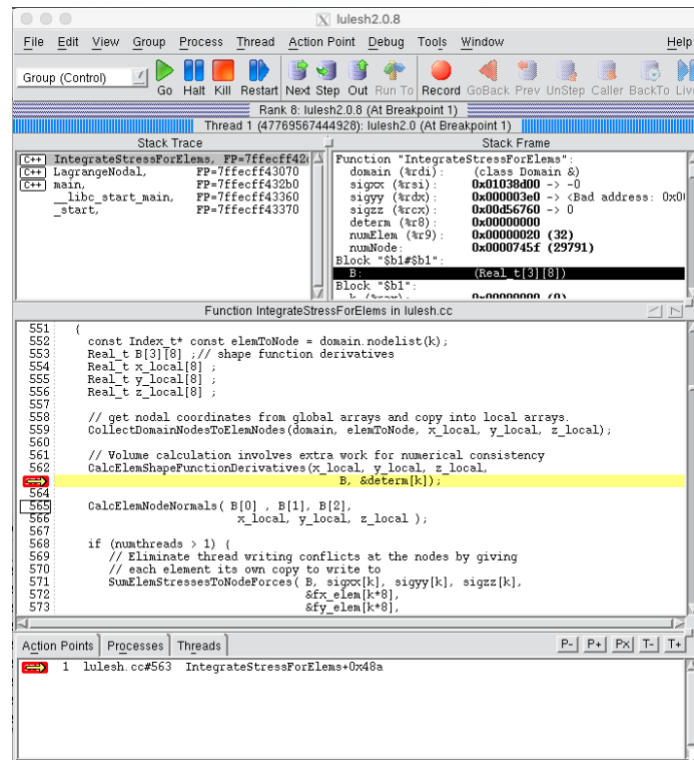
#### View memory:

Ans. Variable window can show the address of variable directly in memory. User needs to check for leaks, Heap status or corrupted memory, but it is very difficult to do. MemoryScape provides user with a graphical window to check and it is handy to observe and debug.

### 7.1 Task 4 (9 points total)

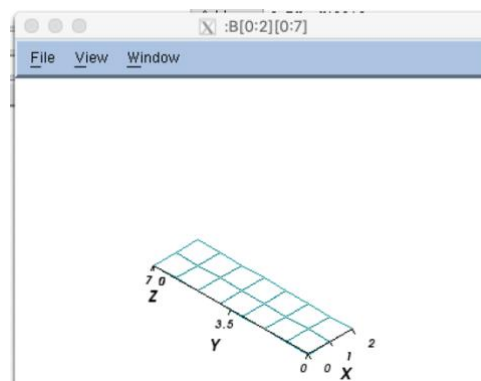
1. State the name of the array you decided to visualize and include a snapshot of its visualization in the report. It is not necessary to understand the actual meaning of the values in the array. (4 points)

Ans. We have decided to visualize an array named B containing the data for shape function derivatives, which is an argument to the function named CalcElemShapeFunctionDerivatives() in the file named lulesh.cc.



The screenshot shows the B - IntegrateStressForElems - 1.1 window. The Expression is B, Address is 0x7fecf42910, Slice is [1], and Type is Real\_t[3][8]. The table below shows the values for the expression B.

Field	Value
[0][0]	2.36012669021855e-310 <denormalized>
[0][1]	2.36012695405e-310 <denormalized>
[0][2]	3.28486856809111e-317 <denormalized>
[0][3]	4.94065645841247e-324 <denormalized>
[0][4]	3.30020831826319e-317 <denormalized>
[0][5]	2.36012846763473e-310 <denormalized>
[0][6]	0
[0][7]	3.29838225163616e-317 <denormalized>
[1][0]	3.29838225163616e-317 <denormalized>
[1][1]	3.87535606537474e-317 <denormalized>
[1][2]	0
[1][3]	0
[1][4]	4.94065645841247e-324 <denormalized>
[1][5]	2.36012814243625e-310 <denormalized>
[1][6]	2.9742751879643e-321 <denormalized>
[1][7]	2.36012677437537e-310 <denormalized>
[2][0]	0
[2][1]	2.36012800109487e-310 <denormalized>
[2][2]	6.95310378251272e-310 <denormalized>
[2][3]	2.36012694124856e-310 <denormalized>
[2][4]	0
[2][5]	6.95310378251272e-310 <denormalized>
[2][6]	4.94065645841247e-324 <denormalized>
[2][7]	2.36012675689128e-310 <denormalized>



2. What is the name of the MPI rank variable in the benchmark? (1 point) What is the name of the MPI size variable in the benchmark? (1 point)

Ans. The MPI rank variable is the “myRank” variable which obtains its value from the `MPI_Comm_rank`. It returns the rank of a process in a communicator.

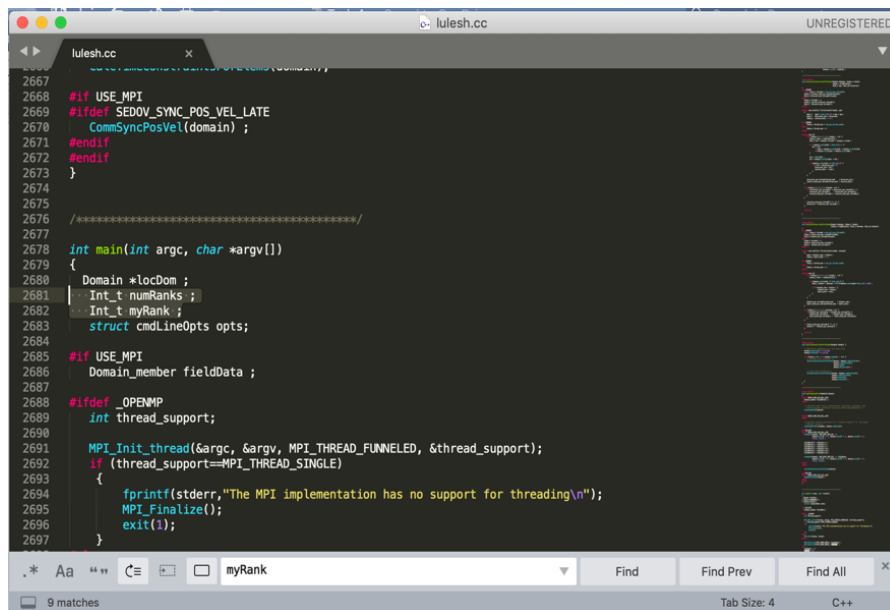


3. Where are these variables first set in the benchmark's source code (file name and line number)? (3 points)

Ans. The MPI size variable is the “numRanks” variable which obtains its value from the `MPI_Comm_size`. It returns the number of a processes in a communicator.

4. Where are these variables first set in the benchmark's source code (file name and line number)? (3 points)

Ans. “numRanks” and “myRanks” variables are first set in the benchmark file named `lulesh.cc` respectively in line no. 2681 and 2682.



```
2667
2668 #if USE_MPI
2669 #ifdef SEDOV_SYNC_POS_VEL_LATE
2670 CommSyncPosVel(domain);
2671 #endif
2672 #endif
2673 }
2674
2675
2676 /*****
2677 int main(int argc, char *argv[])
2678 {
2679     Domain *locDom;
2680     Int_t numRanks;
2681     Int_t myRank;
2682     struct cmdLineOpts opts;
2683
2684     #if USE_MPI
2685     Domain_member fieldData;
2686
2687     #ifdef _OPENMP
2688     int thread_support;
2689
2690     MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &thread_support);
2691     if (thread_support==MPI_THREAD_SINGLE)
2692     {
2693         fprintf(stderr, "The MPI implementation has no support for threading\n");
2694         MPI_Finalize();
2695         exit(1);
2696     }
2697 }
```

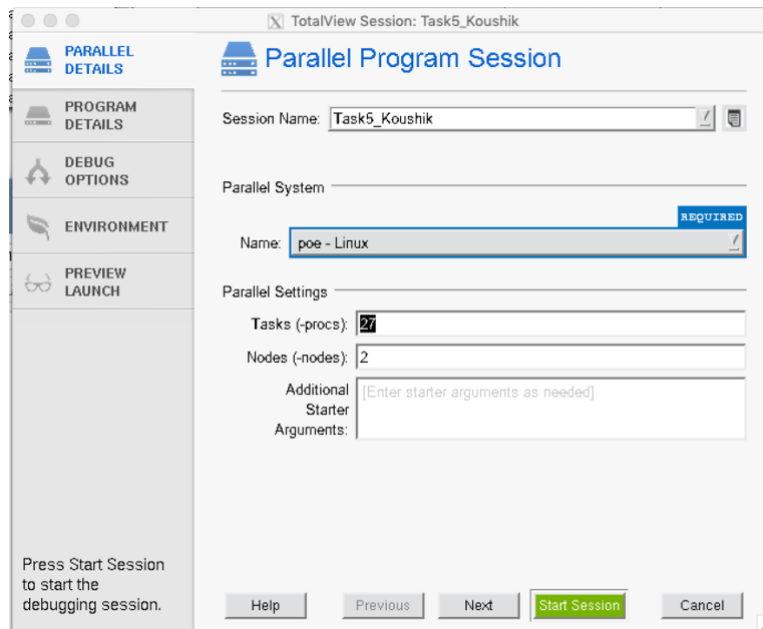
## 8.1 Task 5 (9 points total)

1. Justify in the report the core and thread combination used. (3 points)

Ans. According to the time in assignment 1, one processor with 5 to 26 cores (5-25 threads per processor) has the best performance so we choose the middle point: 16 threads.

2. Explain how thread and process counts are controlled in TotalView. (2 points)

Ans. We can set the required no. of processes in TotalView when creating a new parallel programming session. Under Parallel Settings, we set the number of Tasks that we require for that particular session.



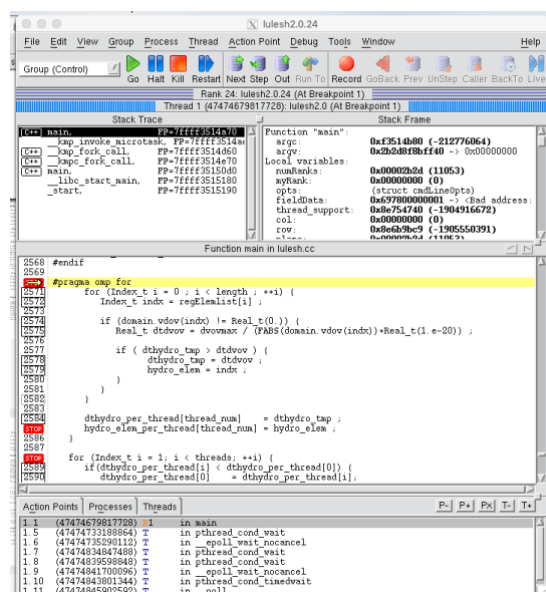
To control the thread counts, we use the `OMP_NUM_THREADS = X` environment variable for threaded parallel runs where x is the no. of threads required.

### 3. Explain what the fork-join model is. (1 point)

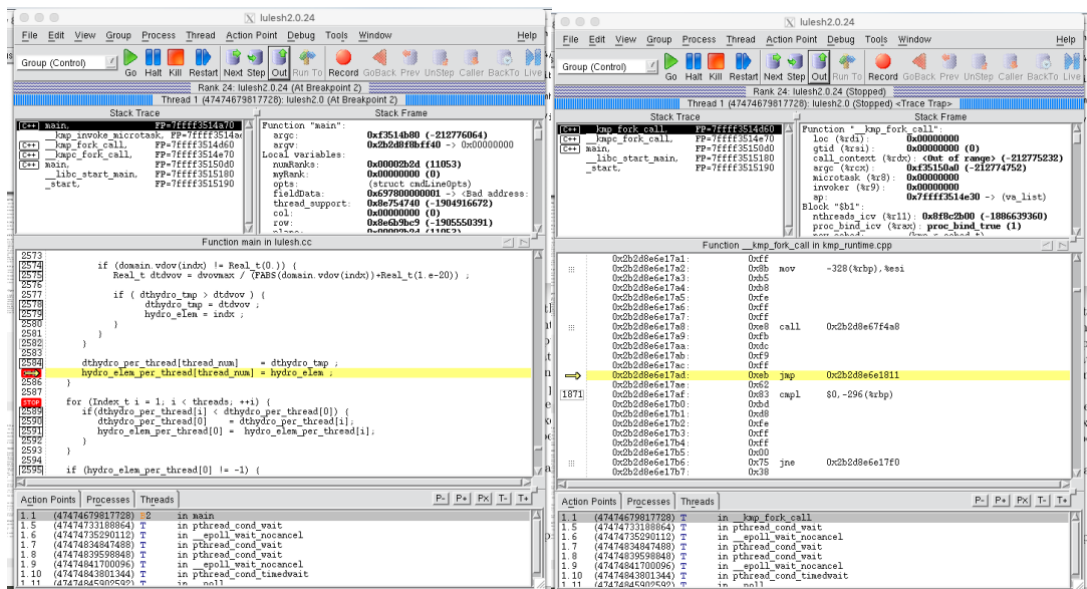
Ans. In parallel computing, the **fork-join model** is a way of setting up and executing parallel programs, such that execution branches off in parallel at designated points in the program, to "join" (merge) at a subsequent point and resume sequential execution. Parallel sections may fork recursively until a certain task granularity is reached. Fork-join can be considered a parallel design pattern.

### 4. Include a screen capture of the before and after effect of the fork-join model by navigating to a parallel region and looking at the Threads Pane in TotalView. (3 points)

Ans. Thread pane before entering the parallel region:

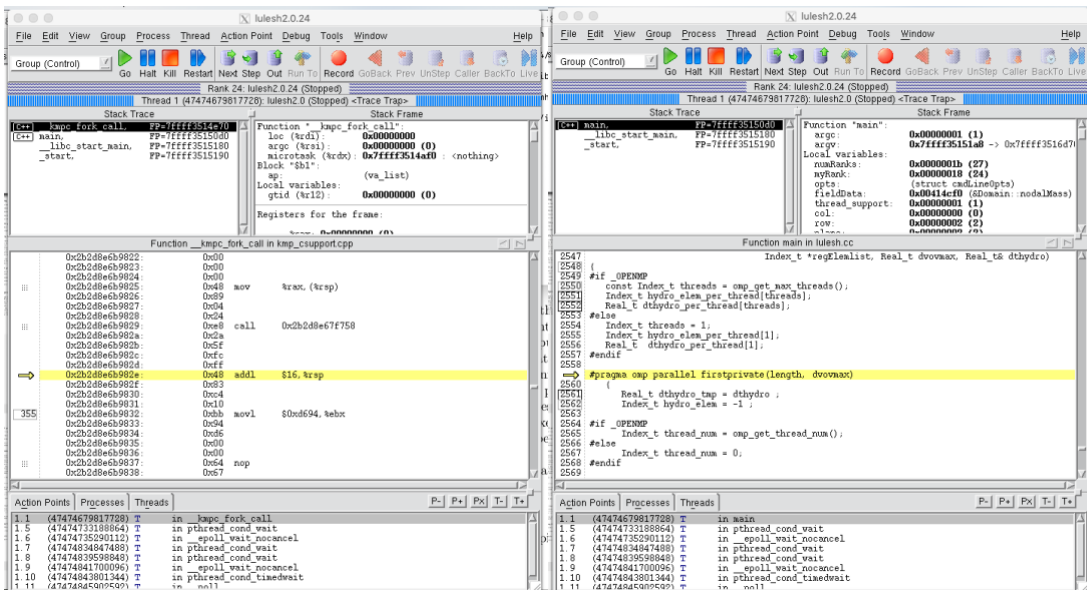


After entering the parallel region:



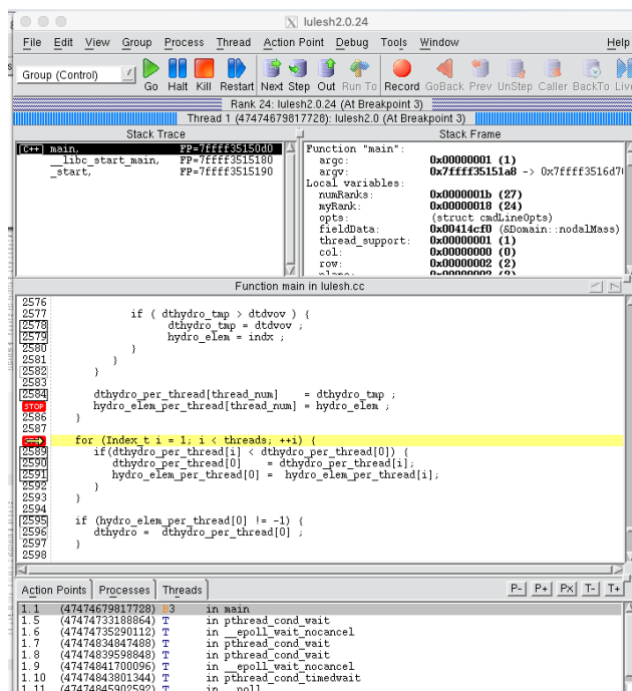
2

3



4

5



6

## 10.1 Task 6 (26 points total)

### 1. What are the main differences in the way events are recorded by gprof and Score-P? (3 points)

Ans. Profiling uses the occurrence of an event to keep track of statistics of performance metrics. These statistics are stored and can be viewed in the profile data files when the program terminates. Tracing records a detailed log of timestamped events and their attributes. This allows the user to see when and where routine transitions, communication and user defined events take place, hence it has a much larger size. Moreover, tracing can be done over more processes, whereas gprof can investigate just one process.

### 2. Does Vampir support post-mortem or online analysis? Explain. (2 points)

Ans. Vampir supports post-mortem analysis, as the analysis is done after the application has finished. It cannot be viewed online, as the data management would be too big.

### 3. Explain the communication bottlenecks that occur in point-to-point (4 points) and collective MPI communication (4 points). For example, late-sender, ...

Ans. In point-to-point bottlenecks, some processes have to wait for data from other processes in order to resume. There can be two types: “late-senders” and “late-receivers”. For example, if a process has finished its task but needs information from another process, it cannot continue until it receives the data. In this case, the process is a late-receiver. Otherwise, a process which sends data later than it was requested is a late-sender.

At collective MPI, because the data is collected from all users there can be more types of bottlenecks. First of all, there is “early reduce time”, in which the root process suffers wait times because it entered the operation (MPI\_Reduce, MPI\_Gather) and there can no data be sent. Second of all, there is “late broadcast time” in which a process loses time by waiting for data from root process which starts broadcasting too late. Basically, in this opposite from the first, other processes lose time, not the root. “Early scan time” is when the process does the scanning, but too early, as not all processes are done yet. “Wait at N x N time” and “N x N Completion time” refer to all the processes. The first one, means that no process can finish the operation until the last process has started it, as they require data from all processes and the second one is the time take it takes for the operation to complete after the first process left the operation.

*From [http://apps.fz-juelich.de/scalasca/releases/scalasca/2.1/help/scalasca\\_patterns-2.1.html](http://apps.fz-juelich.de/scalasca/releases/scalasca/2.1/help/scalasca_patterns-2.1.html)*

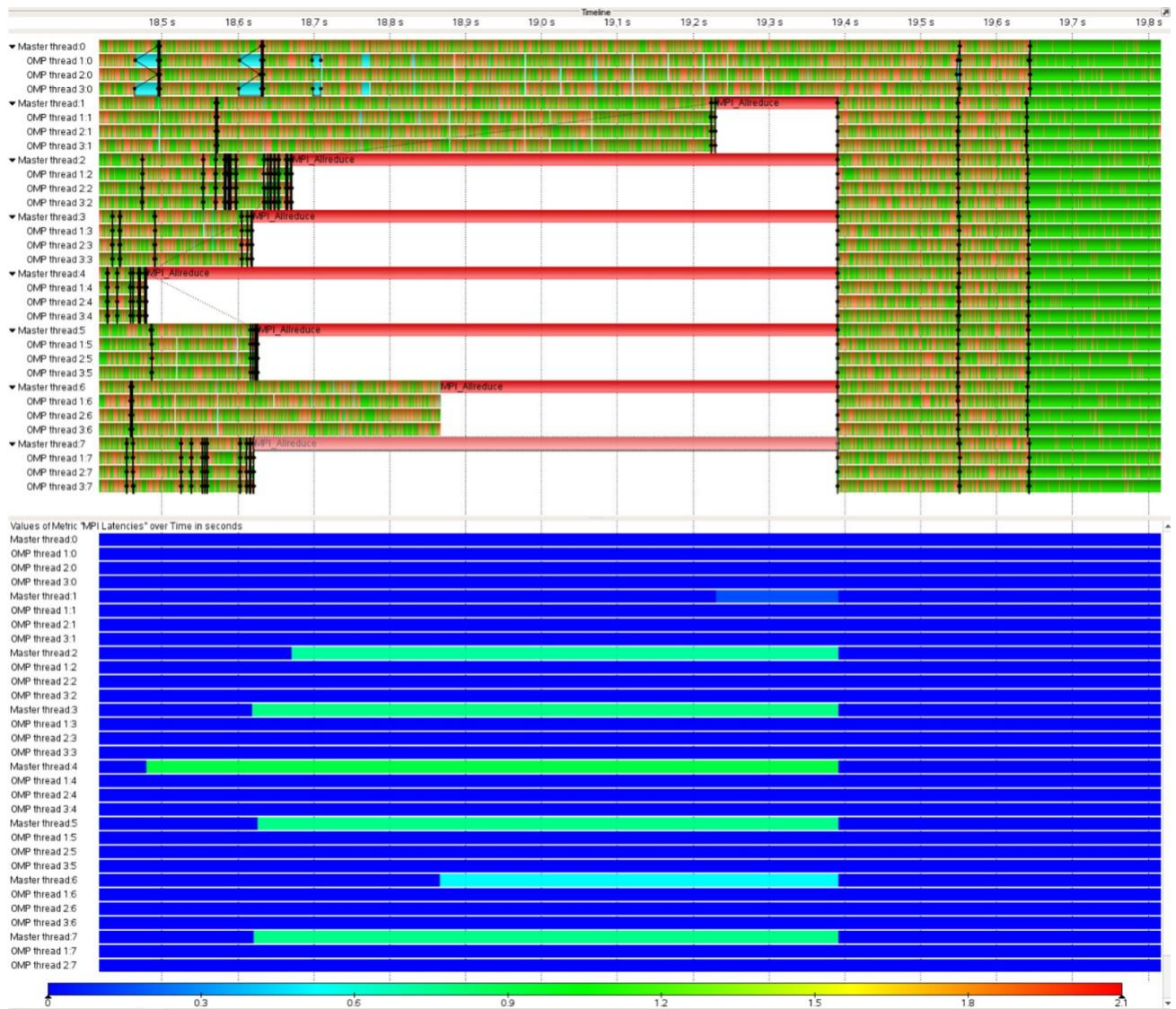
### 4. Take a screen capture of one of the MPI communication bottlenecks from the above question, and describe the situation. (3 points)



Ans. The bottlenecks are practically presented with red and represent the area where the process has to wait through the function `MPI_WAIT`, as it needs to receive data from other processes. For example, Master thread 3 resumes its execution after it receives the data from Master thread 4. It also resumes its execution after it receives data from Master thread 5. I took Master thread 3 as the best example of bottleneck because it has the longest wait duration. Moreover, Master thread 4 also waits for Master thread 0 to finish what it executes to send the message.

### Another Example:





*Fig: all processes have to wait until Master Thread 1 starts MPI\_Allreduce, as they cannot finish the procedure until the last one starts.*

**5. How does the Performance Radar help in analysis? (3 points)**

Ans. Performance Radar helps us analyze different values with the program. For example, it was helpful in analyzing the collective MPI bottleneck, as there were many messages send with 0 bits and they could be counted. It basically counts how many times an event has happened.

**6. Describe the MPI message pattern in the Communication Matrix tab. (2 points)**

Ans. The Communication matrix allows us to see even from the threads of a certain process, what thread send a message to what process. It has different ways of visualization, one of the most useful in the analysis being the “Average Message Data Rate” which shows the number of bits transmitted per second. Furthermore, it can also be seen that each process communicates with each other process within the program, meaning that the wait time durations might be longer.

**7. Take a screen capture of a fork-join, and describe the situation. (2 points)**





Ans. Fork-join() makes the thread, that is calling the function wait until all the other threads have finished their execution and then it finishes or continues. This can be seen within the circled areas on the figure, as the one thread waits for all the others, they all finish at the same time and in case of Master thread 4, after all the threads join, the execution is continued just on the Master thread. This can also be seen outside the circled areas, on Master thread 3,5,6,7, as they join and the execution continues just on the Master thread.

#### 8. Do you observe any performance bottlenecks from Section 3.1 (Task 1) in the code? (3 points)

Ans. For example, on our code there can be found load imbalance, meaning when one thread finishes its work early and has to wait for the rest of threads, as in the situations described in no. 7. Moreover, load imbalance is also found within MPI as some processes have to wait in point-to-point communication or collective communication as other processes have not finished their works yet. Parallelization overhead is also to be found, as some time of the entire program is spent with communications between processes.

Table. Assignment distribution

Task Name	1	2	3	4	5	6	Report
Mihai-Gabriel Robescu	P	P	P	P	P	R	P
Hsieh Yi-Han	P	R	R	P	P	P	R
Yi JU	R	P	P	P	P	P	P

Koushik Karmakar	P	P	P	R	R	P	P
------------------	---	---	---	---	---	---	---

**P** = **P**articipate      **R** = be **R**esponsible to      **A** = **A**bsent