Assignment 2:
Parallel Debugging with TotalView

Emily Bourne (emily.bourne@tum.de)
Abraham Duplaa (abraham.duplaa@tum.de)
Jiho Yang (jiho.yang@tum.de)

December 7, 2017

1 Parallel Programming Concepts

In order to understand a parallel program it is important to understand some important concepts which arise in parallel programming. The following provides summaries of some of the key concepts:

Race Condition :
Race condition is a situation in which multiple threads attempt to access and change the same data on shared memory space. In parallel computing, since the order in which threads access/write same data is not known to user but depends on the scheduler, the users must avoid race conditions when programming.
Deadlock :
Deadlock is a situation in which a process enters a waiting state because a requested system resource is held by another waiting process, again which is waiting for the system resource held by another waiting process to be available. If such relation becomes symmetric (forms a loop of dependent locks), the threads are unable to change its state indefinitely. Such situation is stated to be in a deadlock.
Heisenbug (observer's effect) :
Heisenbug is a jargon in computer programming that means a software bug that seems disappear or alter its behaviour when one attempts to observe it. This term originates from the name of *Werner Heisenberg*, a Physicist who first suggested the observer effect (which states that by simply observing a phenomenon necessarily changes that phenomenon) in quantum mechanics. This occurs because by inserting some lines of code (e.g. "cout" to output values), or removing compiler optimisation flags for debugging purposes changes the execution binary.
Cache Coherency and False Sharing :
Data can be stored in multiple caches at any given time, especially in a parallel program as each thread may have its own cache. If data in one cache is updated then the other caches must be updated accordingly. When all caches are kept up to date, it is said that there is cache coherency. The updating of the caches is done cache line by cache line. Cache lines contain multiple data values. Sometimes it is possible that one thread changes just one value on a cache line. If a different thread accesses a different value on that cache line, cache coherency will force the cache line to be updated even though this is not necessary. This is costly and unnecessary and is referred to as false sharing.
Load Imbalance :
In a parallel program each thread or process has a load which is the quantity of work that it has to do. Load imbalance arises when one thread/process has significantly more to do than the others. This results in threads/processes idling while they wait for the thread with the largest load to complete its work. During this period the program is in effect running in serial and is thus not utilising the benefits of parallel programming.

Amdahl's law :

Amdahl's law states that the performance speedup is limited by the inherently sequential part of the computation. If s is the execution time for inherently sequential computations, the speedup is limited by:

$$speedup(p) = \frac{time(1)}{time(p)} = \frac{time(1)}{s + parallel\_time(p)} <= \frac{time(1)}{s}$$

where s = execution time of sequential part of code and p = # of processors

Clearly as N approaches infinity, the speedup will asymptotically reach $\frac{time(1)}{s}$. This is important for strong scaling applications.

Strong scaling applications :

Scalability (or scaling efficiency) indicates how efficient an application is when increasing numbers of parallel processing elements. When a program is **Strong Scaling**, the problem size stays fixed but the number of processing elements are increased. This is useful for programs that take a large amount of time to run, due to being CPU-bound. A program is CPU-bound if performance is limited by the CPUs but not necessarily the memory. The amount of memory should be sufficient, but there is a need to increase the amount of processing elements in order to speed up performance. In order to determine the right amount of processing elements, the programmer must find an optimized number where the computation finishes quicker, but also limits the wasting of too many cycles due to parallel overhead.

Linear Scaling in Strong Scaling:

As processing elements N increase, the speedup increases linearly. In other words, the speedup is equal to the amount of processing elements. Linear scaling is very difficult to achieve with high N, because of overhead (communication) and inherently serial code (Amdahl's Law).

Strong Scaling Efficiency:

This is a measure for the scaling efficiency of a strong scaling application:

$$\frac{t_1}{(N \cdot t_N)}$$

Weak scaling applications :

When a program is weak scaling, the problem size (workload) assigned to each processing unit isn't fixed but additional elements (data) are used, usually more than can fit on the memory, otherwise known as memory-bound. A program is memory-bound if it is bound by the amount of RAM each processing unit has. Linear scaling for a weak scaling application is defined as run time remains constant while workload is increased in direct proportion to the number of processors. Unlike in strong scaling, the workload assumptions are relaxed and now it varies linearly with N (number of processing elements). With this assumption, the workload W is of the following form:

$$W(N) = N * t$$

Where W grows with N and t remains constant. Therefore, it is easier for a program to scale linear if it is weak scaling.

Most programs that are weak scaling should scale well to larger core counts, due to the fact that these memory-bound programs don't require global communication (communication is mostly done with neighboring processing elements, and therefore communication will usually be in-memory). **EXCEPTION**: Programs that heavily use global communication (eg. FFT)

Parallelization Overhead :

A parallel program requires additional work that is not necessary for a serial program. This work is referred to as parallel overhead. It includes tasks such as starting and terminating threads and synchronising jobs.

Floating-point arithmetic challenges :

Comparisons :

Floating-point numbers are represented on computers with IEEE standards. This representation of a floating-point number is only an approximation, unlike integers (e.g. 1.5 could be 1.4999999999 on computers). Therefore when comparing two floating-point numbers, particularly when one of the values are result of computation, it is not advisable to compare them simply via in/equality operators (==, >, <). Instead, it is safer to compare the subtracted value (difference between the two floats) to some very small value (e.g. 1e-15).

Definition of a zero and signed zeros :

A positive zero (+0) is defined as a positive value too small to be represented as a float/double via IEEE standards, hence casted as zero. A negative zero (-0) is the same but casted from very small negative value.

Cancellation or loss of significance :

Since numeric values always carry machine errors when represented on computers due to value approximation, any arithmetic operations produce errors in the output. This error is characterised by *condition numbers*, which vary

for arithmetic operations. If the condition number is large, the resulting value after the arithmetic operation is subject to large errors. Such condition can also be described as when an arithmetic operation on two numbers increases relative error substantially more than it increases absolute error. Subtraction of two floating-point values with equal sign and similar size, in particular, holds large condition number. The following is an example of such operation, and four significant digits are lost for this example.

$$\begin{aligned} x &= 1.23467 \cdot 10^0 \\ y &= 1.23456 \cdot 10^0 \\ x - y &= 0.00011 \cdot 10^0 \\ &= 1.1???? \cdot 10^{-4} \end{aligned}$$

Amplification and error propagation : As mentioned above, floating-point numbers carry round-off errors. When a sequence of operations subject to rounding error is made the resulting errors may accumulate, causing amplification and propagation of error.

Q1) Which of the concepts affect performance but not correctness? Parallel overhead will of course affect performance as it is extra work for the program to do. It does not however directly affect the result of the program and therefore does not affect correctness. Load imbalance can affect performance although it will not affect correctness. The worse the imbalance, the worse the effect on the performance. In the worst case the code will in effect run in serial (with the parallel overhead in addition to the steps that would have occurred if the program had been run entirely in serial). If cache coherency is assured then the correctness of the program will not be affected, however the process of keeping the cache up to date can be costly, especially in the case of false sharing. This therefore affects performance. Deadlock could also be considered to affect performance. This is arguable since deadlock just stops the program, but to solve deadlocks blocking-operations and/or synchronisation are often required which causes overhead.
Q2) Which of the concepts affect the correctness of the application? If the cache is not coherent then read values may be incorrect, This will obviously lead to incorrect results. Any computations involving floating-point operations are also clearly subject to incorrect results. Race conditions may also lead to incorrect results if read and write operations are conducted in an unintended fashion/order. Finally, although not always applicable, Heisenbug can also be considered since the program may not work as intended when some code lines (e.g. *cout*) for debugging purposes are added/removed.

a) Of these, which are exclusive to parallel programming? Problems due to a lack of cache coherency are exclusive to parallel programming as a serial program has no reason to store the same variable in multiple caches. Another concept exclusive to parallel programming is race condition, since it occurs only for multi-threaded computing paradigm.

b) Of these, which are not exclusive to parallel programming? Floating-point arithmetics are obviously not exclusive to parallel programming, and are common problems in any programming paradigm. Heisenbug (if considered to affect the correctness of the program) is also not exclusive to parallel programming.
Q3) Which of them can occur in OpenMP applications? False sharing can occur in OpenMP as it is a shared memory system and therefore all threads should have access to the same up-to-date information. Load imbalance can occur in OpenMP however, there are multiple load balancing methods (static, dynamic, guided, auto, runtime) which which therefore allow an optimal configuration to be used to try and reduce this. As with all parallel programs, OpenMP has parallel overhead. Both race condition and deadlock can occur in multi-threaded computing paradigm on shared memory systems, hence OpenMP.
Q4) Which of them can occur in MPI applications? As with all parallel programs, MPI has parallel overhead. MPI can also have load imbalance problems. Although it is designed in such a way that all threads carry out the same tasks simultaneously, if loops can result in different loads for different processes and thus load imbalance. Deadlock can occur in MPI applications too, for instance by using MPI_Send and MPI_Recv commands separately (instead of MPI_sendrecv). MPI_Send and MPI_Recv are blocking operations. MPI_Send blocks until the data in sending buffer is received by the receiving buffer, and MPI_Recv blocks until the data is received in the receiving buffer. Similar to thread-wise deadlock explained above, deadlocks can occur in MPI applications if such blocking forms a loop/symmetry.
Q5) Is cache coherency necessary on MPI applications with a single process and a single thread per rank? Explain.

Cache coherency is not necessary when there is a single process and a single thread as there is no parallelism and thus no reason for the same data to be stored in multiple places.

The rank identifies the process within a MPI communication group. When there is only one process, there is no need for communication, however it is still possible to create multiple groups. This arises when a program designed to be run with multiple processes is run with only one process. Thus it is possible to create one thread per rank and therefore use multiple threads. Threads use a shared memory model which means that all threads should have access to the same data, thus in this case cache coherency is necessary.

Q6) Is Amdahl's law applicable to strong scaling applications? Explain. Yes, it is applicable to strong scaling applications. A strong scaling application is not able to scale linearly for large N due to inherently serial code in the application as well as communication overhead. When the amount of processing elements are increased for a strong scaling application, the inherently serial part of the code will not benefit and communication between processing units increases, which will both prevent linear scaling. Therefore, a large scaling application will scale asymptotically towards $\frac{time(1)}{s}$ as N processing elements increases. As stated above, $\frac{time(1)}{s}$ is also the upper limit in Amdahl's Law.

Q7) Is Amdahl's law applicable to weak scaling applications? Explain Amdahl's law is applicable to *fixed workload* applications. However, weak scaling applications do not respect this assumption and therefore, Amdahl's law is not applicable to weak scaling applications. Linear scalability can be achieved with weak scaling applications due to the fixed workload assumption being relaxed. Since the assumption that workload is relaxed and workload varies linearly with N, we are able to have a linear speedup, although the slope of the speedup will most likely be less than 1. However, if workload were to remain constant, then Amdahl's law would be apparent in weak scaling applications, as processing units increase.

Q8) Which of these limit the scalability of applications? As previously mentioned, parallel overhead affects the performance of the program. The more threads are used, the larger the overhead becomes. At some point this may begin to be one of the slowest parts of the program. It therefore limits the scalability. Equally any problems which lead to more overhead such as false sharing, or the synchronisation or mutexes required to avoid race conditions or deadlocks, also limit the scalability of applications.

2 TotalView

TotalView is a parallel debugger available from Rogue Wave Software. Debuggers are useful tools for finding problems in codes. However in order to efficiently utilise a debugger it is important to first understand how it works.

2.1 GUI (Task 2)

The following are brief descriptions of some of the aspects of TotalView's GUI [1]

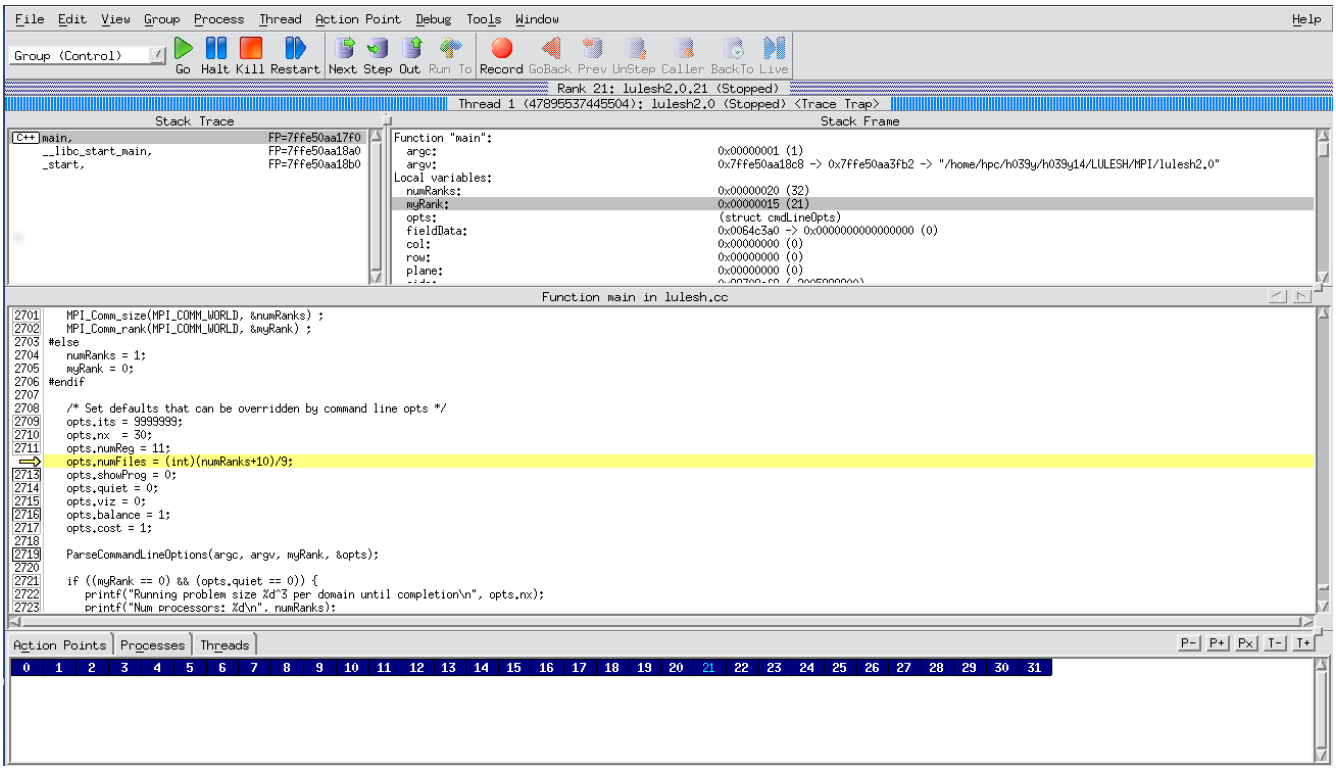
- Session Manager: Very first pane a user sees when invoking *totalview*. It allows the user to launch a new program (serial or parallel), attach to a running program, load a core file, save a debug session for later, or load a previously saved debug session.
- Root Window: Appears when TotalView is started. Provides an overview of all processes and threads, showing the TotalView assigned ID, MPI rank, status, and brief description/name for each. Allows sorting on each column of information that appears. Provides the ability to expand/collapse information under the Hostname column. The "Configure" button allows selection of which information is displayed.
- Process Window: Provides information on debugging/testing process. By default, a single process window will display. For multi-process / multi-threaded programs however, every process and every thread may have its own Process Window if desired. Contains the following 4 panes (fig. 1a)
  - Stack Trace Pane: Shows the call stack of routines the current executable is running. Selection of any routine shown in the call stack will automatically update the Process Window with its information.
  - Stack Frame Pane: Displays the local variables, registers and function parameters for the selected executable. Register abbreviations and meanings are architecture specific.

- Source Pane: Displays source code for the currently selected program or function. Shows program counter, line numbers and any associated action points. Only the lines with a "box" on their line numbers are eligible for debugging.
- Action Points, Processes, Threads Pane: A multi-function pane. By default, it shows any action points that have been set. Action points are either a breakpoint, a barrier point, an eval point, or a watchpoint. The threads tab may also be selected to show associated threads.

- Variable Window: Appears when user dives on a variable or selects a menu item to view variable information. Displays detailed information about selected program variables. Also permits editing, diving, filtering and sorting of variable data. Diving on a variable in TotalView refers to accessing detailed information about the variable.

2.2 Basic Operations (Task 3)

To effectively use a debugger it is important to properly understand how to use it. The main execution controls for TotalView can be seen in figure 1b and are summarised in table 1. They allow the user to progress step by step to find on which line the results diverge from what is expected. This will show which line contains the error.



(a) Process Window

(b) TotalView run controls as described in table 1

(c) TotalView Action points

Figure 1: TotalView process window

Usually however we have an idea where the problem might be. It is therefore not necessary to follow the whole execution step by step but only a certain portion of the execution. A breakpoint is used to help with this. It is set by clicking on the line number where the break is required or by right clicking on the line and selecting "Set Breakpoint" in the dropdown menu. When the breakpoint has successfully been set a red "STOP" will appear on the line number (see figure 1c). When the program is run using the button "Go" it will then carry out all jobs

Command number from figure	Command name	Description
1	Go	Starts the execution of the program
2	Halt	Pauses the execution of the program. This can be useful if you suspect that the program is stuck in an infinite loop.
3	Kill	Stops the execution of the program
4	Restart	Restarts the execution of the program from the beginning
5	Next	Moves to the next step in the current scope (i.e. if a function is called, it is carried out completely and the program stops when it reaches the first step after the function)
6	Step	Moves to the next step in the execution. This usually entails stepping into a function and showing the first step within
7	Out	Moves to the next step in the parent scope (i.e. if the current step is in a function, the function executes and the program stops when it reaches the first step after the function)
8	Run To	When a function is selected, run to allows the program to carry out all steps between its current position and the selected function.

Table 1: Description of the run controls as shown in figure 1b

until it reaches this breakpoint, at which point it will pause all jobs allowing the execution to be continued from this point. This can however be problematic as processes rarely run at exactly the same speed. It would therefore be necessary to force each individual thread to then get to this point (without pressing "Go" as this would make the thread that has reached the breakpoint go past it). To help with this, barriers exist. Barriers are points which all threads must reach before any thread continues. They are set by right clicking on the line and selecting "Set Barrier" in the dropdown menu. When the breakpoint has successfully been set a blue "BARR" will appear on the line number (see figure 1c). Barriers and breakpoints can therefore be used very effectively together.

As previously mentioned, when debugging, it is likely that you have an idea of where the problem may be. It is therefore important to be able to navigate to the correct portion of the code. This is done by diving into functions. Diving into a function opens the source code of that function. This can be done by right clicking on a function and selecting "Dive" in the dropdown menu. During step-by-step execution it can also be done by clicking "Step". This means that all steps carried out by the program can be analysed, including those which are not immediately visible.

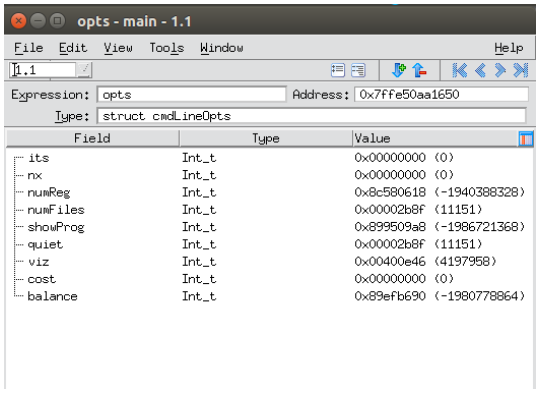


Figure 2: The result of diving on a structure

During the actual debugging process the indicators which allow you to see when your program begins to act differently to what was expected are the variables. It is therefore important to be able to monitor the variables. A list of variables is shown in the stack frame pane along with some basic information about them. More information can be found by diving into the variables. This can be done by double clicking on them in the stack frame pane. This opens a popup window such as the one shown in figure 2. One structure that can be more difficult to visualise is an array. Although we can dive into it as with a structure, it is likely that there will be too many variables to get

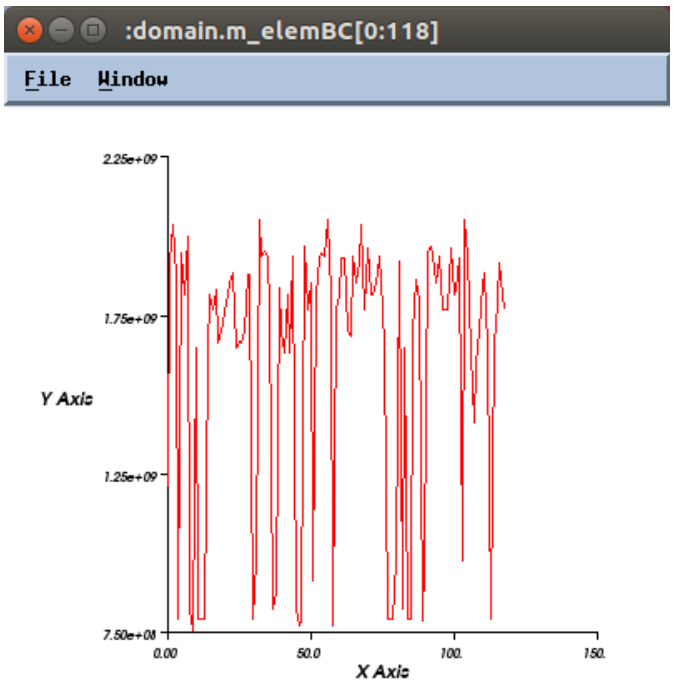


Figure 3: Visualization of domainṁ_elemBC

a clear picture of what the array contains. By clicking "Visualize" in the "Tools" menu of the popup-dive window it is possible to get a view such as the one in figure 3. This allows us to have an idea of the magnitude of the values of the array as well as the uniformity. Thus erroneous values can be quickly located and it is simple to see if one value does not follow the expected pattern.

2.3 MPI with TotalView

The vector chosen to visualize is domain.m_elemBC. The Visualizer tool was used for visualization and it produced the output in Figure 3.

Q1) What is the name of the MPI rank variable in the benchmark?

`myRank`

Q2) What is the name of the MPI size variable in the benchmark?

`numRanks`

Q3) Where are these variables first set in the benchmark's source code (file name and line number)?

Two breakpoints were set. One was set at line 2700 and another set at line 2701 to see when numRanks and myRank are set. The Stack Frame Pane was observed while stepping through the code to make sure each process stores the group size and its own rank.

numRanks value is set in lulesh.cc at line 2700 when MPI_Comm_size(MPI_COMM_WORLD,&numRanks) is called. The number of processes in the group associated with the communicator ,MPI_COMM_WORLD, is returned in numRanks when MPI_Comm_size is called.

myRank value is set in lulesh.cc at line 2701 when MPI_Comm_rank(MPI_COMM_WORLD,&myRank) is called. The rank of each process is returned in myRank when MPI_Comm_rank is called.

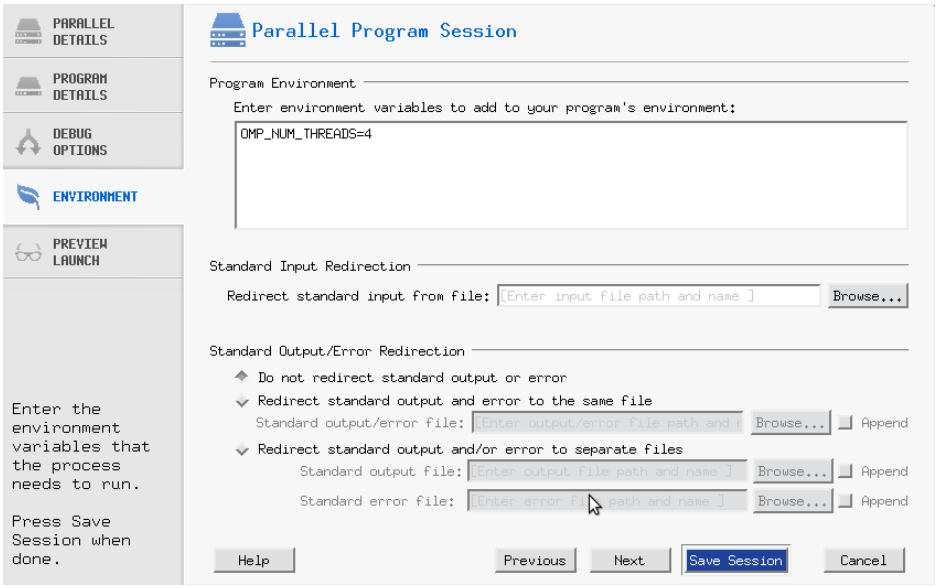


Figure 4: Setting of OpenMP threads to 4

numRanks and myRank are declared in lulesh.cc at lines 2692 and 2693, respectively. These variables are declared as type Int_t. These variables are set shortly afterwards, as explained above.

2.4 MPI+OpenMP with TotalView

Q1) Justification of the core and thread combination used.

Since the Sandy-Bridge node has 2 processors, each with 8 cores the maximum number of tasks (equal to the number of nodes times the number of threads) that can be run on 2 nodes is 32. Due to the restrictions of the Lulesh software only allowing a cubed number of processes (1,8,27...), we chose to run 8 processes with a max number of 4 threads. By doing so we ensure that we reach the maximal number of tasks while using as many processes as possible to take advantage of Lulesh's weak scaling. In order to set the openmp thread number to 4, we set an environment variable `OMP_NUM_THREADS=4`. Please see Figure 4

Q2) How are thread and process counts controlled in TotalView?
TotalView places processes and threads in groups as your program creates them. The purpose of this organization is to aid the user in operating with the threads and processes. These groups are known cleverly as Process/Thread (P/T) groups.

TotalView groups processes and threads into 4 types of groups:

- **Control Group:** Contains all processes. Can be processes which are local or remote.
- **Share Group:** The processes within a control group that share the same code. There will usually be more than one share group.
- **Workers Group:** Contains all the threads that are working in one Control Group. The threads can can be in multiple Share groups but be in one Workers group.
- **Lockstep Group:** This group only is significant when the threads are stopped, since it is created or changed whenever a process or thread reaches an action point or is stopped. This group is a subset of the Workers group.

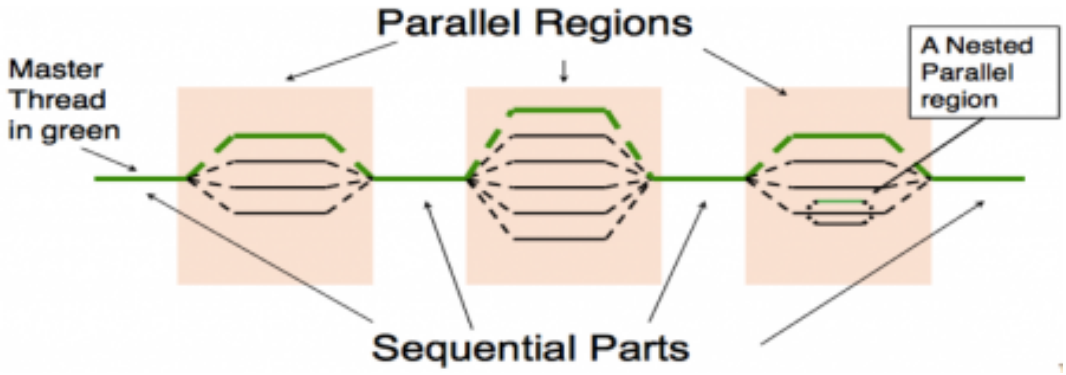


Figure 5: Animation of Fork-Join Model

Q3) What is a fork-join Model?

A fork-join model is what OpenMP uses when a parallel region is reached. Once a process (the master thread) reaches a parallel region, the master thread forks in to many other threads to perform the parallel region concurrently. Once the parallel region has ended and the thread team has completed its work, the threads join back together to one thread again, the master thread. See figure 5 for a graphical interpretation.

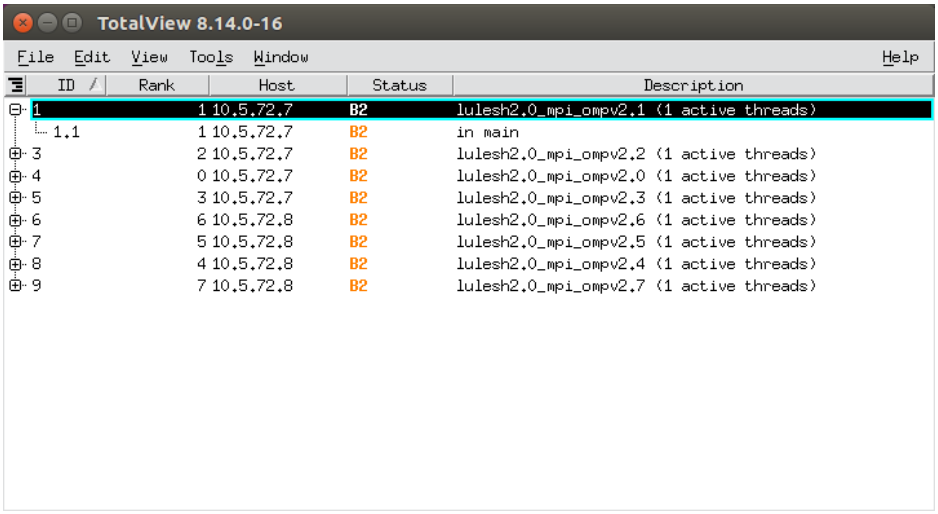


Figure 6: First Breakpoint

Q4) Before and after effect of the fork-join Model in TotalView.

To see TotalView working in action, one must dive and set breakpoints in places aside from the main function. After a thorough investigation of the lulesh code, we found a particularly interesting place to see the effect of the OpenMP fork-join model. In order to note the changes, screenshots were captured before, during, and after the fork-join took place.

Before fork-join

A breakpoint (B2) was set at line 2708, the line immediately after MPI is initialized. At this breakpoint (in the picture, labeled as B2), we can see that all 8 processes are stopped at that location. Furthermore, each process has 1 active thread at this moment. Please see Figure 6.

During fork-join

The second breakpoint (B4) was set at line 2585, inside CalcHydroConstraintForElems and located within a for loop that has had a #pragma added to it. The results are as expected. As shown in the figure, the first Process has 15

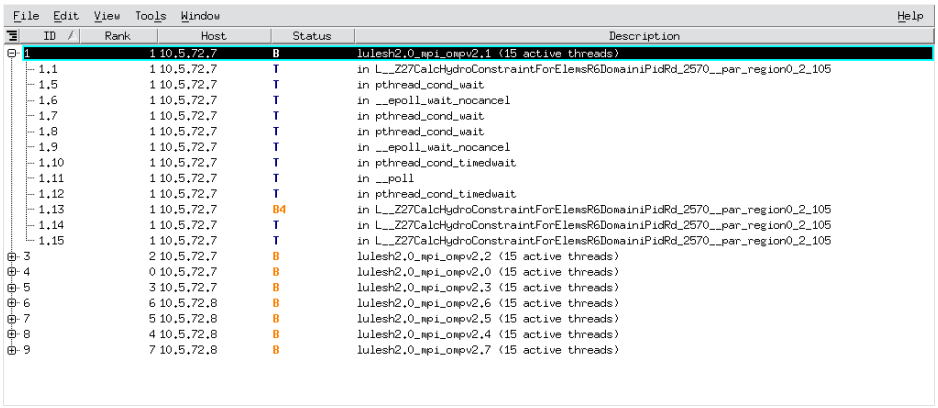


Figure 7: Second Breakpoint

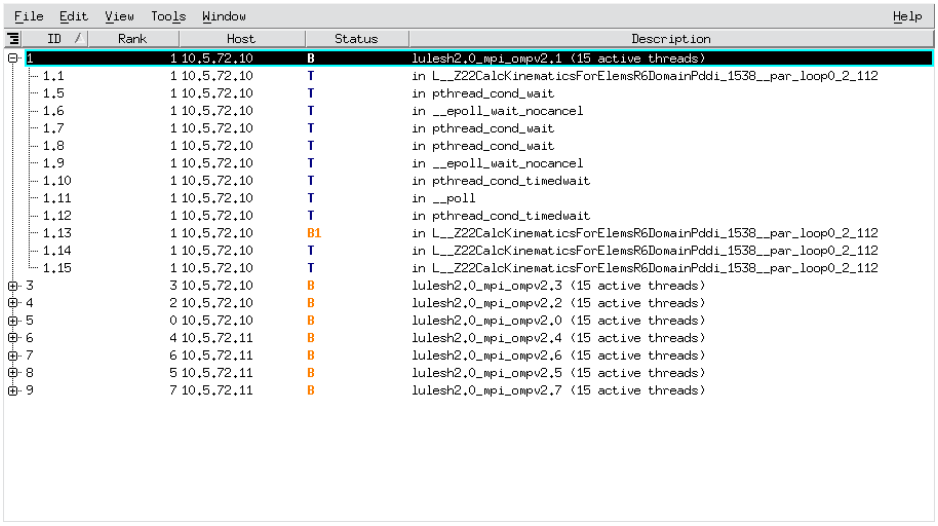


Figure 8: Example Breakpoint

active threads, and 4 are in the parallel region of CalcHydroConstraintForElems. The Process has effectively forked in to 4 threads, which are working in the parallel region in CalcHydroConstraintForElems. **This information is shown in the description of each thread**. Please see Figure 7.

For further explanation, here is another breakpoint set at line 1565, within the function CalcKinematicsForElems. This one exhibits the same behavior of forking. It is evident once again by the description of each thread. Please see Figure 8.

After fork-join

A third breakpoint (B3) was once again set in main but now at line 2785. In this region, we see that each team of threads have joined back together in to main and as expected, one thread for each process is currently located in main.Please see Figure 9.

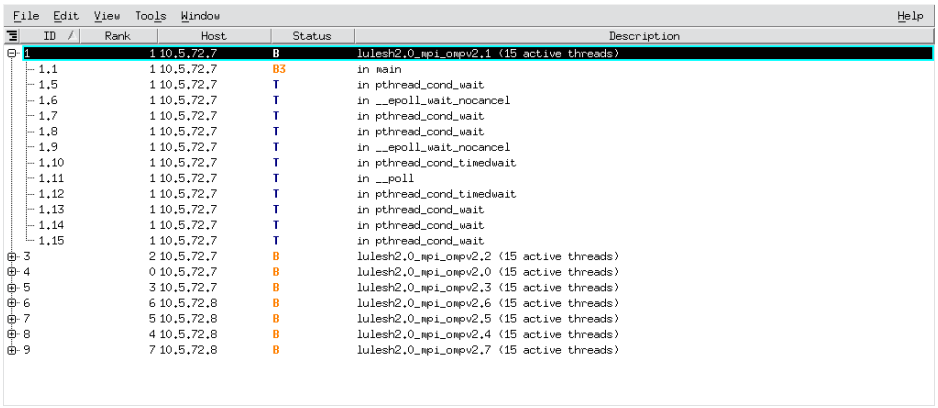


Figure 9: Breakpoint after fork-join section

References

[1] TotalView Debugger
https://computing.llnl.gov/tutorials/totalview/#Windows

[2] GWDG P/T
http://wwwuser.gwdg.de/~parallel/parallelrechner/totalview/totalview_doc/cli_guide/procs_n_threads8.html

[3] Roguewave Documentation
http://docs.roguewave.com/totalview/

[4] NERSC OpenMP Resources
http://www.nersc.gov/users/software/programming-models/openmp/openmp-resources/

[5] TACC U. Texas
http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-basics.html