Organizing Processes and Threads for Debugging

Barry Kingsbury, Ph.D.
TotalView Technologies
24 Prime Park Way
Natick, MA 01760
1-508-652-7704
barryk@totalviewtech.com

ABSTRACT
Debugging parallel programs containing thousands of processes and threads across hundreds or thousands of nodes by individually looking at each process and thread is possible. Possible, however, does not mean practical. The only workable approach is to organize processes and threads into groups and then debug the program by using these groups. This paper discusses the ways in which the TotalView Technologies Debugger (TVD) automatically organizes a program's processes and threads, the way developers can manually organize them when additional grouping capabilities are needed, and the implications of executing processes, threads, and groups asynchronously under debugger control.

Categories and Subject Descriptors
D.2.5 [**Testing and Debugging**]: Debugging aids

General Terms
Algorithms, Design, Theory

1. INTRODUCTION
Debugging parallel programs containing thousands of processes and threads across hundreds or thousands of nodes by individually looking at each process and thread is possible. Possible, however, does not mean practical. The only workable approach is to organize processes and threads into groups and then debug the program by using these groups. Traditional debuggers have only limited means to control program execution. Debugging parallel programs requires that developers be able to asynchronously control a thread, a process, a group of threads, or a group of processes and run these groups on one or more systems and cores.

Just placing threads and processes into groups can be an onerous task. Doing it every time a developer needs to run a program under a debugger's control would probably mean the developer is executing and where the program counter is. Of course, the de spending more time setting up the debugger than actually debugging. A better way is to have the debugger place processes and threads into groups, basing what it does on what the program is executing and where the program counter is. Of course, the debugger must allow for *ad hoc* groupings based on choices made by a developer.

Finally, multiprocess and multithreaded programs must be controlled in ways that minimally disrupt normal runtime patterns. In serial programs, developers could stop the entire program. In these more sophisticated programs, some processes and threads can run, others should not be allowed to.

The problem TotalView Technologies faced when creating and extending its TotalView Debugger was that it needed to create a parallel debugger using the constructs of a serial debugger. That is, the debugger had to look and feel in much the same way as customers were used to, and yet control programs in new ways. For example, even basic debugging activities such as "step" had to be rethought because the "scope" of what was to be stepped, allowed to run freely, or ignored must be understood.

This paper discusses the ways in which the TotalView Technologies Debugger (TVD) automatically organizes a program's processes and threads, the way developers can manually organize them when additional grouping capabilities are needed, and the implications of executing processes, threads, and groups asynchronously under debugger control.

2. AUTOMATIC GROUPING
For many kinds of parallel programs, the debugger can group processes and threads based on how the program is executing. Fortunately, such disciplines as MPI and OpenMP are very regular in how they execute. So long as the debugger can follow the creation of threads and processes, automatically place them under debugger control, and then place them into groups, much of the work can be done for the developer.

Here are the groups into which TVD automatically places a program's processes and threads:

- **Control Group**: All the processes that a program creates. These processes can be local or remote. If the program uses processes that it did not create, TVD places them in separate control groups. For example, client/server programs have two distinct executables that run independently of one another. TVD places each in its own control group. In contrast, processes created by fork() are in the same control group.

PADTAD'07, July 9–12, 2007, London, England, United Kingdom.

- **Share Group**: All the processes within a control group that share the same code. "Same" code means that the processes have the same executable file name and path. In most cases, a program has more than one share group. Share groups, like control groups, can be local or remote.
- **Workers Group**: All the worker threads within a control group. (Worker threads are threads created by your program to do work. This is in contrast to manager or service threads created by a programming environment to assist in managing threads. TVD automatically recognizes that some threads are doing real work and others are created by the runtime environment to manage activities of others)
- **Lockstep Group**: All threads that are at the same PC (program counter). A lockstep group only exists for stopped threads. All members of a lockstep group are within the same workers group. That is, a lockstep group cannot have members in more than one workers group or more than one control group.

The control and share groups only contain processes; workers and lockstep groups only contain threads.

The following figure shows a system running five processes (ignoring daemons and other programs not related to the program) and the threads within the processes. This figure shows a control group and two share groups within the control group.

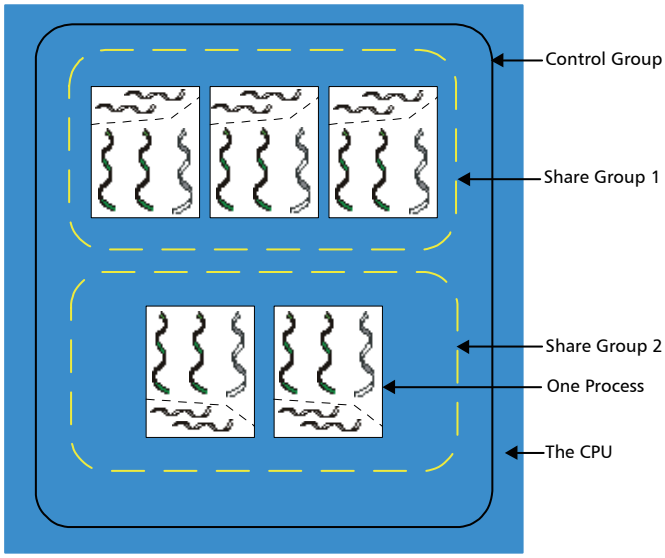

Figure 1. Predefined Sets: Part 1

The elements within this figure are as follows:

- **CPU**: The one outer square represents one CPU or core. All elements in the drawing are within the CPU.
- **Processes**: The five white inner squares represent executing processes. Each process has three threads.
- **Control Group**: The large rounded rectangle surrounding the five processes indicates one control group. This diagram does not show which process is the main procedure.
- **Share Groups**: The two smaller rounded rectangles created using dashed lines surround processes in a share group. This drawing shows two share groups within one control group. The three processes in the first share group have the same executable. The two processes in the second share group share a second executable.

The control group and the share group only contain processes. The next figure shows how TVD organizes the threads in the previous figure. This figure adds a workers group and two lockstep groups.

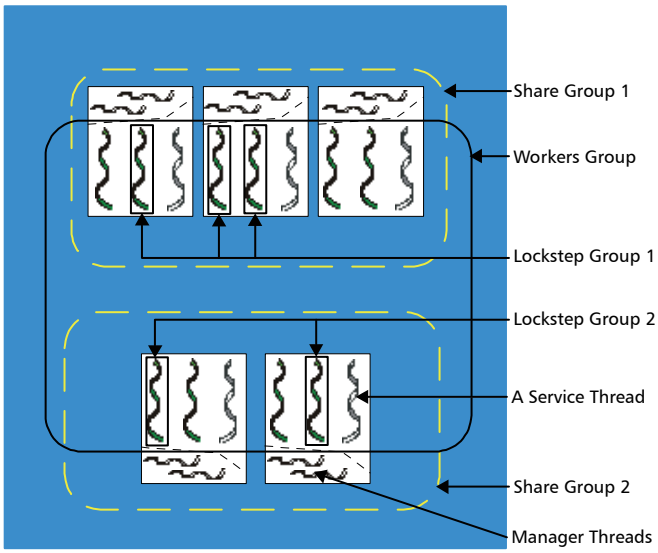

Figure 1. Predefined Sets: Part 2

This figure does not show the control group since it encompasses everything in this figure. That is, this example's control group contains all of the program's lockstep, share, and worker group's processes and threads.

The additional elements in this figure are as follows:

- **Manager Threads**: The ten manager threads are the only threads that do not participate in the workers group.
- **Workers Group**: All nonmanager threads within the control group make up the workers group. This group includes service threads.
- **Lockstep Groups**: Each share group has its own lockstep group. The previous figure shows two lockstep groups, one in each share group.
- **Service Threads**: Each process has one service thread. A process can have any number of service threads, but this figure only shows one. (A service thread is a thread created directly by the program to provide a service to the program. Its computation, however, is not interesting or important while debugging. For example, a program could be using a thread to manage queues. Typically, service routines reside in libraries and are not considered part of the algorithm being debugged.)

The following figure extends the previous figure to show the same kinds of information executing on two systems.

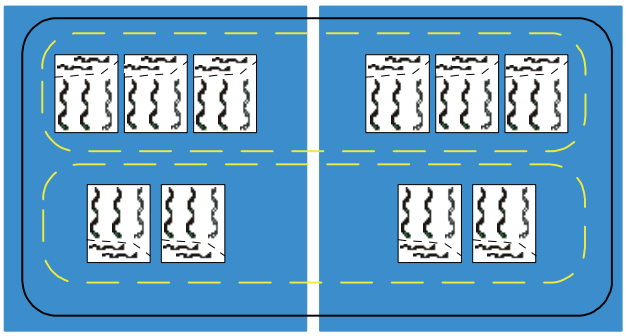

Figure 3. Predefined Sets Across CPUs

Although the number of systems has changed, the number of control and share groups is unchanged. This makes a nice example. Unfortunately, programs are seldom this regular.

3. USING GROUPS

As a program creates processes and threads, TVD tracks these events, automatically attaching to them as they are created. As they are acquired, TVD places them into groups. Because the debugger acquires control over them early, developers can add breakpoints and perform other operations examining state before the process or thread even begins executing.

The TotalView Debugger has two interfaces: a GUI and a command line interface (CLI). One of the prominent features in the GUI is that many windows have a group selector that lets developers choose what the debugger should run. For example, TVD can be told to step a process, thread, or group.

Similarly, the command prompt within the CLI presents the scope of what the command will act upon. The CLI also lets developers execute any command so that it is directed at a group.

4. DETERMINING WHAT RUNS

One of the major innovations built into TVD was that it is designed to understand the way in which processes relate and execute. For example, an MPICH2 program can actually be a collection of independently cooperating programs. While the debugger must have visibility into all programs being run and must be able to control each, it must minimize the way it controls external programs. That is, when the developer halts or kills the execution of one program, the others must be left executing. Or, when one program is stepped, others should continue running freely. Said differently, TVD was designed so that developers could control and minimize the impact of all debugging operations.

The following sections provide a high-level view of how TVD determines what it should be controlling.

> **Note**: *TVD contains an exhaustive set of defaults. These defaults minimize the extent to which developers need to understand these concepts in normal day-in-day-out debugging.*

4.1 Defining the GOI, POI, and TOI

In a multiprocess. multithreaded program, the debugger needs to determine what it should be manipulating. For example, when performing a step operation, what is the scope of the stepping? That is, are other threads stepped and which other threads? While the stepping operation is occurring, do other threads run freely? When the step operation concludes, are other threads stopped or do they continue to run? To decide questions such as these, TVD dynamically determines scoping using the:

- Group of Interest (GOI)
- Process of Interest (POI)
- Thread of Interest (TOI)

Determining the GOI, POI, and TOI gives TVD the information it needs when it defines the scope of the actions to be performed when it executes a command. Depending on the command, TVD determines the TOI, POI, and GOI, and then executes the command's action on that thread, process, or group.

For example, if TVD steps a control group,

- It needs to know what the TOI is so that it can determine what threads are in the lockstep group—TVD will only step a lockstep group.
- The lockstep group is part of a share group.
- This share group is also contained in a control group.

By knowing what the TOI is, the GUI also knows what the GOI is. This is important because, while TVD knows what it will step (the threads in the lockstep group), it also will now know what it will allow to run freely while it is stepping these threads.

The following sections discuss the implications of running at group, process, and thread width. Also, what happens depends upon whether the focus is on a process or a thread group.

4.2 Understanding Group Width Behavior

TVD behavior when stepping at group width depends on whether the GOI is a process group or a thread group, as follows:

> **Note**: *In this discussion, "goal" means the place at which processes and threads should stop executing. For example, when using a step command, the goal is the next line. In contrast, the goal for a run to command is the selected line.*

- **Process group**—TVD examines the group, and identifies which of its processes has a thread stopped at the same location as the TOI (a matching process). TVD runs these matching processes until one of its threads arrives at the goal. When this happens, TVD stops the thread's process. The command finishes when it has stopped all of these matching processes.
- **Thread group**—TVD runs all processes in the control group. However, as each thread arrives at the goal, TVD only stops that thread; the rest of the threads in the same process continue executing. The command finishes when all threads in the GOI arrive at the goal. When the command finishes, TVD stops all processes in the control group.

TVD does not wait for threads that are not in the same share group as the TOI, since they are executing different code and can never arrive at the goal.

4.3 Understanding Process Width Behavior

When stepping at process width, TVD chooses what will run based upon the GOI, which can either can a process group or a thread group, as follows:

- **Process group**—TVD runs all threads in the process, and execution continues until the TOI arrives at its goal, which can be the next statement, the next instruction, and so on. Only when the TOI reaches the goal does TVD stop the other threads in the process.
- **Thread group**—TVD lets all threads in the GOI and all manager threads run. As each member of the GOI arrives at the goal, TVD stops it; other the threads continue executing. The command finishes when all members of the GOI arrive at the goal. At that point, TVD stops the whole process.

4.4 Understanding Thread Width Behavior

Stepping at thread width tells TVD to only run that thread. It does not step other threads. In contrast, process width tells TVD to run all threads in the process that should run while the TOI is stepped. While TVD is stepping the thread, manager threads run freely.

Stepping a thread is not the same as stepping a thread's process, because a process can have more than one thread.

Thread-level single-step operations can fail to complete if the TOI needs to synchronize with a thread that is not running. For example, if the TOI requires a lock that another held thread owns, and it steps over a call that tries to acquire the lock, the primary thread cannot continue successfully. The developer must allow the other thread to run in order to release the lock. In this case, process-width stepping must be used.

5. MANUALY CHOOSING WHAT WILL EXECUTE

Previous sections have discussed how TVD automatically places processes and threads into groups and how it decides what processes and threads it should run. The remainder of this paper discusses how developers can manually define the focus of debugging activities.

> **Note**: *How a developer creates groups that incorporate a set of processes and threads is not discussed as this paper would need to present product details that are not important here. It is sufficient for the reader to know that creating groups is a simple operation.*

When TVD executes a command, it must decide upon which processes and threads to act. Most commands have a default set of threads and processes and, in most cases, the default is exactly what is needed. This collection of processes and threads is called a Process/Thread (P/T) set. Unlike a serial debugger in which each command clearly applies to the only executing thread, TVD can control and monitor many threads with their PCs at many different locations. The P/T set indicates the groups, processes, and threads that are the target of a command.

There are times, however, when the default will not do. This section begins a detailed discussion of how a developer can indicate the processes and threads upon which a command will act. That is, this paper began with a discussion of how TVD automatically sets focus. The subject now changes to how the developer can explicitly tell TVD what it should run when additional capabilities are needed.

5.1 Manually Defining the Thread of Interest (TOI)

The TOI is specified as "**p.t**", where "**p"** is the TVD process ID (PID) and "**t**" is the TVD thread ID (TID). The **p.t** combination identifies the POI (Process of Interest) and TOI. The TOI is the primary thread affected by a command. For example, while a stepping command always steps the TOI, it can run the rest of the threads in the POI and step other processes in the group.

For example, the following list contains names process 2, thread 1, and process 3, thread 2:

```
{p2.1 p3.2}
```

5.2 Setting Process and Thread Widths

P/T sets can be entered in two ways. If groups are not being manipulated, the format is:

[*width_letter*][*pid*][*.tid*]

For example, **p2.3** indicates 'process 2, thread 3."

Although the syntax seems to indicate that there is no need to enter any element, TVD requires at least one. Because TVD tries to determine what it can do based on what is typed, it fills in what is omitted.

The width_letter indicates which processes and threads are part of the focus set. These letters are:

t: Thread width

A command's target is the indicated thread.

p: Process width

A command's target is the process that contains the TOI.

g: Group width

A command's target is the group that contains the POI. This is a process width specifier; thread groups—lockstep and workers —are ignored.

a: All processes

A command's target is all threads in the GOI that are in the POI.

As mentioned previously, the TOI specifies a target thread, while the width specifies how many threads surrounding the TOI are also affected. For example, a step command always requires a TOI, but entering this command can do the following:

- Step just the TOI during the step operation (thread-level single-step).
- Step the TOI and step all threads in the process that contain the TOI (process-level single-step).
- Step all processes in the group that have threads at the same PC as the TOI (group-level single-step).

This list does not indicate what happens to other threads in a program when TVD steps a thread.

Here are two specifiers:

`g2.3` Select process 2, thread 3, and set the width to group.

`t1.7` Commands act only on thread 7 or process 1.

5.3 Specifying Groups in P/T Sets

This section extends the P/T set syntax to include groups. Notice that a group_indicator is added.

[*width_letter*][*group_indicator*][*pid*][*.tid*]

In the description of this syntax, everything appears to be optional. But, while no single element is required, at least one element must be entered. TVD determines other values based on the current focus.

5.3.1 A Group Letter

Developers can name one of TVD's predefined sets. Each set is identified by a letter. For example, the following command sets the focus to the workers group:

```
dfocus W
```

The following are the group letters.

C: Control group: All processes in the control group.

S: Share group: The set of processes in the control group that have the same executable as the TOI.

W: Workers group: The set of all worker threads in the control group.

L: Lockstep group: The set containing all threads in the share group that have the same PC as the TOI. If these threads are stepped as a group, they proceed in lockstep.

TVD lets developers create their own groups. When using these names, the name must be entered using slashes. The following example sets the focus to the set of threads contained in process 3 that are also contained in a group called `my_group`:

```
dfocus p/my_group/3
```

5.4 Combining Width and Group Specifiers

The following table combines all P/T specifiers. It shows how one can modify the other.

Table 1. Combing Width and Group Specifiers

Specifier	Specifies
aC	All threads
aS	All threads.
aW	All threads in all workers groups.
aL	All threads. Every thread is a member of a control group and a member of a share group and a member of a lockstep group. Consequently, three of these definitions mean "all threads".
gC	All threads in the TOI control group.
gS	All threads in the TOI share group
gW	All worker threads in the control group that contains the TOI.
gL	All threads in the same share group within the process that contains the TOI that have the same PC as the TOI
pC	All threads in the control group of the POI. This is the same as **gC**.
pS	All threads in the process that participate in the same share group as the TOI.
pW	All worker threads in the POI
pL	All threads in the POI whose PC is the same as the TOI
tC **tS** **tW** **tL**	Just the TOI. The **t** specifier overrides the group specifier. So, for example, **tW** and **t** both name the current thread.

The following examples add PID and TID numbers to the raw specifier combinations listed in the previous table:

`pW3` All worker threads in process 3.

`gW3` All worker threads in the control group that contains process 3. The difference between this and pW3 is that pW3 restricts the focus to one of the processes in the control group.

`gL3.2` All threads in the same share group as process 3 that are executing at the same PC as thread 2 in process 3. The reason this is a share group and not a control group is that different share groups can reside only in one control group.

`3` Specifies processes and threads in process 3. The POI and TOI are inherited from the existing P/T set, so the exact meaning of this specifier depends on the previous context.

`g3.2/3` The 3.2 group ID is the name of the lockstep group for thread 3.2. This group includes all threads in the process 3 share group that are executing at the same PC as thread 2.

`p3/3` Sets the process to process 3. The Group of Interest (GOI) is set to group 3. If group 3 is a process group, most commands ignore the group setting. If group 3 is a thread group, most commands act on all threads in process 3 that are also in group 3.

5.5 Stepping: Examples

The following shows different ways that a program can be stepped:

- Step a single thread

  While the thread runs, no other threads run (except kernel manager threads).

  Example: `dfocus t dstep`

- Step a single thread while the process runs

  A single thread runs into or through a critical region.

  Example: `dfocus p dstep`

- Step one thread in each process in the group

  While one thread in each process in the share group runs to a goal, the rest of the threads run freely.

  Example: `dfocus g dstep`

- Step all worker threads in the process while nonworker threads run

  Worker threads run through a parallel region in lockstep.

  Example: `dfocus pW dstep`

- Step all workers in the share group

  All processes in the share group participate. The nonworker threads run.

  Example: `dfocus gW dstep`

- Step all threads that are at the same PC as the TOI

  TVD selects threads from one process or the entire share group. This differs from the previous two items in that TVD uses the set of threads are in lockstep with the TOI rather than using the workers group.

  Example: `dfocus L dstep`

6. USING P/T SET OPERATORS

Because membership in groups changes as the program executes and what must be done changes as the program executes, TVD contains operators for managing P/T sets:

|	Creates a union; that is, all members of two sets.
-	Creates a difference; that is, all members of the first set that are not also members of the second set.
&	Creates an intersection; that is, all members of the first set that are also members of the second set.

For example, the following creates a union of two P/T sets:

```
p3 | L2
```

Typically, these operators are used with the P/T set functions. Here are some of the functions that developers can use:

`breakpoint(ptset)`
: Returns a list of all threads that are stopped at a breakpoint.

`comm(process, "comm_name")`
: Returns a list containing the first thread in each process associated within a communicator within the named process. While process is a P/T set it is not expanded into a list of threads.

`error(ptset)`
: Returns a list of all threads stopped due to an error.

`existent(ptset)`
: Returns a list of all threads.

`nonexistent(ptset)`
: Returns a list of all processes that have exited or which, while loaded, have not yet been created.

`running(ptset)`
: Returns a list of all running threads.

`stopped(ptset)`
: Returns a list of all stopped threads.

The following examples clarify how these operators and functions are used. The P/T set "**a**" (all) is the argument to these operators.

`f {breakpoint(a) | watchpoint(a)} dstatus`
: Shows information about all threads that stopped at breakpoints and watchpoints. The "**a**" argument is the standard P/T set indicator for all.

`f {stopped(a) - breakpoint(a)} dstatus`
: Shows information about all stopped threads that are not stopped at breakpoints.

`f {g.3 - p6} duntil 577`
: Runs thread 3 along with all other processes in the group to line 577. However, it does not run anything in process 6.

7. CONCLUSIONS

Debugging parallel programs barely resembles serial debugging. Individually focusing on a process or a thread when the program can have thousands or tens of thousands of each is impractical. While a developer may ultimately want to look at the behavior of one process or thread, the developer must have the ability to organize processes and threads into groups having sharing similar characteristics. While some execution features can help the debugger automatically create predefined groups, only the developer can know what a group's contents should be.

Debugging parallel programs fundamentally alters the way in which the debugger controls interactions. Each group has a context—what was described as the GOI, POI, and TOI—and the debugging action must be based on this context and this concept is in relation to the width of the group.

The TotalView Debugger was designed from the ground up to understand how multiprocess and multithreaded programs. Its group model automatically creates groups and these groups are often all that is needed for disciplines such as MPI and OpenMP. However, its grouping features also give developers the flexibility they need to abstract and model the way in which programs, processes, and threads interact.

8. REFERENCES

[1] TotalView Technologies. *TotalView Debugger Users Guide.* TotalView Technologies, Natick, MA, 2007.

[2] TotalView Technologies. *TotalView Debugger Reference Guide.* TotalView Technologies, Natick, MA, 2007.

[3] High Performance Debugging Forum. High Performance Debugging Standards Effort, ed. J. Francioni and C. Pancake, Dept of Computer Science, Oregon State University, 1998.