Debugging with TotalView Parallel Debugger 
Profiling with Sun Studio

Seppo Saharakorpi, PhD
SEAS IRCS, User Services Lead

SEAS IRCS: ircshelp@seas.harvard.edu
FAS RC: rchelp@fas.harvard.edu

SEAS IRCS Online documentation: http://ircs.seas.harvard.edu/USERDOCS/
FAS RC: http://rc.fas.harvard.edu/
Debugging - TotalView Parallel Debugger

• Introduction

• Getting Started
  – Compiling debuggable code, Launching TotalView (on SEAS HPC systems), and Basic concepts (main GUI, stack trace, and execution, process and thread control)

• Examples
  – e.g. Variable queries, and data visualization, Message Queue Graphs, and Memory / Heap management
  – Live data visualization, and model parameter modification

• Resources for more information

• Summary and Discussion
Profiling – Sun Studio SW Development Kit

- Introduction
- Code Performance and Optimization
  - Amdahl’s law
  - Quick review on optimization options
- Profiling C/C++ or Fortran programs
  - Compiling the code for profiling
  - Executing the code
  - Analyzing the results using Sun Studio
- Matlab Profiler
- Resources for more information
- Summary and Discussion
TotalView Parallel Debugger – Intro

• Built from ground-up for debugging massively parallel codes
• MPI, pthread, and OpenMP process level support for program execution control (depending on the OS kernel support)
• Reverse debugging allows one to step backwards from any point in execution
  – Not available at SEAS due to additional licensing cost
• Supported on most *NIX platforms, but not on Windows
Start using TotalView

• Connect to ‘hpc’, and start an interactive session
  – `ssh -AY hpc.seas.harvard.edu`
  – `qlogin -pe orte 8`

• Load the necessary SW modules
  – `module load compilers/intel/10.1`
  – `module load mpi/openmpi/1.2.8/intel`
  – `module load packages/totalview/8.6.1-1`

• Compile a ‘debuggable’ executable
  – `mpif90 -g -O0 -o mympicode.out mympicode.f90`
Start using TotalView (continued)

• Start the program under TotalView’s control
  – totalview mympicode.out
  – Select ‘Parallel’ tab, and ‘Open MPI’ from the pull down menu.
  – Then select number of tasks (8 in this case). Click OK
  – The main TotalView GUI window shows the source code

• You are now ready to start debugging...
Main View
Control Buttons
All Processes
Stack Trace
Stack Frame
Breakpoint (red)
Current position (yellow arrow)
Source code
Tabs for breakpoints, processes, and threads
Using TotalView - Basics

• Control Buttons
  – Go – Run the code
  – Halt, Kill, and Restart – ‘Stop’, or ‘Kill’ execution, or ‘kill and restart’
  – Next – Step one line of code, stepping over subroutines
  – Step – Step one line of code, stepping into subroutines
  – Out – Execute code to the end of and exit from current subroutine

• Action points, e.g. breakpoints
  – Left click the line number to set or unset a simple breakpoint
  – Right click and ‘Properties’ for additional settings

• Variable values
  – Right click on top of variable, and choose ‘Dive’ or ‘Across Processes’
  – In the pop-up, try e.g. ‘Tools’ -> ‘Visualize’, or ‘Tools’ -> ‘Statistics’
Viewing Data and Basic Visualization

Data values (changed values are in yellow)

Data visualization

Data statistics
Automated Actions, e.g. Visualization

Action point -> Evaluate

\$\text{visualize} (v, "[::4][::4]");
\$\text{visualize} (v, "[127:127]");
\$\text{stop};
Very Basic Data Manipulation, and Message Queue (Graph)

Action point -> Evaluate

ISEND = 3

Value of ISEND is changed and output looks correct, but the program never finishes??

Tools -> Message Queue Graph reveals the problem
More information

• SEAS IRCS TotalView documentation:
  – https://ircs.seas.harvard.edu/display/USERDOCS/How+to+use+the+TotalView+debugger

• Official TotalView documentation:
  – http://www.totalviewtech.com/support/documentation.html

• Using Evaluation Expressions:
Debugging - Summary and Discussion

• **TotalView Parallel Debugger** is designed from ground-up to debug massively parallel codes with full MPI process, and pthread/OpenMP level control.

• **TotalView** is also useful as a multi-purpose SW development and modeling tool beyond debugging:
  – Visualize data structures and model evolution on the fly
  – Modify parameters on the fly and observe resulting changes
Profiling -> Optimization -> Performance

• Profiling, definition in Wiki:
  – A form of dynamic program analysis (as opposed to static code analysis), is the investigation of a program's behavior using information gathered as the program executes.
  – The usual purpose of this analysis is to determine which sections of a program to optimize - to increase its overall speed, decrease its memory requirement or sometimes both.

• Profiling helps understanding the code
  – Time spent per line, per subroutine, and per function
  – Number of calls per line, per subroutine, and per function
  – This information can help and guide the optimization effort substantially
Amdahl’s Law

• Max speedup is capped by whatever fraction of work must be done without optimization.

\[
\text{Speedup} = \frac{\text{Time}_{\text{old}}}{\text{Time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}
\]

Total speed-up assuming 5-times speed-up in some part (%) of the code
Amdahl’s Law – Parallel programs

• Max speedup is capped by whatever fraction of work must be done sequentially.

• Example: If for a serial version of a code:
  • 20 hours spent on part that can be parallelized (P)
  • 1 hour is purely sequential
  • Then max speedup is 20x

Source: http://en.wikipedia.org/wiki/Amdahl%27s_law
Aspects of Serial Performance

CPU Time = \frac{N_{\text{inst}} \times \text{CPI} \times \text{Clock rate}}{\text{program}}
Tips for Enhancing the Code Performance

- Understand your goals for the code and performance
- Prototype in a language/runtime that is “fast” to code in, if not to run.
- Profile to understand your bottlenecks
- Recast algorithms if possible to alleviate bottlenecks
  - Use analytic results where appropriate...
  - Use precalculated lookup tables...
  - Change meshing, integrator, etc. based on empirical results...
- Recode into faster runtime and use fast compiler flags
  - e.g. write key parts in fast low level (pre-compiled) language
- Utilize optimized or parallelized libraries
- Parallelize, using high abstraction level tools first
  - First Scalapack, CULA...Then MPI, OpenMP, CUDA
## Estimates for Gains from Optimization & Parallelization

<table>
<thead>
<tr>
<th>Approach</th>
<th>Potential Gain</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profiling</td>
<td>Know what to focus on!!</td>
<td>Low: learning the tools</td>
</tr>
<tr>
<td>Recast Algorithm or Problem</td>
<td>&gt; 1000 x</td>
<td>Variable, May be high</td>
</tr>
<tr>
<td>Compiler flags</td>
<td>10% - 50%</td>
<td>Low</td>
</tr>
<tr>
<td>Vectorization</td>
<td>Up to 50x</td>
<td>Some code rewriting</td>
</tr>
<tr>
<td>MPI</td>
<td>10x – 100x ...</td>
<td>Significant recoding</td>
</tr>
<tr>
<td>Embarrassingly parallel</td>
<td>10x – 1000x - ...</td>
<td>Minor recoding, scripting</td>
</tr>
</tbody>
</table>
Using Sun Studio for profiling

• Connect to ‘hpc’, and start an interactive session
  – `ssh -AY hpc.seas.harvard.edu`
  – `qlogin -pe orte 8`

• Load the necessary SW modules
  – `module load compilers/sunstudio/12`

• Compile the executable (the same as for debugging)
  – `f90 -g -O0 -o myslowcode.out myslowcode.f`
Using Sun Studio for profiling (continued)

• Execute the program and collect performance data
  – `collect ./myslowcode.out`
  – This will create a new directory `./test.NN.er`, which contains all the information regarding the performance of your program.

• Analyze the results using the GUI tool
  – `sunstudio`
  – Choose first the 'Advanced' pull-down menu, and then 'Open Experiment'. You should select the `./test.NN.er` directory, and then click 'Open'. Sun Studio will open the Experiment and shows by default the time spent in all functions of the program. Choose the 'Source' tab, and you see the source code and the time the program spent on each line.
Using Sun Studio

e.g. Add line-by-line info:
Set Data Presentation -> Tabs -> Lines

7/7/2010
http://ircs.seas.harvard.edu
Using Matlab Profiler

• Connect to ‘wumpus’
  – `ssh -AY wumpus.seas.harvard.edu`

• Load the necessary SW modules
  – `module load packages/matlab/r2010a`

• Start Matlab and Profiler
  – `matlab`
  – `profile on`
  – `...matlab code...`
  – `profile viewer`
More information

• SEAS IRCS Sun Studio documentation:
  – http://ircs.seas.harvard.edu/USERDOCS/How+to+use+the+Sun+Studio+profiler

• Sun Studio documentation:
  – http://developers.sun.com/sunstudio/documentation/

• Matlab Profiling documentation
Profiling – Summary and Discussion

• **Focus, Focus, Focus**
  – Due to Amdahl’s Law it is essential to understand where the time is spent in the program and focus on those parts
  – Profiling reveals the bottlenecks and time sinks

• **Profiling helps understanding the code**
  – Time spent per line, per subroutine, and per function
  – Number of calls per line, per subroutine, and per function
  – This information can help and guide the optimization effort substantially