Module 6: Parallel Debugging

- **Objective**
  - Learn the basics of debugging parallel programs with PTP
- **Contents**
  - Launching a parallel debug session
  - The PTP Debug Perspective
  - Parallel Breakpoints
  - Current Instruction Pointer
  - Process sets: controlling sets of processes
  - Process registration: controlling individual processes
The PTP Debug Perspective (1)

- **Parallel Debug** view shows currently running jobs, including the processes in the current process set.
- **Debug** view shows threads and call stack for individual registered processes.
- **Source** view shows the current instruction pointer for all processes.
The PTP Debug Perspective (2)

- **Breakpoints** view shows breakpoints that have been set (more on this later)
- **Variables** view shows the current values of variables for the currently selected process in the **Debug** view
- **Outline** view (from CDT) of source code
Process Sets (1)

- Traditional debuggers apply operations to a single process
- Parallel debugging operations apply to a single process or to arbitrary collections of processes
- A process set is a means of simultaneously referring to one or more processes
Process Sets (2)

- When a parallel debug session is first started, all processes are placed in a set, called the **Root** set.
- Sets are always associated with a single job.
- A job can have any number of process sets.
- A set can contain from 1 to the number of processes in a job.
Operations on Process Sets

- Use the icons in the toolbar of the **Parallel Debug** view to create, modify, and delete process sets, and to change the current process set.
- Current process set is listed next to job name along with number of processes in the set.
- Debug operations on the **Parallel Debug** View toolbar always apply to the current set:
  - Resume, suspend, stop, step into, step over, step return

---

*Module 6*
Breakpoints

- Two types of parallel breakpoints
  - Global breakpoints
    - Apply to all processes, all jobs
  - Set Breakpoints
    - Apply only to processes in a particular set (which can include the root set) for a single job
    - When the job completes, the breakpoints are no longer available
  - Set breakpoints are colored depending on which processes the breakpoint applies to:
    - Green indicates the breakpoint set is the same as the current set.
    - Blue indicates some processes in the breakpoint set are also in the current set (i.e. the process sets overlap)
    - Yellow indicates the breakpoint set is different from the current set
Setting Breakpoints

- To create a set breakpoint
  - Make sure the current job is selected
  - Select the root process set, or any other set
  - Double-click on the left edge of an editor window, at the line on which you want to set the breakpoint
  - Or, right click and use the context menu

- To create a global breakpoint
  - First make sure that no jobs are selected (click in white part of jobs view if necessary)
  - Double-click on the left edge of an editor window
  - Note that if a job is selected, the breakpoint will apply to the current set
Breakpoint Information (1)

- Hover over breakpoint icon
  - Will show the sets this breakpoint applies to
- Select **Breakpoints** view
  - Will show all breakpoints in all projects
Breakpoint Information (2)

➡️ Use the menu in the breakpoints view to group breakpoints by type
➡️ Breakpoints sorted by breakpoint set (process set)
Current Instruction Pointer (1)

- The current instruction pointer is used to show the current location of suspended processes.
- In traditional programs, there is a single instruction pointer (the exception to this is multi-threaded programs).
- In parallel programs, there is an instruction pointer for every process.
- The PTP debugger shows one instruction pointer for every group of processes at the same location.
Current Instruction Pointer (2)

The highlight color depends on the stack frame:
- Green: Registered Process
- Brown: Unregistered Process
- Blue: Tracks current stack frame

The marker depends on the type of process stopped at that location

Hover for more details about the processes suspend at that location
Process Registration (1)

- Process set commands apply to groups of processes
- For finer control and more detailed information, a process can be registered and isolated in the Debug View
- Registered processes, including their stack traces, appear in the `Debug` view
- Any number of processes can be registered, and processes can be registered or un-registered at any time
Process Registration (2)

- To register a process, double-click its process icon in the Parallel Debug view.
- Note that the process icon is surrounded by a box. The process appears in the debug view.
- To un-register a process, double-click on the same process icon.
- Debug commands in the Debug view control the single process that is currently selected in that view.

Registered processes

Debug commands apply to the currently selected process

Registered processes appear in Debug view
Terminating a Debug Session

- Click on the terminate icon in the **Parallel Debug** view to terminate all processes
- Click on the terminate icon in the **Debug** view to terminate the currently selected process
Basic Debug Commands

1. Launch debug session
   ✪ Using helloMPI program
   ✪ Use same launch configuration used for running
2. Step Over
3. Watch variable change
4. Set a breakpoint
5. Run to breakpoint
Debug Actions on Processes and Process Sets

1. Register a (different) process
2. Step the registered process
3. Create a process set (select process icons first)
4. (a) step and (b) run the set
5. Terminate debug session
Debugging a More Complex Application

- Shallow water weather model
- Finite difference model of shallow water equations based on
  - "The dynamics of finite difference models of the shallow water equations" by R. Sadourny, JAS, 32, 1975
- Code from
  - "An introduction to three-dimensional climate modeling" by Washington and Parkinson
- Already in your workspace
  - Also on tutorial CD in samples folder
Code Structure

- Distributed memory model
- Master/worker
  - Master is task ID 0
  - Master initializes data and sends to slaves
  - Workers perform computation on their data
  - Workers may exchange data during computation
  - Workers send results back to master
- Double-click on main.c
  - `main()` is in `main.c`
- Master code follows test for `tid != 0` (line 87)
Code Structure (cont...)

- Worker code starts at `worker()` in `worker.c`
- Worker receives data from master (line 107)
- Main time step loop starts on line 137
- Computation is performed by call to `calcuvhz()` (line 145)
  - `calcuvhz()` is located in `calc.c`
- Time tendencies are calculated by call to `timetend()` (line 154)
  - `timetend()` is located in `tend.c`
- Results are returned at line 180
Building the Application

- Switch to **C/C++ perspective**
- Select the **Make Targets** view
- Open the **shallow** project
- Double-click on the **all** target
- Bring **Console** view to front to see build progress
Running the Application

- Select Run > Run... to open the run configuration dialog
- Select Parallel Application
- Select the New button
- Select the shallow project if not already visible
- Select Browse and choose the shallow executable for Application program
- Select the Parallel tab and enter 3 for Number of processes
  - must be an odd number
- Select the Debugger tab and choose SDM from the Debugger drop down menu
- Select Apply, then Run
- Eclipse will automatically switch to the PTP Runtime perspective
Examining Output

- Three process icons should be visible in the **Jobs** view
- Double-click on process 0 and the standard output should be visible in the **Process** view
- Examine the value for **Potential energy** in Cycle 950
  - The correct value is **128225.086**
  - There must be a bug!
Debugging the Application

- Select **Run ➤ Debug History ➤ shallow**
- Or just click on the Debug icon on the toolbar
- The application should launch and switch to the **PTP Debug** perspective
Debugging Hints (1)

- There is an error somewhere in the worker calculation
- Place a breakpoint on the `worker()` call in `main.c`
- Continue execution by selecting the resume button (green arrow on **Parallel Debug** view)
  - You’ll notice that process 0 remains running (green) while processes 1 and 2 are suspended (yellow)
- **Step into the `worker()` call**
  - `worker.c` should open and show the current line marker on the first statement
- **Place a breakpoint in the main timestep loop**
  - Line 144 is a good place
- **Continue execution again**
  - The processes should now suspend at the breakpoint
Debugging Hints (2)

- Double click on the process 1 and process 2 icons
  - This will register the processes so you can view variables
- Compare variable values that should be the same across processes
  - Make sure Variable view is visible
  - Select stack frame of each process in turn
  - The ncycle, time and tdt variables are likely candidates since they should be the same for each process
- Continue execution and check values again
- When you locate a difference
  - Inspect code where the variable is updated or modified
  - This is likely to be the source of the error
- When you find the error
  - Exit the debugger, recompile and re-run the code