How to Develop Programs for Beagle2

- How to develop on Beagle2
  - Development Policies
  - Basics
  - Compilers user's guides and general information
  - Optimization
    - Optimization tools
    - Profilers
    - Using CrayPat
    - Sampling
    - Trace
  - Optimization by specific compilers
    - Cray compilers
    - GCC compilers
  - Debugging
    - Using TotalView

Development Policies

- Three nodes have been reserved for development.
- Monday through Friday between 8:00AM and 5:00PM 3 nodes are reserved for jobs that request less than 1 hour walltime.
- No additional job parameters are required, i.e., no particular queue or PBS switch needs to be specified, and any job that fits these requirements will run in the reservation.

Basics

- The development environment on a Cray is controlled by using the module command, see the Getting started: performing basic operations on Beagle2 section of this wiki for details.
- Compilers:
  - The default compiler is PrgEnv-cray, and this can be confirmed by typing module list
  - The list of available compilation environments can be obtained by typing module avail PrgEnv. On Beagle2 the current list contains PGI, Cray and Gnu (Intel and PathScale might show on the list, but they are not currently available), see CrayDoc for more details about the compilers.
  - To change compilation environment, for example from Cray to PGI, module swap should be used, in this example module swap PrgEnv-cray PrgEnv-pgi. No other changes are necessary and all the environmental variables should be properly set according to the new compilation Environment (if they don't, make sure you contact us).
  - To obtain more information type, for example, module show PrgEnv-pgi (Note: the module does not need to be loaded to use module show, it will also give you some information about compilers that are not currently available)
- The driver scripts are always called ftn (fortran), cc (c), CC (c++) independently from the specific compiler being used (e.g., PGI, Cray or Gnu). (Note: technically, there is a difference between producing an executable for testing/profiling on a login node and producing an executable for a compute node. Remember: you cannot execute parallel programs on the login node. Also on login nodes OpenMP can give you misleading results and MPI is not allowed)
- Libraries:
  - Many can be loaded as native libraries, for example:
    - FFTW, via module load fftw. Cray has added the environment variable FFTW_SYSTEM_WISDOM_DIR, which is set to the location of a particular installation's system wisdom directory. The fftw/3.2.x module automatically sets this up
    - See CrayDoc for additional libraries
    - If you can't find the libraries you need on CrayDoc, before you attempt to optimize you own libraries please contact our support, we might have already done it (beagle-support@lists.uchicago.edu)
  - Setting LDFLAGS=-Wl,-z,muldefs can help port programs when there are multiple definitions (work for gcc)

Compilers user's guides and general information

URLs for user's guides:

- GCC http://gcc.gnu.org/onlinedocs/
- Cray (under “Programming Environment”) http://docs.cray.com/cgi-bin/craydoc.cgi?

You can also use the man pages when logged in on Beagle2:

- GCC man gfortran, man gcc, man g++
- Cray man crayftn, man craycc, man crayc++

Note: while the names are different on the man pages, to compile for the compute nodes one should use the driver scripts ftn, cc and CC for all any of the four available environment. The man pages usually work only when the relevant compiler environment is loaded. Please see our Compilation Guide.
Static vs. Dynamic linking

- In ESM libraries are generally linked statically (default) during the building of executables both because of potential conflicts between nodes (which would affect performance) and because the Linux environment on the compute nodes running in ESM is minimized. It is therefore recommended to use static libraries, particularly for larger jobs. However, dynamic linking is also possible, for additional material on how to use shared libraries, see the document from Cray Cray XT Programming Environment’s Implementation of Dynamic Shared Libraries, but be mindful that things might not work as described.
- Setting XTPE_LINK_TYPE=dynamic is usually not sufficient to successfully compile code into executables that make use of shared libraries. To compile dynamic, add "--dynamic" to the compile command line. However, the default Cray Message Passing Toolking, xt-mpt, will work for static linking only because it has issues with dynamic linking. If you wish to link dynamically, please use cray-mpich (MPI) or cray-shmem (SHMEM) instead of cray-mpt, depending on whether your code uses MPI or SHMEM message-passing, e.g., by typing module switch cray-mpt cray-mpich (This later switch is necessary only for older compiler environments)
- Not all compilers (PGI, Gnu or Cray) behave identically with respect to these issues.
- If you need help with dynamically or statically linked executables, please contact us.

Optimization

- Optimization specific for the "Abu Dhabi" processors of Beagle2 can be set up using xtpe-mc12 either as an optimization flag or as module load xtpe-mc12 (load the module specifically optimized for the "Abu Dhabi" processor with 16 cores). This optimization should be set by default to be specific to Beagle2, and it can be verified by typing module list.
- The more aggressive optimization is, the more likely is that at least for some parameter values or simulation configuration it will produce incorrect results. The incorrect results might not always be easy to spot (e.g., the program crashes or produces a completely absurd output). We strongly recommend users to check very carefully whether the optimization produced unacceptable side effects.
- Huge pages are virtual memory pages which are bigger than the default base page size of 4KB. Some applications may perform better when large memory pages are used. Add the option -lhugetlbfs when memory usage, specifically memory which is mapped through the high speed network, exceeds 2GB on a single node. An application should be linked with the libhugetlbfs library to use the larger address range available with huge pages. At run time, set HUGETLB_ELFMAP=W to map static data to huge pages and set HUGETLB_MORECORE=yes to map the private heap to huge pages.

The memory available for huge pages is less than the total memory on the node. You must leave enough memory for CNL and I/O buffers. Also, because of memory fragmentation, less memory is available for huge pages after a node has run other jobs.

There is no guaranteed amount of huge page memory available to an application. Memory allocated as huge pages is unavailable for I/O, whether the application uses the memory or not. Less available memory for I/O buffers may result in performance degradation.

Type man intro_hugepages

Optimization tools

Profilers

To optimize the performance of calculations it is often necessary or at least helpful to have an empirical idea about which are the parts that take the most time to be executed. This is the purpose of profilers: collect information about the behavior of a program while in execution.

Cray provides a suite of tools for the profiling of code developed and/or ported to the Cray XE6. To load them type module load perftools and it will look the three basic packages:

- Craypat
- Apprentice2
- Papi

Most of them will provide sampling (a measure of how much time it is spent in each part of the program) and/or tracing (what is the timeline of the execution of the program) data.

Using CrayPat

CrayPat is a performance analysis tool that collects performance information from a user application. Production codes should NOT use CrayPat. To use CrayPat:

CrayPat consists of three major components:
- pat_build – used to instrument the program to be analyzed
- pat_report – a standalone text report generator that can be used to further explore the data generated by instrumented program execution

Using CrayPat assumes that you have loaded the default Cray environment.

Step 1: Load CrayPat module files

module load perftools ##needs to be loaded before you start building your application!
Step 2: Build application

cc -c myapp.c
cc -o myapp myapp.o ### keep .o files! Compile and link in separate steps.

Step 3: Instrument the original program

pat_build myapp #this will create: myapp+pat

Step 4: Run the instrumented executable

qsub test.pbs ###this produces the data file myapp+pat+PID-nodesdt.xf

vi test.pbs
#!/bin/bash
. /opt/modules/3.2.6.7/init/bash
#PBS -A CI-<my_account>
#PBS -N test
#PBS -j oe
#PBS -l walltime=00:10:00
#PBS -q batch
#PBS -l mppwidth=32
cd $PBS_O_WORKDIR
module load perftools
setenv PAT_RT_EXPFILE_DIR $LUSTREDIR/mydir        ##Specify a Lustre directory when you create a single experiment output file.
aprun -n 1 ./myapp+pat                            ##Run the executable. This will generate a performance data file with the suffix, 'xf'

Step 5: Use pat_report to process the data file

pat_report myapp+pat+PID-nodesdt.xf ###Run pat_report on the data file to view the results!
# This produces an .ap2 file (myapp+pat+PID-nodesdt.ap2), which contains both the report data and the associated mapping from addresses to functions and source line numbers.

Cray compiler wrappers (ftn, cc and CC) must be used for building an executable, instead of native compiler commands (pgf90, pgcc, pgCC, gfortran, gcc, g++, etc.) since pat_build cannot build an instrumented executable from an executable built with a native compiler.

Note: Object files (*.o files) need to be made available to CrayPat to correctly build an instrumented executable for profiling or tracing. In other words, compile and link stage should be separated by using the -c compile flag.

Sampling

CrayPat supports sampling experiments: capture values from the call stack or the program counter (PC indicates where a computer is in its program sequence) at specified intervals or when a specified counter overflows. The default experiment type is to sample the PC at a time interval (i.e., samp_pc_time). Type man intro_craypat.

- In above PBS script include this line: pat_build myapp
- This generates a new executable, myapp+pat. Run this executable on compute nodes, as you would with the regular executable. This will generate a performance data file with the suffix, 'xf'. Example: myapp+pat+xxxx-yyzz.xf where xxxx is the process ID, yy the physical node, zz the type of experiment performed.
- pat_report reads the experiment file(s) and produces a text report and an .ap2 file. Cray Apprentice2 generates and displays graphical reports from the .ap2 file (ap2 is portable).
- Use sampling to obtain a profile and then trace function of interest.

Trace

Tracing counts an event, such as the number of times an MPI call is executed.

- pat_build -w myapp to track the entire program as a whole.
- pat_build -w -T func1,func2 myapp to instrument user-defined functions, _func1 and func2 (tracefunc excludes function)
- pat_build -w -t tracefile myapp to trace a group of functions you list in a text file, tracefile
- pat_build -u myapp to trace all the user-defined function
- pat_build -g mpi,omp,heap -u myapp to specify the function group name
Note: Tracing the entire user functions can slow down the code significantly if it contains many small and frequently called functions.

Optimization by specific compilers

- Please, in general refer to the user's guides or type `man ftn` or `man cc`.
- The default compiler is the Cray compiler. You need to change compilation environment if you want to use a different compiler, see below.

Cray compilers

- The Cray compilers are the default compilers at Beagle2.

GCC compilers

- First, to use the Gnu Compiler Collection, type `module swap PrgEnv-cray PrgEnv-gnu` (unless you already swapped programming environment, of course).
- `cc -Q --help=optimizers` option, displays the list of optimization options and whether they are enabled, disabled, or set to a specific value.
- The output is sensitive to the effects of previous options. Therefore if one is interested in knowing what is the difference between `-O3` and `-O2` it is sufficient to compare the output of `ftn -O3 -Q --help=optimizers` and `ftn -O2 -Q --help=optimizers`.
- Even `-O3` does not turn on all the possible optimization flags. Other flags that can be added are:
  - `-funroll-loops` to unroll loops only if the number of iterations is known at compiler time or on entry to the loop
  - `-funroll-all-loops` to unroll all loops
  - `-ffast-math` sets also a number of other flags (can be checked by typing `ftn -O3 -ffast-math -Q --help=optimizers` and comparing it with `ftn -O3 -Q --help=optimizers`, e.g., using the `diff` command, and looking for the meaning of the flags using `ftn -O3 -Q --help=optimizers`
- Should an optimization level fail to produce reliable and acceptable results, say `-O3` produces a program that produces inaccurate results, it is possible to move to a lower level, say `-O2` and then, should the lower level work, add progressively some of the optimization flags included in the higher level until the problematic optimization(s) are identified and avoided.

Debugging

If you want to use the debugger from the mom nodes, they need to call it like this:

```
qsub -I -q advanced -l mppwidth=32 -X
```

Using TotalView

TotalView is a parallel debugging tool that can be run with up to 512 processors. It provides both X Windows-based User Interface (GUI) and command line interface (CLI) environments for debugging. To use TotalView:

1. `ssh -Y username@login.beagle.ci.uchicago.edu` #to log in on Beagle with an Y window forwarding enabled. `-Y` enables X11 forwarding.
2. `echo $DISPLAY` #verify your DISPLAY settings
3. `localhost:x.x` #this should be result. If nothing shows up you have to install X11 software.
4. `cat /proc/cray_xt/nid` #your current node
5. `module load xt-totalview` #to load the totalview modulefile in order to set the correct environment settings.
6. `cc -g -o a.out performTV.c` #Compiling code to run totalView. Code must be compiled with the `-g` option!
7. `qsub -I -q advanced -l mppwidth=32 -X
   echo $DISPLAY` #After the job is ready verify the node you are on, and your DISPLAY settings.
8. `localhost:x.x` #PBS pointed you to the different node you have to change DISPLAY variable: `setenv DISPLAY workstation:0.0` #to launch the debugger with the totalview command followed by the name of the executable to debug.
9. `totalview aprun -a -n numOfTasks ./a.out` #Partial View startup GUI will pop up and show debugging startup parameters. In the Parallel tab, select: your MPI preference, number of tasks, number of nodes...After reviewing them, click OK.
10. `Main Window` and `Process Window` should appear like this:
11. `Main window` or `root window` (as shown below).

- Will always appear when the TotalView GUI is started.
- Provides an overview of all processes and threads, showing the TotalView assigned ID, MPI rank, host, status and brief description/name for each.
- Allows sorting by ID, rank, host and status.
- Provides the ability to expand/collapse each process to view/hide any included threads in that process.
- Pull-down menus - File, Edit, View, Tools, Window, Help (menus are discussed later)

1. In this example it shows you that aprun is running, but the application is not loaded (Status -).
2. Processes and threads status codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Stopped at a breakpoint</td>
</tr>
<tr>
<td>E</td>
<td>Stopped because of an error</td>
</tr>
<tr>
<td>H</td>
<td>In a Hold state</td>
</tr>
</tbody>
</table>
5. **Process window** (shown below). Usually (but not always) appears with the **Root Window** after **TotalView** is started.

6. **Run the program**: press **GO** selected from the **Process Window**’s execution button panel. A small window will then pop up, asking whether the aprun process should be stopped. Click “Yes”. **TotalView** will connect to the server and load the application.

8. **Note** that since no breakpoints were set, the program simply runs to completion. **Note** also that the program’s output is displayed in the window where you started **totalview**.

10. **Note**: When running a multi-process and multi-threaded program, **totalView** tries to automatically place your executing processes into different groups. **TotalView** lets you group your threads and processes. Most of **TotalView**’s execution control commands can be applied at the **Group**, **Process** or **Thread** scoping level. The right scope depends upon what you want to effect.

**TotalView** automatically organizes your programs and threads into the following **predefined groups**:

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

- **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, your 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. These threads can reside in more than one share group.

- **Lockstep Group**: All threads that are at the same PC (program counter). This group is a subset of a workers group. A lockstep group only exists for stopped threads. By definition, 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. A lockstep group only means anything when the threads are stopped.

---

1. Users can control many aspects of **TotalView**’s appearance and behavior through setting Preferences in Preferences window (Process Window > File Menu > Preferences). The “**Parallel**” panel has an option to “**ask what to do**” when launching a parallel program. When the program is run as a job starts you can select or deselect processes which you would like to **attach to a running process**. In other words, programs don't have to be started within **TotalView** to be debugged by **TotalView**. The “**Action**” panel define what happens when a **breakpoint** is set; stop a process or stop the group, etc.

2. **Set breakpoints** (action points) and begin debugging. **Required if the application doesn't stop (hang) on error**. Breakpoints are used to control a program’s execution. It causes a process/thread to halt execution at the line number - prior to executing that line number. Breakpoints can be set in source code and assembler code. For regular source, only “boxed” line numbers are eligible for breakpoints. For setting the initial breakpoint: select “Group (Share)" and right mouse click anywhere on the desired source line until the pop-up menu appears (right). Then select **Set Breakpoint**. A STOP icon will appear here and also in the **Action Points Pane**, indicating that the breakpoint has been set. After that, you can use other buttons (‘Next’, ‘Step’, ‘Out’, etc.). By default, **TotalView** shares the breakpoint.

- **Lines that are not executable do not have the box**
- **If the user source code is not displayed** in the process window at this stage, select the main routine in the **stack trace pane** (the upper left frame) of the process window.
- **Control group** - all processes of the program (forked)
- **Share group** - all processes of the program that use the same code (executed from the same binary. A share group is a single MPI application. The NEXT, STEP, GO etc... operations work on a shared group.
- **The control and share groups only contain processes; the workers and lockstep groups only contain threads.**
- **The application runs in parallel until the breakpoint**
- **To unset the breakpoint**, simply click on the red STOP icon or select "delete" from the pop-up menu or **Action Point** menu.
- **In many cases**, you are setting a breakpoint at a place where you hope the program is still executing correctly. Because you are debugging a multi-process, multi-threaded program, **set a barrier point so that all threads and process will stop at the same place**.
- **Run the program again**: press **GO** on the process window. When the program hits the breakpoint and stops, notice the Program Counter (yellow arrow).
- **If you’re debugging an MPI program**, the **Tools > Message Queue Graph Window** graphically displays the program’s message queues.
- **Diving**: enables you to view more detail about a data containing object (such as an array variable) by “diving” into it. To enable diving double left clicking on an object. What happens when you dive on an object depends upon the object. The table below describes most cases.

<table>
<thead>
<tr>
<th>Object</th>
<th>Where Object Is Located</th>
<th>What Happens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process or thread</td>
<td>Root Window</td>
<td>Process/thread is displayed in an existing Process Window. If none exists, then a new Process Window appears for the selected process/thread.</td>
</tr>
<tr>
<td>Routine</td>
<td>Process Window Stack Trace Pane</td>
<td>Stack Frame and Source Code panes in the Process Window are updated with information for the selected routine.</td>
</tr>
<tr>
<td>Subroutine</td>
<td>Process Window (in Source Code Pane)</td>
<td>Source code appears in the Process Window</td>
</tr>
</tbody>
</table>
Table: 

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Process Window</th>
<th>Referenced memory area appears in a new Variable Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable, array, address</td>
<td>Process Window</td>
<td>Variable contents appear in a new a Variable Window</td>
</tr>
<tr>
<td>Element of an array or structure</td>
<td>Variable Window</td>
<td>Contents of element appear in the Variable Window. Example of a &quot;nested&quot; dive.</td>
</tr>
</tbody>
</table>

• To undive: Find the undive button located in the upper right corner of the Source Code Pane.
• Viewing and Modifying Data: Process Window > View Menu > Lookup Variable. Leaving a Variable Window open allows you to perform runtime monitoring of variables. TotalView will update its contents each time the program is stopped. You can edit variables from within the Variable Window. Simply click on the variable with the Select (left) mouse button. This will select the variable for field editing. The modified variable has effect when the program resumes execution.

Arrays:
• Displaying array slices: Can be entered in the Slice: field in the Variable Window.
• Data filtering: Can be entered in the Filter: field in the Variable Window.
• Data sorting: Simply click on the Value bar in a Variable Window. The array will sort in ascending order.
• Array statistics: Variable Window > Tools Menu > Statistics
• File Menu > Exit
• Video tutorial