Debugging and Optimization Tools

Richard Gerber
NERSC
User Services Group Lead

Thanks to Woo-Sun Yang and Helen He
Outline

• Take-Aways
• Debugging
• Performance / Optimization
• NERSC “automatic” tools

Videos, presentations, and references:

http://www.nersc.gov/users/training/courses/CS267/

Help at NERSC
consult@nersc.gov
https://help.nersc.gov
Take-Aways

• Tools can help you find errors in your program and locate performance bottlenecks

• In the world of HPC parallel computing, there are few widely adopted standard tools
  – Totalview and DDT debuggers
  – PAPI, Tau, & vendor-specific performance tools

• Common code problems

• How tools work in general

• Use the tools that works for you and are appropriate for your problem

• Be suspicious of outliers among parallel tasks

• Where to get more information
Debugging
What is a Bug?

- A bug is when your code
  - crashes
  - hangs (doesn’t finish)
  - gets inconsistent answers
  - produces wrong answers
  - behaves in any way you didn’t want it to

The term “bug” was popularized by Grace Hopper (motivated by the removal of an actual moth from a computer relay in 1947)
Common Causes of Bugs

• “Serial” (Sequential might be a better word)
  – Invalid memory references
  – Array reference out of bounds
  – Divide by zero
  – Use of uninitialized variables

• Parallel  Let’s concentrate on these
  – Unmatched sends/receives
  – Blocking receive before corresponding send
  – Out of order collectives
  – Race conditions
  – Unintentionally modifying shared memory structures
What to Do if You Have a Bug?

• **Find It**
  – You want to locate the part of your code that isn’t doing what it’s designed to do

• **Fix It**
  – Figure out how to solve it and implement a solution

• **Check It**
  – Run it to check for proper behavior
This is just the oddest bug I've ever seen. It wasn't happening until I found the mistake reading the source code.

More than that, the bug manifests itself only after I 'fix' the broken code. It works otherwise.

Then the broken code is not really broken.

It is, just look at it.

What the hell is that?!

Oh my god, stop looking at it! The paradigm may rip the space time continuum!

I guess I forgot a little debug statement!
Find It: Tools

• **printf, write**
  – Versatile, sometimes useful
  – Doesn’t scale well
  – Not interactive
  – Fishing expedition

• **Compiler / Runtime**
  – Bounds checking, exception handling
  – Dereferencing of NULL pointers
  – Function and subroutine interface checking

• **Serial gdb + friends**
  – GNU debugger, serial, command-line interface
  – See “man gdb”

• **Parallel debuggers**
  – DDT
  – Totalview

• **Memory debuggers**
  – MAP
  – Valgrind

See NERSC web site
https://www.nersc.gov/users/software/debugging-and-profiling/
Parallel Programming Bug

This code hangs because both Task 0 and Task N-1 are blocking on MPI_Recv.

```
if(task_no==0) {
    ret = MPI_Recv(&herBuffer, 50, MPI_DOUBLE, totTasks-1, 0, MPI_COMM_WORLD, &status);
    ret = MPI_Send(&myBuffer, 50, MPI_DOUBLE, totTasks-1, 0, MPI_COMM_WORLD);
}
else if (task_no==(totTasks-1)) {
    ret = MPI_Recv(&herBuffer, 50, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
    ret = MPI_Send(&myBuffer, 50, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
```
NERSC NX – Accelerate Your X Connection

NERSC NX SERVICE - X-WINDOWS ACCELERATION AT NERSC

Introduction

NX is a computer program that handles remote X Window System connections and it provides three benefits for NERSC users:

- **SPEED**: NX can greatly improve the performance of X Windows, allowing users with slow, high-latency connections (e.g., on cell phone network, traveling in Africa) to use complex X Windows programs (such as rotating a plot in Matlab).
- **SESSION**: NX provides sessions that allow a user to disconnect from the session and reconnect to it at a later time while keeping the state of all running applications inside the session.
- **DESKTOP**: NX gives users a virtual desktop that’s running at NERSC. You can customize the desktop according to your work requirement.

![NX Example Image]

Related Information

- Download Tested NX Player
- Download NX Configuration File
- NX FAQ
Compile & Start DDT

Compile for debugging

```bash
edison% make
cr -c -g hello.c
cr -o hello -g hello.o
```

Set up the parallel run environment

```bash
edison% qsub -I -V -lmppwidth=24
edison% cd $PBS_O_WORKDIR
```

Start the DDT debugger

```bash
edison% module load ddt
edison% ddt ./hello
```
Press Go and then Pause when code appears hung.

Task 0 is at line 44

At hang, tasks are in 3 different places.
What About Massive Parallelism?

- With 10K+ tasks/threadsstreams it’s impossible to examine every parallel instance in detail
- Make use of statistics and summaries
- Look for tasks that are doing something different
  - Amount of memory used
  - Number of calculations performed (from counters)
  - Number of MPI calls
  - Wall time used
  - Time spent in I/O
  - One or a few tasks paused at a different line of code
- We (NERSC) have been advocating for this statistical view for some time
Vendors are starting to listen (DDT)

Statistics

Sparklines
DDT video

• http://vimeo.com/19978486

• Or http://vimeo.com/user5729706

• Linked to from http://www.nersc.gov/users/training/courses/CS267/
Debuggers on NERSC machines

• **Parallel debuggers with a graphical user interface**
  – DDT (Distributed Debugging Tool)
  – TotalView

• **Specialized debuggers on Hopper and Edison**
  – STAT (Stack Trace Analysis Tool)
    • Collect stack backtraces from all (MPI) tasks
  – ATP (Abnormal Termination Processing)
    • Collect stack backtraces from all (MPI) tasks when an application fails
  – CCDB (Cray Comparative Debugger)
    • Comparative debugging

• **Valgrind**
  – Suite of debugging and profiler tools

https://www.nersc.gov/users/training/courses/CS267/ for links to recent training presentations
STAT (Stack Trace Analysis Tool)

- Gathers stack backtraces (showing the function calling sequences leading up to the ones in the current stack frames) from all (MPI) processes and merges them into a single file (*.dot)
  - Results displayed graphically as a call tree showing the location in the code that each process is executing and how it got there
  - Can be useful for debugging a hung application
  - With the info learned from STAT, can investigate further with DDT or TotalView

- Works for MPI, CAF and UPC, but not OpenMP

- For more info:
  - ‘intro_stat’, ‘STAT’, ‘statview’ and ‘statgui’ man pages
Hung application with STAT

Ranks 1 & 2 are here

Rank 0 is here

Rank 3 is here
ATP (Abnormal Termination Processing)

• ATP gathers stack backtraces from all processes of a failing application
  – Invokes STAT underneath
  – Output in atpMergedBT.dot and atpMergedBT_line.dot (which shows source code line numbers), which are to be viewed with statview

• By default, the atp module is loaded on Hopper and Edison, but ATP is not enabled; to enable:
  – setenv ATP_ENABLED=1  # csh/tcsh
  – export ATP_ENABLED=1  # sh/bash/ksh

• For more info
  – ‘intro_atp’ man page
CCDB (Cray Comparative Debugger)

• Find a bug introduced in a version, by running two versions side by side and comparing data between them
• GUI
• Supports MPI; doesn’t support threading

For info:
  - ccdb man page and help pages
  - lgdb man page and help pages
Running CCDB

% qsub -IV -lmppwidth=48,walltime=30:00 -q debug
% cd $PBS_O_WORKDIR
% module load cray-ccdb
% ccdb

Request enough nodes to run two apps. simultaneously

Slide Courtesy of Woo-Sun Yang
Valgrind

• Suite of debugging and profiler tools
• Tools include
  – memcheck: memory error and memory leaks detection
  – cachegrind: a cache and branch-prediction profiler
  – callgrind: a call-graph generating cache and branch prediction profiler
  – massif, dhat (exp-dhat): heap profilers
  – helgrind, drd: pthreads error detectors
• For info:
Performance / Optimization
Performance Questions

• How can we tell if a program is performing well? Or isn’t? What is “good”?

• If performance is not “good,” can we identify the causes?

• What can we do about it?
Is Your Code Performing Well?

• No single answer, but
  – Does is scale well?
  – Is MPI time <20% of total run time?
  – Is I/O time <10% of total run time?
  – Is it load balanced?
  – If GPU code, does GPU+Processor perform better than 2 Processors?

• “Theoretical” CPU performance vs. “Real World” performance in a highly parallel environment
  – Cache-based x86 processors: >10% of theoretical is pretty good
  – GPUs, Xeon Phi: >1% in today’s real full HPC applications pretty good? **This your challenge!**
What can we do about it

- Minimize latency effects (aggregate messages)
- Maximize work vs. communication
- Minimize data movement (recalculate vs. send)
- Use the “most local” memory
- Take advantage of vector (SIMD) capabilities
- Use large-block I/O
- Use a balanced strategy for I/O
  - Avoid “too many” tasks accessing a single file, but “too many” files performs poorly (~1000s)
  - Use “enough” I/O tasks to maximum I/O bandwidth, but “too many” causes contention 1/node
Performance Landscape
Can We Identify the Causes? Use Tools

• **Vendor Tools:**
  – CrayPat on Crays
  – INTEL VTune

• **Community Tools:**
  – TAU (U. Oregon via ACTS)
  – PAPI (Performance API)
  – gprof

• **NERSC “automatic” and/or easy-to-use tools**
  – e.g. IPM, Darshan

See NERSC web site
https://www.nersc.gov/users/software/debugging-and-profiling/
Example: CrayPat

- Suite of tools that provides a wide range of performance-related information

- Can be used for both sampling and tracing
  - with or without hardware or network performance counters
  - Built on PAPI

- Supports Fortran, C, C++, UPC, MPI, Coarray Fortran, OpenMP, Pthreads, SHMEM

- Man pages
  - intro_craypat(1), intro_app2(1), intro_papi(1)
Using CrayPat

1. Access the tools
   - module load perftools

2. Build your application; keep .o files
   - make clean
   - make

3. Instrument application
   - pat_build ... a.out
   - Result is a new file, a.out+pat

4. Run instrumented application to get top time consuming routines
   - aprun ... a.out+pat
   - Result is a new file XXXXX.xf (or a directory containing .xf files)

5. Run pat_report on that new file; view results
   - pat_report XXXXX.xf > my_profile
   - view my_profile
   - Also produces a new file: XXXXX.ap2 that can be viewed with apprentice GUI application
Cray perftools and perftools-lite

- **Reports:**
  - execution time
  - memory high water mark
  - aggregate FLOPS rate
  - top time consuming user function
  - MPI information
  - IO information
  - hardware performance counters
  - load balance ...

- **Start with perftools-lite**
- **Available on Hopper and Edison.**
- **Documentation:**
  - [https://www.nersc.gov/users/software/debugging-and-profiling/craypat/](https://www.nersc.gov/users/software/debugging-and-profiling/craypat/)
  - [http://www.nersc.gov/assets/Uploads/UsingCrayPat.pdf](http://www.nersc.gov/assets/Uploads/UsingCrayPat.pdf)
  - [http://www.nersc.gov/assets/Uploads/Perftools-lite-2013.pdf](http://www.nersc.gov/assets/Uploads/Perftools-lite-2013.pdf)
Allinea MAP

- Allinea MAP is a parallel MPI profiler with GUI, small overhead.
- Reports: Memory usage, MPI usage, CPU time, CPU instructions, I/O, etc. as a function of time.
- Available on Hopper, Edison, and Carver.

Documentation:
Tools for the Masses

• **Using even the best tools can be tedious**
  – “Follow these 10 steps to perform the basic analysis of your program” – from a supercomputer center web site for a well-known tool

• **NERSC wants to enable easy access to information that can help you improve your parallel code**
  – automatic data collection
  – provide useful tools through the web

• **Efforts**
  – Work with vendors (e.g., CRAY ARU, Allinea Perf. Report)
  – IPM (MPI profiling, chip HW counters, memory used)
  – Accounting & UNIX resource usage
  – System-level I/O monitoring
  – User-level I/O profiling (Darshan)
## NERSC Completed Jobs

<table>
<thead>
<tr>
<th>#</th>
<th>Host</th>
<th>JobID</th>
<th>Job Name</th>
<th>User</th>
<th>Nds</th>
<th>Complete</th>
<th>Wall hrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>edison</td>
<td>782557</td>
<td>my_job</td>
<td>hdixit</td>
<td>11</td>
<td>02/17/14 13:14</td>
<td>0.160</td>
</tr>
<tr>
<td>1</td>
<td>edison</td>
<td>782431</td>
<td>cp_ref_big</td>
<td>hsinyu</td>
<td>13</td>
<td>02/17/14 13:14</td>
<td>0.849</td>
</tr>
<tr>
<td>2</td>
<td>edison</td>
<td>782591</td>
<td>tvsoi</td>
<td>wangyu</td>
<td>2</td>
<td>02/17/14 13:14</td>
<td>0.005</td>
</tr>
<tr>
<td>3</td>
<td>edison</td>
<td>782560</td>
<td>my_job</td>
<td>hdixit</td>
<td>11</td>
<td>02/17/14 13:13</td>
<td>0.131</td>
</tr>
<tr>
<td>4</td>
<td>edison</td>
<td>719618</td>
<td>nacl_big</td>
<td>carnevav</td>
<td>16</td>
<td>02/17/14 13:12</td>
<td>11.453</td>
</tr>
<tr>
<td>5</td>
<td>edison</td>
<td>782403</td>
<td>slab-Ni-pa...</td>
<td><img src="image" alt="Full Name" /></td>
<td>hdixit</td>
<td>11</td>
<td>02/17/14 13:12</td>
</tr>
<tr>
<td>6</td>
<td>edison</td>
<td>782590</td>
<td>NL_test</td>
<td>khkim</td>
<td>86</td>
<td>02/17/14 13:12</td>
<td>0.008</td>
</tr>
</tbody>
</table>

### Command Information

**APID**: 3140812

**Command**: gribmeanp.x

**Nodes**: 3

**Tasks**: 64

**Threads per Task**: 1

**Tasks per Node**: 24

**Max Task Mem (MB)**: 235

**Run Time (secs)**: 5

**Command Line**:

```
/apopt/cray/alps/default/bin/aprun -n 64 -N 24 /scratch1/scratchdirs/whitaker/edison/newstuff/bin/gribmeanp.x
```

**Node List**: 2628-2630
IPM: An Easy to Use Performance Tool

Just load the module, relink, and run.

```
# host   : s05601/006035314C00_AIX
# start  : 11/30/04/14:35:34
# stop   : 11/30/04/14:36:00
# gbytes : 6.65863e-01 total
# mpi_tasks : 32 on 2 nodes
wallclock : 29.975184 sec
%comm     : 27.72

gbytes : 6.65863e-01 total
gflop/sec : 2.33478e+00 total

[total]           <avg>           min     max
# wallclock        953.272        29.7897  29.6092  29.9752
# user             837.25         26.1641  25.71    26.92
# system           60.6           1.89375  1.52     2.59
# mpi              264.267        8.25834  7.73025  8.70985
# %comm            27.7234        25.8873  29.3705  29.3705
# gflop/sec        2.33478        0.0729619 0.072204  0.0745817
# gbytes           0.665863       0.0208082 0.0195503 0.0237541
# PM_FPU0_CMPL    2.28827e+10  7.15084e+08 7.07373e+08 7.30171e+08
# PM_FPU1_CMPL    1.70657e+10  5.33304e+08 5.28487e+08 5.42882e+08
# PM_FPU_FMA      3.00371e+10  9.3866e+08  9.27762e+08 9.62547e+08
# PM_INST_CMPL    2.78819e+11  8.71309e+09 8.20981e+09 9.21761e+09
# PM_LD_CMPL      1.25478e+11  3.92118e+09 3.74541e+09 4.11658e+09
# PM_ST_CMPL      7.45961e+10  2.33113e+09 2.21164e+09 2.46327e+09
# PM_TLB_MISS     2.45894e+08  7.68418e+06 6.98733e+06 2.05724e+07
# PM_CYC          3.0575e+11   9.55467e+09 9.36585e+09 9.62227e+09

[time]           <calls>           <%%%time>  <%%wall>
# MPI_Send        188.386       639616     71.29     19.76
# MPI_Wait        69.5032       639616     26.30       7.29
# MPI_Reduce      6.34936       639616       2.40       0.67
# MPI_Barrier     0.0177442     32         0.01         0.00
# MPI_Reduce      0.00540609    32         0.00         0.00
# MPI_Barrier     0.00465156    32         0.00         0.00
# MPI_Comm_rank   0.000145341   32         0.00         0.00
```
**IPM Data on NERSC Web Site**

聂瑟数据网站上的IPM数据

<table>
<thead>
<tr>
<th>Metric</th>
<th>Sum</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>CV (%)</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate Floating Point Operations (Flop x 10**9)</td>
<td>3.011e+02</td>
<td>1.470e-01</td>
<td>4.96e-03</td>
<td>3.36e+00</td>
<td>1.395e-01</td>
<td>2.161e-01</td>
</tr>
<tr>
<td>GFlop/sec</td>
<td>6.147e+01</td>
<td>3.002e-04</td>
<td>1.008e+05</td>
<td>3.36e+00</td>
<td>2.847e-04</td>
<td>4.411e-04</td>
</tr>
<tr>
<td>Maximum Memory Usage (Bytes)</td>
<td>4.101e+02</td>
<td>2.002e-01</td>
<td>9.86e-03</td>
<td>4.80e+00</td>
<td>1.781e-01</td>
<td>2.448e-01</td>
</tr>
<tr>
<td>Time Spent in MPI Routines (sec)</td>
<td>1.228e+06</td>
<td>5.995e+02</td>
<td>4.98e+01</td>
<td>8.31e+00</td>
<td>5.177e+02</td>
<td>6.810e+02</td>
</tr>
<tr>
<td>Wallclock Time (sec)</td>
<td>1.003e+06</td>
<td>4.898e+02</td>
<td>6.428e+02</td>
<td>1.31e+02</td>
<td>4.898e+02</td>
<td>4.927e+02</td>
</tr>
</tbody>
</table>

CV = Coefficient of Variance = (Standard Deviation / Mean)

任务分布的 Aggregate Floating Point Operations (Flop x 10**9) - 作为最大值的百分比

该MPI排名是aggregate Floating Point Operations (Flop x 10**9)的总和的百分比。
**Statistics Across Tasks**

### NERSC job details

<table>
<thead>
<tr>
<th>Sum</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>CV (%)</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.011e+02</td>
<td>1.470e-01</td>
<td>4.946e-03</td>
<td>3.36e+00</td>
<td>1.395e-01</td>
<td>2.161e-01</td>
</tr>
<tr>
<td>6.147e-01</td>
<td>3.002e-04</td>
<td>1.008e-05</td>
<td>3.36e+00</td>
<td>2.847e-04</td>
<td>4.411e-04</td>
</tr>
<tr>
<td>4.101e+02</td>
<td>2.002e-01</td>
<td>9.606e-03</td>
<td>4.80e+00</td>
<td>1.781e-01</td>
<td>2.448e-01</td>
</tr>
<tr>
<td>1.228e+06</td>
<td>5.995e+02</td>
<td>4.984e+01</td>
<td>8.31e+00</td>
<td>5.177e+02</td>
<td>6.801e+02</td>
</tr>
<tr>
<td>1.003e+06</td>
<td>4.898e+02</td>
<td>6.428e-02</td>
<td>1.31e-02</td>
<td>4.898e+02</td>
<td>4.927e+02</td>
</tr>
</tbody>
</table>

**9) - as a percentage of maximum**

a.
IPM Examples

Time spent by each task in MPIRecv as a percentage of the maximum value.

The MPI rank represented by each cell in the table is the sum of the cell's column and row indices.

Table Columns: 32 32

Time vs. MPI Rank for MPIRecv
User-Space I/O Profiling

The module is loaded by default. Just build and run.
Estimated I/O Time: 29.5%
Summary

• Debugging and Parallel Code Optimization can be hard

• Tools can help
  – See NERSC web pages for recommendations
  – Use the ones that work for you

• Be aware of some of the more common errors and best practices

• Look for outliers in parallel programs

• Refer to NERSC web pages for details
  – http://www.nersc.gov/users/software/debugging-and-profiling/
  – http://www.nersc.gov/users/training/courses/CS267/
National Energy Research Scientific Computing Center
Job Physical Topology
System-Level I/O Monitoring

Users can see the system-wide I/O activity while their job ran to look for contention.
Time spent by each task in \textit{MPI_Allreduce} as a percentage of the maximum value.

The MPI rank represented by each cell in the table is the sum of the cell’s column and row indices.

Table: 

|        | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 |
|--------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Time vs. MPI Rank for \textit{MPI_Allreduce}
IPM Examples

Task distribution of Maximum Memory Usage (GBs/req) - as a percentage of maximum

The MPI rank is the sum of the column and row indices in the table.

| Row | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|-----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 160 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 192 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 256 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 384 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 512 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 768 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1024| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1536| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table Columns: [32]
Parallel Tools for the Masses

• Using even the best tools can be tedious
  – “Follow these 10 steps to perform the basic analysis of your program”
Vampir w/ CUDA
Compiler runtime bounds checking

Out of bounds reference in source code for program “flip”

allocate(put_seed(random_size))
...
bad_index = random_size+1
put_seed(bad_index) = 67

Intel compiler:

```
ftn -c -g -Ktrap=fp -check bounds flip.f90
ftn -c -g -Ktrap=fp -check bounds printit.f90
ftn -o flip flip.o printit.o -g
```

```
qusub -I -qdebug -lmppwidth=16
% cd $PBS_O_WORKDIR
% aprun -n 16./flip
forrtl: severe (408): fort: (2): Subscript #1 of the array SEED has value 3 which is greater than the upper bound of 2
```
Compiler Documentation

• For a list of compiler options, see the man pages for the individual compilers
  – man pgcc | pgCC | pgf90
  – man icc | icpc | ifort
  – man gcc | g++ | gfortran

• Use your favorite search engine to find vendor manuals on line
Other Debugging Tips

• Try different compilers
  – Diagnostic messages and language spec compliances differ

• Look for memory corruption
  – Bad memory reference in one place (array out of bounds) can make code crash elsewhere
  – It might appear that you’re crashing on a perfectly valid line of code

• Check the arguments to your MPI calls

• Call the NERSC Consultants (800-66-NERSC or 510 486-8600)
Identifying Targets for Optimization

• Hardware Event Counters
  – Special registers count events on processor
  – E.g. number of floating point instructions
  – Many possible events
  – Only a few can be recorded at a time (~4 counters)
  – Can give you an idea of how efficiently you are using the processor hardware
Apprentice Basic View

- Can select new (additional) data file and do a screen dump
- Worthless
- Useful
- Can select other views of the data
- Can drag the “calipers” to focus the view on portions of the run
TAU Assignment

• You will have a homework assignment using TAU
  - %module load tau
  - Define paths in Makefile
  - Modify header file to define TAU macros
  - Add macro calls to the code
  - Compile and submit to batch queue
  - Use pprof to produce readable output

• Good reference
Users Want (Need?) Tools

• Users are asking for tools because HPC systems and programming models are changing

• More and more components to worry about
  – CPU (caches, FPUs, pipelining, ...)
  – Data movement to main memory, GPU memory, levels of cache
  – I/O
  – Network (message passing)
  – CPU Threads (OpenMP)
  – GPU performance
Questions to You

• What tools do you use?
• What tools do you want?
• What would you like centers to support?
• Can you get to exascale without tools?
What I Want in a Tool

• Let the users help themselves
• Work for everyone all (most of?) the time
• Easy to use
• Useful
• Easy to interpret the results
• Affordable & ubiquitous
• Simple, supplement existing complex tools
  — Point the way for a “deeper dive” in problem areas
IPM

- Integrated Performance Monitoring
- Developed by David Skinner at NERSC
- MPI profiling, hardware counter metrics, IO profiling (?)
- IPM requires no code modification & no instrumented binary
- IPM uses hooks already in the MPI library to intercept your MPI calls and wrap them with timers and counters
• How it works (user perspective)
  – % module load IPM*
  – Run program as normal
  – Look at results on the web

• It’s that easy!
  – And extremely low overhead, so IPM is examining your production code

* (As long as your system supports dynamic load libs)
IPM Examples

Click on the metric you want.

<table>
<thead>
<tr>
<th>Executable</th>
<th>Number of tasks</th>
<th>Aggregate GFlop/sec</th>
<th>Average GFlop/sec/task</th>
<th>Average memory (GB)</th>
<th>Average memory/task (GB)</th>
<th>Aggregate MPI calls made</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>512</td>
<td>0.1482</td>
<td>0.0003</td>
<td>32.2626</td>
<td>0.0630</td>
<td>7.027e+07</td>
</tr>
</tbody>
</table>

**IPM Summary Statistics - 512 tasks**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Sum over all tasks</th>
<th>Average (per task)</th>
<th>Task CV (%)</th>
<th>Task Minimum</th>
<th>Task Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate Floating Point Operations (Flop x 10**9)</td>
<td>1.313e+01</td>
<td>2.565e-02</td>
<td>6.10</td>
<td>1.713e-02</td>
<td>2.738e-02</td>
</tr>
<tr>
<td>GFlop/sec</td>
<td>1.482e-01</td>
<td>2.895e-04</td>
<td>6.10</td>
<td>1.936e-04</td>
<td>3.114e-04</td>
</tr>
<tr>
<td>Maximum Memory Usage (GBytes)</td>
<td>3.226e+01</td>
<td>6.301e-02</td>
<td>10.12</td>
<td>5.701e-02</td>
<td>1.547e-01</td>
</tr>
<tr>
<td>Time Spent in MPI Routines (sec)</td>
<td>4.044e+04</td>
<td>7.898e-01</td>
<td>4.05</td>
<td>9.801e+00</td>
<td>8.359e+01</td>
</tr>
<tr>
<td>Nallock Time (sec)</td>
<td>4.537e+04</td>
<td>8.861e+01</td>
<td>0.10</td>
<td>8.848e+01</td>
<td>8.895e+01</td>
</tr>
</tbody>
</table>

Memory in units of gigabytes; time in seconds.

**Hardware counter statistics - 512 tasks**

<table>
<thead>
<tr>
<th>Counter Name</th>
<th>Sum over all tasks</th>
<th>Average (per task)</th>
<th>Task CV (%)</th>
<th>Task Minimum</th>
<th>Task Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_FP_OBB</td>
<td>1.161799e+12</td>
<td>2.259139e+09</td>
<td>6.09</td>
<td>1.515023e+09</td>
<td>2.439529e+09</td>
</tr>
</tbody>
</table>

**MPI Time Statistics - 512 tasks**

<table>
<thead>
<tr>
<th>Call</th>
<th>Sum over all tasks</th>
<th>Average (per task)</th>
<th>Task CV (%)</th>
<th>Task Minimum</th>
<th>Task Maximum</th>
<th>% of MPI</th>
<th>% of well</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Bcast</td>
<td>3.517e+04</td>
<td>6.695e-01</td>
<td>4.48</td>
<td>4.342e-01</td>
<td>7.269e-01</td>
<td>86.967</td>
<td>77.520</td>
</tr>
<tr>
<td>MPI_Scatterv</td>
<td>2.589e+03</td>
<td>5.057e+00</td>
<td>5.79</td>
<td>1.059e+00</td>
<td>5.540e+00</td>
<td>6.403</td>
<td>5.707</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>2.176e+03</td>
<td>4.249e+00</td>
<td>17.82</td>
<td>1.250e+00</td>
<td>4.958e+00</td>
<td>5.380</td>
<td>4.795</td>
</tr>
<tr>
<td>MPI_Gather</td>
<td>4.312e+02</td>
<td>8.422e-01</td>
<td>36.44</td>
<td>3.552e-03</td>
<td>2.271e+00</td>
<td>1.066</td>
<td>0.950</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>5.250e+01</td>
<td>1.025e-01</td>
<td>11.36</td>
<td>7.182e-02</td>
<td>1.259e-01</td>
<td>0.130</td>
<td>0.116</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>1.033e+01</td>
<td>2.017e-02</td>
<td>10.21</td>
<td>1.217e-02</td>
<td>2.613e-02</td>
<td>0.026</td>
<td>0.023</td>
</tr>
<tr>
<td>MPI_Gather</td>
<td>1.021e+01</td>
<td>1.995e-02</td>
<td>50.07</td>
<td>1.391e-02</td>
<td>1.343e-02</td>
<td>0.025</td>
<td>0.023</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>4.563e+01</td>
<td>8.913e-04</td>
<td>4.74</td>
<td>7.799e-04</td>
<td>1.404e+01</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>9.629e+02</td>
<td>1.681e-04</td>
<td>10.65</td>
<td>1.462e-02</td>
<td>4.859e-04</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Init</td>
<td>6.000e+00</td>
<td>0.000e+00</td>
<td>0.000e+00</td>
<td>0.000e+00</td>
<td>0.000e+00</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>6.000e+00</td>
<td>0.000e+00</td>
<td>0.000e+00</td>
<td>0.000e+00</td>
<td>0.000e+00</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Average MPI Time per Task
Identifying Targets for Optimization: Profiling

• **Sampling**
  – Regularly interrupt the program and record where it is
  – Build up a statistical profile of time spent in various routines
  – Concentrate first on longest running sections or routines

• **Tracing**
  – Insert hooks into program to record and time program events (logging)
  – Reasonable for sequential programs
  – Unwieldy for large parallel programs (too much data!)
Using Apprentice

- Optional visualization tool for Cray’s perftools data
- Use it in a X Windows environment
- Uses a data file as input (XXX.ap2) that is prepared by `pat_report`:

  `app2 [--limit_per_pe tags] XXX.ap2`
PAPI

• PAPI (Performance API) provides a standard interface for use of the performance counters in major microprocessors

• Predefined actual and derived counters supported on the system
  – To see the list, run ‘papi_avail’ on compute node via aprun:
    
    qsub -I -lmppwidth=24
    module load perftools
    aprun -n 1 papi_avail

• AMD native events also provided; use ‘papi_native_avail’:
  aprun -n 1 papi_native_avail
• Tuning and Analysis Utilities
• Fortran, C, C++, Java performance tool
• Procedure
  – Insert macros
  – Run the program
  – View results with pprof
• More info than gprof
  – E.g. per process, per thread info; supports pthreads
• http://acts.nersc.gov/tau/index.html
What IPM measures

• IPM “only” gives a high-level, entire-program-centric view

• Still, very valuable guidance
  – Shows whole-run info per MPI task, OpenMP thread, (CUDA under development)
  – Many pieces of data in one place

• Reveals what many users don’t know about their code
  – High-water memory usage (per task)
  – Load balance
  – Call imbalance
  – MPI time
  – I/O time