Cray Optimization and Performance Tools

Harvey Wasserman
Woo-Sun Yang
NERSC User Services Group

February 7-8, 2011

Cray XE6 Workshop
NERSC Oakland Scientific Facility
Outline

• Introduction, motivation, some terminology

• Using CrayPat

• Using Apprentice2

• Hands-on lab
Why Analyze Performance?

- Improving performance on HPC systems has compelling economic and scientific rationales.
  - Dave Bailey: Value of improving performance of a single application, 5% of machine's cycles by 20% over 10 years: $1,500,000
  - Scientific benefit probably much higher

- Goal: solve problems faster; solve larger problems

- Accurately state computational need

- Only that which can be measured can be improved

- The challenge is mapping the application to an increasingly more complex system architecture
  - or set of architectures
Performance Evaluation as an Iterative Process

Vendor

Buy Machine

User

Sell Machine

Improve machine

Improves code

Overall goal: more / better science results
Performance Analysis Issues

• Difficult process for real codes
• Many ways of measuring, reporting
• Very broad space: Not just time on one size
  – for fixed size problem (same memory per processor): Strong Scaling
  – scaled up problem (fixed execution time): Weak Scaling
• A variety of pitfalls abound
  – Must compare parallel performance to best \textit{uniprocessor} algorithm, not just parallel program on 1 processor (unless it’s best)
  – Be careful relying on any single number
• Amdahl’s Law
Performance Questions

• How can we tell if a program is performing well?

• Or isn’t?

• If performance is not “good,” how can we pinpoint why?

• How can we identify the causes?

• What can we do about it?
Supercomputer Architecture

Supercomputing Architecture Issues

Standard Parallel Computer Architecture

- CPU
- RAM
- Disk

Network Switch

Corresponding Memory Hierarchy

- Registers
- Instr. Operands

Cache

Local Memory

Messages

Remote Memory

Pages

Performance Implications

- Increasing Bandwidth
- Increasing Latency
- Increasing Programmability
- Increasing Capacity

- Standard architecture produces a “steep” multi-layered memory hierarchy
Performance Metrics

- **Primary metric: application time**
  - but gives little indication of efficiency

- **Derived measures:**
  - rate (Ex.: messages per unit time, Flops per Second, clocks per instruction), cache utilization

- **Indirect measures:**
  - speedup, efficiency, scalability
Performance Metrics

- **Most basic:**
  - *counts:* how many `MPI_Send` calls?
  - *duration:* how much time in `MPI_Send`?
  - *size:* what size of message in `MPI_Send`?

- **(MPI performance as a function of message size)**

\[
T = T_{\text{msg}} = t_s + t_w L
\]

\[
T = \text{Time} \\
L = \text{Message Size} \\
t_s = \text{startup cost} \\
t_w = \text{cost per word} = \text{Bandwidth}
\]
Two dimensions:

When data collection is triggered:

- Externally (asynchronous): Sampling
  - OS interrupts execution at regular intervals and records the location (program counter) (and/or other event(s))

- Internally (synchronous): Tracing
  - Event based
  - Code instrumentation, Automatic or manual
• Instrumentation: adding measurement probes to the code to observe its execution.

• Different techniques depending on where the instrumentation is added.

• Different overheads and levels of accuracy with each technique.

Karl Fuerlinger, UCB
• Goal is to allow performance measurement without modification of user source code

Timing information:

Time in timer: IMPVMIXT
Timer number 1 = 10.04 seconds
Time in timer: IMPVMIXU
Timer number 2 = 0.51 seconds
Time in timer: TOTAL
Timer number 3 = 17.88 seconds
Time in timer: STEP
Timer number 4 = 17.88 seconds
Time in timer: BAROCLINIC
Timer number 5 = 13.88 seconds
Time in timer: BAROTROPIC
Timer number 6 = 1.27 seconds

POP exiting...
Successful completion of POP run

real 18.09
user 17.97
sys 0.10
• Instrumentation: adding measurement probes to the code to observe its execution.

• Different techniques depending on where the instrumentation is added.

• Different overheads and levels of accuracy with each technique.
• **Approach: use a tool to “instrument” the code**
  1. Transform a binary executable before executing
  2. Include “hooks” for important events
  3. Run the instrumented executable to capture those events, write out raw data file
  4. Use some tool(s) to interpret the data
Performance Data Collection

- **How performance data are presented:**
  - Profile: combine sampled events over time
    - Reflects runtime behavior of program entities
      - functions, loops, basic blocks
      - user-defined “semantic” entities
    - Good for low-overhead performance assessment
    - Helps to expose performance hotspots ("bottleneckology")
  - Trace file: Sequence of events over time
    - Gather individual time-stamped events (and arguments)
    - Learn when (and where?) events took place on a global timeline
    - Common for message passing events (sends/receives)
    - Large volume of performance data generated; generally intrusive
    - Becomes very difficult at large processor counts, large numbers of events
      - Example in Apprentice section at end of tutorial
Performance Analysis Difficulties

- Tool overhead
- Data overload
- User knows the code better than the tool
- Choice of approaches
- Choice of tools
- CrayPat is an attempt to overcome several of these
  - By attempting to include intelligence to identify problem areas
  - However, in general the problems remain
Performance Tools @ NERSC

• IPM: Integrated Performance Monitor
• Vendor Tools:
  – CrayPat
• Community Tools (Not all fully supported):
  – TAU (U. Oregon via ACTS)
  – OpenSpeedShop (DOE/Krell)
  – HPCToolKit (Rice U)
  – PAPI (Performance Application Programming Interface)
Profiling: Inclusive vs. Exclusive

- Inclusive time for main:
  - 100 secs

- Exclusive time for main:
  - 100-20-50-20=10 secs
  - Exclusive time sometimes called “self”

```c
int main()
{
    /* takes 100 secs */
    f1(); /* takes 20 secs */
    /* other work */
    f2(); /* takes 50 secs */
    f1(); /* takes 20 secs */

    /* other work */
}

/* similar for other metrics, such as hardware performance counters, etc. */
```
Woo-Sun Yang

USING CRAYPAT
CrayPat Outline

- Introduction
- Sampling (and example)
- Tracing
- .xf files
- pat_report
- Tracing examples: Heap, MPI, OpenMP
- APA (Automatic Program Analysis)
- CrayPat API
- Monitoring hardware performance counters

- Exercises provided: Exercise info in this box
Introduction to CrayPat

• Suite of tools to provide a wide range of performance-related information

• Can be used for both sampling and tracing user codes
  – with or without hardware or network performance counters

• Supports Fortran, C, C++, UPC, MPI, Coarray Fortran, OpenMP, Pthreads, SHMEM
Access to Cray Tools

• Access via module utility

• Old:
  - module load xt-craypat
  - module load apprentice2

• Now:
  - module load perftools
  - xt-craypat, apprentice2, and xt-papi
    (via xt-craypat) are loaded
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`
   - `vi my_profile`
   - Result is also a new file: `XXXXX.ap2`
Key points to remember:

- MUST load module prior to building your code
  - Error message is obscure!
    ERROR: Missing required ELF section 'link information'

- MUST load module prior to looking at man pages

- MUST run your application in $SCRATCH

- Module name change: xt-craypat → perftools

- MUST leave relocatable binaries (*.o) when compiling
• To sample the program counter (PC) at a given time interval or when a specified hardware counter overflows; runs faster than tracing

• To build, use –S or simply without any tracing flag for pat_build
  –pat_build –S a.out or
  –pat_build a.out
Running a Sampling Experiment

- To run
  - Set `PAT_RT_EXPERIMENT` to a type
    - Default: `samp_pc_time` with default time interval (`PAT_RT_INTERVAL`) of 10,000 microseconds
    - Others: `samp_pc_ovfl`, `samp_cs_time`, `samp_cs_ovfl` (see pat man page)

- `pat_report on .xf` from a sampling experiment generates `.apa` file (later on this)
Sampling Example

> module load perftools
> ftn -c jacobi_serial.f90
> ftn -o jacobi_serial jacobi_serial.o
> pat_build jacobi_serial

Run jacobi_serial+pat in a batch job.

> pat_report
  jacobi_serial+pat+5511-2558sot.xf

---

**Table 1: Profile by Function**

<table>
<thead>
<tr>
<th>Samp %</th>
<th>Samp</th>
<th>Group</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>3598</td>
<td>Total</td>
<td></td>
</tr>
<tr>
<td>99.9%</td>
<td>3596</td>
<td>IUSER</td>
<td></td>
</tr>
<tr>
<td>73.8%</td>
<td>2654</td>
<td>MAIN</td>
<td></td>
</tr>
<tr>
<td>24.0%</td>
<td>891</td>
<td>compute_diff</td>
<td></td>
</tr>
<tr>
<td>1.4%</td>
<td>50</td>
<td>limit_fields</td>
<td></td>
</tr>
</tbody>
</table>

---

**Table 2: Profile by Group, Function, and Line**

<table>
<thead>
<tr>
<th>Samp %</th>
<th>Samp</th>
<th>Group</th>
<th>Function</th>
<th>Source</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>3598</td>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99.9%</td>
<td>3596</td>
<td>IUSER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>73.8%</td>
<td>2654</td>
<td>MAIN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24.0%</td>
<td>891</td>
<td>compute_diff</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4%</td>
<td>50</td>
<td>limit_fields</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Table 3: Wall Clock Time, Memory High Water Mark**

<table>
<thead>
<tr>
<th>Process</th>
<th>Process</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>HiMem</td>
<td>(MBytes)</td>
</tr>
<tr>
<td>57.593670</td>
<td>1153</td>
<td></td>
</tr>
</tbody>
</table>
More on Sampling

• Binary built for sampling doesn’t work for a tracing experiment
• Binary built for tracing can be used for sampling ("trace-enhanced sampling")
  – Set PAT_RT_EXPERIMENT to a sampling type
  – set PAT_RT_SAMPLING_MODE to 1 (raw user sampling) or 3 (bubble user sampling: aggregate all samples inside a call to a traced function)
pat_build for Tracing

• To trace entries and returns from functions
  – Type of experiment: ‘trace’; no need to set PAT_RT_EXPERIMENT to it in general

• –w: trace functions specified by –t and –T
  – If none is specified, “trace” the ‘main’ (i.e., entire code as a whole)

• –u: trace all user functions routine by routine
  – For source files owned and writable by the user
  – Use care: binary runs longer and can fail
    WARNING: Tracing small, frequently called functions can add excessive overhead
    WARNING: To set a minimum size, say 800 bytes, for traced functions, use:
    -D trace-text-size=800
pat_build for Tracing

- `–T function: trace function`
  - `pat_build –w –T field_,grad_ a.out`
  - Learn the Unix `nm` or `readelf`:
    `nm mycode.o | grep “ T “`

- `–T ! function: do not trace function`
  - `pat_build –u –T \!field_ a.out`
    - trace all user functions except field_
    - `'\'` to escape the `‘!’` character in csh/tcsh

- `–t tracefile: trace all functions listed in the file tracefile.`
• –g: trace all functions in certain function groups (e.g., MPI):
  - pat_build -g mpi,heap -u a.out
  - pat_build -g mpi -T \!MPI_Barrier a.out
  trace all MPI calls except MPI_Barrier

• See $CRAYPAT_ROOT/lib/Trace* for files that list what routines are traced

  • mpi
  • omp
  • pthreads
  • caf
  • upc
  • shmem
  • ga
  • heap
  • blas
  • blacs
  • lapack
  • scalapack
  • fftw
  • petsc
  • io
  • netcdf
  • hdf5
  • lustre
  • adios
  • sysio
  • dmapp
  • ...

CrayPat Trace Function Groups
Other pat_build Options

• \texttt{-f}: overwrite an existing instrumented program

• \texttt{-o instr_prog}:
  – use a different name for the instrumented executable instead of a.out+pat
  – can put \texttt{instr_prog} at the end of the command line without `\texttt{-o}'

• \texttt{-O optfile}: use the pat_build options in the file \texttt{optfile}.
  – Special argument `\texttt{-O apa}' will be discussed later

Exercise in perftools/pat_build_examples
**Instrumenting Programs Using Compiler Options**

- Available in Pathscale, GNU, and Cray compilers at NERSC
- Requires recompile, link (selected files)
- Alternative to pat_build –u
  - GNU, PathScale:
    - `cc -finstrument-functions -c pgm.c`
    - `cc -o pgm pgm.c`
  - Cray compiler:
    - `cc -h func_trace -c pgm.c`
    - `cc -o pgm pgm.o`
  - Then
    - `pat_build -w pgm`
.xf Files

- **Experiment data files; binary files**
- **Number of .xf files**
  - Single file (for \( \leq 256 \) PEs) or directory containing multiple (~\( \sqrt{\text{PEs}} \)) files
  - Can be changed with `PAT_RT_EXPFILE_MAX`
- **Name convention**
  - `a.out+pat+<UNIX_PID>-<NODE_ID>[st][dfot].xf` (or `a.out+apa+....xf`; see APA)
  - `[st][dfot]`: [sampling, tracing], [distributed memory, forked process, OpenMP, Pthreads]
- **New one each time you run your application**
- **Can create .xf file(s) in other than the current location by setting `PAT_RT_EXPFILE_DIR`**
• Generates from .xf data file(s) ASCII text report and .ap2 file (to be viewed with Apprentice2)
  – Create .ap2 file right after .xf file becomes available!
    • .xf file requires the instrumented executable in the original directory (not portable)
    • .ap2 doesn’t (self-contained and portable)
• `pat_report` on .xf file (or directory containing multiple .xf files) generates
  – text report to stdout (terminal)
  – .ap2 file
    – .apa file, in case of a sampling experiment
• Running on .ap2 file generates text report to stdout
pat_report Options

- **-d**: data items to display (time data, heap data, counter data, ...)

- **-b**: how data is aggregated or labeled (group, function, pes, thread, ...)

- **-s**: details of report appearance (aggregation, format, ...)

- **-O | -b | -d | -s | -h**: list all available cases for the option
- **-O**: predefined report types; this is what we should use
  - profile, callers (ca), calltree (ct), ca+src, ct+src, heap, load_balance, mpi_callers, hwpc, nids, ...

- heap and load_balance have a few “sub-reports”
  - load_balance = load_balance_program (for entire program)
    + load_balance_group (per function group)
    + load_balance_function (per function)

- Examples:
  - **-O profile**: `-d ti%@0.95,ti,imb_ti,imb_ti%,tr
  -b gr,fu,pe=HIDE`
  - **-O callers+src**: `-d ti%@0.95,ti,tr -b gr,fu,ca,pe=HIDE
  -s show_ca='fu,so,li'
  - **-O load_balance_function**: `-d ti%@0.95,ti,tr
  -b gr,fu,pe=[mmm]`
pat_report Options

- Without `–d`, `–b` or `–o`, a few reports appear by default; dependent on the used trace groups

- `–i instrProg`: specify the path for the instrumented executable (if not in the same directory as the .xf file)
- `–o outputFile`: specify the output file name
- `–T`: disable all thresholds (5%)

- `pat_report` lists the options used in the report – a good place to learn options; try adding an option to the existing ones
- By default, all reports (`-O`) show either no individual PE values or only the PEs having the maximum, median, and minimum values.
- The suffix `_all` can be appended to any of the `pat_report` keyword options to show the data for all PEs
Heap Memory Example

> module load perftools
> ftn -c jacobi_serial.f90
> ftn -o jacobi_serial jacobi_serial.o
> pat_build -g heap -u jacobi_serial

Run jacobi_serial+pat in a batch job.

> pat_report
  jacobi_serial+pat+15243-18tot.xf

<table>
<thead>
<tr>
<th>Time %</th>
<th>Time</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>0.576433</td>
<td>64.0</td>
</tr>
</tbody>
</table>

---

Table 3: Heap Stats during Main Program

<table>
<thead>
<tr>
<th>Tracked</th>
<th>Total</th>
<th>Tracked</th>
<th>Tracked</th>
<th>Tracked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap Allocs</td>
<td>Frees</td>
<td>Objects</td>
<td>MBytes</td>
<td></td>
</tr>
<tr>
<td>HiWater</td>
<td>Not</td>
<td>Not</td>
<td>Freed</td>
<td>Freed</td>
</tr>
<tr>
<td>12.466</td>
<td>11</td>
<td>7</td>
<td>4</td>
<td>4.836</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tracked</th>
<th>Tracked</th>
<th>Tracked</th>
<th>Caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBytes</td>
<td>MBytes</td>
<td>Objects</td>
<td></td>
</tr>
<tr>
<td>Freed %</td>
<td>Freed</td>
<td>Freed</td>
<td></td>
</tr>
<tr>
<td>100.0%</td>
<td>4.836</td>
<td>4</td>
<td>Total</td>
</tr>
</tbody>
</table>

Sometimes not easy to understand

Exercise in perftools/heap
MPI Code Example

- Profiling by MPI functions
- MPI message stats
- Load imbalance among MPI tasks

Run `jacobi_mpi+pat` in a batch job.

```bash
> module load perftools
> ftn -c jacobi_mpi.f90
> ftn -o jacobi_mpi
> pat_build -g mpi -u jacobi_mpi

> pat_report
jacobi_mpi+pat+15207-18tdt.xf
```
**MPI Code Example**

**Table 1: Profile by Function Group and Function**

<table>
<thead>
<tr>
<th>Time %</th>
<th>Time</th>
<th>Imb. Time</th>
<th>Imb.</th>
<th>Calls</th>
<th>Experiment-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>0.078172</td>
<td>--</td>
<td>--</td>
<td>124.0</td>
<td>Total</td>
</tr>
<tr>
<td>83.2%</td>
<td>0.065023</td>
<td>--</td>
<td>--</td>
<td>26.0</td>
<td>USER</td>
</tr>
<tr>
<td>60.0%</td>
<td>0.046878</td>
<td>0.003254</td>
<td>6.5%</td>
<td>1.0</td>
<td>jacobi_mpi_</td>
</tr>
<tr>
<td>18.7%</td>
<td>0.014595</td>
<td>0.001878</td>
<td>11.4%</td>
<td>10.0</td>
<td>compute_diff_</td>
</tr>
<tr>
<td>12.9%</td>
<td>0.010061</td>
<td>--</td>
<td>--</td>
<td>14.0</td>
<td>MPI_SYNC</td>
</tr>
<tr>
<td>7.9%</td>
<td>0.006150</td>
<td>0.006052</td>
<td>49.2%</td>
<td>4.0</td>
<td>mpi_bcast(sync)</td>
</tr>
<tr>
<td>5.0%</td>
<td>0.003812</td>
<td>0.002356</td>
<td>22.5%</td>
<td>10.0</td>
<td>mpi_allreduce(s)</td>
</tr>
<tr>
<td>3.9%</td>
<td>0.003088</td>
<td>--</td>
<td>--</td>
<td>84.0</td>
<td>MPI</td>
</tr>
<tr>
<td>3.0%</td>
<td>0.002330</td>
<td>0.001195</td>
<td>83.5%</td>
<td>20.0</td>
<td>mpi_sendrecv</td>
</tr>
<tr>
<td>0.5%</td>
<td>0.000413</td>
<td>0.000051</td>
<td>11.6%</td>
<td>10.0</td>
<td>mpi_allreduce</td>
</tr>
<tr>
<td>0.5%</td>
<td>0.000327</td>
<td>0.000217</td>
<td>65.8%</td>
<td>1.0</td>
<td>mpi_final</td>
</tr>
<tr>
<td>41.8%</td>
<td>4.0</td>
<td>mpi_bcast</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48.6%</td>
<td>24.0</td>
<td>mpi_comm_size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57.3%</td>
<td>24.0</td>
<td>mpi_comm_rank</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54.6%</td>
<td>1.0</td>
<td>mpi_init</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 2: Load Balance with MPI Message Stats**

<table>
<thead>
<tr>
<th>Time %</th>
<th>Time</th>
<th>MPI</th>
<th>MPI Msg</th>
<th>Avg MPI</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>0.078209</td>
<td>33.0</td>
<td>764697.0</td>
<td>22546.35</td>
<td>Total</td>
</tr>
<tr>
<td>83.2%</td>
<td>0.065031</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>USER</td>
</tr>
<tr>
<td>0.4%</td>
<td>0.068331</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>PE.154</td>
</tr>
<tr>
<td>0.3%</td>
<td>0.065538</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>PE.93</td>
</tr>
<tr>
<td>0.3%</td>
<td>0.053429</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>PE.24</td>
</tr>
<tr>
<td>12.9%</td>
<td>0.010061</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>MPI_SYNC</td>
</tr>
<tr>
<td>0.1%</td>
<td>0.019155</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>PE.68</td>
</tr>
<tr>
<td>0.1%</td>
<td>0.009859</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>PE.51</td>
</tr>
<tr>
<td>0.0%</td>
<td>0.001855</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>PE.155</td>
</tr>
<tr>
<td>4.0%</td>
<td>0.003133</td>
<td>33.0</td>
<td>764697.0</td>
<td>22546.35</td>
<td>MPI</td>
</tr>
<tr>
<td>0.1%</td>
<td>0.014530</td>
<td>34.0</td>
<td>767896.0</td>
<td>22585.18</td>
<td>PE.24</td>
</tr>
<tr>
<td>0.0%</td>
<td>0.002374</td>
<td>34.0</td>
<td>767896.0</td>
<td>22585.18</td>
<td>PE.132</td>
</tr>
<tr>
<td>0.0%</td>
<td>0.001671</td>
<td>34.0</td>
<td>767896.0</td>
<td>22585.18</td>
<td>PE.75</td>
</tr>
</tbody>
</table>

**Equation:**

\[ \text{imb} = \text{max} - \text{avg} \]

\[ \text{imb\%} = \frac{\text{imb}}{\text{max}} \times \frac{\text{npes}}{\text{npes}-1} \times 100\% \]
Table 3: MPI Message Stats by Caller

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Msg</th>
<th>MsgSz</th>
<th>4KB</th>
<th>&lt;16B</th>
<th>16KB</th>
<th>32KB</th>
<th>64KB</th>
<th>&lt;512KB</th>
<th>&lt;1KB</th>
<th>&lt;4KB</th>
<th>&lt;8KB</th>
<th>&lt;16KB</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>764697.0</td>
<td>33.9</td>
<td>14.0</td>
<td>19.9</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**Bins by message size**

**per PE**

**Level of depth**

**callers**

<table>
<thead>
<tr>
<th>Level</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>0.0</td>
</tr>
</tbody>
</table>

**MPI Code Example**

<table>
<thead>
<tr>
<th>Callers</th>
<th>Bins</th>
<th>per PE</th>
<th>Level of depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>jacobi_mpi</td>
<td>40.0</td>
<td>10.0</td>
<td>10.0</td>
</tr>
<tr>
<td>jacobi_mpi</td>
<td>40.0</td>
<td>10.0</td>
<td>10.0</td>
</tr>
<tr>
<td>jacobi_mpi</td>
<td>40.0</td>
<td>10.0</td>
<td>10.0</td>
</tr>
<tr>
<td>mpi_bcast</td>
<td>16.0</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>mpi_bcast</td>
<td>15.0</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>mpi_bcast</td>
<td>15.0</td>
<td>4.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>

**Experiments**

1. MPISEND/RECV
2. jacobi_mpi
3. mpi_bcast
4. mpi_bcast
5. mpi_bcast
6. mpi_bcast
7. mpi_bcast
8. mpi_bcast
9. mpi_bcast
10. mpi_bcast
11. mpi_bcast
12. mpi_bcast
13. mpi_bcast
14. mpi_bcast
15. mpi_bcast
16. mpi_bcast
17. mpi_bcast
18. mpi_bcast
19. mpi_bcast
20. mpi_bcast
21. mpi_bcast
22. mpi_bcast
23. mpi_bcast
24. mpi_bcast
25. mpi_bcast
26. mpi_bcast
27. mpi_bcast
• Time spent waiting at a barrier before entering a collective can be a significant indication of load imbalance.

• MPI_SYNC group: for time spent waiting at the barrier before entering the collectives.

• Actual time spent in the collectives go to the MPI function group.

• Not to separate these groups, set PAT_RT_MPI_SYNC to 0 before aprun.

<table>
<thead>
<tr>
<th>Time %</th>
<th>Time (ms)</th>
<th>Imbalance %</th>
<th>Imbalance</th>
<th>Calls</th>
<th>Experiment=1 Group</th>
<th>Function</th>
<th>PE=&quot;HIDE&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>0.078172</td>
<td>--</td>
<td>--</td>
<td>124.0</td>
<td>Total</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>83.2%</td>
<td>0.065893</td>
<td>--</td>
<td>--</td>
<td>26.0</td>
<td>USER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60.0%</td>
<td>0.068578</td>
<td>0.003254</td>
<td>6.5%</td>
<td>1.0</td>
<td>jacobi_mpi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.7%</td>
<td>0.014959</td>
<td>0.003370</td>
<td>11.4%</td>
<td>10.0</td>
<td>compute_diff</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.8%</td>
<td>0.010086</td>
<td>--</td>
<td>--</td>
<td>14.0</td>
<td>MPI_SYNC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.9%</td>
<td>0.006150</td>
<td>0.006052</td>
<td>49.2%</td>
<td>4.0</td>
<td>mpi_bcast(sync)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.0%</td>
<td>0.003912</td>
<td>0.002366</td>
<td>22.5%</td>
<td>10.0</td>
<td>mpi_allreduce(sync)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.9%</td>
<td>0.003038</td>
<td>--</td>
<td>--</td>
<td>84.0</td>
<td>MPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.0%</td>
<td>0.002330</td>
<td>0.011095</td>
<td>83.0%</td>
<td>20.0</td>
<td>MPI_SENDRECV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.5%</td>
<td>0.000413</td>
<td>0.000051</td>
<td>11.0%</td>
<td>10.0</td>
<td>MPI_ALLREDUCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2%</td>
<td>0.000173</td>
<td>0.000317</td>
<td>65.0%</td>
<td>1.0</td>
<td>mpi_finalize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1%</td>
<td>0.000108</td>
<td>0.000077</td>
<td>41.8%</td>
<td>4.0</td>
<td>mpi_bcast</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1%</td>
<td>0.000046</td>
<td>0.000043</td>
<td>48.6%</td>
<td>24.0</td>
<td>MPI_COMM_SIZE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0%</td>
<td>0.000017</td>
<td>0.000022</td>
<td>57.3%</td>
<td>24.0</td>
<td>mpi_comm_rank</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0%</td>
<td>0.000000</td>
<td>0.000001</td>
<td>54.6%</td>
<td>1.0</td>
<td>MPI_INIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MPI Rank Order Suggestion

- `-O mpi_sm_rank_order`
  
  `[ -s rank_grid_dim=M,N ]`
  
  `[ -s rank_cell_dim=m,n ]`
  
  `[ -s mpi_dest=d ]`

  - Based on sent messages
  - `-s rank_*`: specify a different MPI process topology
    - Global topology, \( M \times N \)
    - topology per node, \( m \times n \)
  - consider \( d \) busiest partners (default, 8)

  > pat_report -O mpi_sm_rank_order
  
  jacobi_mpi+pat+30971-19tdot.xf

  > ls MPICH_RANK_ORDER.*
  
  MPICH_RANK_ORDER.d ➡
  MPICH_RANK_ORDER.u ➡

  Examined the cases 0, 1, 2, and 3 (‘d’ and ‘u’) for
  MPICH_RANK_REORDER_METHOD;
  provided MPICH_RANK_ORDER file for
  the ‘d’ and ‘u’ cases.
Table 1: Sent Message Stats and Suggested MPI Rank Order

<table>
<thead>
<tr>
<th>Number</th>
<th>Rank</th>
<th>Partners</th>
<th>Count</th>
<th>Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>238</td>
<td>1</td>
</tr>
</tbody>
</table>

Sent Msg Total Bytes per Max Avg Max
Total Bytes Total Bytes Rank
1.54e+06 1.53e+06 1

Best: smallest ratios wrt to SMP ord.

24 cores per node: Sent Msg Total Bytes per node

<table>
<thead>
<tr>
<th>Rank</th>
<th>Order</th>
<th>Send Max/ SMP Total Bytes SMP Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>d</td>
<td>6.14e+06 100.0% 1.38e+06 100.0%</td>
</tr>
<tr>
<td>2</td>
<td>u</td>
<td>6.14e+06 100.0% 5.53e+06 400.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.69e+07 2400.0% 3.32e+07 2400.0%</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>3.69e+07 2400.0% 3.67e+07 2655.6%</td>
</tr>
</tbody>
</table>

The custom rank placement in this file is the one labeled 'd' in the report from:

```
    cat MPICH_RANK_ORDER.d
```

To use this file, copy it to MPICH_RANK_ORDER and set the environment variable MPICH_RANK_REORDER_METHOD to 3 prior to executing the program.

```
    pat_report -0 mpi_sm_rank_order \
    -s mpi_dests=8 \ 
    /scratch/scratchdirs/wyang/mpi/jacobi_mpi+pat+30971-19tde ... 
    116,117,118,119,6,7,228,229,4,5,230,231,236,237,1,0,2,3,232,233, 
    3,60,61,174,175,64,65,170,171,58,59,176,177,66,67,168,169,172,173,164 
```
• ‘-O mpi_rank_order
    [-s mro_metric=…]’: based on specified metric
    – ‘time’, if no metric is specified
    – HWPC if a.out+pat was run with this set (later on this)

```
> pat_report -O mpi_rank_order
  jacob_mpi+pat+30971-19tdot.ap2
> ls MPICH_RANK_ORDER.*
MPICH_RANK_ORDER.d ←
```

Examined the cases 0, 1, 2, and 3 (‘d’) for 
MPICH_RANK_REORDER_METHOD; provided MPICH_RANK_ORDER file for ‘d’ case.

Table 1: Suggested MPI Rank Order

<table>
<thead>
<tr>
<th>USER Time per MPI rank</th>
<th>Max</th>
<th>Avg</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER Time</td>
<td>USER Time Rank</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.68e+08</td>
<td>2.55e+08</td>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>

24 cores per node: USER Time per node

<table>
<thead>
<tr>
<th>Rank</th>
<th>USER Time</th>
<th>SMP USER Time</th>
<th>SMP Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.13e+09</td>
<td>6.11e+09</td>
<td>3,13,23,33,43,53,63,73,83,93</td>
</tr>
<tr>
<td>1</td>
<td>5.13e+09</td>
<td>6.11e+09</td>
<td>0,1,2,3,4,5,6,7,8,9,10,11,12</td>
</tr>
<tr>
<td>2</td>
<td>5.13e+09</td>
<td>6.11e+09</td>
<td>3,15,23,36,43,56,63,76,83,96</td>
</tr>
<tr>
<td>d</td>
<td>5.13e+09</td>
<td>6.11e+09</td>
<td>143,113,64,63,79,105,26,54,3</td>
</tr>
</tbody>
</table>
> module load perftools
> ftn -mp=nonuma -c jacobi_omp.f90
> ftn -mp=nonuma -o jacobi_omp
   jacobi_omp.o
> pat_build -g omp -u jacobi_omp

Run jacobi_omp+pat in a batch job

> pat_report
   jacobi_omp+pat+15307-18tot.xf
OpenMP Code Example

Table 1: Profile by Function Group and Function

<table>
<thead>
<tr>
<th>Time %</th>
<th>Time</th>
<th>Imb. Time</th>
<th>Imb.</th>
<th>Calls</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Thread:CHIDE</td>
</tr>
<tr>
<td>100.0%</td>
<td>4.123724</td>
<td>--</td>
<td>--</td>
<td>379.0</td>
<td>Total</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88.9%</td>
<td>3.671375</td>
<td>--</td>
<td>--</td>
<td>253.0</td>
<td>USER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36.8%</td>
<td>1.519932</td>
<td>0.007403</td>
<td>0.5%</td>
<td>20.0</td>
<td>MAIN_LOOP@li.35</td>
</tr>
<tr>
<td>14.1%</td>
<td>0.582412</td>
<td>0.001967</td>
<td>0.4%</td>
<td>20.0</td>
<td>compute_diff_LOOP@li.166</td>
</tr>
<tr>
<td>13.7%</td>
<td>0.565131</td>
<td>0.012983</td>
<td>2.4%</td>
<td>20.0</td>
<td>MAIN_LOOP@li.56</td>
</tr>
<tr>
<td>11.4%</td>
<td>0.471882</td>
<td>0.000751</td>
<td>0.2%</td>
<td>1.0</td>
<td>init_fields_LOOP@li.104</td>
</tr>
<tr>
<td>11.4%</td>
<td>0.470750</td>
<td>0.000434</td>
<td>0.1%</td>
<td>1.0</td>
<td>init_fields_LOOP@li.108</td>
</tr>
<tr>
<td>12.2%</td>
<td>0.049400</td>
<td>0.047252</td>
<td>100.0%</td>
<td>1.0</td>
<td>MAIN_LOOP@li.138</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.0%</td>
<td>0.453033</td>
<td>--</td>
<td>--</td>
<td>184.0</td>
<td>USER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.9%</td>
<td>0.449830</td>
<td>0.430272</td>
<td>100.0%</td>
<td>1.0</td>
<td>init_fields_REGION@li.104(ovhd)</td>
</tr>
<tr>
<td>0.0%</td>
<td>0.001494</td>
<td>0.001429</td>
<td>100.0%</td>
<td>20.0</td>
<td>MAIN_REGION@li.135(ovhd)</td>
</tr>
<tr>
<td>0.0%</td>
<td>0.000885</td>
<td>0.000834</td>
<td>100.0%</td>
<td>20.0</td>
<td>compute_diff_REGION@li.166(ovhd)</td>
</tr>
<tr>
<td>0.0%</td>
<td>0.000343</td>
<td>0.000329</td>
<td>100.0%</td>
<td>1.0</td>
<td>omp_set_num_threads</td>
</tr>
<tr>
<td>0.0%</td>
<td>0.000278</td>
<td>0.000256</td>
<td>100.0%</td>
<td>21.0</td>
<td>set_bc_REGION@li.138(ovhd)</td>
</tr>
<tr>
<td>0.0%</td>
<td>0.000123</td>
<td>0.000118</td>
<td>100.0%</td>
<td>20.0</td>
<td>MAIN_REGION@li.56(ovhd)</td>
</tr>
<tr>
<td>0.0%</td>
<td>0.000079</td>
<td>0.000076</td>
<td>100.0%</td>
<td>21.0</td>
<td>set_bc_REGION@li.145(ovhd)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1%</td>
<td>0.004316</td>
<td>--</td>
<td>--</td>
<td>22.0</td>
<td>PTHREAD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pthread_create</td>
</tr>
</tbody>
</table>

- **24 thread case**
- **Good load balance in parallel regions!**
- **Overhead:**
  - Imb. =100%!, all in the master thread (pat_report’s ‘–O load_balance’ to see this more clearly)
  - largest in init_fields_
jacobi_omp.f90 with ngrind=9,600 and maxiter=20
Automatic Program Analysis (APA)

- One may not know in advance where large run time is spent; tracing all the functions can be overwhelming due to large overhead

1. Have the tool detect the most time consuming functions in the application with a sampling experiment

2. Feed this information back to the tool to instrument for focused data collection

3. Get performance information on the most significant parts of the application

- APA does this for you (you can do the same thing by hand)
Automatic Program Analysis (APA)

1. **pat_build -O apa a.out**
   - Produces the instrumented executable `a.out+pat` for sampling
2. **aprun -n ... a.out+pat**
   - Produces data file, e.g., `a.out+pat+4677-19sdot.xf`
3. **pat_report a.out+pat+4571-19sdot.xf**
   - Produces `a.out+pat+4571-19sdot.apa` (suggested options for tracing)
   - Produces `a.out+pat+4571-19sdot.ap2`
4. **Edit a.out+pat+4571-19sdot.apa, if necessary (next slide)**
5. **pat_build -O a.out+pat+4571-19sdot.apa**
   - Produces `a.out+apa` for tracing
6. **aprun -n ... a.out+apa**
   - Produces `a.out+apa+4590-19tdot.xf`
7. **pat_report a.out+apa+4590-19tdot.xf > out**

**Exercise in perftools/apa**
• Recommended pat_build options for tracing
  – Customize it for your need
    – Include/exclude functions
    – Add/change options
  – Note that the suggestions may not be valid for a very different task/thread configuration

```bash
# You can edit this file, if desired, and use it
# to reinstrument the program for tracing like this:
#
#pat_build -o jacobi_mpi+apa+24908-19sdot.apa
# These suggested trace options are based on data from:
# /scratch/scratchdirs/wyang/mpi+apa+24908-19sdot.apa+24908-19sdot.apa
# HWPC group to collect by default.
-Ortenu=PAT_RT_HWPC=1 # Summary with TLB metrics.
# Libraries to trace.
g mpi
# User-defined functions to trace, sorted by % of samples.
-w # Enable tracing of user-defined functions.
# Note: -u should NOT be specified as an additional option.
# 54.17% 1798 bytes
- T MAIN_
# 16.67% 296 bytes
- T compute_diff_
...
Instrumented program to create
-o jacobi_mpi+apa # New instrumented program.

Command to run
PAT_RT_HWPC set to 1
mpi trace group chosen
Sampling run result
Program to instrument
Trace ‘MAIN’, but not ‘compute_diff’
```
• Assume your code contains initialization and solution sections
• Want to analyze performance only of solution
• How to do this? Several approaches:
  – Init section is only one routine (or just a few routines): eliminate it (or them) from the profile.
  – Init section is many routines: Use API to define a profile region that excludes init
  – What happens if some routines shared by init and solve? Use API to turn profiling on and off as needed

Exercise in perftools/api
Arbitrary, user-assigned id’s

Using the API

```plaintext
including mpi.h
include 'pat_api.h'
...
call pat_record(PAT_STATE_OFF, ierr)
call mpi_init(ierr)
...
| Main solver loop.

call pat_record(PAT_STATE_ON, ierr)

h = 1.0 / n

do k=1,maxiter
    call pat_region_begin(1, 'compute_unew', ierr)
    ...
call set_bc(unew,n,js,je)
    ...
call compute_diff(u,unew,n,js,je,diffnorm)
    ...
call pat_region_begin(2, 'set_u_to_unew', ierr)
u(:,:, :) = unew(:,:, :)
call pat_region_end(2, ierr)
...
endo

call pat_record(PAT_STATE_OFF, ierr)
...
```

Run jacobi_mpi_api+pat in a batch job.

```plaintext
> module load perftools
> ftn -c jacobi_mpi_api.f90
> ftn -o jacobi_mpi_api jacobi_mpi_api.o
> pat_build -g mpi -u jacobi_mpi_api

> pat_report
jacobi_mpi_api+pat+502-19tdot.xf

| Time % | Time | Imb. Time | Imb. | Calls | Group | Function | PE - 'HIDE'
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>2.571582</td>
<td>--</td>
<td>--</td>
<td>122.0</td>
<td>Total</td>
<td></td>
<td></td>
</tr>
<tr>
<td>98.9%</td>
<td>2.543295</td>
<td>--</td>
<td>--</td>
<td>42.0</td>
<td>USER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45.1%</td>
<td>1.158653</td>
<td>0.0008723</td>
<td>0.8%</td>
<td>10.0</td>
<td>#1,compute_unew</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22.0%</td>
<td>0.566156</td>
<td>0.0008723</td>
<td>1.6%</td>
<td>10.0</td>
<td>#2, set_u_to_unew</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21.6%</td>
<td>0.555785</td>
<td>0.005339</td>
<td>1.8%</td>
<td>10.0</td>
<td>compute_diff_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.2%</td>
<td>0.252097</td>
<td>0.174330</td>
<td>40.8%</td>
<td>1.0</td>
<td>MAIN_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8%</td>
<td>0.000608</td>
<td>0.014145</td>
<td>71.7%</td>
<td>10.0</td>
<td>set_bc_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0%</td>
<td>0.000006</td>
<td>0.000004</td>
<td>38.7%</td>
<td>1.0</td>
<td>main</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.9%</td>
<td>0.022937</td>
<td>--</td>
<td>--</td>
<td>10.0</td>
<td>IMPI_SYNC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2%</td>
<td>0.005269</td>
<td>--</td>
<td>--</td>
<td>70.0</td>
<td>IMPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2%</td>
<td>0.004385</td>
<td>0.007111</td>
<td>60.8%</td>
<td>20.0</td>
<td>IMPI_SENDREC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0%</td>
<td>0.000402</td>
<td>0.000064</td>
<td>14.0%</td>
<td>10.0</td>
<td>IMPI_ALLREDUCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0%</td>
<td>0.000025</td>
<td>0.000007</td>
<td>23.4%</td>
<td>20.0</td>
<td>IMPI_COMM_SIZE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0%</td>
<td>0.000007</td>
<td>0.000002</td>
<td>19.0%</td>
<td>20.0</td>
<td>IMPI_COMM_rank</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Hardware Performance Counters

- Registers available on the processor that count certain events
- Minimal overhead
  - They’re running all the time
  - Typically one clock period to read
- Potentially rich source of performance information
Types of Counters

• Cycles
• Instruction count
• Memory references, cache hits/misses
• Floating-point instructions
• Resource utilization
PAPI Event Counters

- 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:
    module load perftools
    aprun -n 1 papi_avail

- AMD native events also provided; use ‘papi_native_avail’:
  aprun -n 1 papi_native_avail
Hardware Performance Monitoring

• Specify hardware counters to be monitored during sampling or tracing
  – Default is “off” (no HW counters measured)
  – Choose up to 4 events

• Can specify individual events:
  
  setenv PAT_RT_HWPC "PAPI_FP_OPS,PAPI_L1_DCM"
  aprun -n ... a.out+pat  (or a.out+apa)

• Or predefined event group number (next slide):
  
  setenv PAT_RT_HWPC 1
  aprun -n ... a.out+pat  (or a.out+apa)

• Multiplexing (monitoring more than 4 events) to be supported in later versions (5.2?)

Exercise in perftools/hwpc
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Summary with instruction metrics</td>
</tr>
<tr>
<td>1</td>
<td>Summary with translation lookaside buffer (TLB) metrics</td>
</tr>
<tr>
<td>2</td>
<td>L1 and L2 cache metrics</td>
</tr>
<tr>
<td>3</td>
<td>Bandwidth information</td>
</tr>
<tr>
<td>4</td>
<td>*** DO NOT USE, not supported on Quad-core or later AMD Opteron processors***</td>
</tr>
<tr>
<td>5</td>
<td>Floating point instructions</td>
</tr>
<tr>
<td>6</td>
<td>Cycles stalled and resources empty</td>
</tr>
<tr>
<td>7</td>
<td>Cycles stalled and resources full</td>
</tr>
<tr>
<td>8</td>
<td>Instructions and branches</td>
</tr>
<tr>
<td>9</td>
<td>Instruction cache values</td>
</tr>
<tr>
<td>10</td>
<td>Cache hierarchy</td>
</tr>
<tr>
<td>11</td>
<td>Floating point instructions (2)</td>
</tr>
<tr>
<td>12</td>
<td>Floating point instructions (vectorization)</td>
</tr>
<tr>
<td>13</td>
<td>Floating point instructions (single precision)</td>
</tr>
<tr>
<td>14</td>
<td>Floating point instructions (double precision)</td>
</tr>
<tr>
<td>15</td>
<td>L3 cache</td>
</tr>
<tr>
<td>16</td>
<td>L3 cache, core-level reads</td>
</tr>
<tr>
<td>17</td>
<td>L3 cache, core-level misses</td>
</tr>
<tr>
<td>18</td>
<td>L3 cache, core-level fills caused by L2 evictions</td>
</tr>
<tr>
<td>19</td>
<td>Prefetches</td>
</tr>
</tbody>
</table>

---
Hardware Performance Monitoring

PAT_RT_HWPC = 1

| User / Main |
|-----------------|-----------------|
| Time% | 72.3% |
| Time | 7.428213 secs |
| Calls | 0.1/sec | 1.0 calls |
| PAPI_L1_DCM | 4.392M/sec | 32625209 misses |
| PAPI_TLB_DM | 1.224M/sec | 9091457 misses |
| PAPI_L1_DCA | 996.063M/sec | 7398982001 refs |
| PAPI_FP_OPS | 2232.437M/sec | 16583040487 ops |
| User time (approx) | 7.428 secs | 15599292538 cycles 100.0%Time |
| Average Time per Call | 7.428213 secs |

CrayPat Overhead : Time | 0.0%

HW FP Ops / User time | 2232.437M/sec | 16583040487 ops 26.6%peak(DP)
HW FP Ops / WCT | 2232.437M/sec |

Computational intensity | 1.06 ops/cycle | 2.24 ops/ref
MFLOPS (aggregate) | 2232.44M/sec |
TLB utilization | 813.84 refs/miss | 0.795 avg uses
D1 cache hit,miss ratios | 99.6% hits | 0.4% misses
D1 cache utilization (misses) | 226.79 refs/miss | 14.174 avg hits

data size=4 (because single precision was used)
cacheline: 64 Bytes
L1 cache: 64 KB, dedicated for each core
L2 cache: 512 KB, dedicated for each core
page: 4 KB (2 MB if huge page)
**PAT_RT_HWPC = 2**

<table>
<thead>
<tr>
<th>USER / MAIN_</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time%</td>
</tr>
<tr>
<td>Time</td>
</tr>
<tr>
<td>Calls</td>
</tr>
<tr>
<td>DATA_CACHE_REFILLS:</td>
</tr>
<tr>
<td>L2_MODIFIED:L2_OWNED:</td>
</tr>
<tr>
<td>L2_EXCLUSIVE:L2_SHARED</td>
</tr>
<tr>
<td>DATA_CACHE_REFILLS_FROM_SYSTEM:</td>
</tr>
<tr>
<td>ALL</td>
</tr>
<tr>
<td>PAPI_L1_DCM</td>
</tr>
<tr>
<td>PAPI_L1_DCA</td>
</tr>
<tr>
<td>User time (approx)</td>
</tr>
<tr>
<td>Average Time per Call</td>
</tr>
<tr>
<td>CrayPat Overhead : Time</td>
</tr>
<tr>
<td>D1 cache hit,miss ratios</td>
</tr>
<tr>
<td>D1 cache utilization (misses)</td>
</tr>
<tr>
<td>D1 cache utilization (refills)</td>
</tr>
<tr>
<td>D2 cache hit,miss ratio</td>
</tr>
<tr>
<td>D1+D2 cache hit,miss ratio</td>
</tr>
<tr>
<td>D1+D2 cache utilization</td>
</tr>
<tr>
<td>System to D1 refill</td>
</tr>
<tr>
<td>System to D1 bandwidth</td>
</tr>
<tr>
<td>D2 to D1 bandwidth</td>
</tr>
</tbody>
</table>
**Hardware Performance Monitoring**

**PAT_RT_HWPC = 5**

```plaintext
USER / MAIN_

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time%</td>
<td>72.9%</td>
</tr>
<tr>
<td>Time</td>
<td>7.654353 secs</td>
</tr>
<tr>
<td>Calls</td>
<td>0.1 /sec</td>
</tr>
<tr>
<td>RETIRED_MMX_AND_FP_INSTRUCTIONS:</td>
<td></td>
</tr>
<tr>
<td>PACKED_SSE_AND_SSE2</td>
<td>964.770M/sec 7384702007 instr</td>
</tr>
<tr>
<td>PAPI_FML_INS</td>
<td>241.562M/sec 1849000244 ops</td>
</tr>
<tr>
<td>PAPI_FAD_INS</td>
<td>301.927M/sec 2311057161 ops</td>
</tr>
<tr>
<td>PAPI_FDV_INS</td>
<td>1 /sec</td>
</tr>
<tr>
<td>User time (approx)</td>
<td>7.654 secs 16074189254 cycles 100.0%Time</td>
</tr>
<tr>
<td>Average Time per Call</td>
<td>7.654353 secs</td>
</tr>
<tr>
<td>CrayPat Overhead : Time</td>
<td>0.0%</td>
</tr>
<tr>
<td>HW FP Ops / Cycles</td>
<td>0.26 ops/cycle</td>
</tr>
<tr>
<td>HW FP Ops / User time</td>
<td>543.488M/sec 4160574055 ops 6.5%peak(DP)</td>
</tr>
<tr>
<td>HW FP Ops / WCT</td>
<td>543.488M/sec</td>
</tr>
<tr>
<td>FP Multiply / FP Ops</td>
<td>44.4%</td>
</tr>
<tr>
<td>FP Add / FP Ops</td>
<td>55.6%</td>
</tr>
<tr>
<td>MFLOPS (aggregate)</td>
<td>543.49M/sec</td>
</tr>
</tbody>
</table>
```

Relative ratios for multiplies and adds.
**Hardware Performance Monitoring**

**PAT_RT_HWPC = 12**

<table>
<thead>
<tr>
<th>USER / MAIN_</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time%</td>
</tr>
<tr>
<td>Time</td>
</tr>
<tr>
<td>Calls</td>
</tr>
<tr>
<td>RETIRED_SSE_OPERATIONS:</td>
</tr>
<tr>
<td>SINGLE_ADD_SUB_OPS:</td>
</tr>
<tr>
<td>SINGLE_MUL_OPS</td>
</tr>
<tr>
<td>RETIRED_SSE_OPERATIONS:</td>
</tr>
<tr>
<td>DOUBLE_ADD_SUB_OPS:</td>
</tr>
<tr>
<td>DOUBLE_MUL_OPS</td>
</tr>
<tr>
<td>RETIRED_SSE_OPERATIONS:</td>
</tr>
<tr>
<td>SINGLE_ADD_SUB_OPS:</td>
</tr>
<tr>
<td>SINGLE_MUL_OPS:OP_TYPE</td>
</tr>
<tr>
<td>RETIRED_SSE_OPERATIONS:</td>
</tr>
<tr>
<td>DOUBLE_ADD_SUB_OPS:</td>
</tr>
<tr>
<td>DOUBLE_MUL_OPS:OP_TYPE</td>
</tr>
<tr>
<td>User time (approx)</td>
</tr>
<tr>
<td>Average Time per Call</td>
</tr>
<tr>
<td>CrayPat Overhead : Time</td>
</tr>
</tbody>
</table>

vector length (for sp) for 128-bit wide SSE2 vector operation

\[
= 16583040480 / 4154398320 = 3.99
\]

Compiled with ‘ftn –fastsse ...’
### Guidelines to Identify the Need for Optimization

<table>
<thead>
<tr>
<th>Derived metric</th>
<th>Optimization needed when*</th>
<th>PAT_RT_HWPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational intensity</td>
<td>&lt; 0.5 ops/ref</td>
<td>0, 1</td>
</tr>
<tr>
<td>L1 cache hit ratio</td>
<td>&lt; 90%</td>
<td>0, 1, 2</td>
</tr>
<tr>
<td>L1 cache utilization (misses)</td>
<td>&lt; 1 avg hit</td>
<td>0, 1, 2</td>
</tr>
<tr>
<td>L1+L2 cache hit ratio</td>
<td>&lt; 92%</td>
<td>2</td>
</tr>
<tr>
<td>L1+L2 cache utilization (misses)</td>
<td>&lt; 1 avg hit</td>
<td>2</td>
</tr>
<tr>
<td>TLB utilization</td>
<td>&lt; 0.9 avg use</td>
<td>1</td>
</tr>
<tr>
<td>(FP Multiply / FP Ops) or (FP Add / FP Ops)</td>
<td>&lt; 25%</td>
<td>5</td>
</tr>
<tr>
<td>Vectorization</td>
<td>&lt; 1.5 for dp; 3 for sp</td>
<td>12 (13, 14)</td>
</tr>
</tbody>
</table>

* Suggested by Cray
• Use PAT_RT_NWPC instead of PAT_RT_HWPC

• See ‘Overview of Gemini Hardware Counters’, S-0025-10
  – http://docs.cray.com
Harvey Wasserman

USING CRAY’S APPRENTICE TOOL
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
  1. module load perftools
  2. ftn -c mpptest.f
  3. ftn -o mpptest mpptest.o
  4. pat_build -u -g mpi mpptest
  5. aprun -n 16 mpptest+pat
  6. pat_report mpptest+pat+PID.xf > my_report
  7. app2 [--limit_per_pe tags] [XXX.ap2]
• Identify files on the command line or via the GUI:
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
Apprentice Call Tree Report

Horizontal size = cumulative time in node’s children

Vertical size = time in computation

Green nodes: no callees

Stacked bar charts: load balancing info.
Yellow=Max
purple=Average
Light Blue=Minimum

Calipers work

Right-click to view source
Red arc identifies path to the highest detected load imbalance.

Call tree stops there because nodes were filtered out. To see the hidden nodes, right-click on the node attached to the marker and select "unhide all children" or "unhide one level".

Double-click on ? for more info about load imbalance.
• Run code with
  `setenv PAT_RT_SUMMARY 0`

• Caution: Can generate enormous data files and take *forever*
Apprentice Traffic Report

- Click here to select this report
- Shows message traces as a function of time
- Look for large blocks of barriers held up by a single processor
- Zoom is important; also, run just a portion of your simulation
- Scroll, zoom, filter: right-click on trace
Mouse hover pops up window showing source location.
Tracing Analysis Example

VAMPIR Analysis with Two Wavefronts

- Identified Message
  - Message sent from Process 15 to Process 11
  - communicator: 0, type: 8000
  - length: 49152
  - sent at 262.76 ms, received at 288.05 ms (diff. 25.29 ms)
  - Data rate: 1.944 MBytes/sec

- Timeline with Processes and Messages:
  - Processes 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
  - Messages: MPI_Recev, MPI_Allreduce
  - User Code

- MPI and Application Events
Mosaic View

- Can right-click here for more options
- Click here to select this report
- Shows Interprocessor communication topology and color-coded intensity
- Colors show average time (green=low, red=high)
- Very difficult to interpret by itself – use the Craypat message statistics with it.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Science Area</th>
<th>Algorithm Space</th>
<th>Base Case Concurrency</th>
<th>Problem Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAM</td>
<td>Climate (BER)</td>
<td>Navier Stokes CFD</td>
<td>56, 240</td>
<td>Strong scaling, D Grid, (~.5 deg resolution); 240 timesteps</td>
</tr>
<tr>
<td>GAMESS</td>
<td>Quantum Chem (BES)</td>
<td>Dense linear algebra</td>
<td>384, 1024 (Same as Ti-09)</td>
<td>DFT gradient, MP2 gradient</td>
</tr>
<tr>
<td>GTC</td>
<td>Fusion (FES)</td>
<td>PIC, finite difference</td>
<td>512, 2048</td>
<td>100 particles per cell</td>
</tr>
<tr>
<td>IMPACT-T</td>
<td>Accelerator Physics (HEP)</td>
<td>PIC, FFT component</td>
<td>256,1024</td>
<td>50 particles per cell</td>
</tr>
<tr>
<td>MAESTRO</td>
<td>Astrophysics (HEP)</td>
<td>Low Mach Hydro; block structured-grid multiphysics</td>
<td>512, 2048</td>
<td>16 32^3 boxes per proc; 10 timesteps</td>
</tr>
<tr>
<td>MILC</td>
<td>Lattice Gauge Physics (NP)</td>
<td>Conjugate gradient, sparse matrix; FFT</td>
<td>256, 1024, 8192</td>
<td>Weak scaling, 8x8x8x9 Local Grid, ~70,000 iters</td>
</tr>
<tr>
<td>PARATEC</td>
<td>Material Science (BES)</td>
<td>DFT; FFT, BLAS3</td>
<td>256, 1024</td>
<td>Strong scaling, 686 Atoms, 1372 bands, 20 iters</td>
</tr>
</tbody>
</table>
NERSC6 Benchmarks
Communication Topology*

MILC
MAESTRO
GTC
PARATEC
IMPACT-T
CAM

*From IPM
**Sample of CI & %MPI**

<table>
<thead>
<tr>
<th>Code</th>
<th>MAESTRO</th>
<th>GAMES</th>
<th>CAM</th>
<th>IMPACT-T</th>
<th>GTC</th>
<th>MILC</th>
<th>PARATEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CI*</td>
<td>0.24</td>
<td>0.61</td>
<td>0.67</td>
<td>0.77</td>
<td>1.15</td>
<td>1.39</td>
<td>1.50</td>
</tr>
<tr>
<td>Cray XT4 % peak per core (largest case)</td>
<td>5%</td>
<td>12%</td>
<td>13%</td>
<td>14%</td>
<td>24%</td>
<td>14%</td>
<td>44%</td>
</tr>
<tr>
<td>Cray XT4 % MPI medium</td>
<td>20%</td>
<td>29%</td>
<td>9%</td>
<td>4%</td>
<td>12%</td>
<td>27%</td>
<td></td>
</tr>
<tr>
<td>Cray XT4 % MPI large</td>
<td></td>
<td>35%</td>
<td>40%</td>
<td>6%</td>
<td>23%</td>
<td>64%</td>
<td></td>
</tr>
<tr>
<td>Cray XT4 % MPI extra large</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>30%</td>
<td>n/a</td>
</tr>
<tr>
<td>Cray XT4 avg msg size med</td>
<td>2 K</td>
<td>n/a</td>
<td>113 K</td>
<td>35 KB</td>
<td>1 MB</td>
<td>16 KB</td>
<td>34 KB</td>
</tr>
</tbody>
</table>

*CI is the computational intensity, the ratio of # of Floating Point Operations to # of memory operations.*
For More Information

- Using Cray Performance Analysis Tools, S–2376–51
  - http://docs.cray.com/books/S-2376-51/S-2376-51.pdf
- man craypat
- man pat_build
- man pat_report
- man pat_help \(\Rightarrow\) very useful tutorial program
- man app2
- man hwpc
- man intro_perftools
- man papi
- man papi_counters
For More Information

- “Performance Tuning of Scientific Applications,” CRC Press 2010
Same code, same problem size, run on the same 24 cores. What is different? Why might one perform better than the other? What performance characteristics are different?
Exercise

• Get the sweep3d code. Untar
• To build: type ‘make mpi’
• Instrument for mpi, user
• Get an interactive batch session, 24 cores
• Run 3 sweep3d cases on 24 cores creating Apprentice traffic/mosaic views:
  – cp input1 input; aprun -n 24 ...
  – cp input2 input; aprun -n 24 ...
  – cp input3 input; aprun -n 24 ...
• View the results from each run in Apprentice and try to explain what you see.
# Performance Metrics

CPU Time = \( N_{\text{inst}} \times \text{CPI} \times \text{Clock rate} \)