IPM - A Tutorial

Nicholas J Wright
Karl Fuerlinger

nwright@sdsc.edu
fuerling@eecs.berkeley.edu

Allan Snavely, SDSC
David Skinner LBNL
Katherine Yelick LBNL & UCB
Menu

• Performance Analysis Concepts and Definitions
  • Why and when to look at performance
  • Types of performance measurement
• Examining typical performance issues today using IPM
• Summary
Motivation

• Performance Analysis is important
  • Large investments in HPC systems
    • Procurement: ~$40 Mio
    • Operational costs: ~$5 Mio per year
    • Electricity: 1 MW/year ~$1 Mio

• Goal: solve larger problems
• Goal: solve problems faster
Concepts and Definitions

• The typical performance optimization cycle

  Code Development

  Measure

  Analyze

  Modify / Tune

  Usage / Production

  functionally complete and correct program

  instrumentation

  complete, correct and well-performing program
**Instrumentation**

- Instrumentation = adding measurement probes to the code to observe its execution

- Can be done on several levels

- Different techniques for different levels

- Different overheads and levels of accuracy with each technique

- No instrumentation: run in a simulator. E.g., Valgrind
**Instrumentation – Examples (1)**

- **Source code instrumentation**
  - User added time measurement, etc. (e.g., `printf()`, `gettimeofday()`)
  - Many tools expose mechanisms for source code instrumentation in addition to automatic instrumentation facilities they offer
  - Instrument program phases:
    - initialization/main iteration loop/data post processing
  - Pramga and pre-processor based
    ```
    #pragma pomp inst begin(foo)
    #pragma pomp inst end(foo)
    ```
  - Macro / function call based
    ```
    ELG_USER_START("name");
    . . .
    ELG_USER_END("name");
    ```
Instrumentation – Examples (2)

• Preprocessor Instrumentation
  • Example: Instrumenting OpenMP constructs with Opari
  • Preprocessor operation

Original source code → Preprocessor → Modified (instrumented) source code

• Example: Instrumentation of a parallel region

```
POMP_Parallel_fork [master]
#pragma omp parallel {
POMP_Parallel_begin [team]
  /* ORIGINAL CODE in parallel region */
POMP_Barrier_Enter [team]
#pragma omp barrier
POMP_Barrier_Exit [team]
POMP_Parallel_end [team]
}
POMP_Parallel_join [master]
```

This is used for OpenMP analysis in tools such as KOJAK/Scalasca/ompP

Instrumentation added by Opari
Instrumentation – Examples (3)

- Compiler Instrumentation
  - Many compilers can instrument functions automatically
  - GNU compiler flag: \texttt{-finstrument-functions}
  - Automatically calls functions on function entry/exit that a tool can capture
  - Not standardized across compilers, often undocumented flags, sometimes not available at all
  - GNU compiler example:

```c
void __cyg_profile_func_enter(void *this, void *callsite) {
    /* called on function entry */
}

void __cyg_profile_func_exit(void *this, void *callsite) {
    /* called just before returning from function */
}
```
**Instrumentation – Examples (4)**

- **Library Instrumentation:**
  - **MPI library interposition**
    - All functions are available under two names: `MPI_xxx` and `PMPI_xxx`, `MPI_xxx` symbols are *weak*, can be over-written by interposition library.
    - Measurement code in the interposition library measures begin, end, transmitted data, etc… and calls corresponding PMPI routine.
    - Not all MPI functions need to be instrumented.
Measurement

• Profiling vs. Tracing

• Profiling
  • Summary statistics of performance metrics
    • Number of times a routine was invoked
    • Exclusive, inclusive time/hpm counts spent executing it
    • Number of instrumented child routines invoked, etc.
    • Structure of invocations (call-trees/call-graphs)
    • Memory, message communication sizes

• Tracing
  • When and where events took place along a global timeline
    • Time-stamped log of events
    • Message communication events (sends/receives) are tracked
    • Shows when and from/to where messages were sent
    • Large volume of performance data generated usually leads to more perturbation in the program
Measurement: Profiling

• Profiling
  • Recording of summary information during execution
    • inclusive, exclusive time, # calls, hardware counter statistics, …
  • Reflects performance behavior of program entities
    • functions, loops, basic blocks
    • user-defined “semantic” entities
  • Very good for low-cost performance assessment
  • Helps to expose performance bottlenecks and hotspots
  • Implemented through either
    • sampling: periodic OS interrupts or hardware counter traps
    • measurement: direct insertion of measurement code
**Profiling: Inclusive vs. Exclusive**

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

- **Inclusive time for main**
  - 100 secs

- **Exclusive time for main**
  - 100-20-50-20=10 secs

- **Exclusive time sometimes called “self time”**

/* similar for other metrics, such as hardware performance counters, etc. */
Tracing Example: Instrumentation, Monitor, Trace

CPU A:
```c
void master {
    trace(EXIT, 1);
    ...
    trace(SEND, B);
    send(B, tag, buf);
    ...
    trace(EXIT, 1);
}
```

CPU B:
```c
void slave {
    trace(EXIT, 2);
    ...
    recv(A, tag, buf);
    ...
    trace(RECV, A);
    ...
    trace(EXIT, 2);
}
```

Event definition:
```
<table>
<thead>
<tr>
<th>Event</th>
<th>CPU</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTER 1</td>
<td>A</td>
<td>58</td>
</tr>
<tr>
<td>ENTER 2</td>
<td>B</td>
<td>60</td>
</tr>
<tr>
<td>SEND B</td>
<td>A</td>
<td>62</td>
</tr>
<tr>
<td>EXIT 1</td>
<td>A</td>
<td>64</td>
</tr>
<tr>
<td>RECV A</td>
<td>B</td>
<td>68</td>
</tr>
<tr>
<td>EXIT 2</td>
<td>B</td>
<td>69</td>
</tr>
</tbody>
</table>
```

MONITOR
Tracing: Timeline Visualization

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>master</td>
</tr>
<tr>
<td>2</td>
<td>slave</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
</tr>
</tbody>
</table>

... 

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>A ENTER 1</td>
</tr>
<tr>
<td>60</td>
<td>B ENTER 2</td>
</tr>
<tr>
<td>62</td>
<td>A SEND B</td>
</tr>
<tr>
<td>64</td>
<td>A EXIT 1</td>
</tr>
<tr>
<td>68</td>
<td>B RECV A</td>
</tr>
<tr>
<td>69</td>
<td>B EXIT 2</td>
</tr>
</tbody>
</table>

...
Measurement: Tracing

- Tracing
  - Recording of information about significant points (events) during program execution
    - entering/exiting code region (function, loop, block, …)
    - thread/process interactions (e.g., send/receive message)
  - Save information in event record
    - timestamp
    - CPU identifier, thread identifier
    - Event type and event-specific information
  - Event trace is a time-sequenced stream of event records
  - Can be used to reconstruct dynamic program behavior
  - Typically requires code instrumentation
**Performance Data Analysis**

- Draw conclusions from measured performance data

- **Manual analysis**
  - Visualization
  - Interactive exploration
  - Statistical analysis
  - Modeling

- **Automated analysis**
  - Try to cope with huge amounts of performance by automation
  - Examples: Paradyn, KOJAK, Scalasca
Trace File Visualization

SAN DIEGO SUPERCOMPUTER CENTER

PMaC
Performance Modeling and Characterization
Trace File Visualization

- Vampir: message communication statistics
3D performance data exploration

- Paraprof viewer (from the TAU toolset)
Automated Performance Analysis

- **Reason for Automation**
  - Size of systems: several tens of thousand of processors
  - LLNL Sequoia: ~1.6 million cores
  - Trend to multi-core
- **Large amounts of performance data when tracing**
  - Several gigabytes or even terabytes
  - Overwhelms user
- **Not all programmers are performance experts**
  - Scientists want to focus on their domain
  - Need to keep up with new machines
- **Automation can solve some of these issues**
Automation - Example

This is a situation that can be detected automatically by analyzing the trace file

-> *late sender* pattern
Menu

• Performance Analysis Concepts and Definitions
  • Why and when to look at performance
  • Types of performance measurement

• Examining typical performance issues today using IPM

• Summary
“Premature optimization is the root of all evil.” - Donald Knuth

- Before attempting to optimize make sure your code works correctly!
  - Debugging before tuning
  - Nobody really cares how fast you can compute
  - the wrong answer
- 80/20 Rule
  - Program spends 80% of its time in 20% of the code
  - Programmer spends 20% effort to get 80% of the total speedup possible
  - Know when to stop!
  - Don’t optimize what does not matter
Practical Performance Tuning

Successful tuning is combination of
- Right algorithm and libraries
- Compiler flags and pragmas / directives (Learn and use them)

• THINKING
Measurement > intuition (~guessing !)
- To determine performance problems
- To validate tuning decisions / optimizations (after each step!)
Typical Performance Analysis Procedure

• Do I have a performance problem at all? What am I trying to achieve?
  • Time / hardware counter measurements
  • Speedup and scalability measurements

• What is the main bottleneck (computation/communication...)?
  • Flat profiling (sampling / prof)
  • Why is it there?
Users Perspective: I Just Want to do My Science! - Barriers to Entry Must be Low

• “Yea, I tried that tool once, it took me 20 minutes to figure out how to get the code to compile, then it output a bunch of information, none of which I wanted, so I gave up.”

• Is it easier than this?
  Call timer
  Code_of_interest
  Call timer

• The carrot works. The stick does not.
MILC on Ranger – Runtime Shows Perfect Scalability

![Graph showing perfect scaling in runtime for MILC on Ranger. The graph plots time in seconds (y-axis) against number of cores (x-axis). The solid line represents total runtime, and the dotted line represents perfect scaling. As the number of cores increases, the runtime decreases, aligning closely with the perfect scaling line.]
Scaling: Good 1st Step: Do runtimes make sense?

Running fish_sim for 100-1000 fish on 1-32 CPUs we see

\[
\text{time} \sim \text{fish}^2 \quad \checkmark
\]
What is Integrated Performance Monitoring?

IPM provides a performance profile on a batch job.
How to use IPM : basics

1) Do “module load ipm”, then “setenv LD_PRELOAD …”

2) Upon completion you get

```bash
##IPMv0.85################################################################
## command : ../exe/pmemd -O -c inpcrd -o res (completed)
# host    : s05405                         mpi_tasks : 64 on 4 nodes
# start   : 02/22/05/10:03:55              wallclock : 24.278400 sec
# stop    : 02/22/05/10:04:17              %comm     : 32.43
# gbytes  : 2.57604e+00 total              gflop/sec : 2.04615e+00 total
#
#********************************************************************************

Maybe that’s enough. If so you’re done.
Have a nice day.
##IPMv0.85#

```plaintext
# command: ../exe/pmemd -O -c inpcrd -o res (completed)
# host: s05405
# mpi_tasks: 64 on 4 nodes
# start: 02/22/05/10:03:55
# wallclock: 24.278400 sec
# gbytes: 2.57604e+00 total

<table>
<thead>
<tr>
<th></th>
<th>[total]</th>
<th>&lt;avg&gt;</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>wallclock</td>
<td>1373.67</td>
<td>21.4636</td>
<td>21.1087</td>
<td>24.2784</td>
</tr>
<tr>
<td>user</td>
<td>936.95</td>
<td>14.6398</td>
<td>12.68</td>
<td>20.3</td>
</tr>
<tr>
<td>system</td>
<td>227.7</td>
<td>3.55781</td>
<td>1.51</td>
<td>5</td>
</tr>
<tr>
<td>mpi</td>
<td>503.853</td>
<td>7.8727</td>
<td>4.2293</td>
<td>9.13725</td>
</tr>
<tr>
<td>%comm</td>
<td>32.43</td>
<td>17.42</td>
<td>14.6398</td>
<td>41.407</td>
</tr>
<tr>
<td>gflop/sec</td>
<td>2.04614</td>
<td>0.0319709</td>
<td>0.02724</td>
<td>0.04041</td>
</tr>
<tr>
<td>gbytes</td>
<td>2.57604</td>
<td>0.0402507</td>
<td>0.0399284</td>
<td>0.0408173</td>
</tr>
<tr>
<td>gbytes_tx</td>
<td>0.665125</td>
<td>0.0103926</td>
<td>1.09673e-05</td>
<td>0.0368981</td>
</tr>
<tr>
<td>gbyte_rx</td>
<td>0.659763</td>
<td>0.0103088</td>
<td>9.83477e-07</td>
<td>0.0417372</td>
</tr>
</tbody>
</table>
```
Want more detail?  IPM_REPORT=full

<table>
<thead>
<tr>
<th></th>
<th>[time]</th>
<th>[calls]</th>
<th>&lt;%mpi&gt;</th>
<th>&lt;%wall&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM_CYC</td>
<td>3.00519e+11</td>
<td>4.69561e+09</td>
<td>4.50223e+09</td>
<td>5.83342e+09</td>
</tr>
<tr>
<td>PM_FPU0_CMPL</td>
<td>2.45263e+10</td>
<td>3.83223e+08</td>
<td>3.3396e+08</td>
<td>5.12702e+08</td>
</tr>
<tr>
<td>PM_FPU1_CMPL</td>
<td>1.48426e+10</td>
<td>2.31916e+08</td>
<td>1.90704e+08</td>
<td>2.8053e+08</td>
</tr>
<tr>
<td>PM_FPU_FMA</td>
<td>1.03083e+10</td>
<td>1.61067e+08</td>
<td>1.36815e+08</td>
<td>1.96841e+08</td>
</tr>
<tr>
<td>PM_INST_CMPL</td>
<td>3.33597e+11</td>
<td>5.21245e+09</td>
<td>4.33725e+09</td>
<td>6.44214e+09</td>
</tr>
<tr>
<td>PM_LD_CMPL</td>
<td>1.03239e+11</td>
<td>1.61311e+09</td>
<td>1.29033e+09</td>
<td>1.84128e+09</td>
</tr>
<tr>
<td>PM_ST_CMPL</td>
<td>7.19365e+10</td>
<td>1.12401e+09</td>
<td>8.77684e+08</td>
<td>1.29017e+09</td>
</tr>
<tr>
<td>PM_TLB_MISS</td>
<td>1.67892e+08</td>
<td>2.62332e+06</td>
<td>1.16104e+06</td>
<td>2.36664e+07</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>352.365</td>
<td>2816</td>
<td>69.93</td>
<td>22.68</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>81.0002</td>
<td>185729</td>
<td>16.08</td>
<td>5.21</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>38.6718</td>
<td>5184</td>
<td>7.68</td>
<td>2.49</td>
</tr>
<tr>
<td>MPI_Allgatherv</td>
<td>14.7468</td>
<td>448</td>
<td>2.93</td>
<td>0.95</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>12.9071</td>
<td>185729</td>
<td>2.56</td>
<td>0.83</td>
</tr>
<tr>
<td>MPI_Gatherv</td>
<td>2.06443</td>
<td>128</td>
<td>0.41</td>
<td>0.13</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>1.349</td>
<td>185729</td>
<td>0.27</td>
<td>0.09</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>0.606749</td>
<td>8064</td>
<td>0.12</td>
<td>0.04</td>
</tr>
<tr>
<td>MPI_Gather</td>
<td>0.0942596</td>
<td>192</td>
<td>0.02</td>
<td>0.01</td>
</tr>
</tbody>
</table>
Want More? – You’ll Need a Webbrowser
Which problems should be tackled with IPM?

- **Performance Bottleneck Identification**
  - Does the profile show what I expect it to?
  - Why is my code not scaling?
  - Why is my code running 20% slower than I expected?

- **Understanding Scaling**
  - Why does my code scale as it does? (MILC on Ranger)

- **Optimizing MPI Performance**
  - Combining Messages
Using IPM to Understand Common Performance Issues

- Dumb Mistakes
- Load balancing
- Combining Messages
- Scaling behavior
- Amdahl (serial) fractions
- Optimal Cache Usage
What’s wrong here?

Communication Event Statistics (100.00% detail)

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Buffer Size</th>
<th>Ncalls</th>
<th>Total Time</th>
<th>Min Time</th>
<th>Max Time</th>
<th>%MPI</th>
<th>%Wall</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allreduce</td>
<td>8</td>
<td>3278848</td>
<td>124132.547</td>
<td>0.000</td>
<td>114.920</td>
<td>59.35</td>
<td>16.88</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>0</td>
<td>35173439489</td>
<td>43439.102</td>
<td>0.000</td>
<td>41.961</td>
<td>20.77</td>
<td>5.91</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>98304</td>
<td>13221888</td>
<td>15710.953</td>
<td>0.000</td>
<td>3.586</td>
<td>7.51</td>
<td>2.14</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>196608</td>
<td>13221888</td>
<td>5331.236</td>
<td>0.000</td>
<td>5.716</td>
<td>2.55</td>
<td>0.72</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>589824</td>
<td>206848</td>
<td>5166.272</td>
<td>0.000</td>
<td>7.265</td>
<td>2.47</td>
<td>0.70</td>
</tr>
</tbody>
</table>
MPI_Barrier

<table>
<thead>
<tr>
<th>Function</th>
<th>Total calls</th>
<th>Total time (sec)</th>
<th>Total buffer size (MB)</th>
<th>Avg. Buffer Size/call (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Barrier</td>
<td>6.02e+05</td>
<td>3.48e+05</td>
<td>44.23%</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>3.18e+07</td>
<td>2.31e+05</td>
<td>29.33%</td>
<td>3.61e+05</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>1.29e+08</td>
<td>1.29e+05</td>
<td>16.36%</td>
<td>5.24e+04</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>5.73e+07</td>
<td>6.08e+04</td>
<td>7.73%</td>
<td>5.39e+04</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>1.08e+08</td>
<td>1.24e+04</td>
<td>1.58%</td>
<td>1.66e+05</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>1.29e+08</td>
<td>6.11e+03</td>
<td>0.78%</td>
<td>5.24e+04</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>1.14e+03</td>
<td>5.92e-01</td>
<td>7.52e-05%</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>6.66e+02</td>
<td>0</td>
<td>0%</td>
<td>0</td>
</tr>
</tbody>
</table>

Is MPI_Barrier time bad? Probably. Is it avoidable?

~three cases:
1) The stray / unknown / debug barrier
2) The barrier which is masking compute balance
3) Barriers used for I/O ordering

Often very easy to fix
Load Balance: Application Cartoon

Unbalanced:
Task 1
Task 2
Task 3
Task 4

Balanced:
Task 1
Task 2
Task 3
Task 4

Universal App
- Sync
- Flop
- I/O

Time saved by load balance
Load Balance: performance data

Communication Time: 64 tasks show 200s, 960 tasks show 230s
while(1) {
    do_flops(N_i);
    MPI_Alltoall();
    MPI_Allreduce();
}
Load Balance: analysis

- The 64 slow tasks (with more compute work) cause 30 seconds more “communication” in 960 tasks
- This leads to 28800 CPU*seconds (8 CPU*hours) of unproductive computing
- All load imbalance requires is one slow task and a synchronizing collective!
- Pair well problem size and concurrency.
- Parallel computers allow you to waste time faster!
Message Aggregation Improves Performance

Before

After
Ideal Scaling Behavior

• **Strong Scaling**
  • Fix the size of the problem and increase the concurrency
    • # of grid points per mpi task decreases as 1/P
    • Ideally runtime decreases as 1/P
  • Run out of parallel work

• **Weak Scaling**
  • Increase the problem size with the concurrency
    • # of grid points per mpi task remains constant
    • Ideally runtime remains constant as P increases
  • Time to solution
Scaling Behavior: MPI Functions

- Local: leave based on local logic
  - MPI_Comm_rank, MPI_Get_count
- Probably Local: try to leave w/o messaging other tasks
  - MPI_Isend/Irecv
- Partially synchronizing: leave after messaging $M<N$ tasks
  - MPI_Bcast, MPI_Reduce
- Fully synchronizing: leave after every else enters
  - MPI_Barrier, MPI_Allreduce
**Strong Scaling: Communication Bound**

64 tasks, 52% comm  
192 tasks, 66% comm  
768 tasks, 79% comm

- MPI_Allreduce buffer size is 32 bytes.

Q: What resource is being depleted here?  
A: Small message latency

1) Compute per task is decreasing  
2) Synchronization rate is increasing  
3) Surface:Volume ratio is increasing
MILC on Ranger – Runtime Shows Perfect Scalability

The graph shows the total runtime and perfect scaling for different numbers of cores (ncores). The runtime decreases as the number of cores increases, indicating perfect scalability.

- **Total Runtime** (solid line): The actual runtime of the MILC on Ranger system.
- **Perfect Scaling** (dotted line): The ideal runtime if the system scaled perfectly with the number of cores.

The x-axis represents the number of cores ranging from 10 to 10,000, and the y-axis represents the time in seconds, ranging from 1000 to 10 seconds.
MILC – Perfect Scalability due to Cancellation of Effects

![Graph showing the scalability of MILC.

- Runtime
- Communication
- Computation
- Computation - Perfect]
MILC – Superlinear Speedup Cache Effect

![Graph showing the relationship between L2 Fraction Hit and Speedup as the number of cores increases.](image-url)
**WRF – Problem Definition**

- WRF – 3D numerical weather prediction
- Explicit Rugga-Kutta solver in 2 dimensions
- Grid is spatially decomposed in X & Y
- Version 2.1.2
- 2.5 km Continental US 1501 x 1201 x 35 grid
- 9 simulated hours
- parallel I/O turned on
WRF Overall Performance

d. WRF large parallel I/O no init
8,703 s on 256 cores of DataStar
WRF-Compute Performance

d. WRF large parallel I/O no init
7,770 s on 256 cores of DataStar
WRF Communication times

d. WRF large parallel I/O no init

- Ranger
- DataStar
- Lonestar
- Franklin

Communication time (s)

Cores
WRF - MPI Breakdown
WRF – Message Sizes Decrease Slowly

- mean
- min
- max
- mean - ideal

message size / bytes

ncores

SAN DIEGO SUPERCOMPUTER CENTER

Performance Modeling and Characterization
WRF – Latency and Bandwidth Dependence
Direct Numerical Simulation (DNS)

Direct Numerical Simulation of turbulent flows
Uses pseudospectral method - 3D FFT’s
1024^3 problem – 10 timesteps
DNS – Overall Performance

a. DNS 1023^3
1,070 s on 32 cores of DataStar
DNS - Compute Performance

Comp speed/core relative to 1.5-GHz DataStar at 32 cores

- Lonestar
- Franklin
- Ranger
- DataStar

a. DNS 1023^3
958 s on 32 cores of DataStar
DNS – MPI Breakdown

a. DNS $1024^3$ on DataStar
DNS communication time

Theory: $1/P^{0.67}$
Measured: $1/P^{0.6} \times 2^{-0.71}$

Communication time (s)

Cores

a. DNS $1024^3$
Overlapping Computation and Communication

MPI_Isend()
MPI_IRecv()
some_code()
MPI_Wait()

- Basic idea – make the time in MPI_Wait goto zero
- In practice very hard to achieve
More Advance Usage: Regions

Uses MPI_Pcontrol Interface

The first argument to MPI_Pcontrol determines what action will be taken by IPM.

Arguments Description
1,"label" start code region "label"
-1,"label" exit code region "label"

Defining code regions and events:

C
MPI_Pcontrol( 1,"proc_a");
MPI_Pcontrol(-1,"proc_a");

FORTRAN
call mpi_pcontrol( 1,"proc_a"/char(0))
call mpi_pcontrol(-1,"proc_a"/char(0))
More Advanced Usage: Chip Counters –
AMD (Ranger & Kraken) Intel(Abe & Lonestar)

• Default set:
PAPI_FP_OPS
PAPI_TOT_CYC
PAPI_VEC_INS
PAPI_TOT_INS

• Alternative (setenv IPM_HPM 2)
PAPI_L1_DCM
PAPI_L1_DCA
PAPI_L2_DCM
PAPI_L2_DCA

• Default set:
PAPI_FP_OPS
PAPI_TOT_CYC

• Alternative (setenv IPM_HPM )
  2 PAPI_TOT_IIS,
     PAPI_TOT_INS
  3 PAPI_TOT_IIS,
     PAPI_TOT_INS
  4 PAPI_FML_INS,
     PAPI_FDV_INS

User defined counters also possible – setenv IPM_HPM PAPI_FP_OPS
     PAPI_TOT_CYC,…
User is responsible for choosing a valid set
See PAPI documentation and papi_avail command for more information
Matvec: Regions & Cache Misses

• What is wrong with this fortran code?

... setenv IPM_HPM 2

call mpi_pcontrol(1,"main"//char(0))
do i = 1,natom
    sum=0.0d0
    do j = 1, natom
        sum=sum+coords(i,j)*q(j)
    end do
    p(i)=sum
end do
call mpi_pcontrol(-1,"main"//char(0))
...

Regions and Cache Misses cont.

...  

<table>
<thead>
<tr>
<th></th>
<th>[total]</th>
<th>&lt;avg&gt;</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>entries</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>wallclock</td>
<td>0.0185561</td>
<td>0.0185561</td>
<td>0.0185561</td>
<td>0.0185561</td>
</tr>
<tr>
<td>user</td>
<td>0.016001</td>
<td>0.016001</td>
<td>0.016001</td>
<td>0.016001</td>
</tr>
<tr>
<td>system</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mpi</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>%comm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gflop/sec</td>
<td>0.0190196</td>
<td>0.0190196</td>
<td>0.0190196</td>
<td>0.0190196</td>
</tr>
</tbody>
</table>

|                               |            |           |         |         |
| PAPI_L1_DCM                   | 352929     | 352929    | 352929  | 352929  |
| PAPI_L1_DCA                   | 8.01278e+06 | 8.01278e+06 | 8.01278e+06 | 8.01278e+06 |
| PAPI_L2_DCM                   | 126097     | 126097    | 126097  | 126097  |
| PAPI_L2_DCA                   | 461965     | 461965    | 461965  | 461965  |

27% cache misses!
What is wrong with this fortran code?

```fortran
... 
do i = 1, natom
    sum=0.0d0
    do j = 1, natom
        sum=sum+coords(i,j)*q(j)
    end do
    p(i)=sum
end do
...

Indices transposed!
```
### Regions and Cache Misses - 4

```plaintext
# region : main       [ntasks] =      1
#
#
#                           [total]       <avg>       min       max
# entries                          1           1           1  1
# wallclock 0.00727696  0.00727696  0.00727696  0.00727696
# user 0.008 0.008 0.008 0.008
# system 0 0 0 0
# mpi 0 0 0 0
# %comm 0 0 0 0
# gflop/sec 0.000636804 0.000636804 0.000636804 0.000636804
#
# PAPI_L1_DCM                   4634          4634 4634 4634
# PAPI_L1_DCA            8.01436e+06   8.01436e+06 8.01436e+06 8.01436e+06
# PAPI_L2_DCM                  4609          4609 4609 4609
# PAPI_L2_DCA                  126108        126108 126108 126108
#
```

3.6% cache misses – Problem solved - Runtime doubled!
Using IPM on Ranger – 1 Running

- In submission script:
  - (csh syntax)
    module load ipm
    setenv LD_PRELOAD $TACC_IPM_LIB/libipm.so
    ibrun ./a.out
  - (bash syntax)
    module load ipm
    export LD_PRELOAD=$TACC_IPM_LIB/libipm.so
    ibrun ./a.out
Using IPM on Ranger – 2 Postprocessing

• Text summary should be in stdout
• IPM also generates an XML file (username.1235798913.129844.0) that can be parsed to produce webpage

    module load ipm
    ipm_parse -html tg456671.1235798913.129844.0

• This generates a directory with the html content in

    tar zxvf ipmoutput.tgz <directory> eg. a.out_2_tg456671...

    scp tar file to your local machine; untar and view with your favorite browser
Summary

• Understanding the performance characteristics of your code is essential for good performance.
• IPM is a lightweight, easy-to-use profiling interface (with very low overhead <2%).
• It can provide information on:
  • An individual jobs performance characteristics
  • Comparison between jobs
  • Workload characterization
• IPM allows you to gain a basic understanding of why your code performs the way it does.
• IPM is installed on various TG machines: Ranger, BigBen, Pople, (Abe, Kraken) see instructions on IPM website [http://ipm-hpc.sf.net](http://ipm-hpc.sf.net)