Performance Analysis and Workload Characterization with IPM

Karl Fuerlinger, David Skinner

fuerling@eecs.berkeley.edu
dskinner@nersc.gov
Outline

- Overview
- IPM philosophy and implementation overview
- Using IPM
  - Measuring load balance
  - Detecting scalability problems
  - Analyzing message sizes
- Scaling study using IPM
  - MILC on Ranger: Lattice QCD code
- Outlook
  - IPM2
Overview: Profiling and Tracing

- **Profiling:**
  - Summary statistics for regions of the code
  - Low overheads
  - Implemented by direct measurement or sampling

- **Tracing**
  - Preserves full temporal ordering of events
  - Usually higher overheads, larger amounts of data
  - Implemented by direct measurement

- **IPM**
  - IPM = Integrated Performance Monitoring (http://ipm-hpc.sourceforge.net)
  - Is a profiling tool using direct measurement
  - Emphasizes ease of use and efficiency
  - High-level characterization of the event inventory
  - Low overhead, yet useful and user-centric performance data
  - „Always on“ operation for computing center usage
Our goal is to get an event inventory of the application
- We go from events to signatures
- Discard information we are not interested in

\[ \sigma : E \rightarrow S \]
\[ \sigma(e) \ldots \text{signature of } e \]

MPI_Send(&buf, 2048, MPI_BYTE, 
33, 1, MPI_COMM_WORLD)
Components of the Event Signature

<table>
<thead>
<tr>
<th>Signature component</th>
<th>Signature function</th>
<th>Data type</th>
<th>Typical Size (#bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wallclock time</td>
<td>time(e)</td>
<td>floating point</td>
<td>32/64</td>
</tr>
<tr>
<td>Sequence number</td>
<td>seq(e)</td>
<td>integer</td>
<td>32</td>
</tr>
<tr>
<td>Type of MPI call</td>
<td>call(e)</td>
<td>integer</td>
<td>8</td>
</tr>
<tr>
<td>Data size</td>
<td>size(e)</td>
<td>integer</td>
<td>32</td>
</tr>
<tr>
<td>Data address</td>
<td>address(e)</td>
<td>integer</td>
<td>64</td>
</tr>
<tr>
<td>Own rank</td>
<td>rank(e)</td>
<td>integer</td>
<td>32</td>
</tr>
<tr>
<td>Partner rank</td>
<td>partner(e)</td>
<td>integer</td>
<td>32</td>
</tr>
<tr>
<td>Callsite ID</td>
<td>csite(e)</td>
<td>integer</td>
<td>16</td>
</tr>
<tr>
<td>Program region</td>
<td>region(e)</td>
<td>integer</td>
<td>8</td>
</tr>
</tbody>
</table>

- IPM is a profiling tool, so we are not including `time()` and `seq()`
  - Generally, many events will have the same signature (signature space is much smaller than the event space, \(|E| >> |S|\))

- For each event signature we record
  - The **number of occurrences**
  - The **statistics on the duration** (\(\text{min}, \text{max}, \text{avg}\))
The implementation in IPM

- Implementation: IPM uses a fixed-size hash table to store signatures
  - The hash **key** is the event signature (bit vector of 64-128 bits)
  - The hash **values** are the timing statistics (approx. 20 bytes)
  - The whole hash table occupies about 1-2MB and easily fits in memory
Analyzing Event Signatures

- The hash table of event signatures contains a lot of interesting data.
Outline

- Overview
- IPM philosophy and implementation overview
  - Using IPM
    - Measuring load balance
    - Detecting scalability problems
    - Analyzing message sizes
  - Scaling study using IPM
    - MILC on Ranger: Lattice QCD code
- Outlook
  - IPM2
IPM – Integrated Performance Monitoring

- IPM provides a performance profile of a job
  - “Flip of a switch” operation

```
Input_123
  ▼
   ▼
  Job_123  IPM
         ▼
Output_123  Profile_123
```
Using IPM: Basics

- Do “module load ipm”, then run normally
- Upon completion you get

```bash
###IPMv0.85
# command : ../exe/pmemd -O -c inpcrd -o res (completed)
# host    : s05405
# start   : 02/22/05/10:03:55
# stop    : 02/22/05/10:04:17
# mpi_tasks : 64 on 4 nodes
# gbytes   : 2.57604e+00
# gbytes   : total
# wallclock : 24.278400 sec
# gbytes   : 2.04615e+00
# gbytes   : total
# %comm    : 32
# %comm    : 43
# gbytes   : 2.57604e+00
# gbytes   : total
# gbytes   : 2.04615e+00
# gbytes   : total

Maybe that’s enough. If so you’re done.
Have a nice day.
```
Want more detail? IPMREPORT=full

```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
#
# wallclock [total] <avg>      min       max               
# user     936.95      14.6398         12.68          20.3
# system   227.7       3.55781         1.51             5
# mpi      503.853     7.8727         4.2293        9.13725
# %comm    32.4268     17.42           41.407
# gflop/sec 2.04614    0.0319709       0.02724       0.04041
# gbytes   2.57604     0.0402507       0.0399284     0.0408173
# gbytes_tx 0.665125   0.0103926       1.09673e-05    0.0368981
# gbyte_rx  0.659763   0.0103088       9.83477e-07    0.0417372
```
Want more detail? IPM_REPORT=full

| # | PM_CYC       | 3.00519e+11 | 4.69561e+09 | 4.50223e+09 | 5.83342e+09 |
|   | PM_FPU0_CMPL | 2.45263e+10 | 3.83223e+08 | 3.3396e+08  | 5.12702e+08 |
|   | PM_FPU1_CMPL | 1.48426e+10 | 2.31916e+08 | 1.90704e+08 | 2.8053e+08  |
|   | PM_FPU_FMA   | 1.03083e+10 | 1.61067e+08 | 1.36815e+08 | 1.96841e+08 |
|   | PM_INST_CMPL | 3.33597e+11 | 5.21245e+09 | 4.33725e+09 | 6.44214e+09 |
|   | PM_LD_CMPL   | 1.03239e+11 | 1.61311e+09 | 1.29033e+09 | 1.84128e+09 |
|   | PM_ST_CMPL   | 7.19365e+10 | 1.12401e+09 | 8.77684e+08 | 1.29017e+09 |
|   | PM_TLB_MISS  | 1.67892e+08 | 2.62332e+06 | 1.16104e+06 | 2.36664e+07 |
| #  | [time] [calls] | <%mpi> | <%wall> |
|   | MPI_Bcast    | 352.365 | 2816 | 69.93 | 22.68 |
|   | MPI_Waitany  | 81.0002 | 185729 | 16.08 | 5.21 |
|   | MPI_Allreduce| 38.6718 | 5184 | 7.68 | 2.49 |
|   | MPI_Allgatherv| 14.7468 | 448 | 2.93 | 0.95 |
|   | MPI_Isend    | 12.9071 | 185729 | 2.56 | 0.83 |
|   | MPI_Gatherv  | 2.06443 | 128 | 0.41 | 0.13 |
|   | MPI_Irecv    | 1.349 | 185729 | 0.27 | 0.09 |
|   | MPI_Waitall  | 0.606749 | 8064 | 0.12 | 0.04 |
|   | MPI_Gather   | 0.0942596 | 192 | 0.02 | 0.01 |
Want More? – You’ll need a web browser
Performance Analysis and Workload Characterization with IPM | 14
IPM: XML log files

- There’s a lot more information in the logfile than you get to stdout. A logfile is written that has the hash table, switch traffic, memory usage, executable information, ...

- Parallelism in writing of the log (when possible)

- The IPM logs are durable performance profiles
Message Sizes: CAM 336 way

Per MPI call

Per MPI call & buffer size
Scalability: Required

32K tasks AMR code

Performance Analysis and Workload Characterization with IPM | 17
More than a pretty picture

![Graph showing performance analysis and workload characterization with IPM]
Application Assessment with IPM

- Provide high level performance numbers with small overhead
  - To get an initial read on application runtimes
  - For allocation/reporting
  - To check the performance weather on systems with high variability

- What’s going on overall in my code?
  - How much computation, communication, I/O?
  - Where to start with optimization?

- How is my load balance?
  - Domain decomposition vs. concurrency (M work on N tasks)
Which problems should be tackled with IPM?

- **Performance Bottleneck Identification**
  - Does the profile show what I expect it to?
  - Why is my code running 20% slower than I expected?

- **Understanding Scaling**
  - Why does my code scale as it does?

- **Optimizing MPI Performance**
  - Combining Messages
When to reach for another tool

- Full application tracing
- Looking for hotspots on the statement level in code
- Want to step through the code
- Data structure level detail
- Automated performance feedback
What’s wrong here?

Communication

% of MPI Time

---

Communication Event Statistics (100.00% detail)

<table>
<thead>
<tr>
<th>Event</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>

Karl Fuerlinger

Performance Analysis and Workload Characterization with IPM | 22
Is MPI_Barrier time bad? Probably. Is it avoidable?

- The stray / unknown / debug barrier
- Barriers used for I/O ordering
- Barrier masking compute imbalance
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

MPI ranks sorted by total communication time
Load Balance: code

```c
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

Karl Fuerlinger

Performance Analysis and Workload Characterization with IPM | 28
**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
Outline

- Overview
- IPM philosophy and implementation overview
- Using IPM
  - Measuring load balance
  - Detecting scalability problems
  - Analyzing message sizes
- Scaling study using IPM
  - MILC on Ranger: Lattice QCD code
- Outlook
  - IPM2
MILC

- QCD code
- Calculate the masses and other properties of quarks and gluons
- 4D lattice with 32 points in each direction

Strong scaling study on TACC Ranger
- Quad-socket, quad-core Opteron based
- Infiniband network
MILC on Ranger – Runtime Shows Perfect Scalability

![Graph showing perfect scalability](image)

- **Total Runtime**
- **Perfect Scaling**

**Axes:**
- **x-axis (ncores):** 10 to 10,000
- **y-axis (time/s):** 10 to 1000
MILC – Perfect Scalability due to Cancellation of Effects

![Graph showing the scalability of MILC](image-url)
MILC – Superlinear Speedup Cache Effect

![Graph showing L2 Fraction Hit and Speedup vs. number of cores. The graph indicates a superlinear speedup with increasing L2 Fraction Hit as the number of cores increases.]
MILC Communication

Performance Analysis and Workload Characterization with IPM | 35
Outline

- Overview
- IPM philosophy and implementation overview
- Using IPM
  - Measuring load balance
  - Detecting scalability problems
  - Analyzing message sizes
- Scaling study using IPM
  - MILC on Ranger: Lattice QCD code

- Outlook
  - IPM2
Under development: IPM2

- With support for
  - MPI
  - OpenMP
  - POSIX-I/O
  - Pthreads
  - ...

- Event flow graphs:
  - Extension of the signature approach: include the last signature in the current signature
    \[ \sigma'(e_0) = (s^0_i, \sigma(e_0)) \]
    \[ \sigma'(e_i) = (\sigma(e_{i-1}), \sigma(e_i)) \quad (i > 0) \]
  - Retains some information about temporal order, blend between profiling and tracing
  - Advanced app characterization: All transitions between comp, comm, i/o recorded
Event Flow Graphs - Example

```c
void main(int argc, char* argv[]) {
    MPI_Init(...);
    MPI_Comm_size(...);
    MPI_Comm_rank(..., &myrank);

    for(i=0; i<10; i++) {
        if(myrank is odd)
            MPI_Send(10 doubles to rank -1);
        else
            MPI_Recv(10 doubles from rank +1);
    }
    MPI_Finalize();
}
```

Event flow graph of an execution with 4 processes:

- 10 iterations
- Message size 80 bytes
- Send to „left“ neighbor
- Receive from „right“ neighbor
Event Flow Graphs – Example

Graphs can get very complex, some method for interactive exploration is needed
Event Flow Graphs – Interactive Exploration

More Information: See our PROPER 2009 (Euro-Par Workshop) paper on this.
IPM: Summary

- Provide high-level performance profile
  - *Event inventory*: which events and how much time did they take
  - How much time in communication operations
  - Less focus on drill-down into application than other tools

- Efficiency
  - Fixed memory footprint (approx. 1-2 MB per MPI rank)
  - Monitorig data is kept in a hash-table, avoid dynamic memory allocation
  - Low CPU overhead: 1-2 %

- Ease of use
  - HTML, or ASCII-based output format
  - Flip of a switch, no recompilation, no user instrumentation
  - Portability

- [http://ipm-hpc.sourceforge.net](http://ipm-hpc.sourceforge.net)

Thank you for your attention!