MPI Performance in a Production Environment: Toward Automatic Application Profiling

LLNL
October 26, 2004
This talk focuses on profiling and optimization of parallel scientific codes in a production environment. Most of the work presented is from data gathered on seaborg, the IBM SP at NERSC, using tools designed to assist users and HPC managers in low overhead collection of performance profile information. Comparisons of different MPI libraries are presented as is initial work done to characterize the diverse scientific workload currently running at NERSC.
Background on NERSC@LBL
Introduction to NERSC@LBL

Facility provides:
- Identifies existing problems
- Access to real user applications

Research provides:
- New tools and solutions
- Rapid introduction of new technology
National Energy Research Scientific Computing Center

- Serves all disciplines of the DOE Office of Science
- ~2000 Users in ~400 projects

- Focus on large-scale computing
NERSC Usage by Scientific Discipline, FY02

NERSC is DOE’s flagship center for non-classified capability computing, providing reliable computing infrastructure, HPC consultancy, accurate resource accounting.
NERSC has many Customers and an Extremely Diverse Workload

One of ~ 400 Research Projects
Focus on Large Scale Computing
Profiling of Parallel Codes

What do we mean by profile?

**Informative Summary**

There is such a thing as too much information.
Motivation for a Profiling Infrastructure

Users, center managers and HPC consultants (me) all want codes to perform well. At NERSC this is a challenge given our diverse workload.

• Common methodology
  – Pick a code, read the source code
  – Instrument, Run, Analyze, Summarize, Modify, Repeat

Why not provide an infrastructure that precomputes the summary and shares it with the user at runtime?

• Provide a synthesis of the data collected from the infrastructure to characterize our workload.
  – HPC Center Managers $\rightarrow$ build better center
  – Researchers and HPC vendors $\rightarrow$ build better computers
  – Identify low hanging fruit, recurrent/glaring problems
Where can profiling happen?

- **Code Developer**
  - Heavy weight tools (TV, TAU, VAMPIR)
  - Detailed, Invasive to code, may require recompilation
  - Difficult to compare users’ metrics across projects.

- **Center Managers**
  - Transparent tools (poe+, ja, hpm)
  - Uniformity, Ease of Use, Limited Scope
  - Possible to centrally stored records

- **HW/OS Vendor**
  - Custom ASICs to record performance system wide
  - Center has limited control
Profiling Tools

- Many tools exist, roughly they vary by

<table>
<thead>
<tr>
<th>Type of Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level of Detail</td>
</tr>
<tr>
<td>Runtime Impact on Code</td>
</tr>
<tr>
<td>Scalability</td>
</tr>
<tr>
<td>Ease of Use</td>
</tr>
</tbody>
</table>

- NERSC needs an easy to use (i.e., almost transparent) low impact profiling tool that provides high level summaries about job performance.
poe+

• From Feb 2002 to Sep 2004 NERSC provided poe+ to users to meet this need
  – poe+ is glue between hpmcount, PMPI, dump and truss, that parses and aggregates reports into a single job performance record to stdout and to a central log.
  – Central log feeds to a web accessible database
  – Collected > 46,000 job profiles from > 300 users.
  – Widely used for ERCAP
  – Very popular with users and USG staff

Now will look at case studies from poe+ and then on to new ideas.
What poe+ provides:

• An easy to use low overhead (to user and code) interface to performance metrics.
  – Uses hpmcount to gather and aggregate HPM data.
  – Can generate an MPI Profile via PMPI interface
  – Load balance information

• Clear, concise performance reports to user and to NERSC center
  – Reports go to stdout and to www.nersc.gov

• There are other options PMAPI / PAPI / HPMLIB

<table>
<thead>
<tr>
<th>ERCAP GFLOP/S</th>
<th>: 502.865839 GFLOP/S</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERCAP MB/TASK</td>
<td>: 96.01953125 MB</td>
</tr>
</tbody>
</table>
How are we using MPI?

- Identifying the time spent in MPI based on MPI call, buffer size, etc. for HPC workloads is in itself a worthwhile activity. A sort of census of parallelism that can be fed in multiple directions.

- At NERSC we have also poe+ in a first attempt at characterizing our workloads

- Most direct interface to this data is via the web
poe+ : Usage

Usage: poe+ [-hpm_group n] [-mpi] executable

- `-hpm_group` selects HPM group
  - Default group 1 for flops and TLB
  - Group 2 for L1 cache load/store misses

- `-mpi` maps MPI* calls to PMPI*
  - MPI calls get wrapped to records data movement and timings
  - ~1.5 microsecond overhead to each MPI call
  - When MPI_Finalize is reached
    - Application level summary
    - Task level summary
    - Load balance histogram
Case Studies

Names have been changed to protect the innocent (and the guilty)
Colony switch fabric in two planes:
400 MB/sec 19 usec latency

380 compute nodes allowing 6080 tasks,
In production as of last month.

12–64 GB Memory

16 CPUs

Lawrence Berkeley National Laboratory
6080 dedicated CPUs, 96 shared login CPUs
Hierarchy of caching, speeds not balanced
Bottleneck determined by first depleted resource
Most common use of poe+: Load imbalance

• How can I quickly determine load balance w/o recompiling or perturbing an existing code?

> poe+ my_code --nodes 8 --tasks_per_node 16

hpmcount (V 2.5.3) summary

Execution time (wall clock time): 133.128812 seconds on 64 tasks

<table>
<thead>
<tr>
<th>Resource Usage Statistics</th>
<th>Average</th>
<th>Total</th>
<th>MIN</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wall Clock Time (in sec.)</td>
<td>132.465758</td>
<td>8477.808501</td>
<td>130.002884</td>
<td>133.128812 s</td>
</tr>
<tr>
<td>Time in user mode (in sec.)</td>
<td>116.304219</td>
<td>7443.470000</td>
<td>107.020000</td>
<td>117.990000 s</td>
</tr>
<tr>
<td>Time in system mode (in sec.)</td>
<td>2.216562</td>
<td>141.860000</td>
<td>1.000000</td>
<td>4.990000 s</td>
</tr>
<tr>
<td>Maximum resident set size</td>
<td>98324</td>
<td>6292764</td>
<td>97952</td>
<td>98996 KB</td>
</tr>
<tr>
<td>Shared mem use in text seg.</td>
<td>37889</td>
<td>2424926</td>
<td>35043</td>
<td>38309 KB*s</td>
</tr>
<tr>
<td>Unshared mem use in data seg.</td>
<td>11265782</td>
<td>721010109</td>
<td>10498632</td>
<td>11365248 KB*s</td>
</tr>
<tr>
<td>Page faults w/out IO activity</td>
<td>26440</td>
<td>1692189</td>
<td>26320</td>
<td>27002</td>
</tr>
<tr>
<td>Page faults with IO activity</td>
<td>14</td>
<td>942</td>
<td>8</td>
<td>37</td>
</tr>
<tr>
<td>Times process was swapped out</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Times file system perf. INPUT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Times file system perf. OUTPUT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IPC messages sent</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IPC messages received</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>signals delivered</td>
<td>315</td>
<td>20196</td>
<td>314</td>
<td>317</td>
</tr>
<tr>
<td>voluntary context switches</td>
<td>2530</td>
<td>161961</td>
<td>594</td>
<td>7705</td>
</tr>
</tbody>
</table>
Case Study: Load Balance in real apps

In this case poe+ reports good load balance
Case Study: Load Balance in real apps

In this case poe+ shows load balance problems (though not in detail)
Case Study: detecting load imbalance

Unbalanced:

Balanced:

Time saved by load balance
Load Balance: Summary

- Imbalance most often a byproduct of data decomposition
  - Must be addressed before further MPI tuning can happen
- How to quickly identify and quantify imbalance?
  - poe+ provides a simple quantitative means
- Good software exists for graph partitioning / remeshing

- For regular grids consider padding or contracting
load balance via poe+

2) poe+ -mpi

Computation

Communication
Case Study: Communication Time

Is MPI_Barrier time bad? Is it avoidable?

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

Often very easy to fix
Case Study: MPI_Wait

A 1920 way QCD application shows the following poe+ MPI profiling

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>#calls</th>
<th>avg. bytes</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_size</td>
<td>184632</td>
<td>0.0</td>
<td>0.039</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>2687528</td>
<td>0.0</td>
<td>0.561</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>214694</td>
<td>10691.1</td>
<td>6.503</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>214694</td>
<td>10691.1</td>
<td>1.665</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>29388</td>
<td>0.0</td>
<td>67.234</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>3</td>
<td>1268.0</td>
<td>0.003</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>2</td>
<td>0.0</td>
<td>0.024</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>3901</td>
<td>31.4</td>
<td>21.345</td>
</tr>
</tbody>
</table>

Common problem: How to analyze time spent in MPI Wait?
- Can determine ranks you were waiting on from MPI_Status
- Can determine buffer sizes similarly
- This came late to poe+ functionality (see IPM later in this talk)
Wait Topology via MPI_Status

#ifdef LINUX_X86
#define IPM_MPI_RANK_STATUS_C irank=status->MPI_SOURCE
#endif
#ifdef AIX_POWER
#define IPM_MPI_RANK_STATUS_C irank=status->source
#endif

#ifdef LINUX_X86
#define MPI_BYTES_STATUS_C bytes = status->count;
#endif
#ifdef AIX_POWER
#define BYTES_STATUS_C bytes = status->val1;
#endif
Visualizing Wait Topologies

```
rep MPI_Wait cg32res.txt u 1:3:(0.5*$6):((8-$7)/$6)
rep MPI_Send cg32res.txt u 1:3:(0.5*$6):((8-$7)/$6)
```
Visualizing Wait Topologies

```
rep MPT_Wait cg32res.txt u 1:3:(0.5*$6):((8-$7)/$6)
```
Case Study: System Health

Having frequent application performance snapshots is good for the center. It tells you when the machine is sick.

- Performance variation reported by users running with > 1024 tasks
- USG/NSG/IBM identified and resolved slight asymmetries in how the CWS polls batch nodes about their health.
- Significant benefit for highly parallel applications
- Process driven by feedback from users about performance variability.
Scaling of MPI_Barrier()
A (very tricky) case study

- poe+ data from a code varies from day to day and week to week by factors of 2-3 (sound familiar?)

- Having good days and bad days points to something outside the application itself.

- Since we do not share nodes, the resource being contended for must be the switch.

- Need finer grained detail to catch contention “in the act”.
Case Study: Cross Application Contention

• Performance of communications shows wide range of performance variability that increase with concurrency

• Why?
Case Study: Cross Application Contention

- Regularity points to identifiable cause
- Detective work like this is laborious, requires both admin/app skills and resources
Case Study: Cross Application Contention

sar read data for GPFS on seaborg

Read intensity (char/sec)
black < red < white
Case Study: Cross Application Contention

Some investigation into flow control tokens, little progress.

Quite possibly an architectural fact of life

Different networks for Disk and MPI (BG/L) is an appealing idea.
Ongoing Work

Poe+ has become widely adopted by users and USG staff alike. It has several shortcoming we’d like to correct for the FY05 Allocation year (Dec1).
Across many research projects:

Warning: Sampling here is biased

Show jobs that ran after
and ran on or before
Current data ends at 11:30 p.m. yesterday.

Submit Query  Reset to Defaults

Summary for 1163 non-interactive jobs, avg. size: 34.73 tasks, avg. MPI pct: 31.1%

<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_Bcast</td>
<td>4.06e+11</td>
<td>2.85e+07</td>
<td>29.41%</td>
<td>1.10e+08</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>1.27e+10</td>
<td>2.44e+07</td>
<td>25.20%</td>
<td>3.24e+07</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>5.05e+10</td>
<td>1.17e+07</td>
<td>12.11%</td>
<td>2.99e+05</td>
</tr>
<tr>
<td>MPI_Allgatherv</td>
<td>1.61e+10</td>
<td>9.22e+06</td>
<td>9.51%</td>
<td>4.54e+06</td>
</tr>
<tr>
<td>MPI_Alltoall</td>
<td>3.68e+07</td>
<td>5.17e+06</td>
<td>5.33%</td>
<td>1.98e+04</td>
</tr>
<tr>
<td>MPI_Probe</td>
<td>6.17e+05</td>
<td>3.14e+06</td>
<td>3.24%</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>8.82e+09</td>
<td>3.08e+06</td>
<td>3.17%</td>
<td>2.61e+07</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>1.79e+08</td>
<td>3.03e+06</td>
<td>3.13%</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Sendrecv</td>
<td>5.29e+09</td>
<td>2.60e+06</td>
<td>2.68%</td>
<td>1.43e+07</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>3.45e+10</td>
<td>1.98e+06</td>
<td>2.05%</td>
<td>6.03e+08</td>
</tr>
<tr>
<td>MPI_Allgather</td>
<td>1.01e+10</td>
<td>1.89e+06</td>
<td>1.95%</td>
<td>1.07e+05</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>2.32e+08</td>
<td>1.07e+06</td>
<td>1.11%</td>
<td>1.45e+05</td>
</tr>
</tbody>
</table>

Percent of MPI Time
poe+ → IPM

• poe+ has been quite useful, but it is not well integrated and is not portable
  – Parsing output from disparate places is a bad idea, things break
  – Poe+ is not good for uniformly sampling workloads

• IPM (Integrate Performance Monitoring) is the follow on to poe+. It aims to solve many of the shortcomings of poe+ and provide more user requested features, log information, and portability.

• IPM optionally allows scalable tracing in addition to profiling.
Toward an Automated Profiling Infrastructure

• Some users care about performance profiling, many do not.
  – Time spent on code is time spent away from obtaining publishable results

• HPC centers care (or are made to care)
  – Evaluating match and conflicts between computers and codes is possible only through
    • Generalities / Folk Wisdom
    • Specifics that users tell us
    • Specifics that we can measure quantitatively
  – Center performance often tied to code+computer performance

Is it possible to provide a solution that works for everyone?
– must be very easy to use, but not get in the way
IPM : Integrated Performance Monitoring

- Platform independent generalization of poe+
  - PAPI + PMPI + lots of tricks
- Includes timestamped tracing as an option
- More detailed and compact (memory) profile of communication timing and topology

```
# Wall = 58.766343 MPI = 10.784593 ( 18.351649 % )
# Wall = 58.766343 MPI = 10.784593 ( 18.351649 % )
rank call            flow      count       size     time     tmin     tmax     %mpi
  25 MPI_Recv          17      25000        520     2.79e+00 6.44e-06 1.24e-02  24.2
  25 MPI_Recv          26      25000        960     2.31e+00 1.04e-05 6.62e-02  20.0
  25 MPI_Wait          17        252     106080  1.99e+00 2.58e-03 8.46e-02  17.3
  25 MPI_Send          17        252     106080  1.54e+00 5.69e-04 7.69e-02  13.4
  25 MPI_Recv          24      25000        960     1.23e+00 9.89e-06 4.69e-02  10.6
  25 MPI_Send          24      25000        960     3.52e-01 1.28e-05 5.04e-05   3.1
  25 MPI_Send          17      25000        520     3.40e-01 1.16e-05 6.28e-04   2.9
  25 MPI_Send          26      252     204000  2.45e-01 7.92e-04 2.98e-03   2.1
  25 MPI_Send          24      252     204000  2.33e-01 7.23e-04 2.36e-03   2.0
  25 MPI_Allreduce      0          4         40      2.82e-02 7.39e-04 1.23e-02   0.2
  25 MPI_Irecv          0        504     204000  1.31e-02 9.27e-06 9.11e-05   0.1
```
IPM: tricks

• By using a hash based approach instead of histogramming, less information is lost in the profile. Smaller footprint as well.

• With help from Bill Tuel we are able to avoid the relinking step in poe+ by doing profiling in libmpi_r.a directly.

• With a machine latency of 19-23 usec we can in many cases afford ~1usec for call timers and a hash table insert. 2% performance hit on NAMD.

• System logs written using DB friendly markup to allow changes and inter-architecture comparisons.
IPM: treats

• Very simple on the SP. Just load a module (which sets MP_EUILIBPATH) and you’re done.

• Interface is through environment variables and MPI_pcontrol. E.g., set “IPM_MPI_FULLTRACE=MPI_Alltoall” to get timestamped traces of MPI_Alltoall

• Default report is very concise, more details stored to system log.

• Handles the case where you don’t reach MPI_Finalize
Default report is concise. Many poe+ users said it produced too much output. System logs are detailed, allows retrospection.
More detail to stdout if you want it.

```plaintext
# IPMv0.8 :: lego 16 tasks 1 hosts i686/Linux 10/25/04/23:00:32
# ../xapp_simple (completed)
#
#       <mpi>      <user>      <wall> (sec)        [gflop/s]        [gbyte]
#        0.04        0.01        0.79              4.661e-04      2.714e-01
#
# counter                 [total] <average> min max
# PAPI_FP_OPS    6.315e+06
# PAPI_TOT_CYC   7.767e+07
#
# rank call            flow      count       size  [time]   tmin tmax  %mpi
#    0 MPI_Sendrecv 15          1       1016 5.88e-02 5.88e-02 5.88e-02 70.0
#    0 MPI_Barrier   *          2          * 2.25e-02 4.64e-04 2.20e-02 26.8
#    0 MPI_Reduce   0          2       1016 1.48e-03 3.88e-04 1.09e-03 1.8
#    0 MPI_Allreduce 0          1       1016 7.06e-04 7.06e-04 7.06e-04 0.8
#    0 MPI_Reduce 0          1       1008 4.24e-04 4.24e-04 4.24e-04 0.5
#    0 MPI_Sendrecv 1          2       1016 1.93e-05 5.61e-06 1.37e-05 0.0
#    0 MPI_Bcast 0          1          4 1.93e-05 1.93e-05 1.93e-05 0.0
#    0 MPI_Bcast 0          1       1016 1.07e-05 1.07e-05 1.07e-05 0.0
#    0 MPI_Reduce 1          2       1016 9.42e-06 4.43e-06 4.98e-06 0.0
#    1 MPI_Sendrecv 2          2       1016 5.38e-02 6.76e-04 5.31e-02 94.3
#    1 MPI_Reduce 1          2       1016 1.84e-03 8.54e-04 9.89e-04 3.2
#    1 MPI_Barrier   *          2          * 9.70e-04 8.24e-05 8.87e-04 1.7
#    1 MPI_Reduce 1          1       1008 3.41e-04 3.41e-04 3.41e-04 0.6
#    1 MPI_Allreduce 0          1       1016 2.02e-05 2.02e-05 2.02e-05 0.0
```
# IPMv0.7 :: s08105 16 tasks 1 hosts 006030894C00/AIX 10/04/04/13:30:15
# ./madbench.x 5000 16 10 4 128 4 (completed)
#
#       <mpi>      <user>      <wall> (sec)        [gflop/s]        [gbyte]
#       76.19      389.82      425.08              5.818e-01      3.331e+00
#

# invD
#       <mpi>      <user>      <wall> (sec)        [gflop/s]       [nexits]
#        5.94       13.50       15.18              4.922e-01              18
#
# call            [time]      %mpi %wall
# MPI_Reduce 3.640e+01   38.3   15.0
# MPI_Recv 2.722e+01   28.6   11.2
# MPI_Bcast 2.285e+01   24.0    9.4
# MPI_Send 8.343e+00    8.8    3.4
# MPI_Testall 1.279e-01    0.1    0.1
# MPI_Isend 6.049e-02    0.1    0.0
# MPI_Allreduce 3.611e-02    0.0    0.0

# W
#       <mpi>      <user>      <wall> (sec)        [gflop/s]       [nexits]
#       24.22      288.69      305.39              7.642e-01              17
#
# call            [time]      %mpi %wall
# MPI_Recv 2.217e+02   57.2    4.5
IPM ongoing work

• IPM has been ported to AIX, Linux, Altix, SX6 & ES, and X1. Not all features work (or are relevant) on all platforms. Currently in use by early adopter users and LBL CRD staff.

• Feedback on directions is valuable at this point since IPM is still somewhat maleable.

• Interest in collaborations and or support to further the project (BG/L consortium and SBIR discussions so far)
Easy to Use, Low Overhead Performance Profiling Benefits Everyone

User Applications:
• Scalar performance (HPM)
• Parallel efficiency (MPI)
• Disk I/O performance (TBA)

Center Policies:
• Runtime settings
• Queues
• SW Versioning

Compute Resources:
• System settings
• MPI Implementations
• Future Machines.

Understanding workload → Getting more science done!
Parallel Environment 4.1

- New MPI library is available, installed on seaborg in October.
- Not based on MPCI PIPES layer but rather over LAPI. Solves PIPES memory issues.
- Latency is currently higher that PE3.2, IBM is working on this
- Several improvements to MPI Collectives
- Though LAPI uses threads, your code need not
  - A pass through library for non “_r” is provided
  - May move to using “_r” all the time
PE 4.1: preliminary performance data

dev2: PE 3.2 (16.43 usec) and 4.1 (21.15 usec)

Graph showing unidirectional bandwidth (MB/sec) versus message size (bytes) for PE 3.2 and PE 4.1.
Average time for 1024 consecutive MPI_Allreduce()

Time (seconds)

# MPI Tasks

PE 3.2
PE 4.1
Hi David,
I finally got some jobs to run through Seaborg in the last three days, and I had put the "module load USG pe" in my run script. I do see a big difference in the time for global sums. It is currently taking around .002 seconds for a global sum on 1200 processors, where it used to take a minimum of .003 seconds and often much longer. This translates into about a **15% improvement in the overall speed** of our code on this size problem. Thanks for the tip!

User123

- One INCITE code (MPI_Alltoall heavy) reports 10% improvement in wall time using PE4.1 betas

- Latency sensitive apps do not like increase, but it is modest in a relative sense