USING INTEL® CLUSTER TOOLS TO OPTIMIZE MPI PROGRAMS
Agenda

Scaling with MPI Performance Snapshot

Tuning MPI Performance with Intel® Trace Analyzer and Collector
MPI SCALING ANALYSIS CHALLENGES
MPI Scaling Analysis Challenges
To Exascale... and Beyond

Application Growth

Cluster Growth

Problem Growth

Application/Tool Performance

Application Size

Application
Analysis Tools
Scalable Tools
MPI PERFORMANCE SNAPSHOT (MPS)
Why MPI Performance Snapshot (MPS)?

- **Advantages**
  - Get an initial profile of the application very quickly
  - Performance variation at scale can be detected and triaged quickly
  - Provides development recommendations to developers based on analysis
    - Intel® Trace Analyzer and Collector or Intel® VTune™ Amplifier XE for deeper analysis
  - Easy to use out of the box functionality

- **Benefits**
  - Difficult performance issues are easier to spot
  - Application performance guidance is obtained easily
  - Experienced & non-experienced developers can adopt quickly
2 MPI Performance Snapshots

- **Bundled version:**
  - is a part Parallel Studio Cluster Edition
  - relies on Intel VTune Amplifier
  - Intel MPI supports MPS (Hydra knows some environment variables)

- **Standalone version:**
  - Can be downloaded with no charge.
  - Contains Application Performance Snapshot (APS) to collect Hardware counters
  - Has got a launcher script
What’s new

Absolutely new collector which can collect information from all MPI functions.

Collector can work with any MPICH-based MPI implementation (OpenMPI is not supported so far)

Only one library to collect statistics and all other metrics (except Gflops, CPI, Memory bound)

Collector uses only MPI standard calls to support compatibility

New mechanism of MPI imbalance collection based on MPI_T_ mechanism – iMPI only

rdpmc timer for really low intrusion

Binary file format for MPI statistics

Each file writes to its own file

Different levels of statistics (MPS_STAT_LEVEL=1...5) 5 is default now.

New parser for binary statistics. (Still supports native iMPI statistics)

Absolutely new HTML report. All metrics on one page.

HTML report now shows top 5 MPI functions
What’s not supported

MPS doesn’t print statistics on finalization

PAPI library – we cannot rely on this library

MPI_Pcontrol() is not supported yet.

No MPI imbalance in other MPI implementations (might be added later)

OpenMP imbalance can be caught from Intel OpenMP library only.

MPI functions should not be called from OpenMP regions.

VTune Amplifier requires RedHat 6 and later and `perf` utility supported by kernel
MPS USAGE
How to run

“Bundle”

$ mpirun -mps -n N app_name

Standalone

$ source mpsvars.sh --vtune
$ mpirun -n N mpsrun.sh app_name

`mpsrun.sh` is a script which sets needed environment variables including LD_PRELOAD.

In VTune mode it runs `amplxe-cl` for each process
MPS Output

Summary

Files and folders:

- stats.txt — MPI statistics
- app_stat.txt — MPS collector statistics
- _mps/results.<node>/ — VTune results

--- GENERAL STATISTICS ---
Total time: 448.391 sec (All ranks)
  MPI: 40.73%
  NON_MPI: 59.27%

WallClock :
  MIN: 89.594 sec (rank 1)
  MAX: 89.975 sec (rank 4)

--- MEMORY USAGE STATISTICS ---
All ranks: 226.969 MB
  MIN: 24.172 MB (rank 2)
  MAX: 96.465 MB (rank 0)

--- MPI IMBALANCE STATISTICS ---
MPI Imbalance: 31.798 sec
  MIN: 2.219 sec
  MAX: 9.157 sec
  7.092% (All ranks)
  2.467% (rank 4)
  10.219% (rank 0)
**HTML Reporting**

**MPI Performance Snapshot Summary**

WallClock time: 21.78 sec

- **MPI Time**: 17.28 sec, 80.10%
- **MPI Imbalance**: 7.44 sec, 34.47%
- **Computation Time**: 4.29 sec, 19.90%
  - **OpenMP Time**: 19.46 sec, 90.19%
  - **OpenMP Imbalance**: 17.05 sec, 79.03%
- **Serial Time**: 0.00 sec, 0.00%

Total application lifetime. The time is elapsed time for the slowest process. This metric includes the MPI Time and the Computation time below.

- **MPI Time**: 17.28 sec
  - Time spent inside the MPI library. High values are usually bad.
    - This value is **HIGH**. The application is **Communication-bound**. More details...
- **MPI Imbalance**: 7.44 sec
  - Mean unproductive wait time per process spent in the MPI library calls when a process is waiting for data. This time is part of the MPI time above. High values are usually bad.
    - This value is **HIGH**. The application workload is **NOT well balanced** between MPI ranks. More details...
- **Computation Time**: 4.29 sec
  - Mean time per process spent in the application code. This is the sum of the OpenMP Time and the Serial time. High values are usually good.
    - This value is **LOW**.
- **OpenMP Time**: 19.46 sec
  - Mean time per process spent in the OpenMP parallel regions. High values are usually good and indicate that the application is well-threaded.
    - This value is **HIGH**.
- **OpenMP Imbalance**: 17.05 sec
  - Mean unproductive wait time per process spent in OpenMP parallel regions (normaly at synchronization barriers). High values are usually bad.
    - This value is **HIGH**. The application’s OpenMP work sharing is **NOT well load-balanced**. More details...
- **Serial Time**: 0.00 sec
  - Mean application time per process spent outside OpenMP parallel regions. High values may be good or bad depending on the application algorithm.
    - This value is **NEGLIGIBLE**. The application is well **parallelized** via OpenMP directives.

Application: build/heart_demo

Number of ranks: 17

Used statistics: app_stat_20160310-035458.txt, stats_20160310-035458.txt

Creation date: 2016-03-15 03:55:21
MPS HTML Report Breakdown – MPI Time

MPI Time – Time spent in MPI calls

If MPI Time or MPI Imbalance are high, use Intel® Trace Analyzer and Collector to investigate and optimize MPI usage

MPI Imbalance – MPI time spent waiting

Lower is better

- **MPI Time:** 17.28 sec  
  Time spent inside the MPI library. High values are usually bad.  
  This value is HIGH. The application is Communication-bound. More details...

- **MPI Imbalance:** 7.44 sec  
  Mean unproductive wait time per process spent in the MPI library calls when a process is waiting for data. This time is part of the MPI time above. High values are usually bad.  
  This value is HIGH. The application workload is NOT well balanced between MPI ranks. More details...
MPS HTML Report Breakdown – OpenMP Time

OpenMP Time – Computation time spent in OpenMP parallel regions – higher is better

OpenMP Imbalance – OpenMP Time spent waiting – lower is better

If OpenMP Imbalance is high – recommend using Intel® VTune™ Amplifier XE

If OpenMP Time is low – Intel® Advisor to find opportunities to add more threading

<table>
<thead>
<tr>
<th>OpenMP Time: 19.46 sec</th>
<th>90.19%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean time per process spent in the OpenMP parallel regions. High values are usually good and indicate that the application is well-threaded. This value is HIGH.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OpenMP Imbalance: 17.05 sec</th>
<th>79.03%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean unproductive wait time per process spent in OpenMP parallel regions (normally at synchronization barriers). High values are usually bad. This value is HIGH. The application's OpenMP work sharing is NOT well load-balanced. More details...</td>
<td></td>
</tr>
</tbody>
</table>
TUNING MPI APPLICATION PERFORMANCE WITH INTEL® TRACE ANALYZER AND COLLECTOR
## Value Proposition

<table>
<thead>
<tr>
<th>What</th>
<th>Why</th>
<th>How</th>
</tr>
</thead>
</table>
| • **Intel's High Performance MPI Communications Profiler & Analyzer for Scalable HPC Development** | • **Scale Performance** – Perform on More Nodes  
• **Scale Forward** – Multicore and Manycore Ready  
• **Scale Efficiently** – Tune & Debug on More Nodes | • **Visualize** – Understand parallel application behavior  
• **Evaluate** – Profiling statistics and load balancing  
• **Analyze** – Automated analysis of common MPI issues  
• **Identify** – Communication hotspots |
Intel® Trace Analyzer and Collector Overview

Intel® Trace Analyzer and Collector helps the developer:

- Visualize and understand parallel application behavior
- Evaluate profiling statistics and load balancing
- Identify communication hotspots

Features

- Event-based approach
- Low overhead
- Excellent scalability
- Powerful aggregation and filtering functions
- Performance Assistance and Imbalance Tuning
- NEW in 9.1: MPI Performance Snapshot
Strengths of Event-based Tracing

**Predict**
- Detailed MPI program behavior

**Record**
- Exact sequence of program states – keep timing consistent

**Collect**
- Collect information about exchange of messages: at what times and in which order

An event-based approach is able to detect temporal dependencies!
## Multiple Methods for Data Collection

<table>
<thead>
<tr>
<th>Collection Mechanism</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run with –trace or preload trace collector library.</td>
<td>Automatically collects all MPI calls, requires no modification to source, compile, or link.</td>
<td>No user code collection.</td>
</tr>
<tr>
<td>Link with –trace.</td>
<td>Automatically collects all MPI calls.</td>
<td>No user code collection.</td>
</tr>
<tr>
<td>Compile with –tcollect.</td>
<td>Automatically instruments all function entries/exits.</td>
<td>Requires recompile of code.</td>
</tr>
<tr>
<td>Add API calls to source code.</td>
<td>Can selectively instrument desired code sections.</td>
<td>Requires code modification.</td>
</tr>
</tbody>
</table>
Views and Charts

Helps navigating through the trace data and keep orientation

Every View can contain several Charts

All Charts in a View are linked to a single:
  - time-span
  - set of threads
  - set of functions

All Charts follow changes to View (e.g. zooming)
Event Timeline

Get detailed impression of program structure

Display functions, messages, and collective operations for each rank/thread along time-axis

Retrieval of detailed event information
Quantitative Timeline

Get impression on parallelism and load balance

Show for every function how many threads/ranks are currently executing it
**Flat Function Profile**

**Statistics about functions**

<table>
<thead>
<tr>
<th>Name</th>
<th>TSelf</th>
<th>TSelf / TTotal</th>
<th>#Cells</th>
<th>TSelf / Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group All_Threads</td>
<td>678,787</td>
<td>49,336</td>
<td>0,013</td>
<td>703 s</td>
</tr>
<tr>
<td>PRECON</td>
<td>580,473</td>
<td>34,344</td>
<td>0,001</td>
<td>859 s</td>
</tr>
<tr>
<td>MP3_SYNC</td>
<td>410,463</td>
<td>131,341</td>
<td>0,000</td>
<td>328 s</td>
</tr>
<tr>
<td>MATMP5L</td>
<td>329,403</td>
<td>91,919</td>
<td>0,000</td>
<td>673 s</td>
</tr>
<tr>
<td>SOLVER</td>
<td>148,746</td>
<td>14,154</td>
<td>0,001</td>
<td>673 s</td>
</tr>
<tr>
<td>User_Code</td>
<td>94,227</td>
<td>9,141</td>
<td>0,001</td>
<td>927 s</td>
</tr>
<tr>
<td>MPB_Bidual</td>
<td>40,822</td>
<td>701</td>
<td>32</td>
<td>1,369</td>
</tr>
<tr>
<td>MPB_Barrier</td>
<td>24,222</td>
<td>489</td>
<td>32</td>
<td>489 s</td>
</tr>
<tr>
<td>MPB_Reduce</td>
<td>23,007</td>
<td>645</td>
<td>32</td>
<td>3,590</td>
</tr>
<tr>
<td>MPB_Wall</td>
<td>17,507</td>
<td>615</td>
<td>32</td>
<td>1,056</td>
</tr>
<tr>
<td>MPB_Comp_dup</td>
<td>11,756</td>
<td>564</td>
<td>32</td>
<td>183</td>
</tr>
<tr>
<td>MPB_Iter</td>
<td>7,039</td>
<td>869</td>
<td>32</td>
<td>0,000</td>
</tr>
<tr>
<td>MPB_Wine</td>
<td>7,490</td>
<td>913</td>
<td>32</td>
<td>0,000</td>
</tr>
<tr>
<td>MPB_Init</td>
<td>4,909</td>
<td>197</td>
<td>32</td>
<td>0,000</td>
</tr>
<tr>
<td>MPB_Final</td>
<td>0,000</td>
<td>286</td>
<td>32</td>
<td>0,000</td>
</tr>
<tr>
<td>MPB_Comp_size</td>
<td>0,001</td>
<td>205</td>
<td>32</td>
<td>0,000</td>
</tr>
<tr>
<td>MPB_Comm_init</td>
<td>0,000</td>
<td>230</td>
<td>32</td>
<td>0,000</td>
</tr>
</tbody>
</table>

**Call Graph**

<table>
<thead>
<tr>
<th>Children of Group All_Threads</th>
<th>TSelf</th>
<th>TSelf / TTotal</th>
<th>#Cells</th>
<th>TSelf / Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_dup</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>MPI_Wall</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 13 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 24 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 25 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 27 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 29 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 30 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 32 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 33 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 35 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 36 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 37 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 39 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 41 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 43 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 45 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 47 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 49 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 51 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 53 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 55 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 57 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 59 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 61 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 63 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 65 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 67 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 69 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 71 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 73 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 75 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 77 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 79 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 81 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 83 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 85 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 87 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 89 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 91 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 93 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 95 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 97 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
<tr>
<td>Process 99 Thread 0</td>
<td>0,013</td>
<td>335 s</td>
<td>1,546</td>
<td>0,000</td>
</tr>
</tbody>
</table>

**Optimization Notice**

Copyright © 2016, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.*
Call Tree and Call Graph

Function statistics including calling hierarchy

- Call Tree shows call stack
- Call Graph shows calling dependencies
Communication Profiles

Statistics about point-to-point or collective communication

Matrix supports grouping by attributes in each dimension

- Sender, Receiver, Data volume per msg, Tag, Communicator, Type

Available attributes

- Count, Bytes transferred, Time, Transfer rate
Zooming
Grouping and Aggregation

Allow analysis on different levels of detail by aggregating data upon group-definitions

Functions and threads can be grouped hierarchically

- Process Groups and Function Groups

Arbitrary nesting is supported

- Functions/threads on the same level as groups
- User can define his/her own groups

Aggregation is part of View-definition

- All charts in a View adapt to requested grouping
- All charts support aggregation
Aggregation Example
Tagging and Filtering

Help concentrating on relevant parts

Avoid getting lost in huge amounts of trace data

Define a set of interesting data

- E.g. all occurrences of function x
- E.g. all messages with tag y on communicator z

Combine several filters:
Intersection, Union, Complement

Apply it

- Tagging: Highlight messages
- Filtering: Suppress all non-matching events
Tagging Example
Filtering Example
Automatic Performance Assistant
Detect common MPI performance issues
Automated tips on potential solutions

Automatically detect performance issues and their impact on runtime
Summary page shows computation vs. communication breakdown

Is your application MPI-bound?

Resource usage

Is your application CPU-bound?

Largest MPI consumers

Next Steps
MPI-3.0 Support

Support for major MPI-3.0 features

- Non-blocking collectives
- Fast RMA
- Large counts
Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2016, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804