MPI Performance Engineering through the Integration of MVAPICH and TAU

Allen D. Malony
Department of Computer and Information Science
University of Oregon
Acknowledgement

- Research work presented in this talk is being done through a collaboration with Ohio State University:

- People who are doing the work are:
  OSU: DK Panda, Hari Subramoni
  UO: Sameer Shende, Srinivasan Ramesh, Aurele Maheo, me.
Outline

- Motivation
  - How do we understand MPI runtime complexities?
  - How do we evolve tools for MPI performance tuning?
- Introduction of MPI Tools Interface (MPI-T)
- Quick overview of the TAU Performance System
- Infrastructure for MPI Performance Engineering
  - Integration of TAU and MVAPICH using MPI-T
  - Extension with plug-in and monitoring framework
- Case Studies
  - Demonstrate MPI performance engineering infrastructure
  - AmberMD, 3DStencil, miniAMR
- Conclusion and Future
Motivation

- MPI libraries are complex software systems
  - Implement the MPI standard (currently, MPI 3.1)
  - Run on different network layers and parallel HPC platforms
  - Many modular components, interacting in complex ways
  - Multiple tunable parameters (platform and application)
  - Current and future HPC hardware complicate matters

- MPI performance engineering is important
  - Use message benchmarks for platform performance analysis
  - Application-based MPI performance engineering is harder
  - Need to evolve our tools
  - Leverage MPI tools interface (MPI_T)
  - Deeper integration of tools within the MPI software stack
What about the MPI Profiling Interface?

- With impressive forethought, MPI was originally designed with support for performance engineering.

- **MPI Profiling Interface (PMPI)**
  - Library interposition mechanism to observe MPI routines
  - Tool implements “wrapper” version of MPI routines
  - Original MPI call is intercepted by the tool version
    - Tool sees both “entry” and “exit”
    - On entry, tool does whatever it does and then calls “PMPI” interface to execute the “real” MPI routine with the user-supplied parameters
    - On exit, tool does whatever else more and then returns with arguments and return value from the “real” MPI routine

- PMPI supports performance engineering with respect to:
  - MPI routines: time spent, # calls, hardware counts, …
  - Message communication: time, size, patterns, …

- Application-level (external) view is not enough
MPI Tools Interface (MPI_T)

- Introduced in the MPI 3.0 standard (latest MPI 3.1)
- Defines two types of variable (access semantics):
  - Performance Variables (PVARs)
  - Control Variables (CVARs)

- PVARs
  - Variables report static and dynamic information of MPI performance
    - counters, metrics, state, …
  - Written by MPI implementation
  - Read by the tool via MPI_T interface

- CVARs
  - Properties and configuration settings used to modify MPI behavior
  - Configuration and dynamic control
  - Written by the tool via MPI_T interface

- Each MPI implementation defines PVARs and CVARs supported
- These are registered through MPI_T for tool access
Benefits of MPI Tools Interface

- PMPI interface does not provide any opportunity to gain insight into MPI library internals, nor any mechanism to enable re-configuration and control of MPI

- MPI_T provides a window on MPI internals
  - Standardized approach (versus earlier attempts, PERUSE)
  - MPI implementations free to decide what is exported
  - Tool discovers what MPI exports and decides what to do
  - Rich information
  - Rank-level view
  - Exposes control
  - Binding lets PVARs and CVARs to be tied to MPI objects

<table>
<thead>
<tr>
<th>Constant</th>
<th>MPI object</th>
</tr>
</thead>
<tbody>
<tr>
<td>MILT_BIND_NO_OBJECT</td>
<td>N/A; applies globally to entire MPI process</td>
</tr>
<tr>
<td>MILT_BIND_MPI_COMM</td>
<td>MPI communicators</td>
</tr>
<tr>
<td>MILT_BIND_MPI_DATATYPE</td>
<td>MPI datatypes</td>
</tr>
<tr>
<td>MILT_BIND_MPI_ERRHANDLER</td>
<td>MPI error handlers</td>
</tr>
<tr>
<td>MILT_BIND_MPI_FILE</td>
<td>MPI file handles</td>
</tr>
<tr>
<td>MILT_BIND_MPI_GROUP</td>
<td>MPI groups</td>
</tr>
<tr>
<td>MILT_BIND_MPI_OP</td>
<td>MPI reduction operators</td>
</tr>
<tr>
<td>MILT_BIND_MPI_REQUEST</td>
<td>MPI requests</td>
</tr>
<tr>
<td>MILT_BIND_MPI_WIN</td>
<td>MPI windows for one-sided communication</td>
</tr>
<tr>
<td>MILT_BIND_MPI_MESSAGE</td>
<td>MPI message object</td>
</tr>
<tr>
<td>MILT_BIND_MPI_Info</td>
<td>MPI info object</td>
</tr>
</tbody>
</table>
Implementations of MPI_T

- **MPICH**
  - 10 PVARs (no binding)
  - 71 CVARs (no binding)

- **OpenMPI**
  - 5 PVARs (4 bound to MPI objects)
  - 1102 CVARs (exporting of MCA parameters, no binding)

- **Intel MPI**
  - 0 PVARs
  - 60 CVARs

- **MVAPICH**
  - 73 PVARs (no binding)
  - 82 CVARs (no binding) (additional CVARs being added)

- **TAU** works with MPICH, Intel MPI, MVAPICH
MVAPICH MPI_T

- PVARs
  - Memory allocation
  - Collective algorithms
  - VBUFs
  - SMP bytes for Eager and Rendezvous
  - RDMA and IB
  - Message receive queue

- CVARs
  - Collective algorithms: message size, all reduce, bcast, ...
  - Modes: eager, rendezvous, ...
  - Garbage collection, RMA, SMP, Nemesis
  - VBUFs

- Some variables are static and some are dynamic
- Some are variables are set at MPI_Init
Using MPI_T

- MPI implementation defines the PVARs and CVARs
- MPI_T specification defines the interface
  - Semantics
  - Process and procedures
  - Parameters and data types
- MPI implementations support the MPI_T interface
- Tools utilize the MPI_T interface
  - MPI_T_PVAR_GET_INFO
  - MPI_T_CVAR_GET_INFO
    - Get performance variables, incorporate in measurements, analyze
    - Set control variables to enable specific MPI operation
- MPI_T is a rank-level interface (like other MPI routines)
- MPI_T allows multiple in-flight performance sessions
  - Different tools can be simultaneously active
TAU Performance System®

- Performance problem solving framework for HPC
  - Integrated, scalable, flexible, portable
  - Target all parallel programming / execution paradigms

- Integrated performance toolkit (open source)
  - Multi-level performance instrumentation
  - Widely-ported, flexible, and configurable performance measurement
  - Performance data management and data mining
**TAU Architecture**

- TAU is a parallel performance framework and toolkit
- Software architecture provides separation of concerns
  - Instrumentation | Measurement | Analysis

![TAU Architecture Diagram]

**Instrumentation**
- **Source**
  - C, C++, Fortran
  - Python, UPC, Java
  - Robust parsers (PDT)
- **Wrapping**
  - Interposition (PMPI)
  - Wrapper generation
- **Linking**
  - Static, dynamic
  - Preloading
- **Executable**
  - Dynamic (Dyninst)
  - Binary (Dyninst, MAQAO)

**Measurement**
- **Events**
  - static/dynamic
  - routine, basic block, loop
  - threading, communication
  - heterogeneous
- **Profiling**
  - flat, callpath, phase, parameter, snapshot
  - probe, sampling, hybrid
- **Tracing**
  - TAU / Scalasca tracing
  - Open Trace Format (OTF)
- **Metadata**
  - system, user-defined

**Analysis**
- **Profiles**
  - ParaProf parallel profile analyzer / visualizer
  - TAUdb parallel profile database
  - PerfExplorer parallel profile data mining
- **Tracing**
  - TAU trace translation
    - OTF, SLOG-2
  - Trace analysis / visualizer
    - Vampir, Jumpshot
- **Online**
  - event unification
  - statistics calculation
TAU Components

- Instrumentation
  - Fortran, C, C++, OpenMP, MPI, Python, Java, UPC, Chapel, ...
  - Source, compiler, library wrapping, binary rewriting
  - Automatic instrumentation

- Measurement
  - Probe-based and sample-based
  - Internode: MPI, OpenSHMEM, ARMCI, PGAS, DMAPP
  - Intranode: Pthreads, OpenMP, hybrid, ...
  - Heterogeneous: GPU, MIC, CUDA, OpenCL, OpenACC, ...
  - Performance data (timing, counters) and metadata
  - Parallel profiling and tracing (with Score-P integration)

- Analysis
  - Parallel profile analysis and visualization (ParaProf)
  - Performance data mining / machine learning (PerfExplorer)
  - Performance database technology (TAUdb)
  - Empirical autotuning
**TAU Instrumentation Mechanisms**

- **Source code**
  - Manual (TAU API, TAU component API)
  - Automatic (robust)
    - C, C++, F77/90/95, OpenMP (POMP/OPARI), UPC
  - Compiler (GNU, IBM, NAG, Intel, PGI, Pathscale, Cray, …)
- **Object code (library-level)**
  - Statically- and dynamically-linked wrapper libraries
    - MPI, I/O, memory, …
  - Powerful library wrapping of external libraries without source
- **Executable code / runtime**
  - Runtime preloading and interception of library calls
  - Binary instrumentation (Dyninst, MAQAO, PEBIL)
  - Dynamic instrumentation (Dyninst)
  - OpenMP (runtime API, CollectorAPI, GOMP, OMPT)
- **Virtual machine, interpreter, and OS instrumentation**
MPI Performance Engineering

- Improving the performance of MPI implementations and use of the MPI library is important and challenging
- How can MPI_T help in this goal?
  - Couple MPI library and performance tool software components
  - Focus on TAU and MVAPICCH
- Identify performance engineering methods
  - Extended performance measurement and analysis
  - MPI optimization based on recommendation
  - Runtime introspection and performance autotuning
  - Performance monitoring across MPI ranks
- Enabling closer software interaction / co-design is a key goal
- Application-level MPI performance engineering
  - Evaluate opportunities in different domains
Infrastructure Design using MPI_T

MPI Applications

TAU

MVAPICH

MPI_T

HPC Environment
(heterogeneous, hierarchical memory, complex networks, multi/many core)

Monitoring System

BEACON
PyCOOLR

Measurement/Analysis
Autotuning Plugins
Runtime Settings

Set CVARs
Get PVARs

PVAR
CVAR

Performance counters
Topology information
Mechanisms/Algorithms
Migration, C/R, ...

MUG 2017
MPI Performance Engineering through the Integration of MVAPICH and TAU
TAU MPI_T Measurement

- TAU can make MPI_T measurements across all ranks.
- Query PVARs at regular intervals (using signal handler).
- Analyze using TAU’s ParaProf parallel profiler.

<table>
<thead>
<tr>
<th>Name</th>
<th>Total</th>
<th>NumSamples</th>
<th>MaxValue</th>
<th>MinValue</th>
<th>MeanValue</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Footprint (VmRSS)</td>
<td>5,187,810.828</td>
<td>24</td>
<td>219,195,453</td>
<td>202,978.5</td>
<td>216,158.785</td>
<td>4,085.753</td>
</tr>
<tr>
<td>Message size for all-gather</td>
<td>12,778,020,648</td>
<td>3,130</td>
<td>9,806,616</td>
<td>9,806,616</td>
<td>9,806,616</td>
<td>0.177</td>
</tr>
<tr>
<td>Message size for all-reduce</td>
<td>570,400</td>
<td>20,112</td>
<td>944</td>
<td>9</td>
<td>28,501</td>
<td>21,839</td>
</tr>
<tr>
<td>Message size for broadcast</td>
<td>59,867,064</td>
<td>300</td>
<td>9,906,616</td>
<td>4</td>
<td>196,223,547</td>
<td>1,104,554.497</td>
</tr>
<tr>
<td>Message size for gather</td>
<td>1,496.43</td>
<td>0.984</td>
<td>38,307,094</td>
<td>0.062</td>
<td>1,520,183</td>
<td>7,478.003</td>
</tr>
<tr>
<td>Peak Memory Usage Resident Set Size (VmHWM)</td>
<td>5,191,435,234</td>
<td>24</td>
<td>219,248,062</td>
<td>202,978.5</td>
<td>216,309,801</td>
<td>4,144.185</td>
</tr>
</tbody>
</table>

[GROUP=MAX_MARKER] Message size for broadcast
- 8,223,804
- Query PVARs at regular intervals (using signal handler)
- Analyze using TAU’s ParaProf parallel profiler

[GROUP=MAX_MARKER] Message size for gather
- 1,496.43
- 0.984
- 38,307,094
- 0.062
- 1,520,183
- 7,478.003

[GROUP=MAX_MARKER] mem_allocated (Current level of allocates)
- 39,163,581
- 0.064
- 38,307,094
- 38,307,094
- 38,307,094

[GROUP=MAX_MARKER] mem_allocated (Maximum level of memory)
- 39,163,581
- 0.064
- 38,307,094
- 38,307,094

TAU MPI_T Measurement

- TAU can make MPI_T measurements across all ranks.
- Query PVARs at regular intervals (using signal handler).
- Analyze using TAU’s ParaProf parallel profiler.
Case Study Applications

- **AmberMD** is a popular molecular dynamics code
  - Focus on improving the performance of parallel MD engine
  - Substantial runtime is in MPI communication routines
  - MPI_Wait dominates in runtime
  - MPI_Isend and MPI_Irecv dominate in # calls
- **3DStencil** is a simple synthetic stencil application
  - Performs non-blocking point-to-point communication in a grid
  - Computes between communication
  - Look at communication-computation overlap achieved
  - Large, fixed-size message used
- **MiniAMR** is a Mantevo mini-app for 3D stencil computation
  - Memory bound application
  - Significant MPI_Wait for small point-to-point messages (1-2 KB)
  - Significant MPI_Allreduce for 8-byte messages (latency sensitive)
  - Part of a check-summing routine
Experimental Setup

- Experiments with AmberMD
  - Stampede, a 6400 node Infiniband cluster at TACC
  - Stampede compute node: two Xeon E5-2680 8-core “Sandy Bridge” processors and one first-generation Intel Xeon Phi SE10P KNC MIC
  - All our experiments using pure MPI on the Xeon host with 16 MPI processes on a node (1 per core)
    - MV2_ENABLE_AFFINITY turned on
    - A total of 8 nodes (128 processes) used

- Experiments with MiniAMR and 3DStencil
  - ri2 cluster at e Ohio State University
  - ri2 computer node: two 14-core Intel Xeon E5-2680 v4 processors
  - All experiments used pure MPI on Intel Xeon hosts with 28 MPI processes on a node (1 per core)
    - MV2_ENABLE_AFFINITY turned on
    - 3DStencil: 16 nodes (448 processes) used
    - MiniAMR: 8 nodes (224 processes) used
Hardware Offloading of Collectives

- MVAPICH2 now supports offloading of MPI_Allreduce to network hardware using the SHArP protocol
  - Hardware offloading is mainly beneficial to applications where communication is sensitive to latency

- Measurement
  - TAU collects statistics about the average message size involved in MPI_Allreduce operation
  - TAU collects the time spent within MPI_Allreduce versus the overall application time

- Analysis and recommendation
  - If the message size is below a certain threshold and the percentage of total runtime spent within MPI_Allreduce is above a certain threshold, trigger possible recommendation
  - Set CVAR MPIR_CVAR_ENABLE_SHARP
Hardware Offloading of Collectives (2)

- ParaProf recommendation for miniAMR

- Performance improvement for miniAMR

<table>
<thead>
<tr>
<th>Run</th>
<th># Processes</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>224</td>
<td>648</td>
</tr>
<tr>
<td>SHArP enabled</td>
<td>224</td>
<td>618</td>
</tr>
</tbody>
</table>

You could see potential improvement in performance by configuring MVAPICH with `--enable-sharp` and enabling MPIR_CVAR_ENABLE_SHARP in MVAPICH version 2.3a and above.
Eager Limit / Freeing Unused Buffers

- MVAPICH uses internal communication buffers (VBUFs) to temporarily hold messages that are yet to be transferred to the receiver in point-to-point communications
  - There are multiple VBUF pools which vary in size of the VBUF
  - At runtime, MVAPICH performs a match based on the size of the message and accordingly selects a VBUF pool to use
  - VBUFs are used to send short messages in an Eager manner to reduce communication latency
  - Longer messages use the Rendezvous protocol without VBUFs
- Using Eager protocol can result in a greater amount of memory being used for VBUFs
  - Could cause other performance problems to arise
- Monitor and control usage of virtual buffers
Eager Limit / Freeing Unused Buffers (2)

- Use of virtual buffers can offer significant performance improvement to applications performing heavy point-to-point communication, such as stencil based codes.
- MVAPICH2 offers a number of PVARs that monitor the current usage level, availability of free VBUFS in different VBUF pools, maximum usage levels, and the number of allocated VBUFS at process-level granularity.
- Accordingly, it exposes CVARs that modify how MVAPICH2 allocates and frees these VBUFS at runtime.
- Usage level of VBUF pools can vary with time and between processes.
  - Unused VBUFS represent wasted memory resource.
  - Identifying opportunities to free could save memory.
Eager Limit / Freeing Unused Buffers (3)

- PVARs of interest
  - `mv2_vbuf_allocated_array`
  - `mv2_vbuf_max_use_array`
  - `mv2_total_vbuf_memory`

- CVARs of interest
  - `MPIR_CVAR_IBA_EAGER_THRESHOLD`
  - `MPIR_CVAR_VBUF_TOTAL_SIZE`
  - `MPIR_CVAR_VBUF_POOL_CONTROL`
  - `MPIR_CVAR_VBUF_POOL_REduced_VALUE`

- Increasing the value of the Eager limit could lead to improved overlap between communication and computation as larger messages are sent eagerly
  - Overall execution time for the application may reduce
Higher Eager threshold on 3DStencil application
- Improves computation-communication overlap
- Increases VBUF memory size

![3DStencil Diagram](image)

<table>
<thead>
<tr>
<th>Run</th>
<th>Number of Processes</th>
<th>Message Size(Bytes)</th>
<th>Communication-Computation Overlap</th>
<th>Eager Threshold</th>
<th>Total VBUF Memory(Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>448</td>
<td>32,768</td>
<td>11.1</td>
<td>MVAPICH2 Default</td>
<td>1,436,574</td>
</tr>
<tr>
<td>Eager</td>
<td>448</td>
<td>32,768</td>
<td>68.7</td>
<td>33,000</td>
<td>2,573,345</td>
</tr>
<tr>
<td>TAU runtime tuning</td>
<td>448</td>
<td>32,768</td>
<td>69.7</td>
<td>33,000</td>
<td>1,208,782</td>
</tr>
</tbody>
</table>
AmberMD

- Consider total VBUF memory usage for AmberMD application when the Eager threshold is raised

![Graph](a) Increase in total VBUF memory when increasing Eager threshold

![Graph](b) Decrease in total VBUF memory when freeing unused VBUFs

- AmberMD demonstrates a behavior where virtual buffers (VBUFs) from all pools except one remain largely unused
- Freeing unused VBUFs can lead to significant memory savings
AmberMD (2)

- Eager threshold is set statically right after MPI_Init
  - MPIR_CVAR_IBA_EAGER_THRESHOLD
- Increasing the Eager threshold from the MVAPICH2 default value to 64000 Bytes had the effect of reducing application runtime by 38.5%
- This was achieved at the cost of increasing the total VBUF memory across all processes by 80%

<table>
<thead>
<tr>
<th>Run</th>
<th>Number of Processes</th>
<th>Eager Threshold</th>
<th>MD Timesteps</th>
<th>Application Runtime(Seconds)</th>
<th>Total VBUF Memory(Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>128</td>
<td>MVAPICH2 Default</td>
<td>4,000</td>
<td>210</td>
<td>695,150</td>
</tr>
<tr>
<td>Eager</td>
<td>128</td>
<td>64,000</td>
<td>4,000</td>
<td>129</td>
<td>768,188</td>
</tr>
<tr>
<td>TAU runtime autotuning</td>
<td>128</td>
<td>64,000</td>
<td>4,000</td>
<td>129</td>
<td>629,511</td>
</tr>
</tbody>
</table>

- Dynamically monitored VBUF usage and freed the unused ones, while maintaining same runtime
Enabling Runtime Introspection

- TAU gathers performance data exposed through MPI T
  - Interrupt is triggered at regular intervals
  - In signal handler, the MPI T interface is queried and the values of all the performance variables exported are stored at process level granularity
  - TAU registers internal atomic events for each of these performance variables, and every time an event is triggered (while querying the MPI T interface), the running average, minimum value, the maximum value and other statistics

- What to do with the data?
  - Save for offline analysis
  - Analyze online and take tuning action
Plugin Architecture for Runtime Autotuning

- TAU can be extended with a plugin that analyzes performance
- Based on policies, the plugin can make decisions about how control the runtime software
- Generic plugin architecture being developed
  - Policy specification
- Apply in MPI_T for MVAPICH tuning
- See poster!
Global Monitoring for Application Control

- Application tuning requires understanding distributed performance
- UO is building global monitoring framework
  - **BEACON** (Backplane for Event and Control Notification) from DOE Argo project
  - **SOS** (Scalable Observation System) from DOE MONA project
- Use BEACON with MPI_T
  - Gather PVARs from multiple ranks
  - Set CVARs for multiple ranks
  - Analyze and visualize (PYCOOLR)
- See poster!
Conclusion and Future

- UO and OSU are integrating TAU and MVAPICH using the MPI_T interface defined in the MPI 3.1 standard
- Base functionality is in place
- MVAPICH is being enhanced with PVARs and CVARs
- TAU is being enhanced with analysis functionality, online monitoring, and runtime tuning
- Compelling reasons to integrate performance analysis and optimization across the parallel software stack
- Support for runtime performance awareness and control is important to address dynamic performance variation
- Future complex HPC systems will require this
More Information


- Expanded paper just accepted in the Parallel Computing journal