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:

  "SI2-SSI: Collaborative Research: A Software Infrastructure for MPI Performance Engineering: Integrating MVAPICH and TAU via the MPI Tools Interface," D. Panda (PI, OSU), S. Shende (PI, UO), A. Malony (co-PI, UO), NSF Software Infrastructure for Sustained Innovation – SI2, 9/2015-8/2019, Grants: ACI-1450440 and ACI-1450471.

- 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

Constant	MPI object
MPI_T_BIND_NO_OBJECT	N/A; applies globally to entire MPI process
MPI_T_BIND_MPI_COMM	MPI communicators
MPI_T_BIND_MPI_DATATYPE	MPI datatypes
MPI_T_BIND_MPI_ERRHANDLER	MPI error handlers
MPI_T_BIND_MPI_FILE	MPI file handles
MPI_T_BIND_MPI_GROUP	MPI groups
MPI_T_BIND_MPI_OP	MPI reduction operators
MPI_T_BIND_MPI_REQUEST	MPI requests
MPI_T_BIND_MPI_WIN	MPI windows for one-sided communication
MPI_T_BIND_MPI_MESSAGE	MPI message object
MPI_T_BIND_MPI_INFO	MPI info object

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

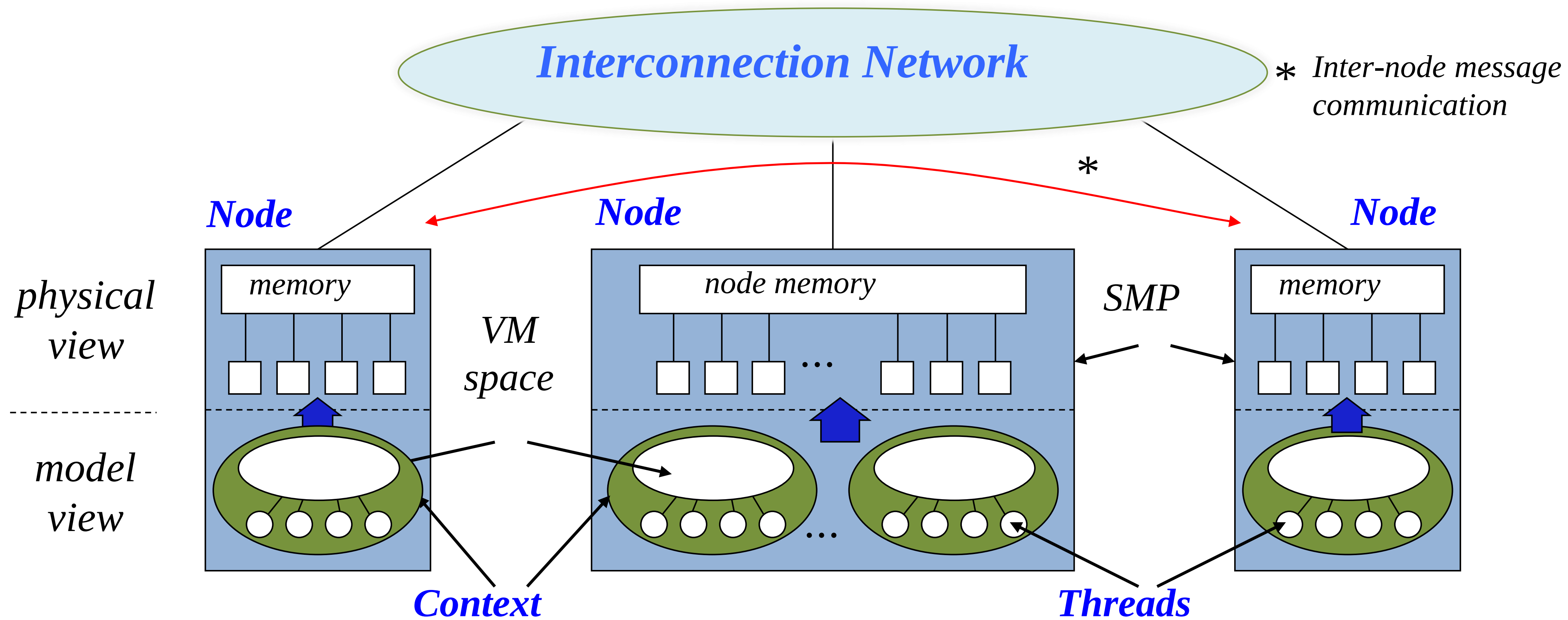
- PVARs
  - Memory allocation
  - Collective algorithms
  - VBUFs
  - SMP bytes for Eager and Rendezvous
  - RDMA and IB
  - Message receive queue
- CVARs
  - Collective algorithms: message size, all reduce, bcast, …
  - Modes: eager, rendezvous, ...
  - Garbage collection, RMA, SMP, Nemesis
  - VBUFs
- 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®

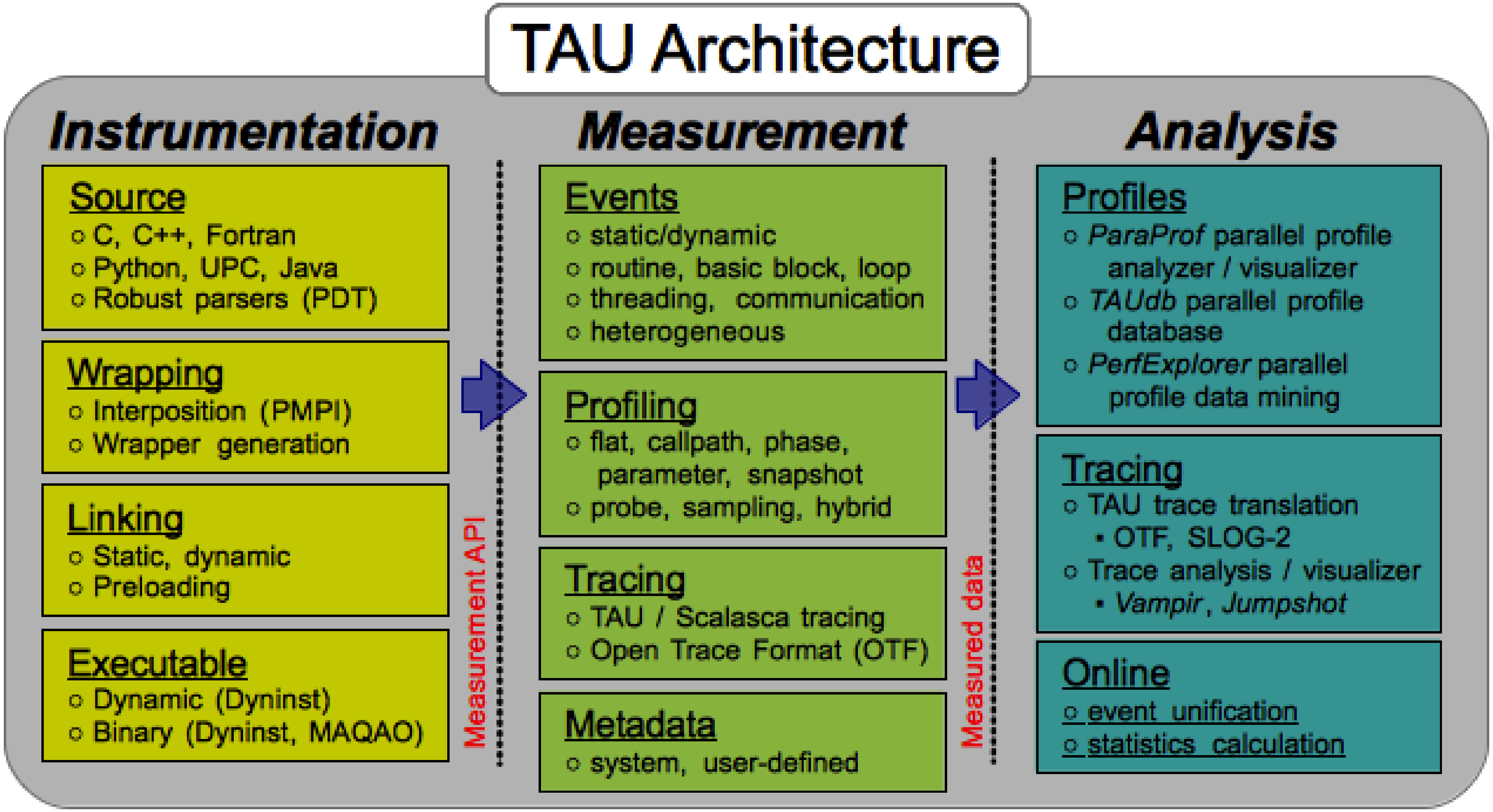
- Performance problem solving framework for HPC
  - Integrated, scalable, flexible, portable
  - Target all parallel programming / execution paradigms



- 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



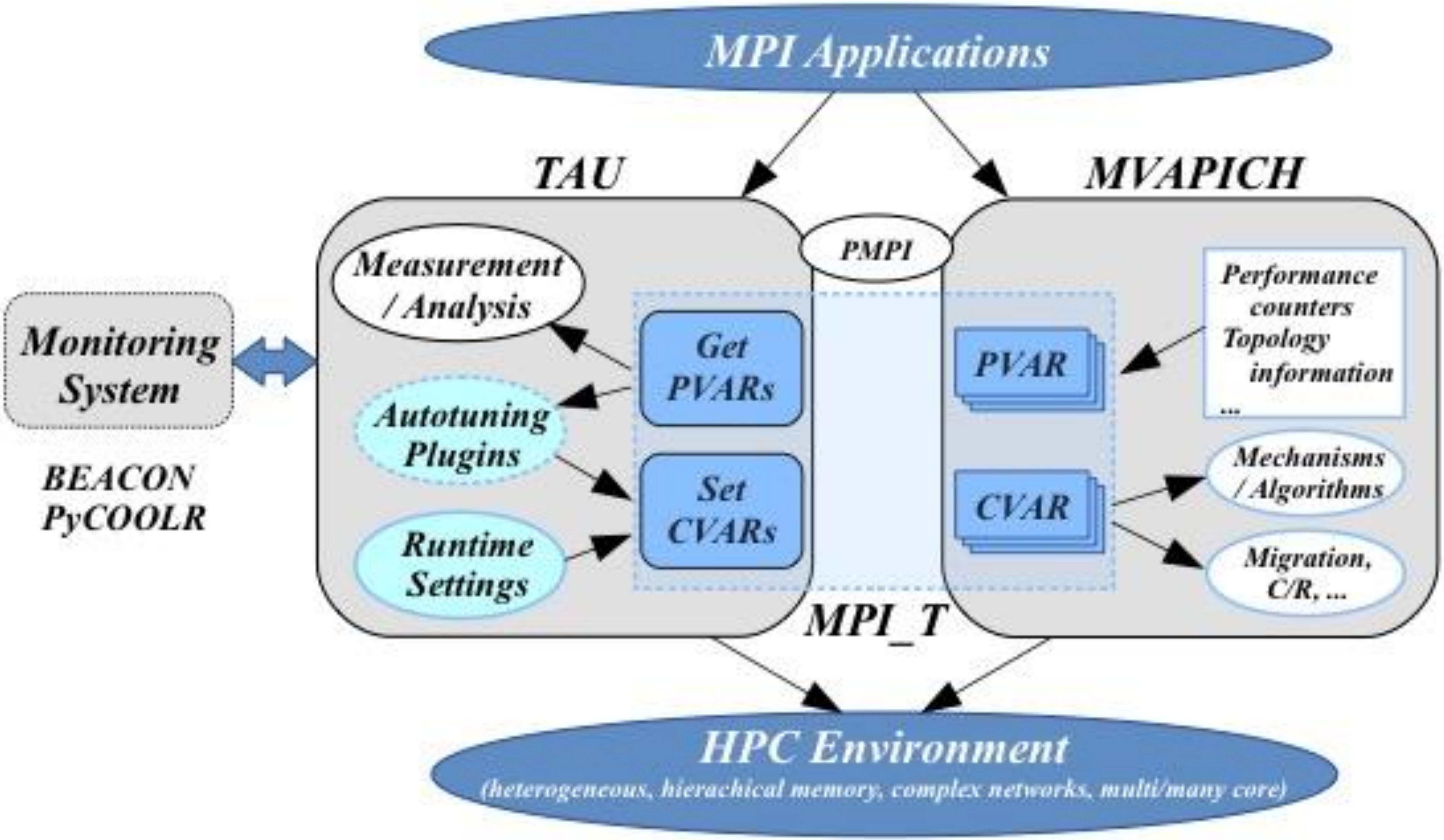
TAU Components

- 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

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 MVAPICH
- 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



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

TAU: ParaProf: Mean Context Events – Default.ppk (on godzilla)

File Options Windows Help

Name △	Total	NumSamples	MaxValue	MinValue	MeanValue	Std. Dev.
.TAU application						
Memory Footprint (VmRSS) (KB)	5,187,810.828	24	219,195.453	202,978.5	216,158.785	4,085.753
Message size for all-gather	12,778,020,648	1,303	9,806,616	9,806,616	9,806,616	0.177
Message size for all-reduce	570,400	20,013	944	8	28.501	21.839
Message size for broadcast	58,867,064	300	9,806,616	4	196,223.547	1,104,554.497
Message size for gather	1,496.43	0.984	38,307.094	0.062	1,520.183	7,478.003
Peak Memory Usage Resident Set Size (VmHWM) (KB)	5,191,435.234	24	219,248.062	202,978.5	216,309.801	4,144.185
[GROUP=MAX_MARKER] Message size for broadcast	8,223,804	4	4,953,852	256	2,055,951	2,140,002.131
[GROUP=MAX_MARKER] Message size for gather	149.637	0.004	38,307.094	38,307.094	38,307.094	0
[GROUP=MAX_MARKER] mem_allocated (Current level of allocatec	39,163.581	0.047	835,489.734	835,489.734	835,489.734	0
[GROUP=MAX_MARKER] mem_allocated (Maximum level of memory	39,163.581	0.047	835,489.734	835,489.734	835,489.734	0
[GROUP=MAX_MARKER] mpit_progress_poll (CH3 RDMA progress (	3,303,001.969	0.402	8,223,534.352	8,195,271.551	8,209,402.951	14,131.4
[GROUP=MAX_MARKER] mv2_coll_allreduce_2lvl (Number of times	2,675.004	2	1,698.004	977	1,337.502	360.502
[GROUP=MAX_MARKER] mv2_coll_allreduce_shm_intra (Number of	2,675.004	2	1,698.004	977	1,337.502	360.502
[GROUP=MAX_MARKER] mv2_coll_allreduce_shm_rd (Number of tin	10.48	0.125	106.375	61.312	83.844	22.531
[GROUP=MAX_MARKER] mv2_coll_bcast_shmem (Number of times	2,674.992	2	1,697.992	977	1,337.496	360.496
[GROUP=MAX_MARKER] mv2_ibv_channel_exact_recv_count (Numb	372.814	1.629	316.027	149.215	228.874	75.115
[GROUP=MAX_MARKER] mv2_num_shmem_coll_calls (Number of tin	5,389.996	2	3,420.996	1,969	2,694.998	725.998
[GROUP=MAX_MARKER] mv2_rdmafp_exact_recv_count (Number o	27,359.951	2.539	15,048.816	6,531.043	10,775.611	3,831.531
[GROUP=MAX_MARKER] mv2_rdmafp_out_of_order_packet_count (	372.814	1.629	316.027	149.215	228.874	75.115
[GROUP=MAX_MARKER] mv2_reg_cache_hits (Number of registrat	14,479.537	1.879	9,870.988	5,554.168	7,706.365	2,125.758
[GROUP=MAX_MARKER] mv2_reg_cache_misses (Number of regist	3.562	0.117	34.219	26.566	30.393	3.826
[GROUP=MAX_MARKER] mv2_smp_eager_received (Number of SMP	15,433,665.765	1.094	14,471,985.461	13,756,999.242	14,110,780.128	340,356.783
[GROUP=MAX_MARKER] mv2_smp_eager_sent (Number of SMP byt	20,379,122.18	1.152	18,129,837.477	17,253,827.977	17,684,933.146	429,103.764
[GROUP=MAX_MARKER] mv2_smp_read_progress_poll (CH3 SMP re	26,667,654.829	1.004	26,603,456.156	26,524,323.113	26,563,889.635	39,566.521
[GROUP=MAX_MARKER] mv2_smp_read_progress_poll_success (Ui	356,569.965	2	226,482.82	130,087.145	178,284.982	48,197.838
[GROUP=MAX_MARKER] mv2_smp_write_progress_poll (CH3 SMP w	26,667,654.525	1.004	26,603,455.855	26,524,322.809	26,563,889.332	39,566.523
[GROUP=MAX_MARKER] mv2_total_vbuf_memory (Total amount of	39,075.459	0.062	635,728	614,686.688	625,207.344	10,520.656
[GROUP=MAX_MARKER] mv2_vbuf_allocate_time (Average time fo	0.123	0.062	2	1.926	1.963	0.037
[GROUP=MAX_MARKER] mv2_vbuf_allocated (Number of VBUFs allc	2.171	0.051	42.75	42.75	42.75	0
[GROUP=MAX_MARKER] mv2_vbuf_freed (Number of VBUFs freed)	77,020.583	1.934	49,644.5	30,021.234	39,832.867	9,811.633
[GROUP=MAX_MARKER] mv2_vbuf_inuse (Number of VBUFs inuse)	1.715	0.062	28.039	26.828	27.434	0.605
[GROUP=MAX_MARKER] mv2_vbuf_inuse_array (Number of VBUFs i	0.003	0.008	0.414	0.414	0.414	0
[GROUP=MAX_MARKER] mv2_vbuf_max_use (Maximum number of V	8.418	0.172	49.906	48.113	48.979	0.788
[GROUP=MAX_MARKER] mv2_vbuf_max_use_array (Maximum numb(	65.251	0.473	138.637	137.465	138.051	0.586
[GROUP=MAX_MARKER] num_free_calls (Number of MPIT_free calls	91,890.897	1.996	58,016.719	34,054.004	46,035.361	11,981.357
[GROUP=MAX_MARKER] num_malloc_calls (Number of MPIT_malloc	97,974.531	1.996	61,368.258	36,798.004	49,083.131	12,285.127
[GROUP=MAX_MARKER] num_memalign_calls (Number of MPIT_men	119.366	0.59	208.258	196.48	202.369	5.889
[GROUP=MAX_MARKER] posted_recvq_length (length of the poste	6.916	0.695	10.934	9.188	9.947	0.836
[GROUP=MAX_MARKER] posted_recvq_match_attempts (number (	1,992,098.262	2	1,263,322.938	728,775.324	996,049.131	267,273.807
[GROUP=MAX_MARKER] time_failed_matching_postedq (total time	159,308.051	2	100,101.18	59,206.871	79,654.025	20,447.154
[GROUP=MAX_MARKER] time_matching_unexpectedq (total time s	159,310.516	2	100,103.551	59,206.965	79,655.258	20,448.293
[GROUP=MAX_MARKER] unexpected_recvq_buffer_size (total buff	101.901	0.039	2,616.344	2,600.969	2,608.656	7.688
[GROUP=MAX_MARKER] unexpected_recvq_length (length of the u	1.186	0.398	3.066	2.887	2.977	0.09
[GROUP=MAX_MARKER] unexpected_recvq_match_attempts (num	50,195.291	1.984	32,140.996	18,459.695	25,295.265	6,721.212
[GROUP=MIN_MARKER] Message size for all-reduce	8	1	8	8	8	0

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

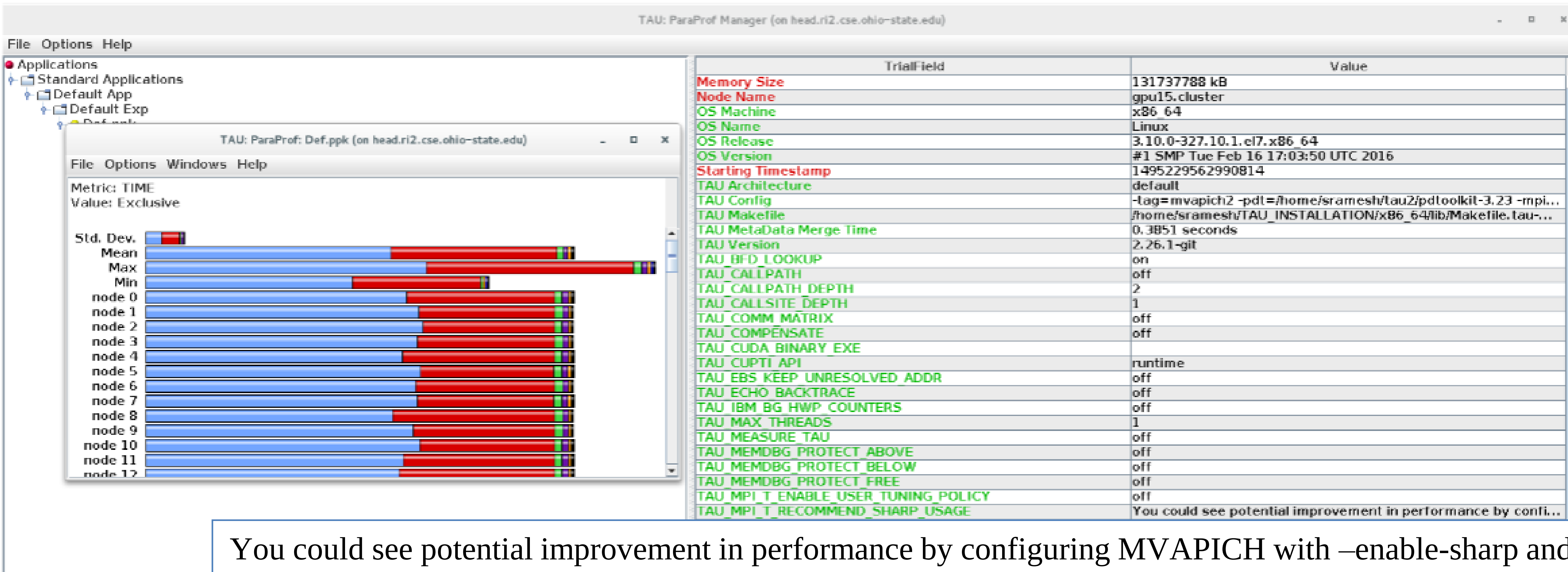
- Experiments with AmberMD
  - Stampede, a 6400 node Infiniband cluster at TACC
  - Stampede compute node: two Xeon E5- 2680 8-core “Sandy Bridge” processors and one first-generation Intel Xeon Phi SE10P KNC MIC
  - All our experiments using pure MPI on the Xeon host with 16 MPI processes on a node (1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



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

- Performance improvement for miniAMR

Run	***# Processes***	***Execution time***
Default	224	648
SHArP enabled	224	618

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)

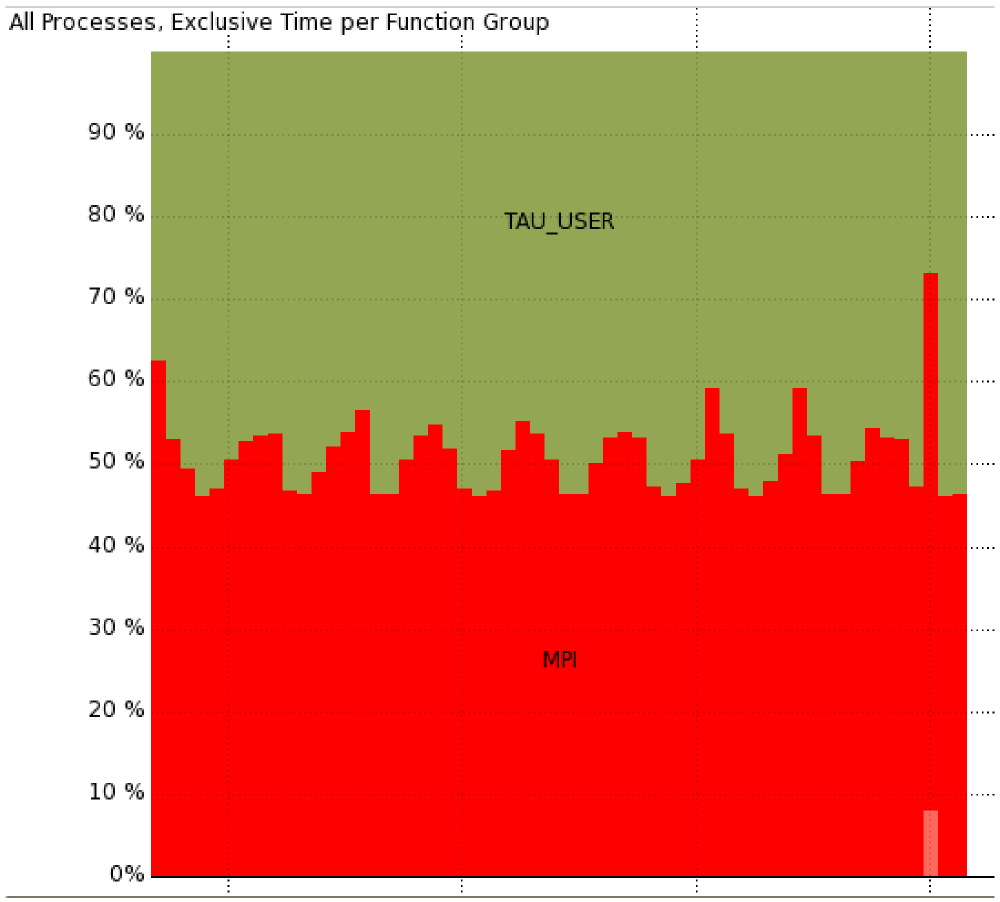
- Use of virtual buffers can offer significant performance improvement to applications performing heavy point-to-point communication, such as stencil based codes
- MVAPICH2 offers a number of PVARs that monitor the current usage level, availability of free VBUFs in different VBUF pools, maximum usage 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)

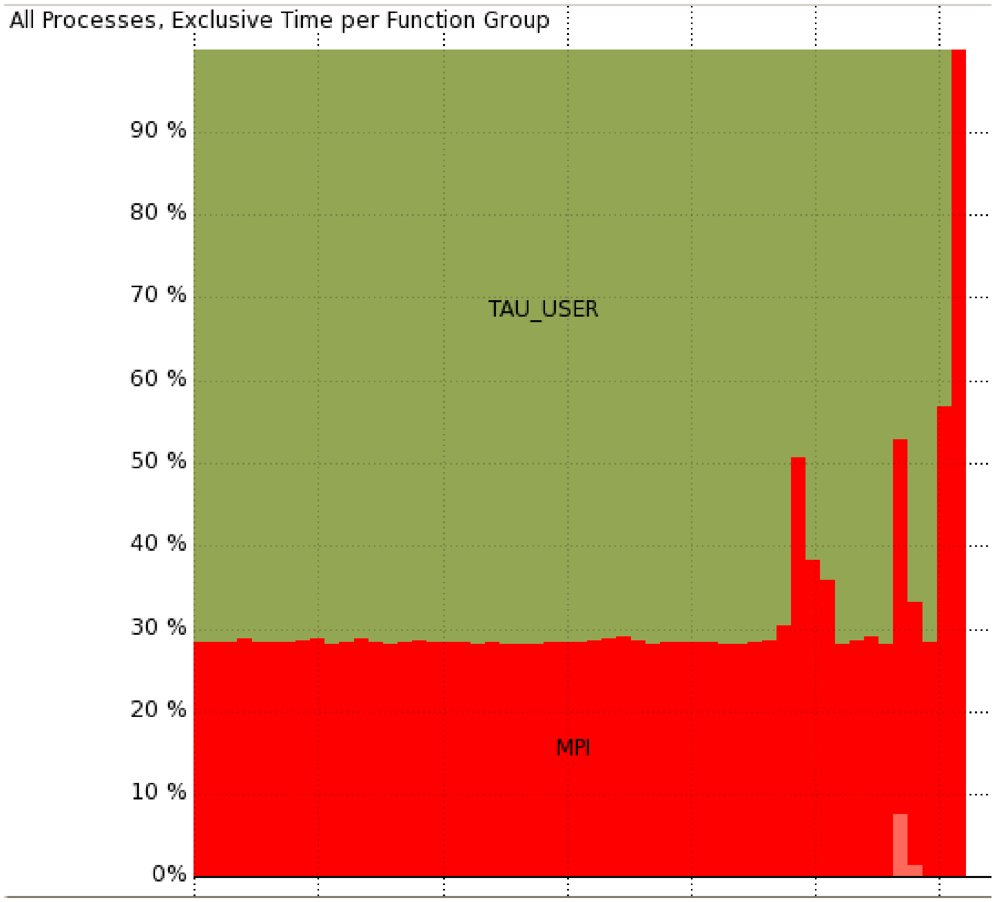
- PVARs of interest
  - *mv2_vbuf_allocated_array*
  - *mv2_vbuf_max_use_array*
  - *mv2_total_vbuf_memory*
- CVARs of interest
  - *MPIR_CVAR_IBA_EAGER_THRESHOLD*
  - *MPIR_CVAR_VBUF_TOTAL_SIZE*
  - *MPIR_CVAR_VBUF_POOL_CONTROL*
  - *MPIR_CVAR_VBUF_POOL_REDUCED_VALUE*
- Increasing the value of the Eager limit could lead to improved overlap between communication and computation as larger messages are sent eagerly
  - Overall execution time for the application may reduce

3DStencil

- Higher Eager threshold on 3DStencil application
  - Improves computation-communication overlap
  - Increases VBUF memory size



(a) Before Eager threshold tuning

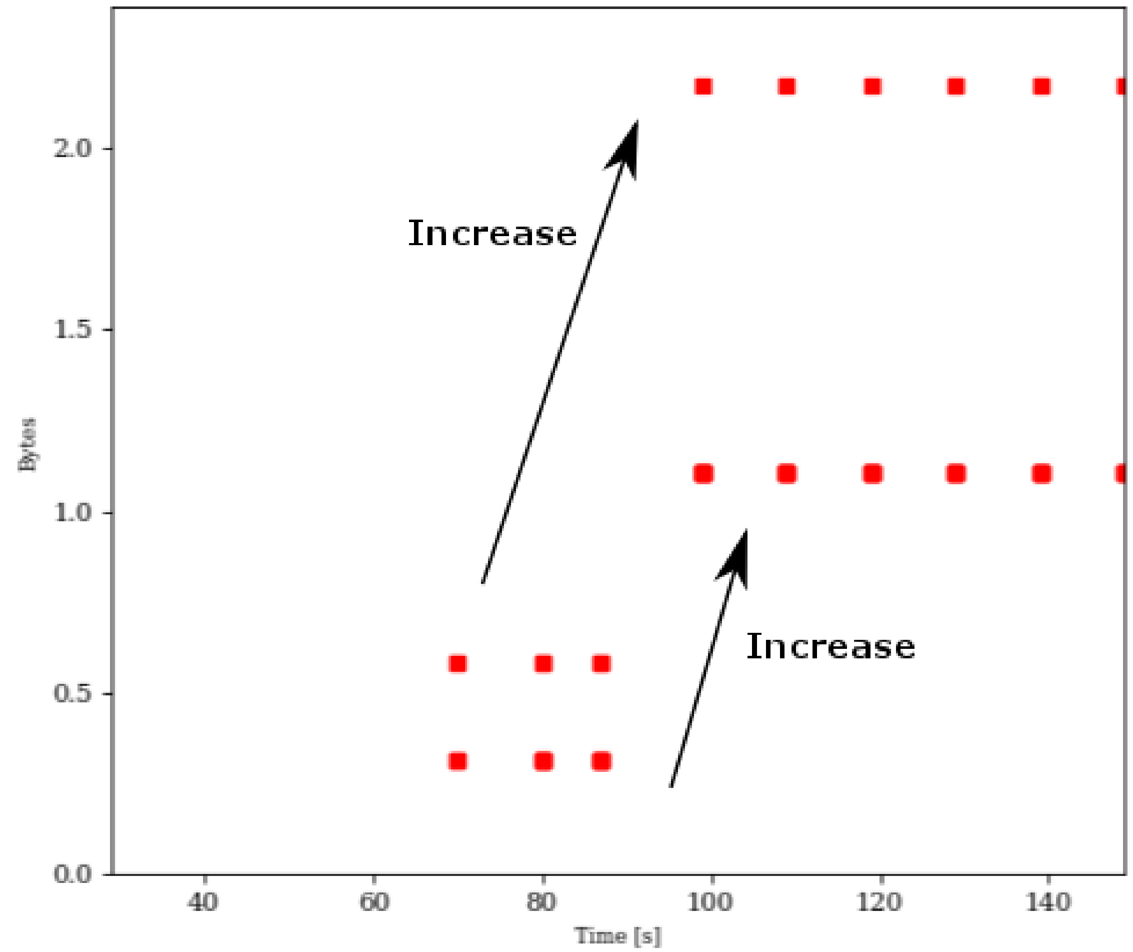


(b) After Eager threshold tuning

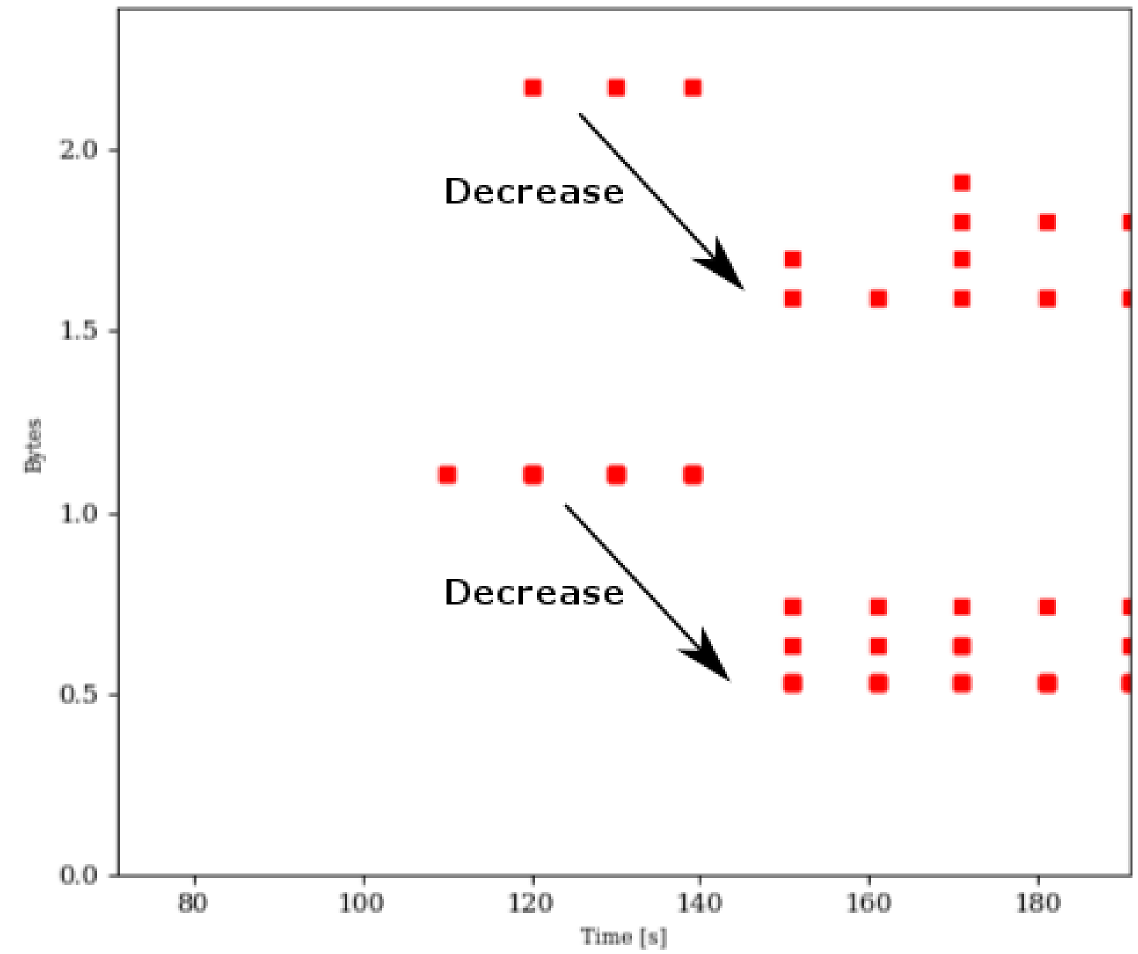
Run	Number of Processes	Message Size(Bytes)	Communication-Computation Overlap	Eager Threshold	Total VBUF Memory(Bytes)
Default	448	32,768	11.1	MVAPICH2 Default	1,436,574
Eager	448	32,768	68.7	33,000	2,573,345
TAU runtime tuning	448	32,768	69.7	33,000	1,208,782

AmberMD

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



(a) Increase in total VBUF memory when increasing Eager threshold



(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%

Run	Number of Processes	Eager Threshold	MD Timesteps	Application Runtime(Seconds)	Total VBUF Memory(Bytes)
Default	128	MVAPICH2 Default	4,000	210	695,150
Eager	128	64,000	4,000	129	768,188
TAU runtime autotuning	128	64,000	4,000	129	629,511

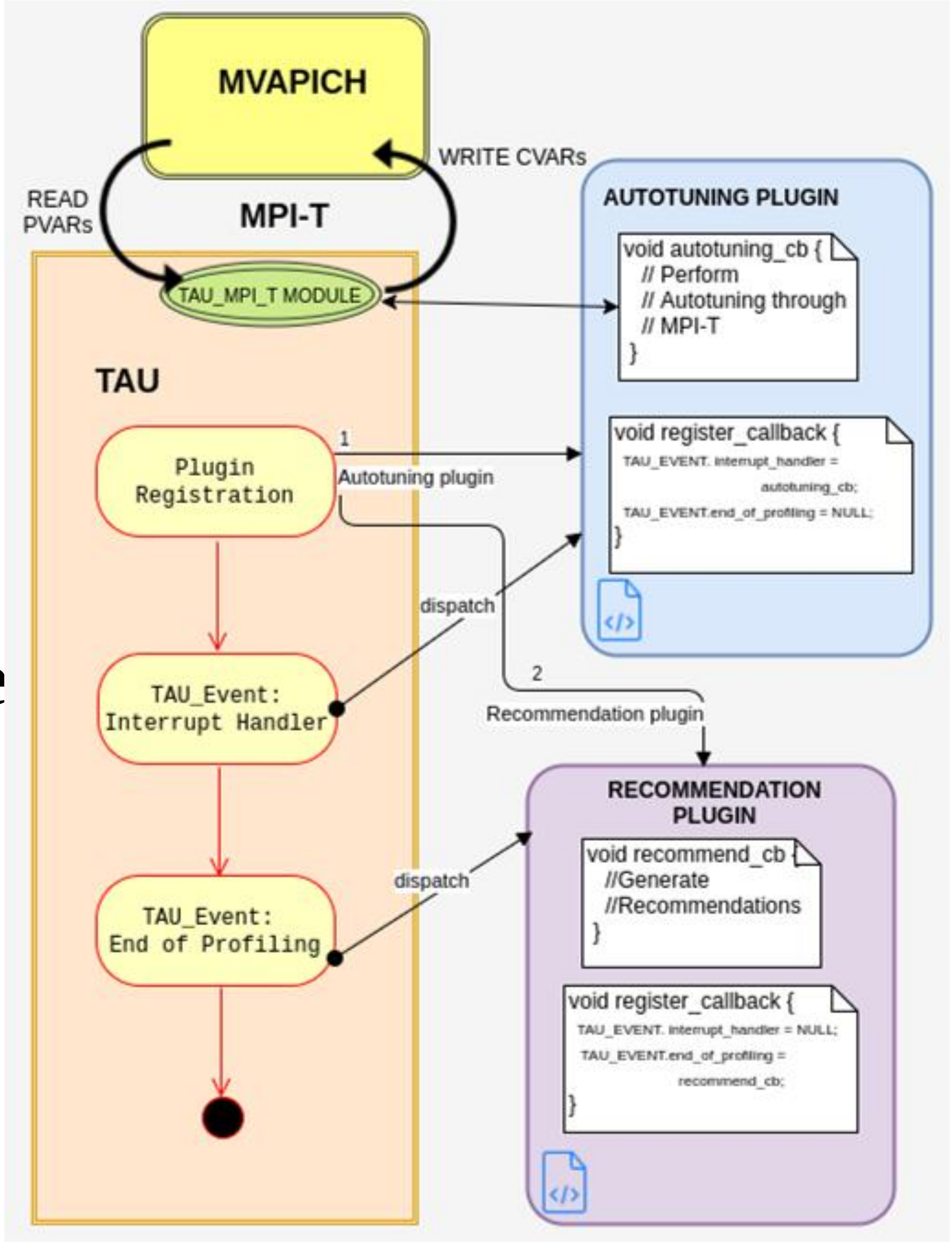
- Dynamically monitored VBUF usage and freed the unused ones, while maintaining same runtime

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

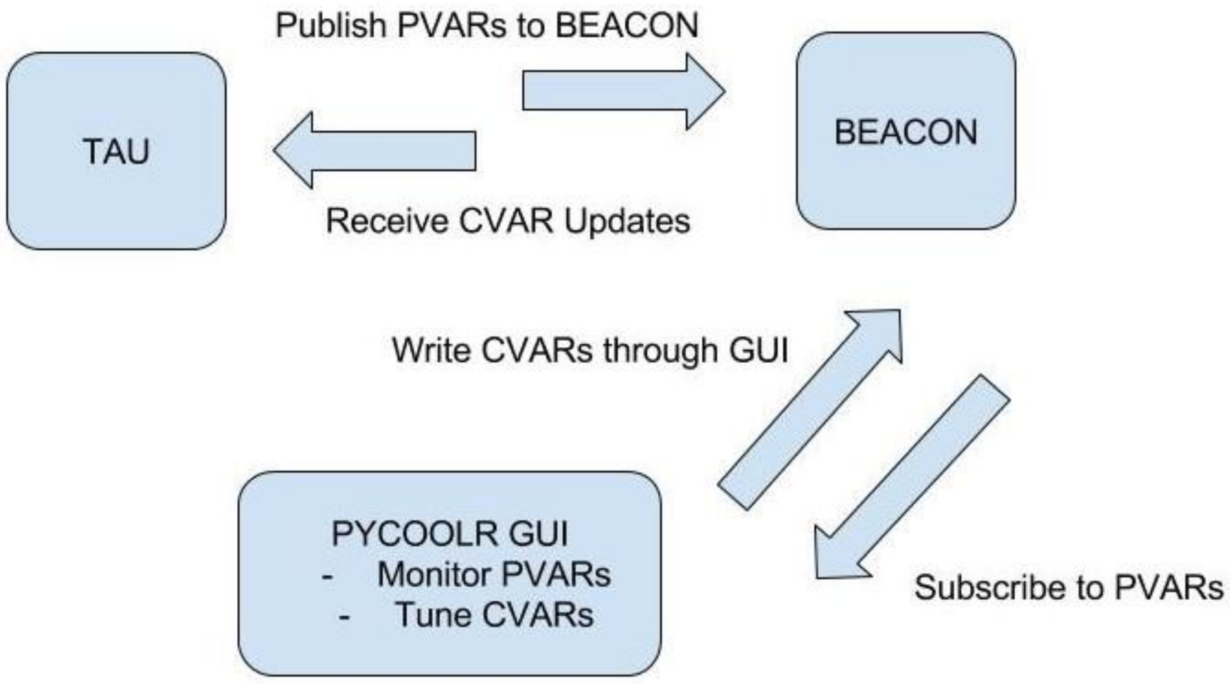
Plugin Architecture for Runtime Autotuning

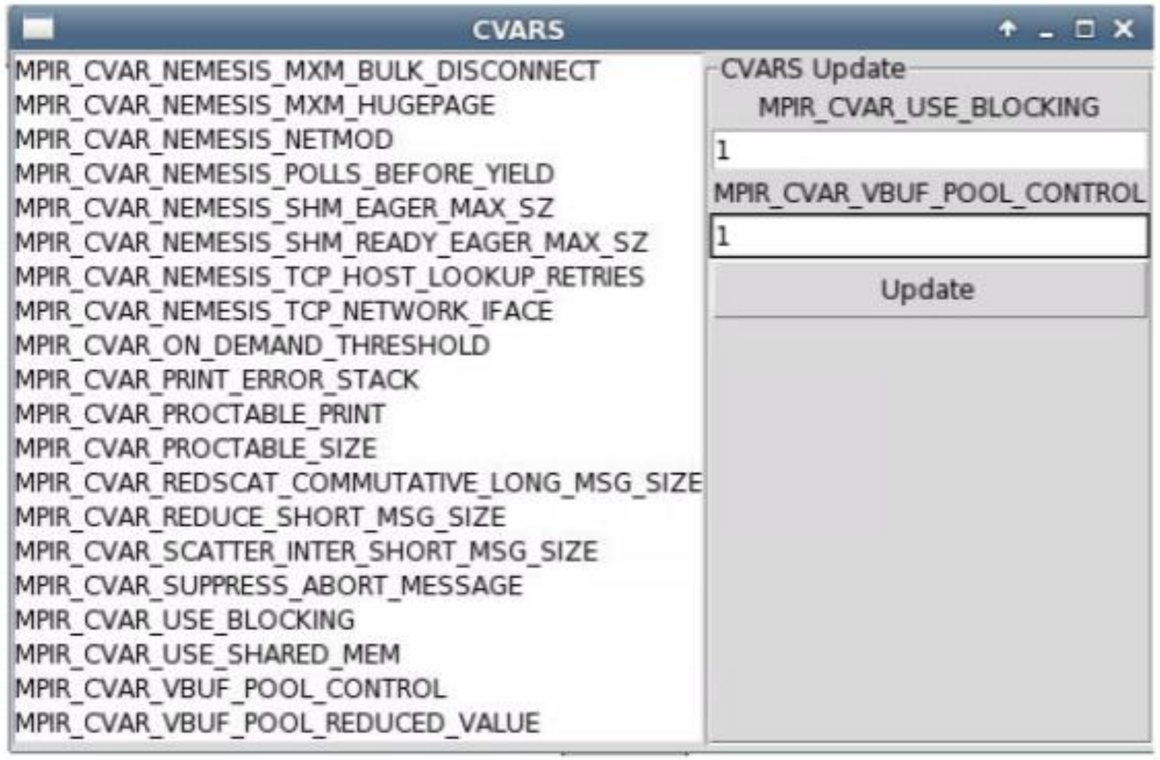
- TAU can be extended with a plugin that analyzes performance
- Based on policies, the plugin can make decisions about how control the runtime software
- Generic plugin architecture being developed
  - Policy specification
- Apply in MPI_T for MVAPICH tuning
- See poster!



Global Monitoring for Application Control

- Application tuning requires understanding distributed performance
- UO is building global monitoring framework
  - *BEACON* (Backplane for Event and Control Notication) 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

- S. Ramesh, A. Maheo, S. Shende, A. Malony, H. Subramoni, D. Panda, "MPI Performance Engineering with the MPI Tool Interface: the Integration of MVAPICH and TAU," EuroMPI/USA, September, 2017. Best paper finalist!
- *A. Maheo, "Integrate TAU, MVAPICH, and BEACON to Enable MPI Performance Monitoring," MUG 2017 poster.*
- *S. Ramesh, "Integrating MVAPICH and TAU through the MPI Tools Interface - A Plugin Architecture to Enable Autotuning and Recommendation Generation," MUG 2017 poster.*