Multi-Level Performance Instrumentation for Kokkos Applications using TAU

Sameer Shende, Nicholas Chaimov, Allen D. Malony
ParaTools, Inc.
{sameer,nchaimov,malony}@paratools.com

Neena Imam
Oak Ridge National Laboratory
imamm@ornl.gov

ProTools Workshop, SC19, Denver, CO
Sunday, November 17, 2019, 4pm – 4:15pm
Room 704-706
Slides:
http://tau.uoregon.edu/TAU_Kokkos_SC19.pdf
Motivation: Kokkos
https://github.com/kokkos/kokkos

• Provides abstractions for node level parallelism (X in MPI+X)
• Productive, portable, and performant shared-memory programming model
• Helps you create single source performance portable codes
• Provides data abstractions
• C++ API for expressing parallelism in your program
• Aggressive compiler transformations using C++ templates
• Low level code targets backends such as OpenMP, Pthread, CUDA
• Creates a problem for performance evaluation tools
• Gap: performance data and higher-level abstractions
• Solution: Kokkos profiling API for mapping performance data
• This talk: experience extending TAU to support Kokkos
TAU Performance System®

- Tuning and Analysis Utilities (20+ year project)
- Comprehensive performance profiling and tracing
  - Integrated, scalable, flexible, portable
  - Targets all parallel programming/execution paradigms
- Integrated performance toolkit
  - Instrumentation, measurement, analysis, visualization
  - Widely-ported performance profiling / tracing system
  - Performance data management and data mining
  - Open source (BSD-style license)
- Easy to integrate in application frameworks
- http://tau.uoregon.edu
Understanding Application Performance using TAU

• **How much time** is spent in each application routine and outer *loops*? Within loops, what is the contribution of each *statement*?

• **How many instructions** are executed in these code regions? Floating point, Level 1 and 2 *data cache misses*, hits, branches taken?

• **What is the memory usage** of the code? When and where is memory allocated/de-allocated? Are there any memory leaks?

• **What are the I/O characteristics** of the code? What is the peak read and write *bandwidth* of individual calls, total volume?

• **What is the extent of data transfer** between host and a GPU? In an Kokkos, OpenMP, OpenCL program.

• **What is the contribution of each phase** of the program? What is the time wasted/spent waiting for collectives, and I/O operations in Initialization, Computation, I/O phases?

• **How does the application scale**? What is the efficiency, runtime breakdown of performance across different core counts?
Types of Performance Profiles

**Flat profiles**
- Metric (e.g., time) spent in an event
- Exclusive/inclusive, # of calls, child calls, ...

**Callpath profiles**
- Time spent along a calling path (edges in callgraph)
- “main=>f1=>f2=>MPI_Send”
- Set the TAU_CALLPATH and TAU_CALLPATH_DEPTH environment variables

**Callsite profiles**
- Time spent along in an event at a given source location
- Set the TAU_CALLSITE environment variable

**Phase profiles**
- Flat profiles under a phase (nested phases allowed)
- Default “main” phase
- Supports static or dynamic (e.g. per-iteration) phases
Kokkos Profiling Interface

extern "C" void kokkosp_init_library(...);
extern "C" void kokkosp_finalize_library();
extern "C" void kokkosp_begin_parallel_for(const char* name, ...);
extern "C" void kokkosp_begin_parallel_reduce(const char* name, ...);
extern "C" void kokkosp_begin_parallel_scan(const char* name, ...);
/* corresponding end parallel constructs */
extern "C" void kokkosp_push_profile_region(const char* name);
extern "C" void kokkosp_pop_profile_region();
... /* other APIs for sections, data transfers, memory allocation, ...*/
TAU

- Tracks kernel names specified as the first parameter in parallel API
- When name is not specified, TAU uses the template instantiation
- TAU needs to demangle mangled names of C++ entities
- TAU maps Kokkos profiling regions to TAU phases
- In a TAU phase, all functions called directly/indirectly are flattened into a flat profile under the phase
- Other runtime system calls (CUDA, Pthread, OpenMP) are also tracked alongside Kokkos calls
- Multi-level instrumentation support in TAU can help us slice through multiple runtime layers
TAU Architecture and Workflow

**TAU Architecture**

**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
  - PerfDMF 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’s Support for Runtime Systems

**MPI**
- PMPI profiling interface
- MPI_T tools interface using performance and control variables

**Kokkos**
- Kokkos profiling API
- Push/pop interface for region, kernel execution interface

**Pthread**
- Captures time spent in routines per thread of execution

**OpenMP**
- OMPT tools interface to track salient OpenMP runtime events
- Opari source rewriter
- Preloading wrapper OpenMP runtime library when OMPT is not supported
TAU’s Support for Runtime Systems (contd.)

**OpenCL**
- OpenCL profiling interface
- Track timings of kernels

**OpenACC**
- OpenACC instrumentation API
- Track data transfers between host and device (per-variable)
- Track time spent in kernels

**CUDA**
- Cuda Profiling Tools Interface (CUPTI)
- Track data transfers between host and GPU
- Track access to uniform shared memory between host and GPU

**ROCm**
- Rocprofiler and Roctracer instrumentation interfaces
- Track data transfers and kernel execution between host and GPU

**Python**
- Python interpreter instrumentation API
- Tracks Python routine transitions as well as Python to C transitions
Examples of Multi-Level Instrumentation

**MPI + OpenMP**
- MPI_T + PMPI + OMPT may be used to track MPI and OpenMP

**MPI + CUDA**
- PMPI + CUPITI interfaces

**OpenCL + ROCm**
- Rocprofiler + OpenCL instrumentation interfaces

**Kokkos + OpenMP**
- Kokkos profiling API + OMPT to transparently track events

**Kokkos + pthread + MPI**
- Kokkos + pthread wrapper interposition library + PMPI layer

**Python + CUDA**
- Python + CUPITI + pthread profiling interfaces (e.g., Tensorflow, PyTorch)

**MPI + OpenCL**
- PMPI + OpenCL profiling interfaces
Simplifying the use of TAU!

Uninstrumented code:

• % make
• % mpirun -np 256 ./a.out

With TAU using event based sampling (EBS):

• % mpirun -np 256 tau_exec -ebs ./lu.B.64
• % paraprof (GUI)
• % pprof -a | more

NOTE:

• Requires dynamic executables (-dynamic link flag on Cray XC systems).
• Source code should be compiled with -g for access to symbol table.
• Kokkos support is on by default in tau_exec
TAU Execution Command (tau_exec)

Uninstrumented execution
• % mpirun -np 256 ./a.out

Track GPU operations
• % mpirun –np 256 tau_exec –rocm ./a.out
• % mpirun –np 256 tau_exec –cupti ./a.out
• % mpirun –np 256 tau_exec –opencl ./a.out
• % mpirun –np 256 tau_exec –openacc ./a.out

Track MPI performance
• % mpirun -np 256 tau_exec ./a.out

Track I/O, and MPI performance (MPI enabled by default)
• % mpirun -np 256 tau_exec -io ./a.out

Track OpenMP and MPI execution (using OMPT for Intel v19)
• % export TAU_OMPT_SUPPORT_LEVEL=full;
  % mpirun –np 256 tau_exec -T openmp,ompt,v5,mpi -ompt ./a.out

Track memory operations
• % export TAU_TRACK_MEMORY_LEAKS=1
• % mpirun –np 256 tau_exec –memory_debug ./a.out (bounds check)

Use event based sampling (compile with –g)
• % mpirun –np 256 tau_exec –ebs ./a.out
• Also -ebs_source=<PAPI_COUNTER> -ebs_period=<overflow_count>
  -ebs_resolution=<file | function | line>
Kokkos API use in ExaMiniMD

```c
void CommMPI::update_halo() {
    Kokkos::Profiling::pushRegion("Comm::update_halo");
    N_ghost = 0;
system;
    pack_buffer_update = t_buffer_update((T_X_FLOAT*)pack_buffer.data(),pack_indices_all.extent(1));
    unpack_buffer_update = t_buffer_update((T_X_FLOAT*)unpack_buffer.data(),pack_indices_all.extent(1));
    for(phase = 0; phase<6; phase++) {
        pack_indices = Kokkos::subview(pack_indices_all,phase,Kokkos::ALL());
        if(proc_grid[phase/2]>1) {
            Kokkos::parallel_for("CommMPI::halo_update_pack",
                Kokkos::RangePolicy<TagHaloUpdatePack, Kokkos::IndexType<T_INT>> >(0,proc_num_send[phase]),
                *this);
            MPI_Request request;
            MPI_Status status;
            MPI_Recv(unpack_buffer.data(),proc_num_recv[phase]*sizeof(T_X_FLOAT)/sizeof(int),MPI_INT, proc_neighbors_recv[phase],1000002,MPI_COMM_WORLD,&request);
            MPI_Send (pack_buffer.data(),proc_num_send[phase]*sizeof(T_X_FLOAT)/sizeof(int),MPI_INT, proc_neighbors_send[phase],1000002,MPI_COMM_WORLD);
            s = *system;
            MPI_Wait(&request,&status);
            const int count = proc_num_recv[phase];
            if(unpack_buffer_update.extent(0)<count) {
                unpack_buffer_update = t_buffer_update((T_X_FLOAT*)unpack_buffer.data(),count);
            }
            Kokkos::parallel_for("CommMPI::halo_update_unpack",
                Kokkos::RangePolicy<TagHaloUpdateUnpack, Kokkos::IndexType<T_INT>> >(0,proc_num_recv[phase]),
                *this);
        } else {
            //printf("HalUpdateCopy: %i %i
",phase,proc_num_send[phase],pack_indices.extent(0));
            Kokkos::parallel_for("CommMPI::halo_update_self",
                Kokkos::RangePolicy<TagHaloUpdateSelf, Kokkos::IndexType<T_INT>> >(0,proc_num_send[phase]),
                *this);
        }
        N_ghost += proc_num_recv[phase];
    }
    Kokkos::Profiling::popRegion();
};
```
ExaMiniMD: TAU Phase

Comm::update_halo phase in TAU ParaProf’s Thread Statistics Table
## ExaMiniMD: ParaProf Node Window

<table>
<thead>
<tr>
<th>Phase: Comm::update_halo</th>
<th>Metric: TIME</th>
<th>Value: Exclusive</th>
<th>Units: seconds</th>
<th>Time</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>29.268</td>
<td><strong>MPI_Send()</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.875</td>
<td>OpenMP_Sync.Region_Barrier 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.58</td>
<td>OpenMP_Sync.Region_Barrier void 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.489</td>
<td>OpenMP_Sync.Region_Barrier 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.08</td>
<td>OpenMP_Sync.Region_Barrier void 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.041</td>
<td>OpenMP_Sync.Region_Barrier 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.004</td>
<td>Comm::update_halo</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.003</td>
<td>Kokkos::parallel_for Comm:: halo_update_self [device=0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.002</td>
<td>Kokkos::parallel_for Comm:: halo_update.unpack [device=0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.002</td>
<td>Kokkos::parallel_for Comm:: halo_update.pack [device=0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7.8E-4</td>
<td>MPI_Wait()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7.2E-4</td>
<td>MPI_Irecv()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6.9E-4</td>
<td>OpenMP_Sync.Region 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.4E-4</td>
<td>OpenMP_Sync.Region 1</td>
</tr>
</tbody>
</table>
Event-based Sampling (EBS): CabanaMD

EBS with Kokkos API

Kokkos sample within Comm::update_halo

Kokkos sample within top-level application code

Instrumented Kokkos::parallel_for

Instrumented Kokkos::parallel_reduce
CabanaMD: CUDA Events

Phase: .TAU application
Name: void
Kokkos::Impl::cuda_parallel_launch_constant_memory<Kokkos::Impl::ParallelFor
<ForceLJ<Cabana::VerletList<Kokkos::Device<Kokkos::Cuda,
Kokkos::CudaUVMSpace>, Cabana::FullNeighborTag, Cabana::VerletLayout2D> >,
Kokkos::RangePolicy<ForceLJ<Cabana::VerletList<Kokkos::Device<Kokkos::Cuda
, Kokkos::CudaUVMSpace>, Cabana::FullNeighborTag, Cabana::VerletLayout2D>
> ::TagFullNeigh, Kokkos::IndexType<int> >, Kokkos::Cuda> ()
Metric Name: TAUGPU_TIME
Value: Exclusive
Units: microseconds

152387 151120
max min std. dev.
449.383
mean
node 0, thread 1
node 1, thread 1
node 2, thread 1
node 3, thread 1
Jumpshot Trace Visualizer
Vampir [TU Dresden] Timeline Display

% export TAU_TRACE=1; export TAU_TRACE_FORMAT=otf2
% mpirun -np 16 tau_exec ./a.out
% vampir traces.otf2
An API model for other runtimes?

- Kokkos profiling interface is very elegant
- Kokkos API calls are defined by the tools
- Kokkos maintains compatibility between its releases
- No need for a header file, no #defines, type names
- Additional calls may be added to the API, but are not necessary for an older version of the tool to support a new version of Kokkos
- No need for tool to be compiled with a given version of Kokkos
- At startup, if KOKKOS_PROFILE_LIBRARY environment variable is defined, it loads the library, if not, profiling calls are disabled
- Profiling hooks are activated by loading the agent library
- Hooks may be disabled during configuration, enabled by default
- Is this a good model for other runtimes to adopt?
- Is there a need for a single tool to provide access to events across different runtimes/programming models?
Acknowledgment

“This work was supported by the United States Department of Defense (DoD) and used resources of the Computational Research and Development Programs, the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory, and the Performance Research Laboratory at the University of Oregon. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work benefited from access to the University of Oregon high performance computer, Talapas. The authors would like to thank Sam Reeve (LLNL), for his assistance with CabanaMD.”
Support Acknowledgments

US Department of Energy (DOE)
- ORNL
- Office of Science contracts, ECP
- SciDAC, LBL contracts
- LLNL-LANL-SNL ASC/NNSA contract
- Battelle, PNNL and ANL contract

US Department of Defense (DoD)
- HPCMP, ORNL

National Science Foundation (NSF)
- SI2-SSI, CSSI

NASA

CEA, France

Partners:
- University of Oregon
- The Ohio State University
- ParaTools, Inc.
- University of Tennessee, Knoxville
Acknowledgment

“This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation’s exascale computing imperative.”

http://exascaleproject.org
Download TAU

http://tau.uoregon.edu
http://taucommander.com
https://e4s.io

[ E4S: Extreme-Scale Scientific Software Stack]

For more info on E4S: E4S BoF, Tues, 12:15pm, Room #405-407

Free download, open source, BSD license
Reference
Installing and Configuring TAU

• Installing PDT:
  
  • wget tau.uoregon.edu/pdt_lite.tgz
  
  • ./configure –prefix=<dir>; make ; make install

• Installing TAU:
  
  • wget tau.uoregon.edu/tau.tgz;  
    tar zxf tau.tgz; cd tau-2.<ver>
  
  • wget http://tau.uoregon.edu/ext.tgz ; tar xf ext.tgz
  
  • ./configure -bfd=download -pdt=<dir>
    -iowrapper -mpi -dwarf=download -unwind=download –
    otf=download –papi=<dir>
  
  make install

• Using TAU:
  
  • export TAU_MAKEFILE=<taudir>/<arch>/lib/Makefile.tau-
    <TAGS>
  
  • make CC=tau_cc.sh  CXX=tau_cxx.sh  F90=tau_f90.sh
Compile-Time Options

Optional parameters for the TAU_OPTIONS environment variable:
% tau_compiler.sh

- **-optVerbose**
  Turn on verbose debugging messages

- **-optCompInst**
  Use compiler based instrumentation

- **-optNoCompInst**
  Do not revert to compiler instrumentation if source instrumentation fails.

- **-optTrackIO**
  Wrap POSIX I/O call and calculates vol/bw of I/O operations
  (Requires TAU to be configured with `–iowrapper`)

- **-optTrackGOMP**
  Enable tracking GNU OpenMP runtime layer (used without `–opari`)

- **-optMemDbg**
  Enable runtime bounds checking (see TAU_MEMDBG_* env vars)

- **-optKeepFiles**
  Does not remove intermediate .pdb and .inst.* files

- **-optPreProcess**
  Preprocess sources (OpenMP, Fortran) before instrumentation

- **-optTauSelectFile=’<file>’**
  Specify selective instrumentation file for `tau_instrumentor`

- **-optTauWrapFile=’<file>’**
  Specify path to `link_options.tau` generated by `tau_gen_wrapper`

- **-optHeaderInst**
  Enable Instrumentation of headers

- **-optTrackUPCR**
  Track UPC runtime layer routines (used with `tau_upc.sh`)

- **-optLinking=’’**
  Options passed to the linker. Typically
  `$(TAU_MPI_FLIBS) $(TAU_LIBS) $(TAU_CXXLIBS)`

- **-optCompile=’’**
  Options passed to the compiler. Typically
  `$(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)`

- **-optPdtF95Opts=’’**
  Add options for Fortran parser in PDT (f95parse/gfparse) …
Optional parameters for the TAU_OPTIONS environment variable:

% tau Compiler.sh

- `optShared` Use TAU’s shared library (libTAU.so) instead of static library (default)
- `optPdtCxxOpts=""` Options for C++ parser in PDT (cxxparse).
- `optPdtF90Parser=""` Specify a different Fortran parser
- `optPdtCleanscapeParser` Specify the Cleanscape Fortran parser instead of GNU gfpars
- `optTau=""` Specify options to the tau_instrumentor
- `optTrackDMAPP` Enable instrumentation of low-level DMAPP API calls on Cray
- `optTrackPthread` Enable instrumentation of pthread calls

See tau Compiler.sh for a full list of TAU_OPTIONS.
## TAU’s Runtime Environment Variables

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAU_TRACE</td>
<td>0</td>
<td>Setting to 1 turns on tracing</td>
</tr>
<tr>
<td>TAU_CALLPATH</td>
<td>0</td>
<td>Setting to 1 turns on callpath profiling</td>
</tr>
<tr>
<td>TAU_TRACK_MEMORY_FOO PRINT</td>
<td>0</td>
<td>Setting to 1 turns on tracking memory usage by sampling periodically the resident set size and high water mark of memory usage</td>
</tr>
<tr>
<td>TAU_TRACK_POWER</td>
<td>0</td>
<td>Tracks power usage by sampling periodically.</td>
</tr>
<tr>
<td>TAU_CALLPATH_DEPTH</td>
<td>2</td>
<td>Specifies depth of callpath. Setting to 0 generates no callpath or routine information, setting to 1 generates flat profile and context events have just parent information (e.g., Heap Entry: foo)</td>
</tr>
<tr>
<td>TAU_SAMPLING</td>
<td>1</td>
<td>Setting to 1 enables event-based sampling.</td>
</tr>
<tr>
<td>TAU_TRACK_SIGNALS</td>
<td>0</td>
<td>Setting to 1 generate debugging callstack info when a program crashes</td>
</tr>
<tr>
<td>TAU_COMM_MATRIX</td>
<td>0</td>
<td>Setting to 1 generates communication matrix display using context events</td>
</tr>
<tr>
<td>TAU_THROTTLE</td>
<td>1</td>
<td>Setting to 0 turns off throttling. Throttles instrumentation in lightweight routines that are called frequently</td>
</tr>
<tr>
<td>TAU_THROTTLE_NUMCALLS</td>
<td>100000</td>
<td>Specifies the number of calls before testing for throttling</td>
</tr>
<tr>
<td>TAU_THROTTLE_PERCALL</td>
<td>10</td>
<td>Specifies value in microseconds. Throttle a routine if it is called over 100000 times and takes less than 10 usec of inclusive time per call</td>
</tr>
<tr>
<td>TAU_CALLSITE</td>
<td>0</td>
<td>Setting to 1 enables callsite profiling that shows where an instrumented function was called. Also compatible with tracing.</td>
</tr>
<tr>
<td>TAU_PROFILE_FORMAT</td>
<td>Profile</td>
<td>Setting to “merged” generates a single file. “snapshot” generates xml format</td>
</tr>
<tr>
<td>TAU_METRICS</td>
<td>TIME</td>
<td>Setting to a comma separated list generates other metrics. (e.g., ENERGY,TIME,P_VIRTUAL_TIME,PAPI_FP_INS,PAPI_NATIVE_&lt;event&gt;:&lt;subevent&gt;)</td>
</tr>
</tbody>
</table>
## Runtime Environment Variables

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAU_TRACE</td>
<td>0</td>
<td>Setting to 1 turns on tracing</td>
</tr>
<tr>
<td>TAU_TRACE_FORMAT</td>
<td>Default</td>
<td>Setting to “otf2” turns on TAU’s native OTF2 trace generation (configure with — otf=download)</td>
</tr>
<tr>
<td>TAU_EBS_UNWIND</td>
<td>0</td>
<td>Setting to 1 turns on unwinding the callstack during sampling (use with tau_exec —ebs or TAU_SAMPLING=1)</td>
</tr>
<tr>
<td>TAU_EBS_RESOLUTION</td>
<td>line</td>
<td>Setting to “function” or “file” changes the sampling resolution to function or file level respectively.</td>
</tr>
<tr>
<td>TAU_TRACK_LOAD</td>
<td>0</td>
<td>Setting to 1 tracks system load on the node</td>
</tr>
<tr>
<td>TAU_SELECT_FILE</td>
<td>Default</td>
<td>Setting to a file name, enables selective instrumentation based on exclude/include lists specified in the file.</td>
</tr>
<tr>
<td>TAU_OMPT_SUPPORT_LEVEL</td>
<td>basic</td>
<td>Setting to “full” improves resolution of OMPT TR6 regions on threads 1.. N-1. Also, “lowoverhead” option is available.</td>
</tr>
<tr>
<td>TAU_OMPT_RESOLVE_ADDRESS_EAGERLY</td>
<td>1</td>
<td>Setting to 1 is necessary for event based sampling to resolve addresses with OMPT. Setting to 0 allows the user to do offline address translation.</td>
</tr>
</tbody>
</table>
# Runtime Environment Variables

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAU_TRACK_MEMORY_LEAKS</td>
<td>0</td>
<td>Tracks allocates that were not de-allocated (needs –optMemDbg or tau_exec –memory)</td>
</tr>
<tr>
<td>TAU_EBS_SOURCE</td>
<td>TIME</td>
<td>Allows using PAPI hardware counters for periodic interrupts for EBS (e.g., TAU_EBS_SOURCE=PAPI_TOT_INS when TAU_SAMPLING=1)</td>
</tr>
<tr>
<td>TAU_EBS_PERIOD</td>
<td>100000</td>
<td>Specifies the overflow count for interrupts</td>
</tr>
<tr>
<td>TAU_MEMDBG_ALLOC_MIN/MAX</td>
<td>0</td>
<td>Byte size minimum and maximum subject to bounds checking (used with TAU_MEMDBG_PROTECT_*)</td>
</tr>
<tr>
<td>TAU_MEMDBG_OVERHEAD</td>
<td>0</td>
<td>Specifies the number of bytes for TAU’s memory overhead for memory debugging.</td>
</tr>
<tr>
<td>TAU_MEMDBG_PROTECT_BELOW/ABOVE</td>
<td>0</td>
<td>Setting to 1 enables tracking runtime bounds checking below or above the array bounds (requires –optMemDbg while building or tau_exec –memory)</td>
</tr>
<tr>
<td>TAU_MEMDBG_ZERO_MALLOC</td>
<td>0</td>
<td>Setting to 1 enables tracking zero byte allocations as invalid memory allocations.</td>
</tr>
<tr>
<td>TAU_MEMDBG_PROTECT_FREE</td>
<td>0</td>
<td>Setting to 1 detects invalid accesses to deallocated memory that should not be referenced until it is reallocated (requires –optMemDbg or tau_exec –memory)</td>
</tr>
<tr>
<td>TAU_MEMDBG_ATTEMPT_CONTINUE</td>
<td>0</td>
<td>Setting to 1 allows TAU to record and continue execution when a memory error occurs at runtime.</td>
</tr>
<tr>
<td>TAU_MEMDBG_FILL_GAP</td>
<td>Undefined</td>
<td>Initial value for gap bytes</td>
</tr>
<tr>
<td>TAU_MEMDBG_ALINGMENT</td>
<td>Sizeof(int)</td>
<td>Byte alignment for memory allocations</td>
</tr>
<tr>
<td>TAU_EVENT_THRESHOLD</td>
<td>0.5</td>
<td>Define a threshold value (e.g., .25 is 25%) to trigger marker events for min/max</td>
</tr>
</tbody>
</table>