TAU Update

OpenSHMEM in the Era of Exascale

BoF, SC18

Wednesday, Nov 14, 2018, 5:15pm – 6:45pm, C145

Sameer Shende

ParaTools, Inc.

sameer@paratools.com

http://tau.uoregon.edu

http://taucommander.com

Acknowledgment

This material is based upon work supported by Extreme Scale Systems Center, ORNL under subcontract 4000146681 Mod 5.

FY18 Accomplishments

- Memory allocation tracking feature in the TAU Peformance System®:
  - Track memory allocations by object type.
  - Integrated with Open MPI implementation of OpenSHMEM.
- Improvements the TAU Performance System®:
  - Support for hybrid CUDA+OpenSHMEM applications.
  - Support for CUPTI derived metric API.
  - Support for new OpenSHMEM API calls added in OpenSHMEM 1.4 specification.
  - Support for multithreaded OpenSHMEM application using shmem_init_thread.
- Improvements in TAU Commander:
  - Support for hybrid CUDA+OpenSHMEM and multithreaded OpenSHMEM applications added to user interface.
- Paper accepted to OpenSHMEM'18: *Tracking Memory Usage in OpenSHMEM Runtimes with the TAU Performance System.*

Memory Tracking

- Goal is to evaluate scalability of OpenSHMEM runtimes in terms of runtime memory usage as the number of PEs increases.
  - By keeping arrays of sizes proportional to the number of PEs, an OpenSHMEM implementation may be limited in its scalability to millions of PEs.
- We extended TAU to track memory allocations within OpenSHMEM runtimes.
  - Trigger atomic events with a value of memory usage from each PE.
  - Trigger **separate** events according to the **data type** of the allocated objects, allowing determination of scaling behavior for different runtime object types.
- Postprocess data to chart memory usage by object type as number of PEs grows.

Allocation Classes

- New calls in TAU for tracking allocations
  - Track “flat” allocations (no relationships maintained)
    - Tau_track_class_allocation(name, size)
  - Track hierarchical allocations
    - Maintain allocation stack for context
    - Tau_start_class_allocation(name, size, include_in_parent)
    - Tau_stop_class_allocation(name, write_record)
  - Included in profile alongside timing data
  - Option to use context events: show *where* allocations occurred in the runtime
    - Two context stacks: timer stack and allocation stack
    - export TAU_MEM_CONTEXT=1
  - Default weak empty implementation allows enabling and disabling instrumentation at runtime.

```
Tau_start_class_allocation(“a”, 10, 0);
Tau_start_class_allocation(“b”, 25, 0);
Tau_stop_class_allocation(“b”, 1);
Tau_stop_class_allocation(“a”, 1);
Tau_start_class_allocation(“b”, 10, 0);
Tau_stop_class_allocation(“b”, 1);
```

→ 10 bytes allocated in object of type A

→ 25 bytes allocated in object of type B (child of A)

→ 10 bytes allocated in object of type B (not child)

Stored in profile:

```
alloc a          10
alloc b          35
alloc b <= a     25
```

Instrumenting Open MPI

- OpenMPI OPAL object system allows centralized instrumentation of allocations of OPAL objects
  - Insert `Tau_start_class_allocation`, `Tau_stop_class_allocation` into `opal_obj_new` in `opal/class/opal_object.h`
- Tracking child objects requires manual instrumentation at the point of allocation
  - Dynamically-allocated members are allocated outside the constructor
  - Accomplished with dummy allocation regions
    - Reopen allocation region with `Tau_start_class_allocation` as normal.
    - Record child allocations
    - Close parent allocation region with write_record = 0

Tracking Flat Allocations

- Tracking allocations by type requires one line of code inserted into Open MPI runtime
  - *static inline* prevents use of library wrapper

```
static inline opal_object_t *opal_obj_new(opal_class_t * cls)
{
  opal_object_t *object;
  assert(cls->cls_sizeof >= sizeof(opal_object_t));
  Tau_track_class_allocation(cls->cls_name, cls->cls_sizeof);
  [...]
}
```

Tracking Hierarchical Allocations

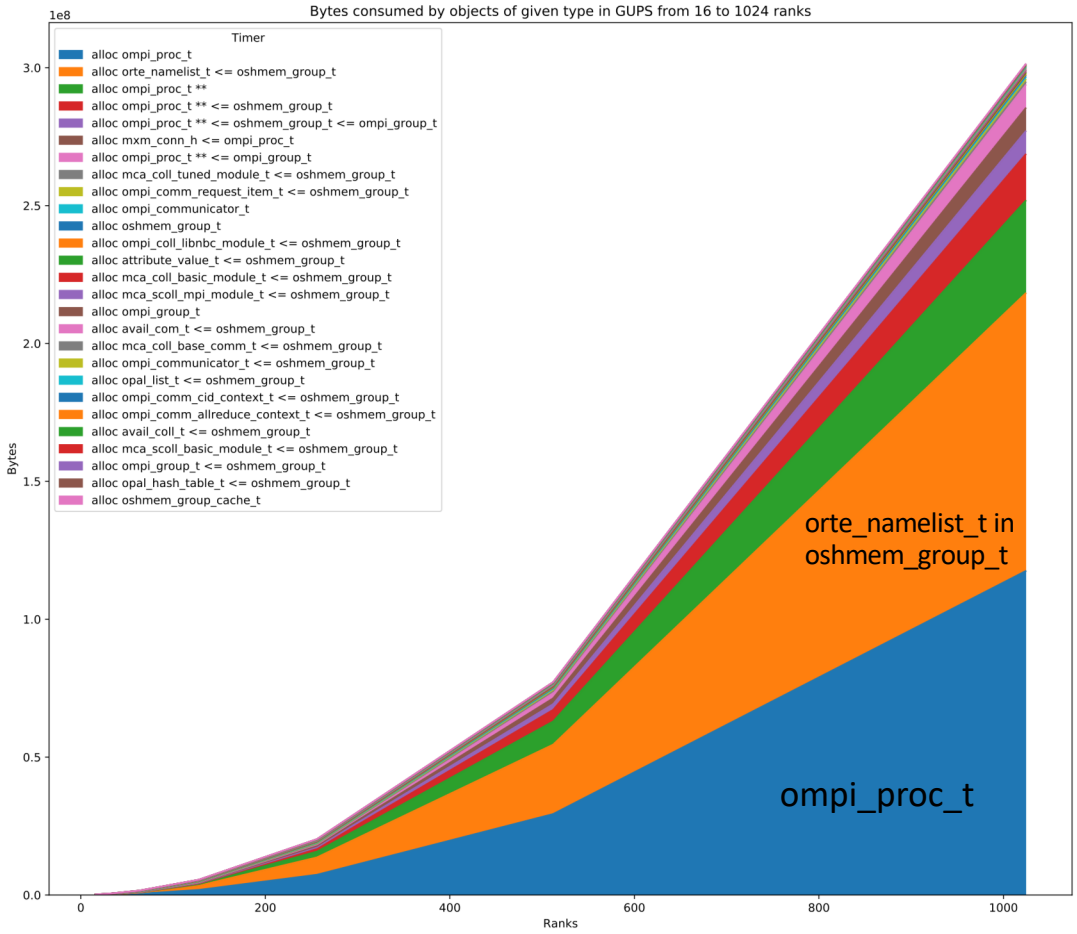
```
static inline opal_object_t *opal_obj_new(opal_class_t * cls
{
    opal_object_t *object;
    assert(cls->cls_sizeof >= sizeof(opal_object_t));

    Tau_start_class_allocation(cls->cls_name, cls->cls_sizeof, 0);

#if OPAL_WANT_MEMCHECKER
    object = (opal_object_t *) calloc(1, cls->cls_sizeof);
#else
    object = (opal_object_t *) malloc(cls->cls_sizeof);
#endif
    if (opal_class_init_epoch != cls->cls_initialized) {
        opal_class_initialize(cls);
    }
    if (NULL != object) {
        object->obj_class = cls;
        object->obj_reference_count = 1;
        opal_obj_run_constructors(object);
    }
    Tau_stop_class_allocation(cls->cls_name, 1);
    return object;
}
```

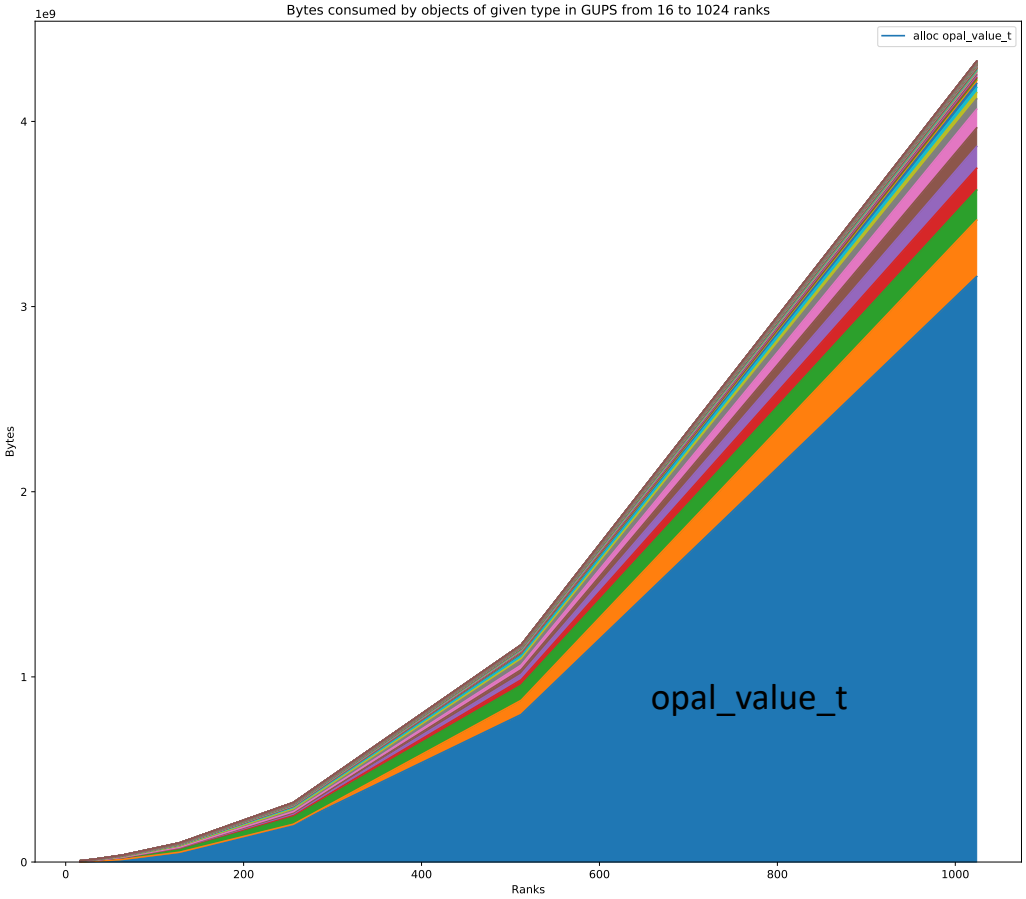
Allocations during constructors automatically attributed to enclosing allocation region

Preliminary Results with GUPS



- Scaling GUPS from 16 to 1024 PEs on Oregon Talapas system
- Record allocations of objects of interest and their child allocations
- Object types with largest allocations in runtime at 1024 PEs among selected types
  - ompi_proc_t (117 MB)
  - orte_namelist_t (child of oshmem_group_t) (100 MB)
  - ompi_proc_t list (child of oshmem_group_t) (33.5 MB)

All Types



- Looking at *all* runtime types shows opal_value_t is by far the largest user of memory
- Usages are spread out as children of many other object types.
- Reducing memory usage will improve scalability.

OpenSHMEM'18 Paper

Tracking Memory Usage in OpenSHMEM Runtimes with the TAU Performance System

Nicholas Chaimov[1], Sameer Shende[1], Allen Malony[1], Manjunath Gorentla Venkata[2], and Neena Imam[2]

[1] ParaTools, Inc.
2836 Kincaid St., Eugene, OR 97405, USA
{nchaimov,sameer,malony}@paratools.com
[2] Oak Ridge National Laboratory
1 Bethel Valley Rd, Oak Ridge, TN 37831, USA
{manjugv,imamn}@ornl.gov

Abstract. As the exascale era approaches, it is becoming increasingly important that runtimes be able to scale to very large numbers of processing elements. However, by keeping arrays of sizes proportional to the number of PEs, an OpenSHMEM implementation may be limited in its scalability to millions of PEs. In this paper, we describe techniques for tracking memory usage by OpenSHMEM runtimes, including attributing memory usage to runtime objects according to type, maintaining data about hierarchical relationships between objects and identification of the source lines on which allocations occur. We implement these techniques in the TAU Performance System using atomic and context events and demonstrate their use in OpenSHMEM applications running within the Open MPI runtime, collecting both profile and trace data. We describe how we will use these tools to identify memory scalability bottlenecks in OpenSHMEM runtimes.

Keywords: Open MPI · TAU · memory · scalability.

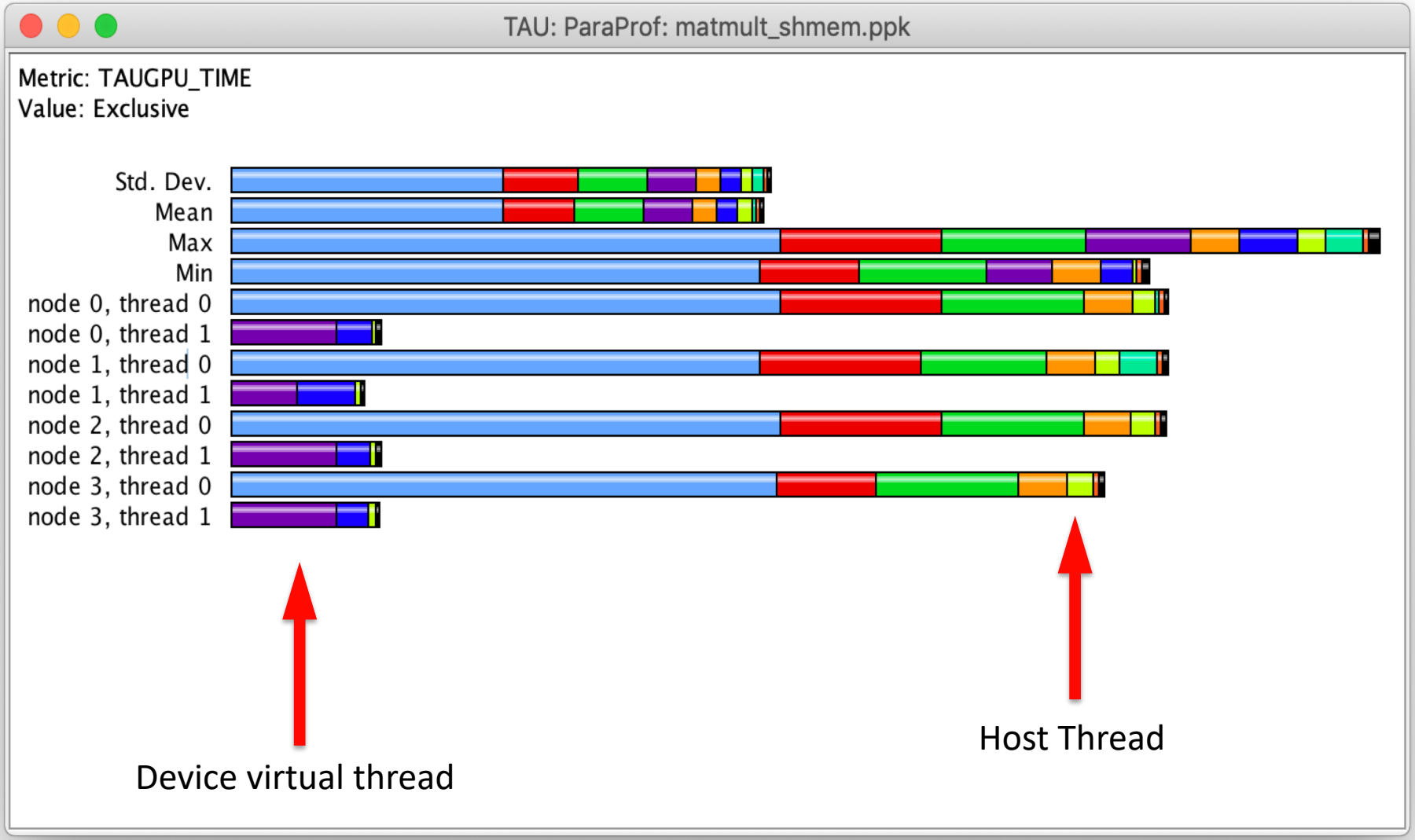
Acknowledgments. This work was sponsored by the U.S. Department of Energy's Office of Advanced Scientific Computing Research.This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retainsand the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Abstract. As the exascale era approaches, it is becoming increasingly important that runtimes be able to scale to very large numbers of pro- cessing elements. However, by keeping arrays of sizes proportional to the number of PEs, an OpenSHMEM implementation may be limited in its scalability to millions of PEs. In this paper, we describe techniques for tracking memory usage by OpenSHMEM runtimes, including attributing memory usage to runtime objects according to type, maintaining data about hierarchical relationships between objects and identification of the source lines on which allocations occur. We implement these techniques in the TAU Performance System using atomic and context events and demonstrate their use in OpenSHMEM applications running within the Open MPI runtime, collecting both profile and trace data. We describe how we use these tools to identify memory scalability bottlenecks in OpenSHMEM runtimes.

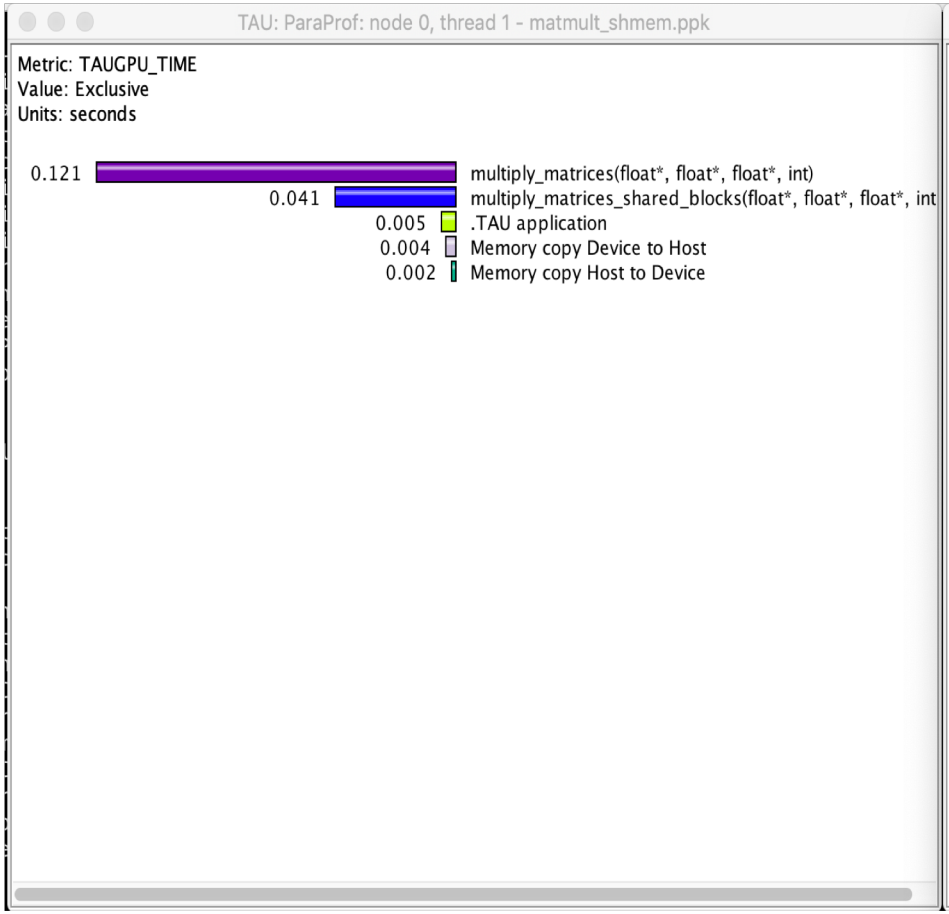
Hybrid CUDA+OpenSHMEM

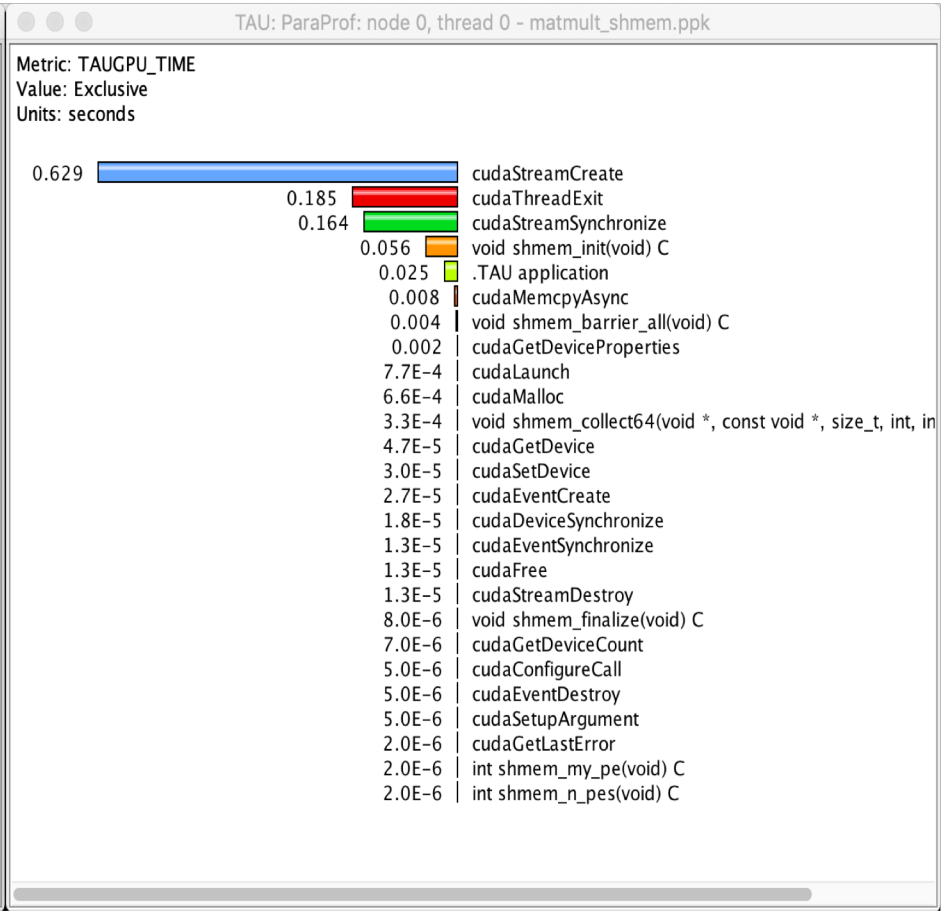
- We have added support to TAU for collecting data on GPU kernels executing in the CUDA runtime for OpenSHMEM applications.
  - Data for each GPU device upon which kernels are executed by a PE are stored in profiles for “virtual threads” representing the GPU device.
  - Host-level CUDA API calls are recorded alongside OpenSHMEM API calls in the profile for the PE’s main thread.
  - Feature parity in TAU between CUDA+MPI and CUDA+OpenSHMEM.

Hybrid CUDA+OpenSHMEM in ParaProf



Hybrid CUDA+OpenSHMEM in ParaProf





Device virtual thread
Kernels and Copies

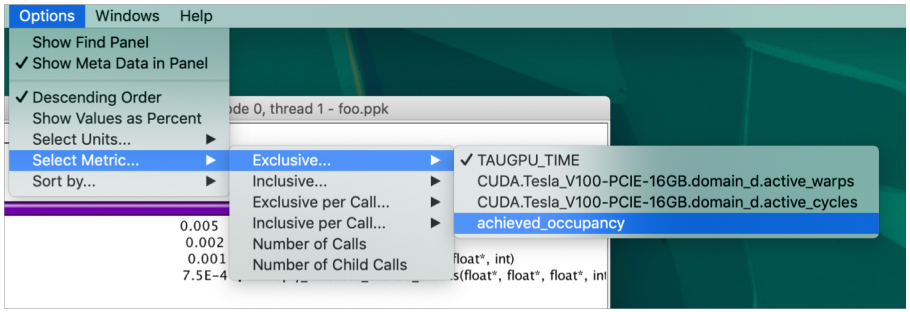
Host Thread
CUDA API calls and
OpenSHMEM API calls

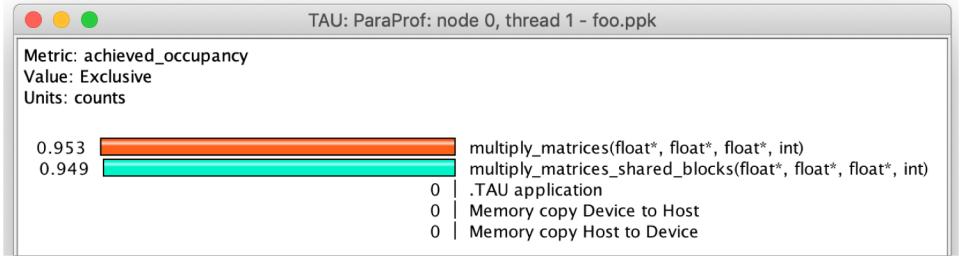
CUPTI Derived Metrics

- CUPTI Derived Metrics are now supported in TAU.
  - Based on a set of underlying CUPTI events and device-specific constants.
  - Specifying a derived metric automatically selects the underlying events and outputs the derived metric as a TAU metric in the profile.
- `tau_cupti_avail -m` prints supported CUPTI derived metrics.
- Add to `TAU_METRICS` environment variable to enable.

CUPTI Derived Metrics Example

- `export TAU_METRICS=TIME:achieved_occupancy`
- Automatically enables `domain_d.active_cycles` and `domain_d.active_warps` and calculates achieved occupancy based on those events.





OpenSHMEM 1.4 Support

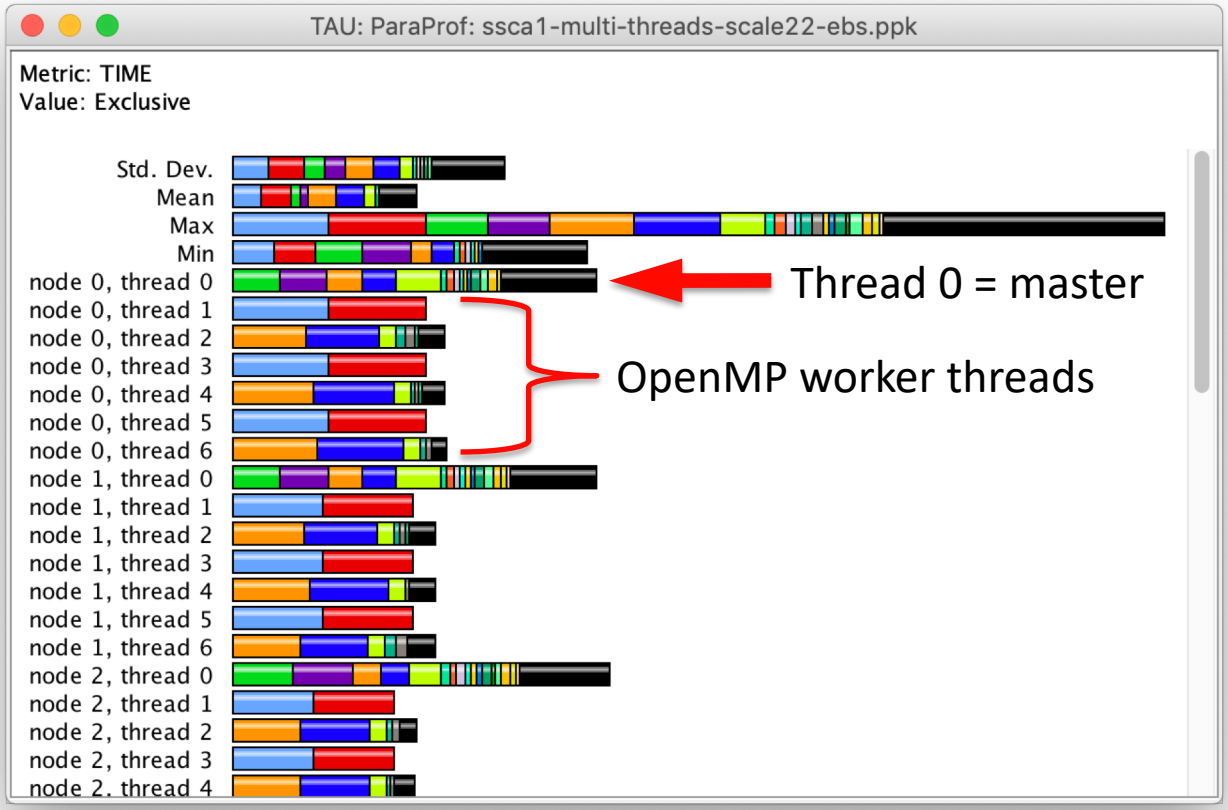
- TAU's OpenSHMEM library wrapper generator now automatically generates wrapper functions for API calls added in the OpenSHMEM 1.4 specification.
  - Implemented support in TAU for the new initialization function (`shmem_init_thread`)
  - Implemented support in TAU for the new context-based communication functions (`shmem_ctx_create`, `shmem_ctx_destroy`, `shmem_ctx_get`, etc.).

Multithreaded OpenSHMEM support

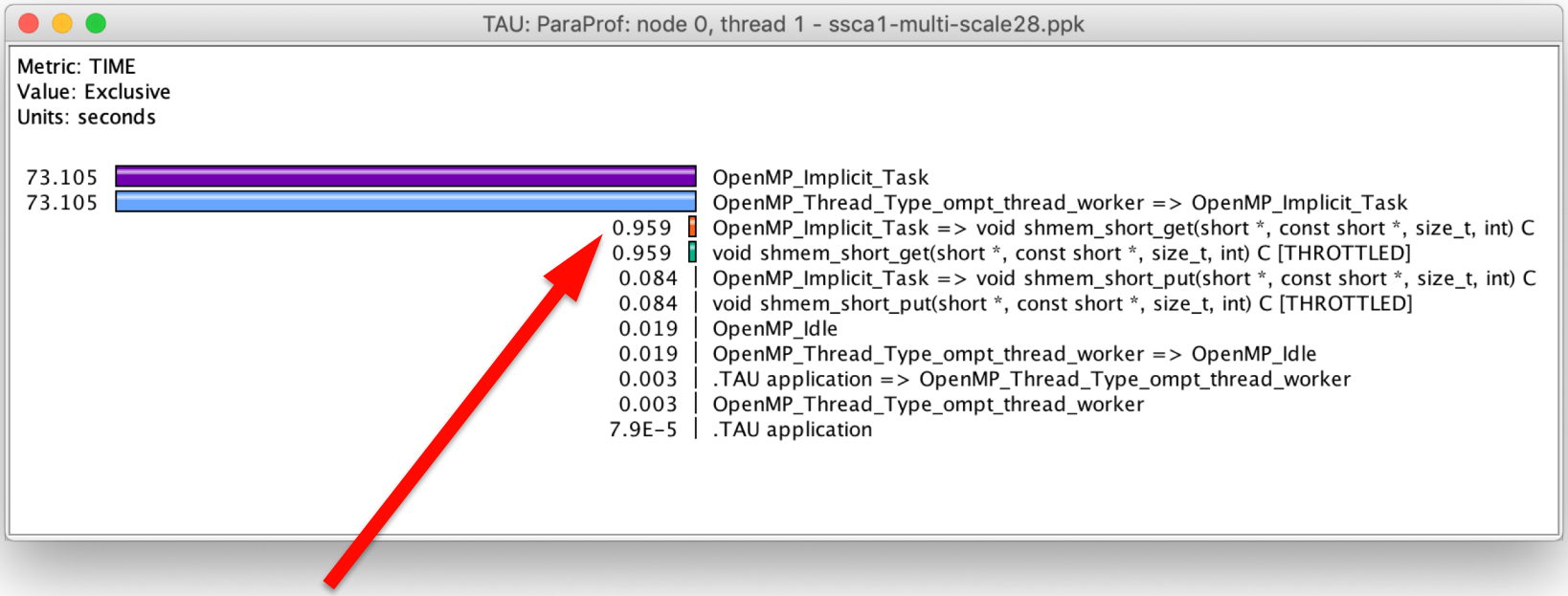
- Implemented support in TAU for collecting performance profiles and traces for OpenSHMEM applications which use the SHMEM_THREAD_MULTIPLE mode.
  - Enabled when TAU is configured with both OpenSHMEM and a threading library (pthreads or OpenMP).
  - When used with OpenMP, OMPT-TR6 support in TAU can be used to tune the level of overhead experienced by enabling or disabling sets of events.

Threaded SSCA OpenMP+SHMEM profile

- SSCA threaded benchmark from Oak Ridge OpenSHMEM Benchmarks.



TAU support for OpenSHMEM and OMPT



OpenSHMEM library wrapper + OMPT records OpenSHMEM API calls within OpenMP worker threads

TAU v2.28 released with all these features at SC18: http://tau.uoregon.edu