Tracking Memory Usage in OpenSHMEM Runtimes with the TAU Performance System

Nicholas Chaimov (ParaTools), Sameer Shende (ParaTools), Allen Malony (ParaTools), Manjunath Gorentla Venkata (Oak Ridge National Laboratory), and Neena Imam (Oak Ridge National Laboratory)

OpenSHMEM 2018
22 August 2018
Hanover, MD
Outline

• Motivation: per-object memory tracking
• What is TAU?
• Implementation
  – Library wrapping
  – Context events
  – Allocation classes
  – Profiling vs Tracing
  – Data analysis
• Preliminary results
• Future work and conclusions
• 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.
• Extend 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.
Tuning and Analysis Utilities (25+ 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-port ed performance profiling / tracing system
- Performance data management and data mining
- Open source (BSD-style license)

Integrates with application frameworks
TAU Supports All HPC Platforms

- C/C++
- Fortran
- CUDA
- UPC
- Python
- MPI
- Java
- OpenMP
- Cray
- Sun
- AIX
- Intel MIC
- LLVM
- PGI
- Cray
- OpenACC
- MinGW
- Intel
- GNU
- Linux
- Windows
- Android
- BlueGene
- Fujitsu
- ARM
- MPC
- OpenSHMEM

Insert yours here
Measurement Approaches

Profiling

Shows how much time was spent in each routine

Tracing

Shows when events take place on a timeline
Performance Data Measurement

Direct via Probes

call TAU_START(‘name’)  // code
call TAU_STOP(‘name’)  

• Exact measurement
• Fine-grain control
• Calls inserted into code

Indirect via Sampling

• No code modification
• Minimal effort
• Relies on debug symbols (-g option)
Questions TAU Can Answer

• **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, *vector instructions*?

• **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 **time spent waiting for collectives**?

• How does the application **scale**?
Atomic Events

• Event types
  – Interval events (begin/end events)
    • Measures exclusive & inclusive durations between events
    • Metrics monotonically increase
  – Atomic events (trigger with data value)
    • Used to capture performance data state
    • Shows extent of variation of triggered values (total, samples, min/max/mean/standard deviation statistics)
  – Context Events
    • Atomic event + context (disaggregated according to timer stack when event triggered)
Interval and Atomic Events in TAU

<table>
<thead>
<tr>
<th>%Time</th>
<th>Exclusive msec</th>
<th>Inclusive total msec</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive usec/call</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>0.187</td>
<td>1.105</td>
<td>1</td>
<td>44</td>
<td></td>
<td>1105659 int main(int char **) C</td>
</tr>
<tr>
<td>93.2</td>
<td>1.030</td>
<td>1.030</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1030654 MPI_Init()</td>
</tr>
<tr>
<td>5.9</td>
<td>0.879</td>
<td>65</td>
<td>40</td>
<td>320</td>
<td></td>
<td>1637 void func(int int)</td>
</tr>
<tr>
<td>4.6</td>
<td>51</td>
<td>51</td>
<td>40</td>
<td>0</td>
<td></td>
<td>1277 MPI_Barrier()</td>
</tr>
<tr>
<td>1.2</td>
<td>13</td>
<td>13</td>
<td>120</td>
<td>0</td>
<td></td>
<td>111 MPI_Recv()</td>
</tr>
<tr>
<td>0.8</td>
<td>9</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td></td>
<td>9328 MPI_Finalize()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.137</td>
<td>0.137</td>
<td>120</td>
<td>0</td>
<td></td>
<td>1 MPI_Send()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.086</td>
<td>0.086</td>
<td>40</td>
<td>0</td>
<td></td>
<td>2 MPI_Bcast()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.002</td>
<td>0.002</td>
<td>1</td>
<td>0</td>
<td></td>
<td>2 MPI_Comm_size()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.001</td>
<td>0.001</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1 MPI_Comm_rank()</td>
</tr>
</tbody>
</table>

**Interval events**

- e.g., routines
- (start/stop) show **duration**

**Atomic events**

- (triggered with value) show **extent of variation**
- (min/max/mean)

% export TAU_CALLPATH_DEPTH=0
% export TAU_TRACK_HEAP=1
Atomic Events, Context Events

<table>
<thead>
<tr>
<th>%Time</th>
<th>Exclusive msec</th>
<th>Inclusive total msec</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive Name usec/call</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>0.253</td>
<td>1.106</td>
<td>1</td>
<td>44</td>
<td>1106701 int main(int, char **) C</td>
</tr>
<tr>
<td>93.2</td>
<td>1.031</td>
<td>1.031</td>
<td>1</td>
<td>0</td>
<td>1031311 MPI_Init()</td>
</tr>
<tr>
<td>6.0</td>
<td>1</td>
<td>66</td>
<td>40</td>
<td>320</td>
<td>1850 void func(int, int) C</td>
</tr>
<tr>
<td>5.7</td>
<td>63</td>
<td>63</td>
<td>40</td>
<td>0</td>
<td>1588 MPI_Barrier()</td>
</tr>
<tr>
<td>0.8</td>
<td>9</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>9119 MPI_Finalize()</td>
</tr>
<tr>
<td>0.1</td>
<td>1</td>
<td>1</td>
<td>120</td>
<td>0</td>
<td>10 MPI_Recv()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.141</td>
<td>0.141</td>
<td>120</td>
<td>0</td>
<td>1 MPI_Send()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.005</td>
<td>0.005</td>
<td>40</td>
<td>0</td>
<td>2 MPI_Bcast()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.001</td>
<td>0.001</td>
<td>1</td>
<td>0</td>
<td>1 MPI_Comm_size()</td>
</tr>
<tr>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 MPI_Comm_rank()</td>
</tr>
</tbody>
</table>

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0

Atomic events
Context events = atomic event + executing context

% export TAU_CALLPATH_DEPTH=1

% export TAU_TRACK_HEAP=1

Controls depth of executing context shown in profiles
% export TAU_CALLPATH_DEPTH=2
% export TAU_TRACK_HEAP=1

Context Events (Default)

<table>
<thead>
<tr>
<th>Time</th>
<th>Exclusive msec</th>
<th>Inclusive total msec</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive usec/call</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>0.357</td>
<td>1.114</td>
<td>1</td>
<td>44</td>
<td>1114040</td>
<td>int main(int, char **) C</td>
</tr>
<tr>
<td>92.6</td>
<td>1.031</td>
<td>1.031</td>
<td>1</td>
<td>0</td>
<td>1031066</td>
<td>MPI_Init()</td>
</tr>
<tr>
<td>6.7</td>
<td>72</td>
<td>74</td>
<td>40</td>
<td>320</td>
<td>1865</td>
<td>void func(int, int) C</td>
</tr>
<tr>
<td>0.7</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>8002</td>
<td>MPI_Finalize()</td>
</tr>
<tr>
<td>0.1</td>
<td>1</td>
<td>1</td>
<td>120</td>
<td>0</td>
<td>12</td>
<td>MPI_Recv()</td>
</tr>
<tr>
<td>0.1</td>
<td>0.608</td>
<td>0.608</td>
<td>40</td>
<td>0</td>
<td>15</td>
<td>MPI_Barrier()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.136</td>
<td>0.136</td>
<td>120</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0.0</td>
<td>0.095</td>
<td>0.095</td>
<td>40</td>
<td>0</td>
<td>2</td>
<td>MPI_Bcast()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.001</td>
<td>0.001</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>MPI_Comm_size()</td>
</tr>
<tr>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>MPI_Comm_rank()</td>
</tr>
</tbody>
</table>

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0

<table>
<thead>
<tr>
<th>NumSamples</th>
<th>MaxValue</th>
<th>MinValue</th>
<th>MeanValue</th>
<th>Std. Dev.</th>
<th>Event Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>365</td>
<td>5.139E+04</td>
<td>44.39</td>
<td>3.091E+04</td>
<td>1.234E+04</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>1</td>
<td>44.39</td>
<td>44.39</td>
<td>44.39</td>
<td>0</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>1</td>
<td>2068</td>
<td>2068</td>
<td>2068</td>
<td>0</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>1</td>
<td>2066</td>
<td>2066</td>
<td>2066</td>
<td>0</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>1</td>
<td>5.139E+04</td>
<td>5.139E+04</td>
<td>5.139E+04</td>
<td>0</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>1</td>
<td>57.58</td>
<td>57.58</td>
<td>57.58</td>
<td>0</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>40</td>
<td>5.036E+04</td>
<td>2068</td>
<td>3.011E+04</td>
<td>1.228E+04</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>40</td>
<td>5.139E+04</td>
<td>3098</td>
<td>3.114E+04</td>
<td>1.227E+04</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>40</td>
<td>5.139E+04</td>
<td>1.13E+04</td>
<td>3.134E+04</td>
<td>1.187E+04</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>120</td>
<td>5.139E+04</td>
<td>1.13E+04</td>
<td>3.134E+04</td>
<td>1.187E+04</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>120</td>
<td>5.139E+04</td>
<td>1.13E+04</td>
<td>3.134E+04</td>
<td>1.187E+04</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>365</td>
<td>5.139E+04</td>
<td>2065</td>
<td>3.116E+04</td>
<td>1.21E+04</td>
<td>Heap Memory Used (KB) : Exit</td>
</tr>
</tbody>
</table>

Context event^2,7 =atomic event + executing context
Library Wrapping (malloc)

• TAU provides wrapper libraries around `malloc`, `free`, et al. which replace the system version

• Each wrapper
  – Starts a timer
  – Records a context event indicating the size and source line of the allocation or deallocation
  – Call the underlying system version of the function
  – Stop the timer

• Loaded into unmodified application with `LD_PRELOAD` or through linker script at link time
Library Wrapping (OpenSHMEM)

- During configure, TAU
  - parses shmem.h
  - generates a wrapper library for each function defined in shmem.h
    - For each function, the wrapper
      - Starts a timer
      - For communications, records a context event indicating the size, source, destination and source code line of the communication.

- Parsing header works around differences between SHMEM implementations and versions.
- Loaded into unmodified application with LD_PRELOAD or through linker script at link time

```c
extern void __real_shmem_get128(void * a1, const void * a2, size_t a3, int a4) ;
extern void __wrap_shmem_get128(void * a1, const void * a2, size_t a3, int a4) {

    TAU_PROFILE_TIMER(t,"void shmem_get128(void *, const void *, size_t, int)", ",", TAU_USER);
    TAU_PROFILE_START(t);
    TAU_TRACE_SENDMSG_REMOTE(TAU_SHMEM_TAGID_NEXT, Tau_get_node(), 16*a3, a4);
    __real_shmem_get128(a1, a2, a3, a4);
    TAU_TRACE_RECVMSG(TAU_SHMEM_TAGID, a4, 16*a3);
    TAU_PROFILE_STOP(t);
}
```
TAU’s existing memory tracking support gives us heap usage overall and per function, but does not tell us where in the runtime or of what data type the allocation belongs to.
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);
10 bytes allocated in object of type A
Tau_start_class_allocation("b", 25, 0);
25 bytes allocated in object of type B (child of A)
Tau_stop_class_allocation("b", 1);
Tau_stop_class_allocation("a", 1);
Tau_start_class_allocation("b", 10, 0);
10 bytes allocated in object of type B (not child)
Tau_stop_class_allocation("b", 1);
```

Stored in profile:
```
alloc a 10
alloc b 35
alloc b <= a 25
```
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 allocations by type requires one line of code inserted into Open MPI runtime
  – static inline prevents use of library wrapper

```c
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);
    [...]
}
```
which indicates that 10 bytes of object were allocated, 35 bytes of object were allocated, and 25 bytes of object were child allocations of object.

Allocations of most objects in the Open MPI runtime are captured by instrumenting, as before, the `opal_obj_new` function in `opal/class/opal_object.h`, which is called to construct most objects in the OPAL class hierarchy. In this implementation, we wrap the call to the class constructors for the object being instantiated, which automatically captures any child allocations which occur inside the constructor:

```c
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;
}
```

By using allocation regions, this automatically captures any child allocations that occur within the constructor of the class. For example, the constructor for `orte_rml_posted_recv_t` allocates an object of type `orte_rml_recv_request_t`, and this allocation is recorded as `alloc orte_rml_posted_recv <= orte_rml_recv_request` in the profile.

This technique does not capture any child objects which are allocated outside of the constructor for a class. There is no central location where such tracking could be implemented, so any such child allocations are instrumented manually. To do this, `dummy allocation regions` are used to indicate the parent of an allocation without actually recording an atomic event for the parent, which was already recorded through `opal_obj_new`. To this, `Tau_start_class_allocation` is called normally, child allocations are recorded, and `Tau_stop_class_allocation` is then called with the `record` parameter set to 0.

Allocations during constructors automatically attributed to enclosing allocation region.
Memory Allocation Profiles

![Memory Allocation Profiler Window](image_url)
## Memory Allocation Profiles

Sorted By: Number of Samples

<table>
<thead>
<tr>
<th>Total</th>
<th>NumSamples</th>
<th>Max</th>
<th>Min</th>
<th>Mean</th>
<th>Std. Dev</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2755848</td>
<td>131280</td>
<td>8208</td>
<td>8</td>
<td>20.992</td>
<td>320.138</td>
<td>Message size sent to all nodes</td>
</tr>
<tr>
<td>524288</td>
<td>65536</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Message size received from all nodes</td>
</tr>
<tr>
<td>159756</td>
<td>945</td>
<td>168</td>
<td>168</td>
<td>168</td>
<td>0</td>
<td>alloc mca_base_var_t</td>
</tr>
<tr>
<td>159756</td>
<td>945</td>
<td>168</td>
<td>168</td>
<td>168</td>
<td>0</td>
<td>alloc mca_base_var_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>71944</td>
<td>529</td>
<td>136</td>
<td>136</td>
<td>136</td>
<td>0</td>
<td>alloc mca_base_var_group_t</td>
</tr>
<tr>
<td>71944</td>
<td>529</td>
<td>136</td>
<td>136</td>
<td>136</td>
<td>0</td>
<td>alloc mca_base_var_group_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>27552</td>
<td>287</td>
<td>96</td>
<td>96</td>
<td>96</td>
<td>0</td>
<td>alloc mca_base_component_repository_item_t</td>
</tr>
<tr>
<td>27552</td>
<td>287</td>
<td>96</td>
<td>96</td>
<td>96</td>
<td>0</td>
<td>alloc mca_base_component_repository_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>54328</td>
<td>194</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>54328</td>
<td>194</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>20944</td>
<td>119</td>
<td>176</td>
<td>176</td>
<td>176</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>20944</td>
<td>119</td>
<td>176</td>
<td>176</td>
<td>176</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>4764</td>
<td>90</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>4764</td>
<td>90</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>4320</td>
<td>90</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>4320</td>
<td>90</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>4408</td>
<td>70</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>4352</td>
<td>60</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>3680</td>
<td>46</td>
<td>88</td>
<td>88</td>
<td>88</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>3680</td>
<td>46</td>
<td>88</td>
<td>88</td>
<td>88</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>1488</td>
<td>35</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>1488</td>
<td>35</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>1504</td>
<td>33</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>1504</td>
<td>33</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>1792</td>
<td>28</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>1728</td>
<td>24</td>
<td>72</td>
<td>72</td>
<td>72</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>1344</td>
<td>21</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>1672</td>
<td>19</td>
<td>88</td>
<td>88</td>
<td>88</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>1672</td>
<td>19</td>
<td>88</td>
<td>88</td>
<td>88</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>5472</td>
<td>19</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>5472</td>
<td>19</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>1984</td>
<td>17</td>
<td>112</td>
<td>112</td>
<td>112</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>1984</td>
<td>17</td>
<td>112</td>
<td>112</td>
<td>112</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>816</td>
<td>17</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>816</td>
<td>17</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>1792</td>
<td>16</td>
<td>112</td>
<td>112</td>
<td>112</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>1792</td>
<td>16</td>
<td>112</td>
<td>112</td>
<td>112</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>128</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>128</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>128</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>128</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
<tr>
<td>768</td>
<td>16</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t</td>
</tr>
<tr>
<td>768</td>
<td>16</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>0</td>
<td>alloc mca_base_component_list_item_t : void shmem_init(void) C</td>
</tr>
</tbody>
</table>

---

### Copyright © ParaTools, Inc.
• We want to use memory tracking for scaling studies.
• ParaProf and PerfExplorer (TAU’s visualizers) do not provide the right kind of charts for visualizing the scaling of context events.
• To increase flexibility, we added a parser which generates Pandas dataframes from TAU profile files.
  – Allows use of Jupyter notebooks and the wide array of Python visualization libraries on data collected from TAU.
Running an application

• Build your app with the instrumented runtime, then run with

```bash
<launcher> <launcher args>
tau_exec -T shmem,pdt -shmem -ebs -memory <app> <app args>
```

• Example: GUPS on University of Oregon Talapas system (Slurm)

```bash
-srun tau_exec -T shmem,pdt -shmem -ebs -memory ./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)
• 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.
This instrumentation gives us *total allocations*.  
- Sum of all allocations throughout application execution.
- Does not distinguish between data types whose objects persist for the lifetime of the application and those that, for example
  - are only used during initialization; or,
  - are subject to repeated allocation and deallocation.

To distinguish these cases, we need to keep traces or phase-based profiles, not context profiles.
OTF2 dramatically improves on SLOG2:
- Smaller trace files
- Richer trace data, e.g. RMA events
- Better trace visualization (Vampir, Ravel)

TAU can now generate OTF2 files natively:
- No Score-P required!
- Uses OpenSHMEM internally for event reduction
- Writes context events to OTF2 trace
ISx in Vampir
Different Nodes, Different Timelines
Get/Put Recorded as RMA Events
Instrumentation of Open MPI destructors enables tracking frees by data type as well as mallocs, enabling a cumulative allocations context event.

```c
static inline void opal_obj_run_destructors(opal_object_t * object) {
    opal_destruct_t* cls_destruct;
    assert(NULL != object->obj_class);

    Tau_start_class_deallocation(object->obj_class->cls_name,object->obj_class->cls_sizeof, 0);
    cls_destruct = object->obj_class->cls_destruct_array;
    while( NULL != *cls_destruct ) {
        (*cls_destruct)(object);
        cls_destruct++;
    }
    Tau_stop_class_deallocation(object->obj_class->cls_name, 1);
}
```
Tracing opal_value_t in GUPS

Full view

Zoomed to show fluctuation during initialization
Future Work in Memory Allocation Tracking

• Identification of targets for optimization by sharing across PEs on a node
  – If there are object types which are written to only during initialization, or which have fields written to only during initialization, these might be shared between PEs on the same physical node
  – Identification requires tracking writes to allocations attributed to particular data types.

• Large scaling studies on Summit, etc.
Future Work in OpenSHMEM in TAU

• For SC18 release:
  – Support for OpenSHMEM 1.4 standard, including threading modes and contexts.
  – Support for hybrid CUDA/OpenSHMEM applications.
Conclusions

• With no runtime instrumentation, TAU can capture runtime allocations by source line of origin.

• With minimal runtime instrumentation (4 lines of code), TAU can capture runtime allocations by both source line of origin and data type of the allocation.

• Now available in TAU v2.27.2:
  – http://tau.uoregon.edu/tau.tgz

• Instrumented runtime at:
  – https://github.com/paratoolsinc/ompi