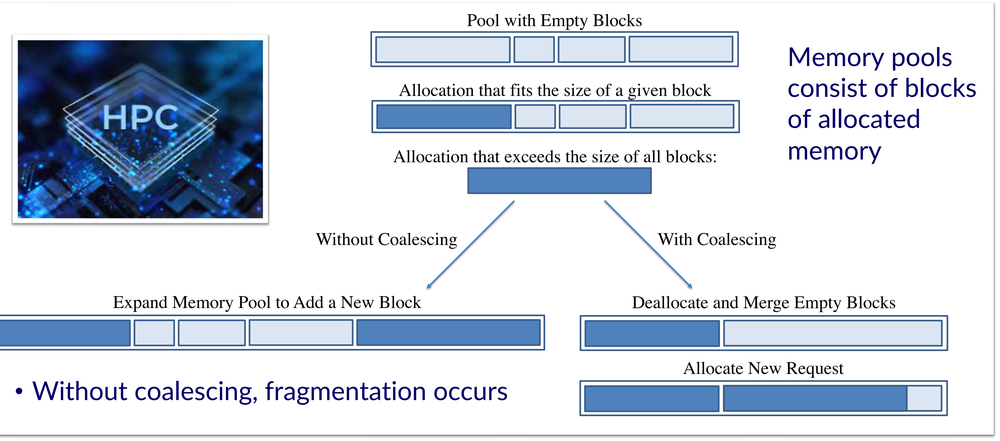
Umpire: Performing Experiments to Determine Consistency of Coalescing Heuristics Across Different Memory Pools

Avnoor Sidhu and Kristi Belcher

Lawrence Livermore National Laboratory, Livermore, CA, USA

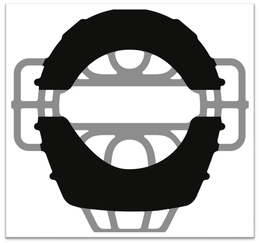
Introduction

- The Umpire API provides memory management abstractions, including memory pools, for high performance applications.



- Without coalescing, fragmentation occurs

- Umpire's heuristics determine when to coalesce and what factors should trigger the call.



- Previous experiments have concluded it is best to call the coalescing function when a specific number of blocks are releasable (Blocks-Releasable heuristic).
- My work will ensure that the previous coalescing heuristic results are valid across different pools (DynamicPoolList and QuickPool) and are consistent from one application from another.

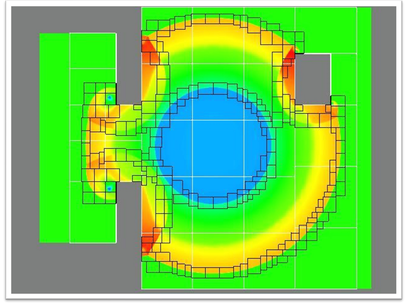
Discussion

- By the end of summer my work will focus on these questions:

- Does the Blocks-Releasable heuristic give consistent performance benefits from one application to another?
- Does the Blocks-Releasable heuristic work equally well for the DynamicPoolList as it does for QuickPool?
- Is the Blocks-Releasable heuristic a good coalescing heuristic to use as a default heuristic in Umpire?

Methods

- Umpire provides a replay tool that can be used to analyze the performance of an application's allocation of memory and reproduce bugs. The replay tool generates a .ult file which can be used with PyDV, a Python data visualizer, to generate plots. To conduct the experiment I will use SAMRAI, a C++ framework for block-structured AMR application development.



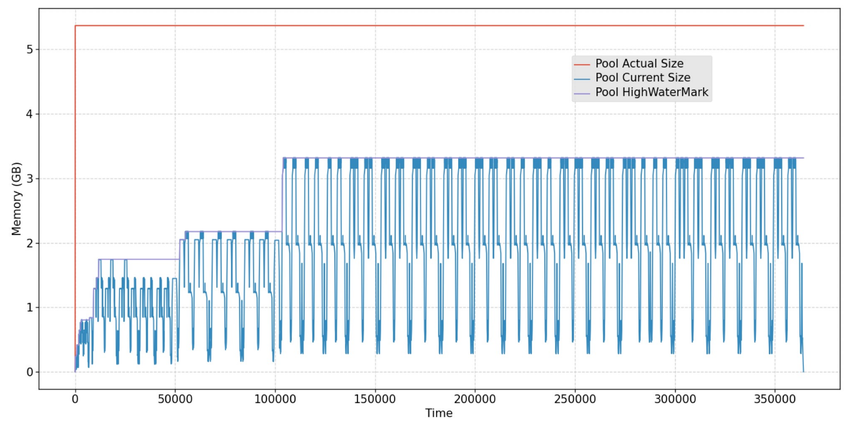
- SAMRAI allows for mesh refinement where high resolution is required.

- The programs used to test will be running on NVIDIA GPUs.

```
// umpire::ResourceManager allows to interact with Umpire API
auto& rm = umpire::ResourceManager::getInstance();

// umpire::Allocator can be used to allocate and deallocate memory
umpire::Allocator allocator = rm.getAllocator("HOST");

// Create a heuristic that will return true when 2 blocks are releasable
auto heuristic_function = umpire::strategy::DynamicPool::blocks_releasable(2);

// Create a dynamic pool with the heuristic
auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPool>("HOST_POOL",
allocator, 1024ul, 1024ul, 16, heuristic_function);

// Pool will contain 4 blocks each of size 1024 bytes
void* a[4];
for(int i = 0; i < 4; ++i)
  a[i] = pooled_allocator.allocate(1024);

// Deallocate
pooled_allocator.deallocate(a[0]);
pooled_allocator.deallocate(a[1]);

// Two blocks left --> Pool will now coalesce the two empty blocks

pooled_allocator.deallocate(a[2]);
pooled_allocator.deallocate(a[3]);
```

- Example code of where the coalescing would occur if there were multiple allocations and deallocations within a pool.

Initial Results



- Umpire's replay tool used to show properties about SAMRAI test application.

- The Current Size fluctuates as the program allocates and deallocates memory while the HighWaterMark stays at the maximum value of the Current Size.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

LLNL-POST-852231