Accelerated Signed Distance Queries For Performance Portable Physics Codes

Jordan Backes (Co-Author) **Evan DeSantola (Co-Author)** George Zagaris (Advisor) Kenneth Weiss (Advisor) Matthew Larsen (Advisor) Cyrus Harrison (Advisor)

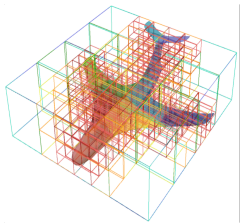
College of Engineering, University of Missouri
Carnegie Mellon University, School of Computer Science

Overview

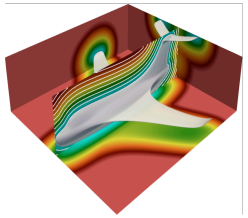
Signed distance is commonly employed to numerically represent material interfaces with complex boundaries in multi-material numerical simulations. However, the performance of computing the signed distance field is hindered by the complexity and size of the input. Recent trends in High-Performance Computing architecture consist of multi-core CPUs and accelerators that collectively expose tens to thousands of cores to the application. Harnessing this massive parallelism for computing the signed distance field presents significant challenges. Chief among them is the design and implementation of a performance portable solution that can work across architectures. Addressing these challenges to accelerate signed distance queries is the primary contribution of this work. Specifically, in this work we employ the RAJA programming model, which provides a loop-level abstraction that decouples the loop-body from the parallel execution and insulates application developers from non-portable compiler and platform-specific directives.

Accelerated Signed Distance

- We employed a Bounding Volume Hierarchy acceleration structure to partition the surface elements of our test configuration

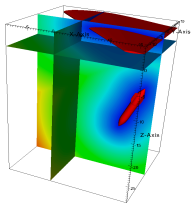


BVH subdivision of a generic jet

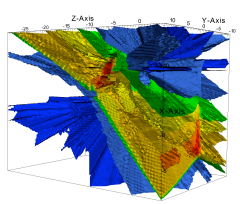


Signed Distance field on 32^3 grid

- We query the BVH subdivision to reconstruct exact signed distances from an arbitrary set of points to the surface.
- Queries to the BVH subdivision are independent and therefore ripe for the parallelization enabled by next generation architectures.



Union Signed Distance of our wing-store configuration



Spatially varying cost of querying the BVH representation

RAJA Parallelization and Portability

We used RAJA's loop abstraction model to minimize the changes necessary to run our loops using different execution policies.

Sequential Execution

```
RAJA::forall<RAJA::seq_exec>(0, nnodes, signed_distance);
```

OpenMP Execution

```
RAJA::forall<RAJA::omp_parallel_for_exec>(0, nnodes, signed_distance);
```

Balanced OpenMP Execution using IndexSet

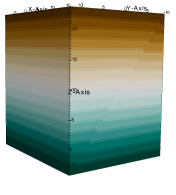
```
RAJA::IndexSet idxSet;
for(int y = 0; y <= params.ny; ++y) {
  for(int z = 0; z <= params.nz; z++) {
    int start = (params.nx + 1) * (y +  z * (params.ny + 1));
    idxSet.push_back(RAJA::RangeSegment(start, start + params.nx + 1));
  }
}
typedef RAJA::IndexSet::ExecPolicy<RAJA::omp_parallel_for_segit,RAJA::simd_exec>
RAJA::forall<exec_pol> (idxSet, signed_distance);
```

Loop Kernel:

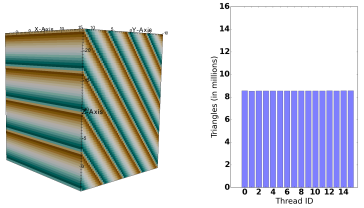
```
auto signed_distance = [=] (int index) {
    quest::Point<double,3> pt;
    umesh->getMeshNode(index, pt.data());
    phi_store[index] =
        sd->computeDistance(pt);
    double dist = phi_store[index];
    phi_union[index] =
        std::min(dist, phi_wing[index]);
}
```

Load Balancing

- We used the number of triangles tested per thread as a hardware independent workload metric
- We discovered a **510%** difference between the fastest and slowest threads in the default RAJA parallelization.
- This was reduced to **0.3%** after we applied RAJA `IndexSets` to change the thread partitioning.



Before load balancing. Default thread mapping for a grid of 275k query points (left, colored by Thread ID) and the number of triangles (in millions) tested by each of 16 threads (right)



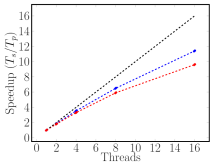
After load balancing: `IndexSet` remapping of threads to query points (left, colored by Thread ID) and the number of triangles (in millions) tested by each of 16 threads (right) on same grid

Performance Results

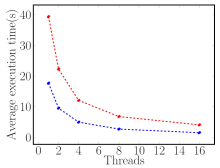
After Load Balancing
Before Load Balancing
Ideal

- Strong scaling yields a **9.7x** speedup with 16 threads and **12x** after load balancing
- Load balancing increased the speedup by 23.7% and had nearly linear weak scaling efficiency

Strong Scaling

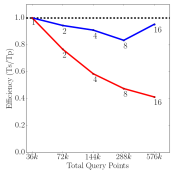


Speedup on 275k points

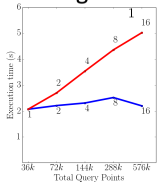


Execution Time on 275k points

Weak Scaling



Efficiency between threads and query points



Execution between threads and query points

Conclusion

- We described a performance portable parallelization strategy for threading signed distance queries
- Our preliminary performance evaluation indicated significant load imbalances among threads
- With the RAJA IndexSet abstraction, we address the load imbalances and improve the overall performance of our algorithm

Future Work

- **Use the RAJA Performance Portability Layer to**
  - Evaluate our implementation on different architectures, such as machines equipped with GPUs or many-core processors, such as the Xeon Phi
  - Exploring and comparing our implementation with other programming models
- **Improve our BVH queries through**
  - Implementing a Surface Area Heuristic (SAH) to the BVH to achieve a more optimal BVH decomposition
  - Compaction of internal BVH data layout to optimize cache utilization and overall memory footprint

Acknowledgements

The authors would like to acknowledge support by the Institute for Scientific Computing Research (ISCR) Summer Scholar Program for making this work possible. In addition, the authors would like to thank Rich Hornung and Rob Neely, from LLNL, for their keen interest in this work and Will Killian, from University of Delaware for useful discussions and help with RAJA.

Videos and Other Media

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-POST-698437