

Parallelization of Adaptive Mesh Refinement Using OpenMP and CUDA

Ronan A. Pereira Jr¹, Ayla Y. Nishino¹, Vinícius R. Kury¹, Hélio C. Guardia¹

¹Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
São Carlos – SP – Brasil

{ronanpereira, aynishino, vinicius.kury}@estudante.ufscar.br

helio.guardia@ufscar.br

Abstract. *This work presents parallel implementations of Adaptive Mesh Refinement (AMR) using OpenMP and CUDA. AMR dynamically refines mesh resolution in regions requiring higher precision, optimizing computational resources. We compare CPU parallelization with OpenMP and GPU acceleration with CUDA against a sequential baseline. Results show that OpenMP achieves super-linear scaling up to 32 threads ($32.40\times$ speedup, 101.3% efficiency) on a 40-core Intel Xeon E7-4870 architecture distributed across 4 sockets, peaking at a $54.40\times$ speedup with 256 overcommitted threads due to cumulative L3 cache size and aggregated memory bandwidth. Consequently, the multi-socket CPU execution outperforms the CUDA implementation, which demonstrates a $28.65\times$ speedup on a Tesla T4 GPU, remaining constrained by warp divergence and irregular memory patterns inherent to AMR.*

1. Introduction

Adaptive Mesh Refinement (AMR) is a fundamental technique for numerical simulations of partial differential equations and computational fluid dynamics [Berger and Colella 1989]. In large-scale problems, uniform grids with high resolution become computationally prohibitive, requiring billions of degrees of freedom.

AMR addresses this challenge by selectively refining only the critical regions of the computational domain, saving resources without compromising numerical accuracy. This work evaluates the parallelization of AMR using Morton codes [Morton 1966] via OpenMP (multi-core) and CUDA (GPU), identifying performance gains and architectural bottlenecks.

2. Theoretical Background

Morton codes encode spatial coordinates (x, y, z) into a single 64-bit integer via *bit-interleaving*, preserving spatial locality. The linear tree maintains cells ordered by Morton code, with a 2 : 1 balance between adjacent levels for numerical stability.

The refinement process consists of four main phases:

- **Evaluation:** Verify the refinement criterion for each cell — $O(n)$
- **Expansion:** Create 2^d children per refined cell — $O(n)$ with a larger constant
- **2:1 Balancing:** Ensure a maximum difference of 1 level between neighbors — $O(n \log n)$

- **Merge:** Maintain Morton ordering — $O(n \log n)$

The total sequential complexity is $O(n \log n)$, being dominated by the merge and balancing phases.

2.1. Related Work

Libraries such as p4est [Burstedde et al. 2011] and t8code [Holke et al. 2021] implement distributed AMR with MPI. Recent studies explore GPU acceleration [Qin et al. 2019], but systematic comparisons between OpenMP and CUDA for AMR using Morton codes remain scarce.

3. Parallelization Strategies

The complete loop of the Parallelized Adaptive Mesh Refinement (AMR) implemented in this work is summarized in Algorithm 1, highlighting the refinement phases, load balancing, and the temporal evolution of the simulation.

```

procedure AMR_EXECUTION(mesh, T)
1: for each step  $t \leftarrow 1$  to  $T$  do
2:   for each cell  $c$  in parallel do
3:     evaluate criterion and mark in  $flags[c]$ 
4:   end for
5:   for each marked cell in parallel do
6:     create  $2^d$  children
7:   end for
8:   merge children into the sorted mesh
9:   for each cell  $c$  in parallel do
10:    check neighbors (2:1 balancing)
11:  end for
12:end for
end procedure

```

Figure 1. Global algorithm of the Parallel AMR Execution.

3.1. Parallelization with OpenMP

- **Lines 3-5:** Utilizing `#pragma omp parallel for` — threads evaluate disjoint subsets of cells, operating without race conditions.
- **Lines 6-9:** Each thread creates child cells in a local buffer (`std::vector`), which are subsequently merged in parallel using `std::merge` (C++17).
- **Lines 10-12:** Balancing is performed using sets constructed in a *thread-local* manner. The resulting complexity of this stage is $O(n \log n/p + \log p)$.

3.2. Parallelization with CUDA

- **Lines 3-5:** The `evaluateOracleKernel<<<blocks, TPB>>>` kernel (B: number of blocks, TPB: threads per block) assigns one thread per cell to evaluate the refinement criterion (L_2 norm of the derivatives), writing the resulting boolean *flags* to global memory. The complexity is $O(n/B)$.

- **Lines 6-9:** The `scanKernel` kernel executes a parallel *prefix sum* (based on Blelloch’s algorithm) utilizing *shared memory*, reducing access latency from ~ 400 cycles (global memory) to ~ 4 cycles, a local *speedup* of $\sim 100\times$. Each block processes up to 1024 elements with explicit synchronization barriers via `__syncthreads()`.

The `refineLeaves` kernel uses the indices pre-computed by the *scan* to write the 2^d children directly to their correct positions in memory, eliminating the need for atomic operations (`atomicAdd`) that would serialize the execution flow.

- **Lines 10-12:** The `balanceKernel` kernel verifies neighbors via Morton codes. The irregular (non-coalesced) access pattern significantly degrades memory *throughput*, a behavior inherent to the structure of the AMR problem. The complexity is defined by $O(n \log n / (B \times \text{TPB}))$, added to the synchronization and CPU $\leftrightarrow$ GPU transfer *overheads*.

4. Experimental Results

The experiments were conducted on two distinct platforms, with each configuration repeated 5 times and results reported as means, with the following configurations:

- **CPU:** Intel Xeon E7-4870 (4 sockets, 40 physical cores / 80 logical threads in total, 120 MiB total L3 cache segmented into 4 instances of 30 MiB), compiled with GCC 11.4 using the `-O3 -march=native -fopenmp` flags.
- **GPU:** NVIDIA Tesla T4 (2560 CUDA cores, 40 SMs, 64 KB of shared memory per SM) hosted in a Google Colaboratory virtualized environment, compiled with NVCC 12.0 using the `-O3 -arch=sm_75` flags.

The workload consisted of a 2D mesh of size 256×256 , executed for 8 iterations, totaling approximately 1.5M cells. The execution time obtained by the sequential *baseline* (run on a single thread) was 184605.0 ms.

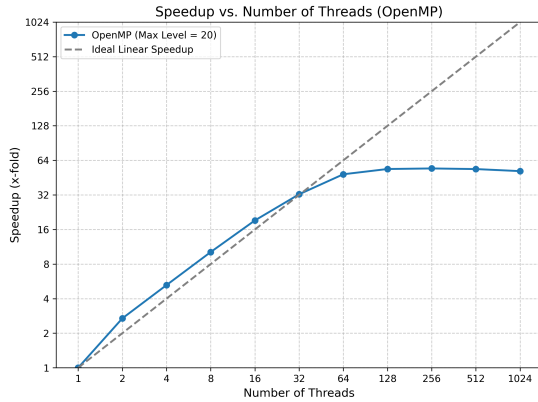


Figure 2. OpenMP scalability compared to ideal linear speedup.

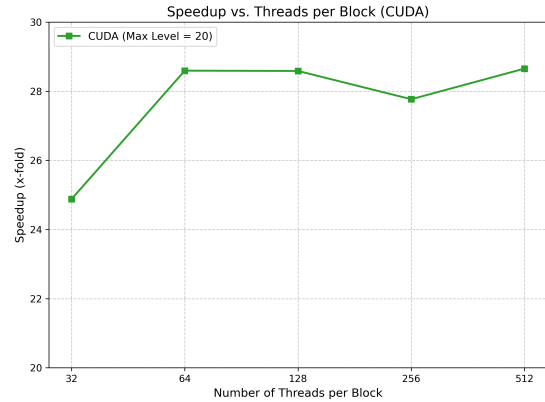


Figure 3. CUDA scalability in relation to CPU execution.

4.1. OpenMP Performance Analysis

OpenMP demonstrates superlinear scaling up to 32 threads ($32.40\times$ speedup, 101.3% efficiency), driven by the multi-socket NUMA architecture which aggregates memory

controllers and expands L3 cache capacity to mitigate irregular memory accesses (Figure 2). Leveraging Hyper-Threading beyond the 40 physical cores yields a peak speedup of $54.40\times$ at 256 threads. However, scaling beyond 256 threads on a system strictly limited to 80 logical threads degrades performance due to system overcommitting, increased scheduler management costs, and synchronization barrier overhead.

4.2. CUDA Performance Analysis

The CUDA implementation achieved a $28.65\times$ speedup (512 threads/block), underperforming the OpenMP peak by $1.9\times$ on this multi-socket infrastructure (Figure 3). Smaller blocks (32 threads) yielded only a $24.88\times$ gain due to SM underutilization. This $\sim 29\times$ speedup falls considerably short of theoretical estimates ($50\text{--}100\times$) due to three critical limitations: (1) warp divergence ($\sim 15\text{--}20\%$) during refinement decisions, (2) non-coalesced global memory accesses severely degrading the theoretical 320 GB/s bandwidth, and (3) parallel prefix sum overheads in large blocks. These irregular access patterns inherent to AMR were further compounded by unprofiled, yet high, PCIe transfer latencies required for continuous dynamic mesh updates.

5. Conclusion

This work evaluated the performance trade-offs of parallelizing AMR across multi-core CPUs and GPUs. OpenMP proved superior in the multi-socket enterprise environment, reaching a peak $54.40\times$ speedup (256 threads). The NUMA architecture and cumulative L3 cache effectively mitigated the latency of irregular memory accesses during tree balancing and merge phases.

Conversely, the Tesla T4 CUDA implementation ($28.65\times$) was severely bottlenecked by warp divergence, non-coalesced memory accesses, and virtualized cloud data-transfer latencies. Ultimately, the CPU’s high physical core count and aggregated bandwidth inverted the expected GPU advantage for this irregular workload, though this comparison reflects a high-end multi-socket server against a mid-tier GPU, and a top-tier HPC accelerator may narrow this gap. Future work will explore hybrid MPI+OpenMP approaches to scale across distributed HPC clusters.

References

- Berger, M. J. and Colella, P. (1989). Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64–84.
- Burstedde, C., Wilcox, L. C., and Ghattas, O. (2011). p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133.
- Holke, J., Knapp, D., et al. (2021). t8code: Modular adaptive mesh refinement in the exascale era. *arXiv preprint arXiv:2109.02729*.
- Morton, G. M. (1966). A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd.
- Qin, X., LeVeque, R. J., and Motley, M. R. (2019). Accelerating an adaptive mesh refinement code for depth-averaged flows using gpus. *Journal of Advances in Modeling Earth Systems*, 11(8):2606–2629.