Highly Asynchronous and Scalable Algorithms for Distributed-Memory Adaptive Mesh Refinement at Extreme Scales

Akhil Langer†, Laxmikant V. Kale

Parallel Programming Laboratory, Department of Computer Science
University of Illinois at Urbana-Champaign
{alanger, kale}@illinois.edu

***Abstract*—In this paper, we present our developments of a novel approach for distributed memory Adaptive Mesh Refinement (AMR). Our approach is highly asynchronous and fully distributed that makes it suitable for extreme-level scaling. It takes negligible memory to store mesh structure as compared to the traditional approaches which are not scalable. It accomplishes adaptive mesh restructuring in just 1 global collective call as compared to $O(d)$ calls in the traditional approaches. A new distributed load balancer has been developed that led to an improvement in performance by 18%. We present our scaling results on up to $131,072$ cores of BG/Q supercomputer.**

I. Introduction

Advanced numerical simulations on large meshes is a computationally daunting task. Adaptive Mesh Refinement (AMR) reduces the computation significantly by adaptively refining the mesh in the zones of interest while keeping the lesser or non-interesting zones at a coarser level. This allows users to solve large problems (12 levels or more) that are intractable on a uniform grid. AMR applications span a diverse set of fields starting from computational fluid dynamics to astrophysics, climate modeling, mantle convection modeling, combustion, biophysics, turbulence and many others.

II. Related Work

Paramesh[1], SAMRAI[2], FLASH[3], p4est[4], Enzo[5], Chombo[6], deal.II[7] are some of the many existing frameworks with parallel AMR implementation. We focus on the oct-tree based formulations. In these implementations, typically a mesh is divided into blocks and these blocks are distributed amongst the processes. The assignment is typically done using a space-filling curve[8] so that blocks that are spatially close to each other are assigned to the same or nearby processes. To maintain the AMR restriction of neighboring blocks to be within ± 1 level of refinement of each other, requires each process to replicate the entire refinement tree meta-data so that the tree can be adapted dynamically. Therefore, this traditional approach suffers from memory bottleneck. As the size of the supercomputers is increasing, available memory per core decreases and the total number of cores are increasing, making it infeasible to store the entire tree structure on each process.

†Student Author

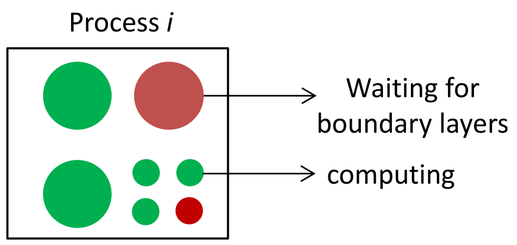

Fig. 1: Overdecomposition allows overlap of computation and communication of blocks on the same physical process

In traditional approaches, the adaptive mesh restructuring is accomplished through level-by-level - from the most refined leaf octant to the least refined leaf octant - decision making and mesh adaptation. This takes $O(d)$ global reductions - 2 for each refinement level (d is the depth of the refinement tree). These synchronization overheads in the traditional approaches inhibits their efficiency for large meshes and scalability at extreme scale. We present our novel algorithms that addresses all the bottlenecks in the traditional approach by proposing a fully distributed and highly asynchronous design capable of extreme scaling.

III. Our Approach

Instead of a process, we model a block as a first class entity - a basic schedulable unit that acts as a virtual processor. Multiple blocks are assigned to the same processor. These blocks can be at different refinement levels. Virtualization allows overlap of computation with communication of other blocks on the same physical process (Figure 1). A block can be uniquely identified with a bit-vector corresponding to its location in the refinement tree (Figure 2). It can be dynamically placed on any physical process, thus facilitating dynamic load balancing.

AMR enforces that REFINE decisions are given higher priority over STAY decisions , and STAY have a higher priority over COARSEN decisions. In order to maintain the difference of ± 1 refinement level amongst neighbors, a block may have

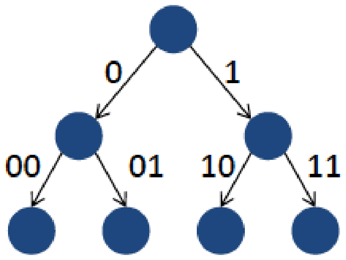

Fig. 2: Bit-vector indexing scheme used for identifying blocks in the refinement tree

to refine itself if any of its refined neighbor decides to refine, even when its personal decision was to stay at the current refinement level. This can trigger a chain reaction leading to refinement of far-off blocks. For this adaptive restructuring of mesh, we proposed a novel mesh restructuring algorithm that minimizes the number of messages exchanged and reaches a consensus on the refinement decisions by detecting a system quiescence state. Our new design offers several benefits over the traditional approach:

- $O(\frac{\#blocks}{P})$ per process memory complexity (by using a distributed hash table) vs $O(\#blocks)$ in the traditional approach
- Asynchronous progress in computation
- $O(d)$ global reductions for remeshing in the traditional approach are reduced to just 1 quiescence detection (d is the depth of the tree)
- Fully distributed load balancing that has $O(\log P + \frac{\#blocks}{P})$ time complexity vs centralized load balancing that has $O(\#blocks)$ space and time complexity in the traditional approach
- Simplified expression of program logic

IV. Results

We implemented our new design in Charm++[9] runtime system, which supports dynamic collections of migratable parallel objects in the form of *chare arrays*[10]. An element of chare array is called a chare. Each chare is assigned one block of the mesh. We therefore, use the words *block* and *chare* interchangeably. Keeping the block as a unit of algorithmic expression reduces the implementation complexity (refer to [9] for source lines of code count of our implementation). As the simulation progresses, blocks are refined or coarsened leading to load imbalance. Our fully distributed load balancer (referred to as *Grapevine*) uses partial information about the global state of the system to perform good load balancing with less overhead. The load balancing strategy uses gossip protocol to spread the information and performs probabilistic transfer of load (details in [11]). An efficiency of 80 % (as compared to just 58% without load balancing) was obtained while strong scaling a fixed-size 2D Advection benchmark (with maximum of 15 levels) from $1,024$ to $16,384$ cores of the Blue Gene/Q supercomputer. Our design promises high performance for much more deeply refined computations than are currently practiced and gives high efficiency at extreme scales. Complete working code for the algorithms and benchmark are available online [12] . Specific details of the algorithms can be found in the paper [13]. We have made several algorithmic improvements since the last publication. We did a major overhaul of the code and developed algorithms to extend our implementation to do 3d AMR simulations. Because of dynamic creation and destruction of chares, there is frequent load imbalance in the program runs. Centralized load balancing leads to a serial bottleneck that compensates for any performance benefit due to the created load balance. Hence, a distributed load balancer was developed which led to a significant improvement in performance. Additionally, we reduced the number of system quiescence required for mesh restructuring from 2 to 1, by buffering the messages directed to not yet created chares and delivering them once they are created.

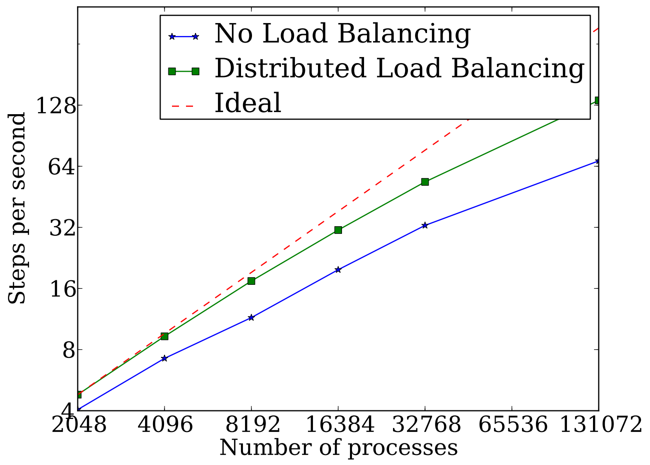

Fig. 3: Strong scaling results (steps per second) of Advection AMR simulation on up to 128k cores of BG/Q The plot shows the improvement in performance by using a distributed load balancer

V. Ongoing Work

Adaptive restructuring of the mesh, in our approach, is done in two phases - the refinement decision making phase, called phase 1 and the block creation and destruction phase, called phase 2. Completion of both of these phases is detected by system quiescence. Before proceeding to the next compute iteration, it is required to wait for quiescence to ensure that the new blocks have been created before the boundary messages are sent to them. However, this requirement can be relaxed by buffering the messages that are directed to chares that have not yet been created. These messages are then delivered when the chare gets created. With this approach, the second quiescence requirement becomes redundant.

Additionally, in phase 1, not all blocks need to wait for system quiescence to know their final decision. For example, a block all of whose neighbors are currently at the same refinement level and whose personal decision is also to stay at the current refinement level, then irrespective of the neighbors

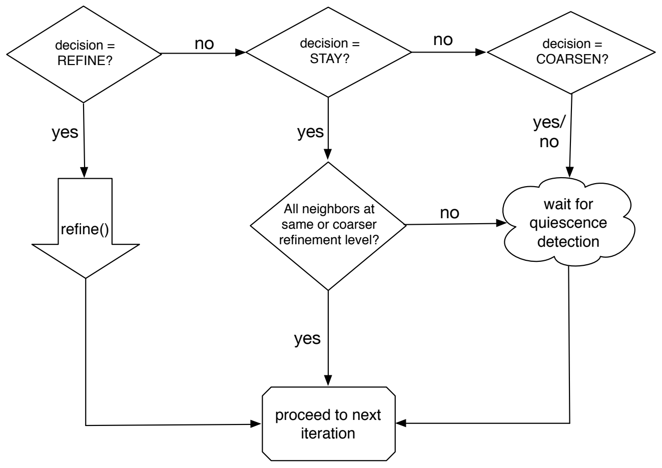
decision = REFINE?
no
decision = STAY?
no
decision = COARSEN?
yes
yes
yes/ no
refine()
All neighbors at same or coarser refinement level?
no
wait for quiescence detection
yes
proceed to next iteration

Fig. 4: Reducing the overhead of phase 1 by allowing chares to proceed to next iteration when they need not wait for quiescence

final decision, the block will not change its decision and hence can proceed to the next compute iteration. This and other rules shown in Figure 4 allow us to overlap computation with the decision making process of the remaining blocks. Additionally, only the blocks that need to wait for quiescence to know their final decision need to participate in the quiescence detection. This requires a modular quiescence detector in which the quiescence of only a selected set of chares and specific remote method calls needs to be detected. We are working on these ideas and on the measurement of the performance improvement due to them.

VI. Conclusion

Our approach is highly asynchronous and has led to several algorithmic advancements in a very important computational science application. We have demonstrated our results on two major supercomputers, BG/Q and Titan. Currently, two major AMR production codes namely, Enzo (renamed Cello) and Chombo, are being rewritten to incorporate these algorithms into their code.

References

[1] P. MacNeice, K. M. Olson, C. Mobarry, R. deFainchtein, and C. Packer, "PARAMESH : A Parallel Adaptive Mesh Refinement Community Toolkit.," *Computer Physics Communications*, vol. 126, pp. 330–354, 2000.

[2] A. M. Wissink, R. D. Hornung, S. R. Kohn, S. S. Smith, and N. Elliott, "Large Scale Parallel Structured AMR Calculations Using the SAMRAI Framework," in *Supercomputing, ACM/IEEE 2001 Conference*, pp. 22–22, IEEE, 2001.

[3] A. Dubey, L. B. Reid, and R. Fisher, "Introduction to FLASH 3.0, with Application to Supersonic Turbulence," *Physica Scripta*, vol. T132, p. 014046, 2008.

[4] C. Burstedde, L. C. Wilcox, and O. Ghattas, "p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees," *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011.

[5] B. Oshea, G. Bryan, J. Bordner, M. Norman, T. Abel, R. Harkness, and A. Kritsuk, "Introducing Enzo, an AMR Cosmology Application," *Adaptive mesh refinement-theory and applications*, pp. 341–349, 2005.

[6] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen, "Chombo Software Package for AMR Applications-Design Document," 2000.

[7] W. Bangerth, R. Hartmann, and G. Kanschat, "deal. IIA General-Purpose Object-Oriented Finite Element Library," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 4, pp. 24–es, 2007.

[8] H. Sagan, *Space-filling Curves*, vol. 2. Springer-Verlag New York, 1994.

[9] L. Kale, A. Arya, N. Jain, A. Langer, J. Lifflander, H. Menon, X. Ni, Y. Sun, E. Totoni, R. Venkataraman, and L. Wesolowski, "Migratable objects + active messages + adaptive runtime = productivity + performance a submission to 2012 HPC class II challenge," Tech. Rep. 12-47, Parallel Programming Laboratory, November 2012.

[10] O. S. Lawlor and L. V. Kalé, "Supporting dynamic parallel object arrays," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 371–393, 2003.

[11] H. Menon and L. Kale, "Epidemic Algorithm for Distributed Load Balancing," in *International Conference on Supercomputing 2013 (Under Review)*, (Eugene, Oregon, USA).

[12] "AMR Algorithm and Benchmark Source Code," 2012. Source code and scripts available at `git://charm.cs.illinois.edu/benchmarks/amr.git`.

[13] A. Langer, J. Lifflander, P. Miller, K.-C. Pan, , L. V. Kale, and P. Ricker, "Scalable Algorithms for Distributed-Memory Adaptive Mesh Refinement," in *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2012). To Appear*, (New York, USA), October 2012.