Formation of the First Galaxies
Enzo-P / Cello Adaptive Mesh Refinement

James Bordner, Michael L. Norman, Brian O'Shea

June 20, 2013

Abstract

Enzo [8] is a mature and highly successful parallel adaptive mesh refinement (AMR) application for computational astrophysics and cosmology. In the early 1990's when development on Enzo began, "extreme scale" meant hundreds of CPU's, not today's hundreds of thousands of cores; thus, the design decisions made in the formative years of Enzo's development were based on being scalable to only hundreds of processors. Attempting to improve Enzo's scalability by over $\times 1000$ to keep pace with the relentless increase in HPC platform parallelism has become increasingly difficult—many of Enzo's current scaling bottlenecks cannot be resolved without a fundamental overhaul of its parallel AMR design and implementation. This motivated us to develop a "petascale Enzo" fork of the Enzo code base called *Enzo-P*, using a new, highly scalable AMR framework called *Cello* that has been developed concurrently. Key features of Cello are that it implements a "forest of octrees" AMR approach, and Cello is parallelized using Charm++ rather than MPI. This report summarizes the work to date, including Cello's adaptive mesh refinement design and implementation and preliminary performance results.

1 Introduction

Enzo-P [4][8] is a "petascale" fork of the Enzo code base, which uses an AMR framework called *Cello* [1] [2] that is designed to be efficient, highly scalable, and usable on petascale class machines. Key features of Cello are that it implements a "forest of octrees" AMR approach, and Cello is parallelized using Charm++ [5] rather than MPI. Octrees are used due to their relatively simple refinement algorithm, smaller tree node size, and greater number of nodes (more available parallelism). A "forest" of octrees was used because it provides a flexible initial refinement granularity, it naturally allows for non-cubical domains, and its demonstrated scalability in practice [3]. Charm++ was selected over other parallelization strategies such as MPI or UPC due to its data-driven asynchronous execution, latency-tolerance, load-balancing algorithms, and fault-tolerance.

While Enzo-P / Cello is designed from the ground up to be extremely scalable, its performance and scalability have been untested in practice. Our goals for this project have been to design a realistic demonstration simulation, run scalability tests, identify any performance or scalability limitations, and optimize Enzo-P / Cello for Blue Waters. Due to unexpected delays in the design and implementation of Cello's parallel AMR—on which this project depends—we have yet to fully meet these goals. However, we will continue work under the auspices of NCSA and the NSF until all of the above goals have been met.

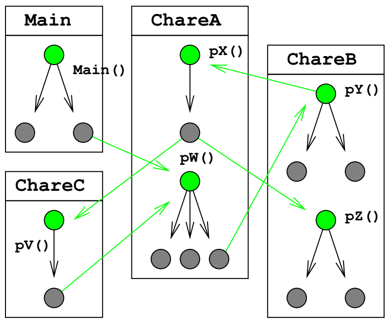
We begin this report with a short review of Charm++, followed by a description of Cello's adaptive mesh refinement design and implementation, including the core mesh adaptation and ghost zone refresh phases. We conclude with preliminary performance results for a small adaptive mesh refinement problem.

2 Cello Adaptive Mesh Refinement

Cello's adaptive mesh refinement (AMR) uses a forest of octrees approach, with each leaf of an octree containing a block of data. This block may contain both field and particle data (only field data is currently implemented), and the size of blocks (number of zones in a block's field data) is tunable to optimize parallel task granularity. Parallel communication, synchronization, and task scheduling are all under the control of Charm++.

2.1 Charm++

The fundamental object in Charm++ is a *chare*. Chares are C++ objects that contain special methods called *entry methods*. Entry methods may be invoked remotely by other chares via *proxies*, and communication is enabled by *messages* passed as arguments through entry methods. Chares may be grouped together into a *chare array*, which enables referencing multiple chares using a single array proxy plus element indices. Additionally, the Charm++ runtime system supports automatic dynamic load balancing of chares within chare arrays. The *runtime system* manages chares, assigning their position in distributed memory, migrating chares to dynamically balance load, communicating message data between chares, and scheduling and executing entry methods.



Charm++ arrays also support global collectives, via contribute() and an associated reduction client. Also provided is *quiescence detection*, defined as "the state in which no processor is executing an entry point, no messages are awaiting processing, and there are no messages in-flight." [6]. Quiescence detection is currently used in Cello to simplify book-keeping, since the number of incoming messages in some communication phases is not necessarily known a priori.

To encapsulate Charm++ within Cello, blocks are implemented using two classes, *CommBlock* chare objects and *Block* C++ objects. The CommBlock object contains the essential parallel synchronization, communication and parallel flow-of-control methods and entry methods, whereas the Block object contains all of the data and methods associated with application data on a block. Each CommBlock object contains exactly one Block object. CommBlocks control the synchronization and communication of the main adaptive mesh refinement operations, including mesh creation, mesh adaptation, ghost data exchange between adjacent CommBlocks, and coordinating parallel I/O.

2.2 Mesh creation and adaptation

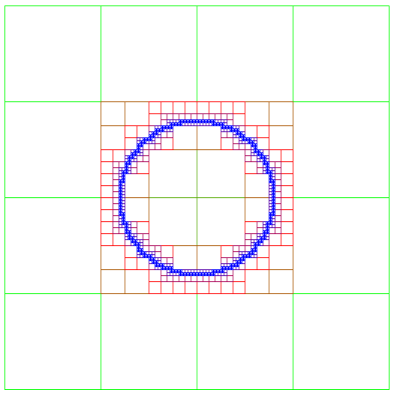
Cello's AMR mesh is represented as a chare array in Charm++, which allows for indexing of remote elements through Charm++'s internal distributed hash table. Charm++ uses a default key size of three 32-bit ints. Cello currently assigns one int per coordinate axis, with each 32-bit int partitioned into 10 bits for the array index, 20 bits for the octree index, and the remaining 2×3 bits collectively encoding the refinement level of the associated CommBlock. This allows for a forest of up to 1024^3 root CommBlocks, plus an additional 20 levels of octree mesh refinement. This implementation is sufficient for most cosmological structure formation simulations, though it may be desirable to tune for different problems, e.g. if more refinement levels are required, as in galaxy or star formation simulations. If even greater forest sizes or AMR depths are required than the available 96-bits allow, Charm++ can be configured and re-compiled to use even longer hash table keys.

These CommBlock chare array indices are encapsulated in a C++ Index class in Cello. Any CommBlock can reference its parent, children, and adjacent CommBlocks through simple bit operations on its own Index. This allows CommBlocks to invoke entry methods on neighboring CommBlocks in the chare array without explicitly having to store and maintain their individual proxies. However, an entry method cannot be invoked on a non-existent chare, so each CommBlock must still maintain the state of neighboring CommBlocks.

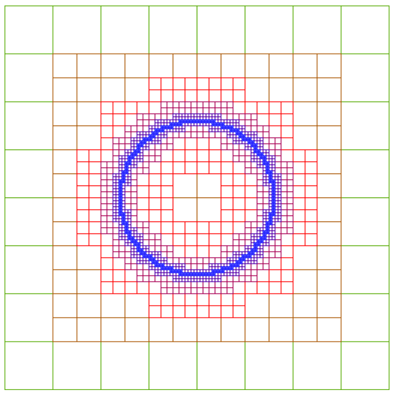
At the beginning of the mesh adaptation step, each CommBlock applies a set of refinement criteria on its Block to determine whether the Block should be refined, coarsened, or remain the same. Refinement and coarsening steps are currently separated by a quiescence detection, though this will be optimized out. Our initial implementation errs on the side of correctness versus performance, since it uses more synchronization than necessary; for example, the initial mesh creation step may be implemented using at most a single global synchronization independent of number of levels [7]. We describe the refinement and coarsening steps below in Sections 2.2.1 and 2.2.2.

2.2.1 Refining CommBlocks

If a CommBlock is tagged for refinement, then it creates eight new child CommBlocks and interpolates (*prolongs*) its Block data to the new child Blocks. Additionally, each CommBlock must maintain the state of neighboring CommBlocks so that they can call the appropriate ghost refresh entry methods on the appropriate neighboring CommBlocks. This state information currently consists of the refinement level of neighboring CommBlocks along each face, including



Unbalanced mesh: resolution differences between adjacent CommBlocks may be greater than $r = 2$



Balanced mesh: 2-to-1 refinement constraint is enforced

Figure 1: Generated meshes with mesh balancing off (left) and on (right)

corners (0-faces) and edges (1-faces) as well as facets (2-faces). Face levels for a given CommBlock and given face are set and updated using a p_set_face_level(level) entry method.

Refinement may trigger a *balancing refinement* in adjacent coarse blocks to maintain the 2-*to*-1 *refinement constraint* that adjacent blocks must be within neighboring refinement levels (see Figure 1). If the CommBlock that is tagged for refinement is adjacent to a coarser CommBlock, then the CommBlock tags its coarser CommBlock neighbor for refinement so that the 2-to-1 refinement constraint is not violated. This refinement may recursively trigger further balance refinements, though only in CommBlocks in the next-coarser level, so this process necessarily terminates.

The refined parent CommBlock calls p_set_face_level(level) on each face of each new child CommBlock, and on each shared face of each neighbor. Figure 2 illustrates the different cases involved in updating face levels of child CommBlocks and adjacent CommBlocks. Different steps are required depending on whether the adjacent CommBlock on a given face is a sibling, cousin, uncle, or nibling. A *sibling* is an adjacent CommBlock in the same mesh level that shares the same parent, a *cousin* is an adjacent CommBlock in the same mesh level that is not a sibling, an *uncle* is an adjacent CommBlock in the next-coarser level, and a *nibling* is an adjacent CommBlock in the next-finer level. Each case in the Figure shows all neighbors in the same refinement level, though in general different neighbors on different faces will be in different refinement levels.

Care must be taken since these face level updates are performed asynchronously. First, when updating the face level for refinement, the maximum level is used rather than assignment to avoid race conditions. Secondly, when updating an uncle's face levels, the update must test whether the uncle has itself refined. If so, p_set_face_level() is called once recursively on the uncle's child CommBlocks that are adjacent to the original parent CommBlock.

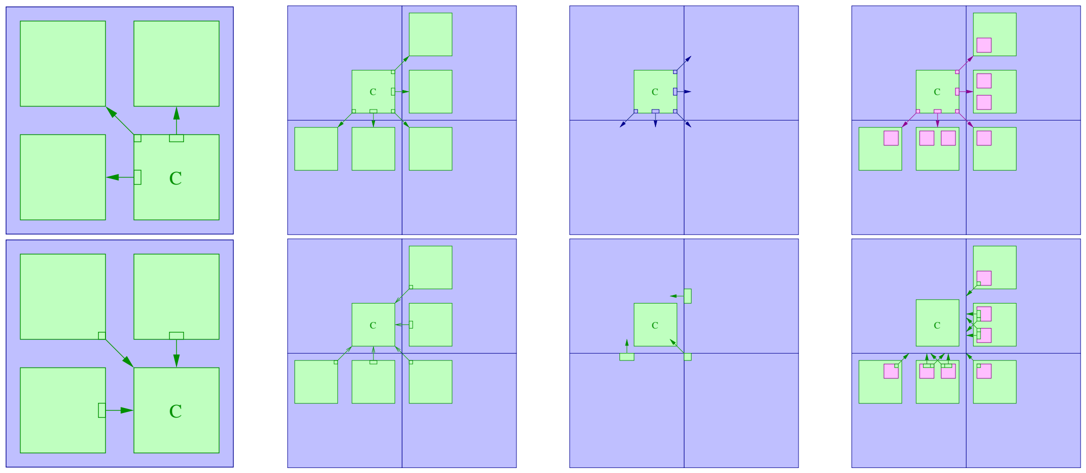
2.2.2 Coarsening CommBlocks

If a CommBlock is tagged for coarsening, then it sends its coarsened (*restricted*) Block data to its parent by calling the parent's p_child_can_coarsen() entry method. If this is called by all child CommBlocks of a parent, then the parent copies all the collected child Block data into its own Block, and deletes the child CommBlocks.

The coarsening operation is simpler than refinement since it cannot trigger a balance refinement (a CommBlock would not be able to coarsen in the first place if doing so would violate the 2-to-1 refinement constraint), and because child CommBlock face levels do not need to be updated since the child CommBlocks are to be deleted.

Figure 3 illustrates the different cases involved in updating face levels of adjacent CommBlocks. Additionally, the face levels of the parent CommBlock must be updated since it will become a new leaf in the octree, and all leaves must maintain up-to-date face level state information. This is done by including the face level information of child CommBlock's in calls to p_child_can_coarsen(), which the parent CommBlock copies into a temporary array, and accesses when it is able to coarsen.

Again, care must be taken since the face level updates are performed asynchronously. Similarly to the refinement



Sibling CommBlocks: *new child CommBlock's sibling face levels are set to level+1 and reciprocal face levels are set to level+1*

Cousin CommBlocks: *new child CommBlock's cousin face levels are set to level+1 and reciprocal face levels are set to level+1*

Uncle CommBlocks: *new child CommBlock's uncle face levels are set to level and reciprocal face levels are (recursively) set to level+1*

Nibling CommBlocks: *new child CommBlock's nibling face levels are set to level+2 and reciprocal face levels are set to level+1*

Figure 2: CommBlock refinement: updating face levels for sibling, cousin, uncle, and nibling neighbors

step, when updating the face level for coarsening, the minimum level is used rather than assignment to prevent race conditions.

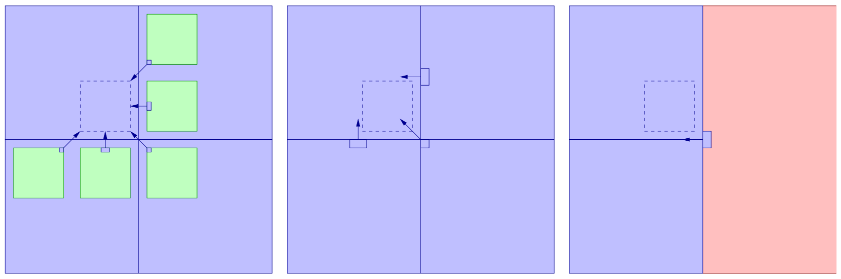
2.3 Ghost zone refresh

Ghost data must be refreshed each cycle to ensure that numerical computations on each Block are accurate. There are three types of ghost data refreshing operations, depending on whether the neighboring CommBlock is in a coarser level, the same level, or a finer level. Whether a neighbor on a given face is coarser, finer, or in the same level is determined by the face level state information maintained in the mesh adaptation phase described above in Section 2.2.

The fine-to-coarse and coarse-to-fine steps are illustrated in Figure 4 (the ghost refresh step between CommBlocks in the same level are similar, but do not involve any data restrictions or prolongations, just copies.) For sending face data from a CommBlock to a coarse neighbor, the Block's face data are first restricted, then sent to the coarse neighbor as a Charm++ message argument through the x_refresh_coarse() entry method. The receiving CommBlock then copies the neighbor's face data to its corresponding ghost zones in its Block. Similarly, for sending ghost data from a CommBlock to a fine neighbor, the Block's face data are sent to the fine neighbor as a Charm++ message argument through the x_refresh_fine() entry method. The receiving CommBlock then prolongs the neighbor's face data before copying it to its corresponding ghost zones in its Block. By performing the restriction before sending to a coarser CommBlock and performing the prolongation after sending to a finer CommBlock, $8\times$ less data is sent than if the restriction was done at the receiving end or the prolongation at the sending end.

3 Performance

The problem we use to test performance is a variation of the Sedov Blast hydrodynamics test problem. This 3D problem involves a regular array of strong explosions induced by points of high energy in a uniform medium, with periodic boundary conditions. Due to delays in software development of Cello from design and implementation modifications since the beginning of this project, we can only show performance for a very small parallel problem here. In this case, $P = 16$ processors are used for a $L = 3$ level adaptive mesh hierarchy.

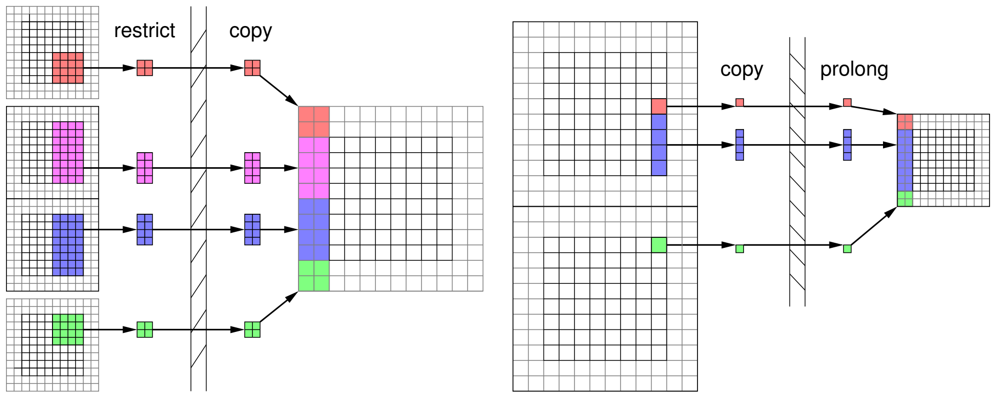


Cousin CommBlocks: *deleted child CommBlock's cousin face levels are set to level*

Uncle CommBlocks: *deleted child CommBlock's uncle face levels are set to level*

Great-uncle CommBlocks: *deleted child CommBlock's great-uncle face levels are set to level*

Figure 3: CommBlock coarsening: updating face levels in cousin, uncle, and great-uncle neighbors



Fine-to-coarse: *The fine CommBlock first restricts its face values, then sends the restricted values to the neighboring coarse CommBlock*

Coarse-to-fine: *The coarse CommBlock sends its face values to the neighboring fine CommBlock, which then prolongs the values*

Figure 4: Refreshing ghost zones

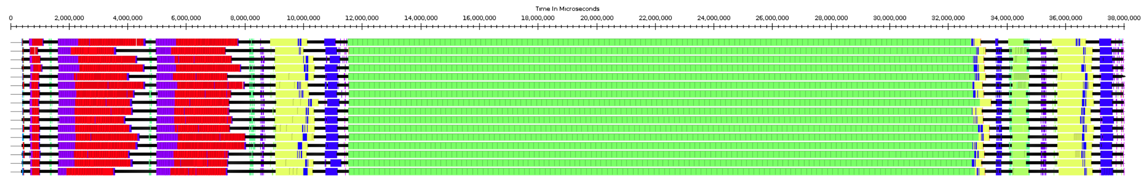


Figure 5: Timeline for a small AMR test problem

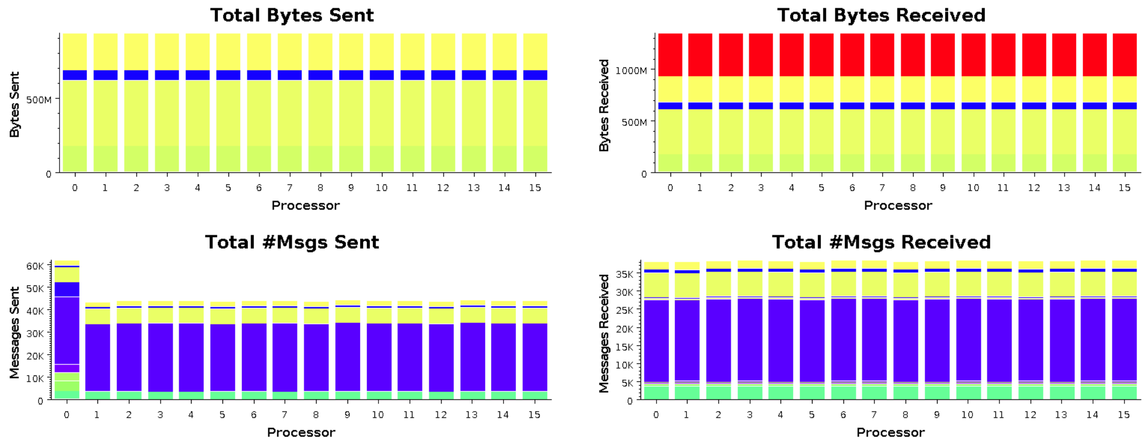


Figure 6: Communication for a small AMR test problem

In Figure 5 is a timeline showing a single computational cycle, including the initial mesh creation. The simulation begins with three phases of mesh refinement from $t \approx 1s$ to $9s$, with each refinement phase creating an additional refinement level. Magenta entry methods are p_adapt_start(), which begin each refine / coarsen cycle, and red entry methods are calls to the CommBlock constructor EnzoBlock(). Green methods at $\approx 1.4s$, $4.8s$ and $8.2s$ are calls to remoteDoneInserting(), which is a Charm++ entry method used to mark the end of a phase of inserting elements in a chare array. Ghost refresh begins with yellow p_refresh_begin() calls at $\approx 9s$, and x_refresh_fine() (yellow), x_refresh_coarse() (blue), and x_refresh_same() (blue) calls at around $10s$. A quiescence detection phase ends with q_refresh_end() in blue at $11s$. PPM computation then begins in green at $11.5s$ and continues to $33s$. We note that 2192 CB's are generated, each of size 32^3. Earlier serial performance tests have indicated that PPM on such a grid completes a cycle in about $0.1s$, indicating a relative parallel efficiency of roughly $\frac{2192\times0.1s}{16}/(33s-11.5s)=0.64$.

In the top of Figure 6 is the distribution of communication amount in terms of "Total Bytes Sent" and "Total Bytes Received" between CommBlocks. Colors in "Total Bytes Sent" indicate, from the bottom up, p_child_can_coarsen() (yellowish green), x_refresh_same() (greenish yellow), x_refresh_fine() (blue) and x_refresh_coarse() (yellow). For "Total Bytes Received", the colors are the same, but with the addition of red on top, representing the CommBlock constructor EnzoBlock(). In terms of the amount of data sent between processors, communication is nearly perfectly balanced, though the underlying problem is inherently load balanced.

In the bottom of Figure 6 is the distribution of communication in terms of numbers of messages sent and received through Charm++ entry methods. For processors 1 through 16, the magenta bars are p_set_face_level(), and the other four correspond to p_child_can_coarsen(), x_refresh_same(), x_refresh_fine(), and x_refresh_coarse() as in "Total Bytes Sent". The other four entry methods for the root processor in "Total #Msgs Sent" are, from the bottom up, q_adapt_end(), q_adapt_stop(), q_adapt_next(), and q_refresh_end(). By convention, Cello callback methods involved in quiescence detection are identified with a prefix of q_, so these all mark the end of a quiescence detection phase. q_adapt_end() ends the mesh creation phase, q_adapt_stop() ends each refine / coarsen mesh creation step, and q_adapt_next() ends the refine step. Here the number of messages received is balanced, but number of messages sent is greater for the root process. Since the additional messages are all from quiescence detection callback methods, reducing or eliminating the need for quiescence detection should bring the communication more into balance.

4 Conclusions and Future Work

We are excited to have reached the milestone in Enzo-P / Cello development where Enzo's PPM hydrodynamics and PPML ideal MHD solver kernels can be applied to problems using both Charm++ for parallelism and Cello's forest-of-octree adaptive mesh refinement. Preliminary performance tests also look promising, especially given that the initial implementation errs on the side of correctness versus performance.

Since our performance work has been necessarily pushed back due to software development delays, we still have much work to do in performance measurement, analysis, and optimization. Some immediately known performance improvements include the following:

1. Allow for simultaneous coarsening and refining of CommBlocks to eliminate the synchronization between phases.
2. Remove or eliminate quiescence detection calls, which introduce unnecessary global synchronization and imbalance in number of messages sent.
3. Do not send restricted child Block data to parent CommBlocks in p_child_can_coarsen(), since the data sent is never accessed unless all eight child CommBlocks are able to coarsen.
4. Remove the need for synchronization between multiple adapt phases at startup. This is a one-time cost, but has been shown in [7] to be possible.
5. Only refresh ghost zones once per cycle instead of twice. Currently twice is required since the mesh refinement criteria is based on the relative slope of the density field, which requires accessing ghost zones.
6. Remove the restriction on ProlongLinear that requires an even number of ghost zones, so that Enzo-P can run with PPM's 3 ghost zones rather than the current 4.
7. Support adaptive time-stepping, which will reduce computation (hence power consumption) by $O(L)$ on average for an AMR mesh with L levels. Currently a single global timestep is used.

References

[1] James Bordner. The Cello Project (website). `<http://cello-project.org>`, 2013.

[2] James Bordner and Michael L. Norman. Enzo-P / Cello: Scalable adaptive mesh refinement for astrophysics and cosmology. In *Proceedings of the Extreme Scaling Workshop*. Blue Waters and XSEDE, ACM Digital Library, 2012.

[3] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[4] Enzo Development Community. The Enzo Project (website). `<http://enzo-project.org>`, 2013.

[5] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[6] Parallel Programming Laboratory. The Charm++ parallel programming system manual, version 6.5.0, 2013.

[7] Akhil Langerz, Jonathan Lifflanderz, Phil Millerz, Kuo-Chuan Pan, Laxmikant V. Kalez, and Paul Ricker. Scalable algorithms for distributed-memory adaptive mesh refinement. In *International Symposium on Computer Architecture and High Performance Computing*, 2012.

[8] Brian O'Shea, Greg Bryan, James Bordner, Michael Norman, Tom Abel, Robert Harkness, and Alexei Kritsuk. Introducing Enzo, an AMR cosmology application. In Tomasz Plewa, Timur Linde, and V. Gregory Weirs, editors, *Adaptive Mesh Refinement - Theory and Applications*, volume 41 of *Lecture Notes in Computational Science and Engineering*, pages 341–349. Springer Berlin Heidelberg, 2005. 10.1007/3-540-27039-6_24.