Enzo-P / Cello
Scalable Adaptive Mesh Refinement
for Astrophysics and Cosmology

James Bordner\textsuperscript{1}  Michael L. Norman\textsuperscript{1}  Brian O’Shea\textsuperscript{2}

\textsuperscript{1}University of California, San Diego
San Diego Supercomputer Center

\textsuperscript{2}Michigan State University
Department of Physics and Astronomy

Extreme Scaling Workshop 2012
Blue Waters / XSEDE
Parallel astrophysics and cosmology
- implemented in C++ / Fortran
- approximately 150K SLOC
- parallelized using MPI / OpenMP

Vast range of scales
- astrophysical fluid dynamics
- hydrodynamic cosmology

Adaptive mesh refinement (AMR)

Growing development community

[ Norman et al ]
Enzo’s physics and algorithms

- Eulerian hydrodynamics
  - piecewise-parabolic method (PPM)
- Lagrangian dark matter
  - particle-mesh method (PM)
- Self-gravity
  - FFT’s on root-level grid
  - multigrid on refinement patches
- Local physics
  - heating, cooling, chemistry, etc.
- MHD, RHD (ray-tracing or implicit FLD)
Enzo’s pursuit of scalability

- Enzo born in early 1990’s
- “Extreme” meant 100 processors
- Continual scalability improvements
  - MPI/OpenMP parallelism
  - “neighbor-finding” algorithm
  - I/O optimizations
- Further improvement getting harder
  - increasing scalability requirements
  - easy improvements already made
- Motivates concurrent rewriting
  - **Enzo-P** “petascale” Enzo fork
  - **Cello** AMR framework
Enzo’s AMR data structure

- Patch-based SAMR
- Each patch is a C++ object
- Patches assigned to processes

- $N_P$ root patches, grid size $\approx 64^3, 128^3$
- Refinement patches generally smaller
- Refinement patches initially local to parent
- Load balancing relocates refinement patches
- Patch data (grids, particles) are distributed
- AMR hierarchy structure is replicated

[ Tom Abel, John Wise, Ralf Kaehler ]
Enzo’s timestepping

- Adaptive timestepping by level
  - parallel within level
  - less computation by $O(N_L)$
  - reduced parallel efficiency

- EvolveLevel($L, dt_{L-1}$)
  1. refresh level $L$ ghosts
  2. compute timestep $dt_L$
  3. advance level $L$ by $dt_L$
  4. EvolveLevel($L+1, dt_L$)
  5. correct fluxes
Enzo’s scaling issues

- Memory usage
  - AMR structure is non-scalable
  - ghost zone layer three zones deep
  - memory fragmentation
- Data locality
  - disrupted by load balancing
- Parallel task definition
  - widely varying patch sizes
  - granularity determined by AMR
- Parallel task scheduling
  - parallel within a level
  - synchronization between levels
Talk outline

1. Enzo
   1. overview
   2. design
   3. scaling issues

2. Talk outline

3. Enzo-P / Cello
   1. overview
   2. design
   3. scaling solutions
   4. implementation using Charm++
   5. recursively generated parallel data structures
**Enzo-P / Cello overview**

- **Enzo-P** intended to be “petascale Enzo”
- **Cello** is a scalable AMR framework
- Parallelism using **Charm++**
  - MPI as a backup
- 25K SLOC
- Work in progress
  - PPM HD / PPML MHD on distributed Cartesian grids
  - prototyping Charm++ AMR implementations
Tree-based SAMR

- **Patches** define *unit of refinement*
  - cubical, varying size
- **Blocks** define *parallel tasks*
  - flexible size and shape
- One Block per Charm++ chare
- Tree structure optionally distributed
Cello’s timestepping

- Optional adaptive timestepping
  - includes by Patch or Block
  - local synchronization
  - parallelism between levels

- Quantized timesteps
  - avoids “sliver timesteps” $dt \approx 0$
  - but $dt$’s smaller than optimal
  - reduced numerical roundoff
Cello’s improvements to scaling
Reducing replicated AMR structure

Fewer replicated patches

- “patch-merging” technique
- truncates full subtrees
- \( \approx 2 \) to \( 3\times \) reduction
Cello’s improvements to scaling
Reducing replicated AMR structure

Fewer replicated patches
- “patch-merging” technique
- truncates full subtrees
- $\approx 2\text{ to } 3\times$ reduction

Smaller replicated patches
- Enzo: $|\text{grid}| = 1544$ bytes
- Cello: $|\text{Node}| = 16$ bytes
- $\approx 100\times$ reduction!
Cello’s improvements to scaling

Distributed AMR Structure

Forest of Trees
- each assigned a process range
- simple indexing
- load balancing issues
Cello’s improvements to scaling

Distributed AMR Structure

Forest of Trees
- each assigned a process range
- simple indexing
- load balancing issues

Space-filling curve
- improved load balancing
- requires global scan
- greater surface area
Cello’s improvements to scaling
Dynamically load balancing AMR block data

Dynamic Load Balancing
- Use Charm++ load balancing
- Space-filling curves
  - equally distribute load
  - maintain data locality
  - no parent-child communication
- Use measured performance
  - computation
  - memory usage
Task definition

- Flexible choice of size / shape
- Reduced size variability
  - 😊 constant grid Block sizes
  - 😞 variable subcycling, ≠ particles
Cello’s improvements to scaling

Parallel tasks

**Task definition**
- Flexible choice of size / shape
- Reduced size variability
  - 😊 constant grid Block sizes
  - 😞 variable subcycling, # particles

**Task scheduling**
- Charm++: asynchronous, data-driven
- Blocks advance when ghosts refreshed
Cello’s implementation using Charm++

Charm++ program structure

- Charm++ program
  - Charm++ objects are *chares*
  - invoke *entry methods*
  - communicate via *messages*

- Charm++ runtime system
  - maps *chares* to processors
  - schedules entry methods
  - migrates *chares* to load balance

- Additional scalability features
  - fault tol.: checkpoint/restart
  - dynamic load balancing
Cello’s implementation using Charm++

Charm++ collections of chares

**Chare Arrays**

- distributed array of chares
- migratable elements
- flexible indexing
Cello’s implementation using Charm++

Charm++ collections of chares

**Chare Arrays**
- distributed array of chares
- migratable elements
- flexible indexing

**Chare Groups**
- one chare per processor (non-migratable)
Cello’s implementation using Charm++

Charm++ collections of chares

Chare Arrays
- distributed array of chares
- migratable elements
- flexible indexing

Chare Groups
- one chare per processor (non-migratable)

Chare Nodegroups
- one chare per node (non-migratable)
Cello’s implementation using Charm++

Three implementation strategies

1. Single chare array
   - efficient: single access
   - restricted Tree depth

Hierarchy

```
H
TPBBPBBTPBBBTPBBBTPBBBTPBBBTP
```
Cello’s implementation using Charm++

Three implementation strategies

1. Single chare array
   - efficient: single access
   - restricted Tree depth

2. Composite chare arrays
   - “tree” of chare arrays
   - less restricted Tree depth
Cello’s implementation using Charm++

Three implementation strategies

1. Single chare array
   - efficient: single access
   - restricted Tree depth

2. Composite chare arrays
   - “tree” of chare arrays
   - less restricted Tree depth

3. Singleton chares
   - no depth restrictions
   - possible performance issues
   - how to generate...?
Recursively generated parallel data structures
Generating a “software network”

1. Start with single Seed
   - conceptually complete p.d.s.
   - with processor range

2. `grow()` spawns remote Seeds
   - conceptually partitioned p.d.s.
   - with processor subranges
   - Seeds interlinked

3. Recurse to individual elements
   - final Seeds complete p.d.s.
   - `scaffolding`: previous Seeds
Recursively generated parallel data structures
Using the software network

- Three types of Seed links
  - *parent*: reductions
  - *neighbor*: collaborations
  - *child*: distributions

- Scalable
  - links per node: $O(1)$
  - generation: $\approx O(\log N)$

- Load balancing
  - space-filling curves
  - hierarchical

- Usable in Cello
  - grids / octrees “easy”
  - SeedGrid, SeedTree, etc.

- Usable with MPI
  - send encoded seed creation
  - link: process rank + pointer
Summary

<table>
<thead>
<tr>
<th>Parallelization</th>
<th>Enzo</th>
<th>Enzo-P / Cello</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMR</td>
<td>MPI</td>
<td>Charm++</td>
</tr>
<tr>
<td>AMR structure</td>
<td>patch-based</td>
<td>tree-based</td>
</tr>
<tr>
<td>Remote patches</td>
<td>replicated</td>
<td>optionally distributed</td>
</tr>
<tr>
<td>Timestepping</td>
<td>1544 bytes</td>
<td>16 bytes</td>
</tr>
<tr>
<td>Block sizes</td>
<td>level-adaptive</td>
<td>block-adaptive</td>
</tr>
<tr>
<td>×1000 variation</td>
<td>level-parallel</td>
<td>constant</td>
</tr>
<tr>
<td>Task scheduling</td>
<td>patch migration</td>
<td>dependency-driven</td>
</tr>
<tr>
<td>Load balancing</td>
<td>LB conflict</td>
<td>space-filling curves</td>
</tr>
<tr>
<td>Data locality</td>
<td></td>
<td>no LB conflict</td>
</tr>
</tbody>
</table>

http://cello-project.org

NSF PHY-1104819, AST-0808184