Petascale Enzo: Software Infrastructure Development and Community Engagement

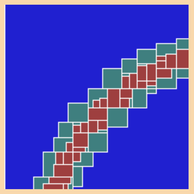
Michael L. Norman[1] – **James Bordner**[1] – Tom Abel[2] – Greg Bryan[3] – David Collins[4] – Brian O'Shea[5] – Daniel Reynolds[6] – Matt Turk[7] – John Wise[8]

[1]U.C. San Diego – [2]Stanford U. – [3]Columbia U. – [4]Florida State U. – [5]Michigan State U. – [6]Southern Methodist U. – [7]U. of Illinois – [8]Georgia Tech

Project Motivation

Enzo's struggles

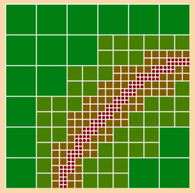
Enzo is a community-developed adaptive mesh refinement simulation code designed for rich multi-physics hydrodynamic astrophysical calculations. Unfortunately, Enzo's scalability is limited by its design and implementation.



- **Memory usage**
  - ▷ AMR structure is not scalable
  - ▷ permanent 3-layer ghost zones
  - ▷ memory fragmentation
- **Data locality**
  - ▷ disrupted by load balancing
- **Mesh quality**
  - ▷ 2-to-1 refinement violated
- **Parallel task definition**
  - ▷ varying patch sizes
  - ▷ determined by AMR
- **Parallel task scheduling**
  - ▷ parallel within a level
  - ▷ synchronization between levels

Enzo-P/Cello's solutions

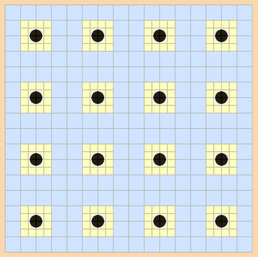
Enzo-P is being developed to bring Enzo's powerful multi-physics capabilities to petascale and beyond. Our approach is to build on Cello, a scalable forest-of-octree AMR framework implemented using Charm++.



- **Memory usage**
  - ▷ AMR structure fully distributed
  - ▷ *ghosts allocated on demand*
  - ▷ uniformity reduces fragmentation
- **Data locality**
  - ▷ nearest-neighbor communication
- **Mesh quality**
  - ▷ 2-to-1 refinement maintained
- **Parallel task scheduling**
  - ▷ asynchronous, data-driven
  - ▷ *block-local time stepping*
- **Parallel task definition**
  - ▷ uniform sized blocks
  - ▷ user-specified size

Enzo-P Scaling

Preliminary scaling tests on Blue Waters indicate that Cello AMR can scale extremely well, especially in terms of memory.

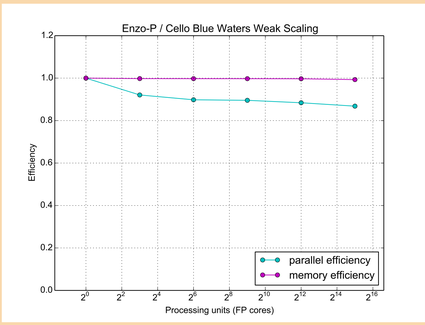


Test Problem

- 3D "array" of Sedov blasts
- One blast per Blue Waters FP core
- 16^3 grid zones per block
- **10M** blocks in 3-level AMR mesh
- Impossible with Enzo: **35GB** / process!

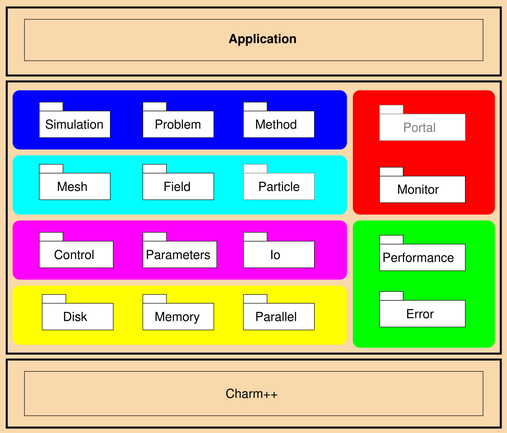
Parallel Scaling

- Efficiency at **P = 32K** FP cores:
  - ▷ parallel: $\epsilon_P = 0.868$
  - ▷ memory: $\epsilon_M = 0.993$



Cello Components

Cello is organized into layered components for representing scientific problems, operating on fully-distributed data-structures, and interfacing with hardware, the user, and (later) other applications.



High-level

`Simulation`:	represents a numerical simulation
`Problem`:	defines a multiphysics problem
`Method`:	implements a computational method

Data structures

`Mesh`:	represents an AMR mesh
`Field`:	represents Eulerian field data
`Particle`:	represents Lagrangian particle data

Middle-level

`Control`:	handles sequencing and synchronization
`Parameters`:	reads and manages input file parameters
`Io`:	coordinates disk data output

Hardware interface

`Disk`:	interfaces to file format libraries
`Memory`:	manages dynamic memory allocation
`Parallel`:	interface to parallel library (depreciated)

User interface

`Portal`:	*interacts with other applications*
`Monitor`:	informs user of simulation progress

Cross-cutting

`Performance`:	collects performance data
`Error`:	detects and evaluates anamolies

Enzo-P/Cello Classes

Cello's classes allow Enzo-P and other Cello application developers to easily extend their application's functionality through inheritance.

Cello classes

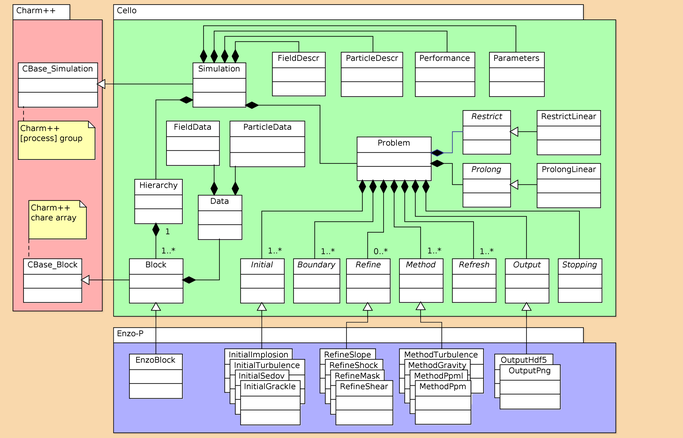
`Simulation`:	Simulation "chare group"
`Problem`:	Problem definition
`Block`:	Mesh block "chare array"

Cello interfaces

`Initial`:	Initial conditions
`Boundary`:	Boundary conditions
`Refine`:	Refinement criteria
`Method`:	Physics methods
`Output`:	Disk output

Enzo-P implementations

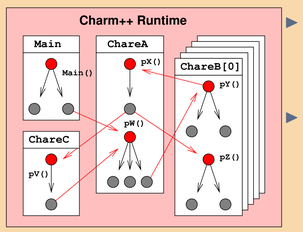
`InitialFoo`:	Enzo-P problem initializers
`BoundaryFoo`:	Boundary condition types
`RefineFoo`:	Specific refinement criteria
`MethodFoo`:	Computational kernels
`OutputFoo`:	Data or image output



Parallelization

Charm++

Cello is implemented using Charm++, a C++-based OOP parallel programming system developed independently at the University of Illinois, Urbana-Champaign. Its design tenets include efficient portability, latency tolerance, dynamic load balancing, and code reuse.

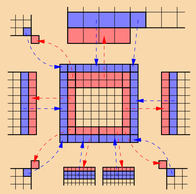


- Charm++ programs
  - ▷ Charm++ objects are *chares*
  - ▷ remote callable *entry methods*
  - ▷ communicate via *messages*
- Charm++ runtime system
  - ▷ maps chares to processors
  - ▷ schedules entry methods
  - ▷ load balances chares
  - ▷ supports checkpoint / restart

Cello AMR

Cello implements a fully-distributed forest-of-octree mesh hierarchy using a Charm++ *chare array* named *`Block`*. `Blocks` are indexed using a bit coding of the array (forest) and octree indices. Charm++ caches array indices for high performance and scalability.

- Asynchronous execution
  - ▷ `Field` data sent when available
  - ▷ computes when all ghosts received
- Dynamic scheduling
  - ▷ multiple `Blocks` per process
  - ▷ automatic overlap of comp./comm.
- Charm++ provides
  - ▷ dynamic load balancing
  - ▷ checkpoint / restart



Cello Applications

1. Parameter file: `heat.in`

```
% list field variables
Field { list = [ "temperature" ]; }
% define initial conditions
Initial {
  list = [ "value" ];
  value {
    temperature = [ 10.0,
        (0.3<x && x<0.7) &&
        (0.3<y && y<0.7),
                      1.0 ];
  }
}
% define method parameters
Method {
  list = [ "heat" ];
  heat {
    alpha = 1.0;
  }
}
```

Developing with Cello

Writing scalable AMR applications with Cello is straightforward. For example, to use Cello to solve the heat equation with the Forward Euler method, two main steps are required:

1. *Create an input parameter file*
2. *Add a new `Method` class*

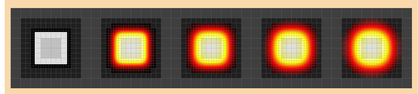
Other minor modifications are also needed for reading method parameters, calling the method's constructor, and updating a Charm++ control file.

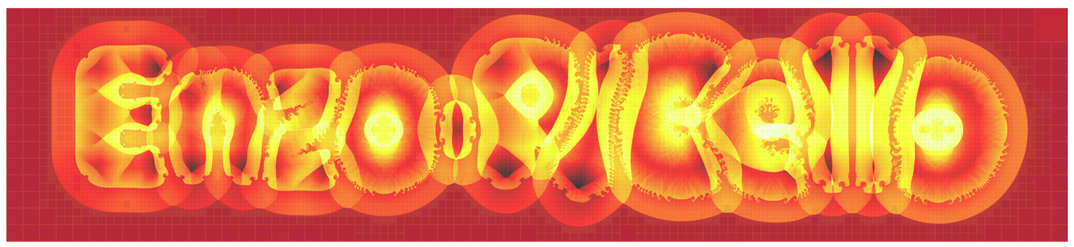
2a. Include file: `MethodHeat.hpp`

```
class MethodHeat : public Method {
  // Create a MethodHeat object
  MethodHeat ( double a ) :a_(a){};
  // Apply FE to the block
  virtual void compute(Block *);
  // Compute the CFL restriction
  virtual double timestep (Block *);
}
```

2b. Source code: `MethodHeat.cpp`

```
MethodHeat::compute(Block * block) {
  // Get field block attributes
  Field field = block()->data()->field();
  iU = field.field_id ("temperature");
  U = (double *) field.values (iU);
  field.dimensions (iU,&mx,&my);
  field.size          (&nx,&ny);
  field.ghosts      (iU,&gx,&gy);
  field.cell_width    (&hx,&hy);
  rx = 1.0/(hx*hx); ry = 1.0/(hy*hy);
  // Apply forward Euler method
  for (int iy=gy; iy<ny+gy; iy++) {
    for (int ix=gx; ix<nx+gx; ix++) {
      i = ix + mx*iy;
      Uxx=(U[i-dx]-2*U[i]+U[i+dx])*rx;
      Uyy=(U[i-dy]-2*U[i]+U[i+dy])*ry;
      Unew[i]=U[i] + a_*dt*(Uxx + Uyy);
} } }
```





Project Development

Summary

This project serves as a transition from single-developer to distributed community open development.

Development is pipelined, with staged Cello framework capabilities enabling new Enzo-P physics.

This material is based upon work supported by the National Science Foundation under Grant No. SI2-SSE-1440709.

Roadmap

Code release	Cello development	Enzo-P development
V1.0 (*hydro*)		
	Elliptic problems	Chemistry
V2.0 (*chemistry*)	↓ ↘	↓
	Particle support	Self-gravity / RHD
V3.0 (*gravity*)	↓ ↘	↓
	MHD support	P^3M / TreePM
V4.0 (*particles*)	↓ ↘	↓
	Ray support	MHD
V5.0 (*magnetism*)	↓ ↘	↓
	optimization	Adaptive ray tracing
V6.0 (*radiation*)		

Status

Enzo-P / Cello is available under the BSD New Open Source license.

- Previous capabilities
  - ▷ PPM dual-energy hydrodynamics
  - ▷ PPML compressible ideal MHD
- Recent capabilities
  - ▷ self-gravity (Krylov)
  - ▷ particles (static AMR)
- Upcoming capabilities
  - ▷ self-gravity (MG, HG, P^3M)
  - ▷ particles (dynamic AMR)
  - ▷ chemistry (Grackle library)