

# Asynchronous distributed-memory task-parallel algorithm for hydrodynamics on unstructured 3D grids\*

J. Bakosi<sup>a,\*</sup>, R. Bird<sup>a</sup>, F. Gonzalez<sup>c</sup>, C. Junghans<sup>a</sup>, W. Li<sup>b</sup>, H. Luo<sup>b</sup>, A. Pandare<sup>a</sup>, J. Waltz<sup>a</sup>

<sup>a</sup>*Los Alamos National Laboratory, Los Alamos, NM, United States*

<sup>b</sup>*North Carolina State University, Raleigh, NC, United States*

<sup>c</sup>*Strong Analytics, Chicago, IL, United States*

---

## Abstract

We discuss the implementation, verification, and performance of a finite element flux-corrected transport method for the Euler equations of high-speed compressible flows using unstructured 3D meshes. The algorithm is implemented for distributed-memory machines using task-parallel asynchronous communication primitives using the Charm++ runtime system. We demonstrate scalability up to  $25 \times 10^9$  cells at  $\mathcal{O}(10^4)$  compute cores and the benefits of automatic load balancing for irregular workloads. The source code with documentation is available at <https://quinoacomputing.org>.

*Keywords:* shock hydrodynamics; finite element method; flux-corrected transport; Charm++; automatic load balancing

---

## 1. Objective

This paper discusses the numerical implementation, verification, and performance of a finite element method for compressible high-speed flows for 3D unstructured grids using the Charm++ runtime system. The numerical method discussed is not new: it closely follows [1] and we make no claims on its numerical accuracy, other than verifying its correctness, its design order of convergence for smooth solutions, and its convergence for discontinuous solutions. The main contribution is the discussion of the software implementation on Charm++ using an asynchronous parallel computing paradigm targeting both shared-, and distributed-memory architectures. The main benefit of Charm++ is to allow arbitrary overlap of computation and communication, automatic checkpoint/restart, automatic fault tolerance, and automatic load balancing based on real-time CPU load measurement that is independent of the algorithm, leading to separation of concerns, code modularity, usability, and significantly increased performance. These features make such an implementation especially suitable for large multiphysics simulations with a priori unknown, heterogeneous, and/or dynamic parallel load distribution.

The structure of the paper is as follows. In Sec. 2 we present the governing equations solved. Sec. 3 describes the numerical method. Sec. 4 gives a brief overview of the Charm++ runtime system, discussing the features used by the flow solver, and gives some details on the high level software design, algorithmic choices, and various implementation details. Sec. 5 discusses a set of test problems used for solution and implementation verification. Sec. 6 discusses scalability, parallel performance, and load balancing. Finally, Sec. 7 contains a summary.

---

\*Research presented in this report was supported by the Laboratory Directed Research and Development program of Los Alamos National Laboratory under project number LDRD-20170127-ER.

\*Corresponding author, [jbakosi@lanl.gov](mailto:jbakosi@lanl.gov)

<sup>20</sup> **2. The equations of compressible flow**

The governing equations under consideration are the 3D unsteady Euler equations, governing inviscid compressible flow, written here in the flux-conservative form

$$\frac{\partial U}{\partial t} + \frac{\partial F_j}{\partial x_j} = S, \quad U = \begin{Bmatrix} \rho \\ \rho u_i \\ \rho E \end{Bmatrix}, \quad F_j = \begin{Bmatrix} \rho u_j \\ \rho u_i u_j + p \delta_{ij} \\ u_j (\rho E + p) \end{Bmatrix}, \quad S = \begin{Bmatrix} S_\rho \\ S_{u,i} \\ S_E \end{Bmatrix}, \quad (1)$$

where the summation convention has been applied for indices  $i$  and  $j$  and, in the usual notation,  $\rho$  is the density,  $u_i$  the velocity vector,  $E = u_i u_i / 2 + e$  is the specific total energy,  $e$  is the specific internal energy, and  $p$  is the pressure.  $S_\rho$ ,  $S_{u,i}$ , and  $S_E$  are source terms that arise from the application of the method of manufactured solutions, used for verification; these source terms are zero when computing practical engineering problems. The system is closed with the ideal gas law equation of state

$$p = \rho e(\gamma - 1), \quad (2)$$

where  $\gamma$  is the ratio of specific heats. The exact form of the equation of state is of secondary importance for this work, in principle any analytic equation of state could be used.

**3. The flow solver**

This section describes discretization in space (Sec. 3.1) and time (Sec. 3.2), and flux-corrected transport <sup>25</sup> (Sec. 3.3). The method is based on a continuous Galerkin finite element method for linear tetrahedra.

*3.1. Discretization in space*

To arrive at a continuous Galerkin finite element method we start from the weak form of the governing equations (1),

$$\int_{\Omega} N^v \left( \frac{\partial U}{\partial t} + \frac{\partial F_j}{\partial x_j} - S \right) d\Omega = 0, \quad v = 1, 2, \dots, n, \quad (3)$$

which requires that the error in the numerical solution, sampled at  $n$  discrete points,  $v$ , using the weighting functions  $N^v$ , vanish on the whole domain,  $\Omega$ , in an integral sense. Numerically approximating the solution as  $U \approx N^w \hat{U}_w$ , where  $\hat{U}_w$  denotes the unknowns at the discrete node  $w$ , leads to the Galerkin weak statement

$$\int_{\Omega} N^v \left[ N^w \frac{\partial \hat{U}_w}{\partial t} + \frac{\partial}{\partial x_j} F_j(N^w \hat{U}_w) - S(N^w \hat{U}_w) \right] d\Omega = 0. \quad (4)$$

Integrating the flux term by parts, neglecting the resulting boundary integral (assuming zero flux on the problem boundary), and applying  $F_j(N^w \hat{U}_w) \approx N^w F_j(\hat{U}_w)$  and  $S(N^w \hat{U}_w) \approx N^w S(\hat{U}_w)$  see [2], yields the final weak form for the whole domain,  $\Omega$ ,

$$\int_{\Omega} N^v N^w d\Omega \frac{\partial \hat{U}_w}{\partial t} - \int_{\Omega} N^w d\Omega \frac{\partial N^v}{\partial x_j} F_j(\hat{U}_w) - \int_{\Omega} N^v N^w d\Omega S(\hat{U}_w) = 0. \quad (5)$$

All integrals in Eq. (5) are evaluated by breaking up the domain,  $\Omega$ , into sub-domains as a sum over integrals over discrete elements,  $\Omega_e$ ,

$$\sum_{\Omega_e \in v} \sum_{w \in \Omega_e} \int_{\Omega_e} N^v N^w d\Omega_e \frac{\partial \hat{U}_w}{\partial t} = \sum_{\Omega_e \in v} \sum_{w \in \Omega_e} \int_{\Omega_e} N^w d\Omega_e \frac{\partial N^v}{\partial x_j} F_j(\hat{U}_w) + \sum_{\Omega_e \in v} \sum_{w \in \Omega_e} \int_{\Omega_e} N^v N^w d\Omega_e S(\hat{U}_w), \quad (6)$$

where the inner summation is over points  $w$  forming  $\Omega_e$  (gather) and the outer summation is over tetrahedra  $\Omega_e$  connected to point  $v$  (scatter). Eq. (6) results in the following semi-discrete system of equations

$$\mathbf{M}_c \hat{\mathbf{U}}_{,t} = \mathbf{r}(\hat{\mathbf{U}}). \quad (7)$$

The size of the consistent mass matrix  $\mathbf{M}_c$  is  $n \times n$ , where  $n$  is the number of nodes of the computational mesh. According to the sum in Eq. (6), only those elements contribute to a given row  $v$  of  $\mathbf{M}_c$  which contain node  $v$  (scatter), thus  $\mathbf{M}_c$  is sparse.

30    3.2. Discretization in time

Eq. (7) is discretized in time using a Lax-Wendroff (Taylor-Galerkin) scheme [3, 4, 5] implemented using two stages:

$$\begin{aligned} U^{n+1/2} &= U^n + \frac{\Delta t}{2} U_{,t}^n = U^n - \frac{\Delta t}{2} \frac{\partial}{\partial x_j} F_j(U^n) + \frac{\Delta t}{2} S(U^n), \\ \Delta U &= U^{n+1} - U^n = \Delta t U_{,t}^{n+1/2} = -\Delta t \frac{\partial}{\partial x_j} F_j(U^{n+1/2}) + \Delta t S(U^{n+1/2}). \end{aligned} \quad (8)$$

The above scheme combined with linear shape functions is identical to a two-stage Runge-Kutta Galerkin finite element method, and thus central differencing without damping. Stabilization is obtained by using constant shape functions for the half step solution where the gather results in element quantities, followed by a scatter step using linear shape functions resulting in nodal quantities. Assuming linear tetrahedra, the 35 combined spatial and temporal discretization that achieves this yields the following staggered scheme [2],

$$\begin{aligned} U_e^{n+1/2} &= \frac{1}{4} \sum_{w=1}^4 \hat{U}_w^n - \frac{\Delta t}{2} \sum_{w=1}^4 \frac{\partial N^w}{\partial x_j} F_j(\hat{U}_w^n) + \frac{\Delta t}{2} \frac{1}{4} \sum_{w=1}^4 S(\hat{U}_w^n), \\ \sum_{\Omega_e \in v} \sum_{w \in \Omega_e} \int_{\Omega_e} N^v N^w d\Omega_e \Delta \hat{U}_w &= \Delta t \int_{\Omega_e} \frac{\partial N^v}{\partial x_j} F_j(U_e^{n+1/2}) d\Omega_e + \Delta t \int_{\Omega_e} N^v S(U_e^{n+1/2}) d\Omega_e \end{aligned} \quad (9)$$

where  $U_e^{n+1/2}$  is the vector of element-centered solutions at the half step. Note that the first step discretizes the flux integral *before* integration by parts, Eq. (4), while the second one *after* integration by parts, Eq. (5), hence the difference in sign.

3.3. Flux-corrected transport

40    Flux-corrected transport (FCT) is a solution to circumvent the consequence of Godunov's theorem [6], which states that no linear scheme of order greater than 1 will yield monotonic solutions. Accordingly, FCT is a nonlinear scheme that combines a high-, and a low-order scheme using limiting.

In the FCT scheme we use the high-order solution at the new time step can be written as

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \Delta \mathbf{U}^h = \mathbf{U}^n + \Delta \mathbf{U}^l + (\Delta \mathbf{U}^h - \Delta \mathbf{U}^l) = \mathbf{U}^l + (\Delta \mathbf{U}^h - \Delta \mathbf{U}^l), \quad (10)$$

where  $\Delta \mathbf{U}^h$  and  $\Delta \mathbf{U}^l$  denote the solution increments of the high-, and low-order schemes, respectively. In Eq. (10), it is the last term that is limited in a way to avoid spurious oscillations as

$$\mathbf{U}^{n+1} = \mathbf{U}^l + \text{lim}(\Delta \mathbf{U}^h - \Delta \mathbf{U}^l), \quad (11)$$

The high-order scheme scheme, given in Eq. (9), is symbolically written as

$$\mathbf{M}_c \Delta \mathbf{U}^h = \mathbf{r}. \quad (12)$$

From Eq. (12) we construct a low order scheme by lumping the mass matrix and adding mass diffusion

$$\mathbf{M}_l \Delta \mathbf{U}^l = \mathbf{r} + \mathbf{d} = \mathbf{r} - c_\tau (\mathbf{M}_l - \mathbf{M}_c) \mathbf{U} \quad (13)$$

where  $\mathbf{M}_l$  is the lumped mass matrix and  $c_\tau$  is a diffusion coefficient. The low-order solution is guaranteed to be monotone for a diffusion coefficient of  $c_\tau = 1$ , see [2]. If we rewrite Eq. (12) as

$$\mathbf{M}_l \Delta \mathbf{U}^h = \mathbf{r} + (\mathbf{M}_l - \mathbf{M}_c) \Delta \mathbf{U}^h \quad (14)$$

the difference between the right hand sides of the high and low order schemes can be recognized as

$$\text{AEC} = \Delta \mathbf{U}^h - \Delta \mathbf{U}^l = \mathbf{M}_l^{-1} (\mathbf{M}_l - \mathbf{M}_c) (c_\tau \mathbf{U}^n + \Delta \mathbf{U}^h), \quad (15)$$

also called as the anti-diffusive element contributions (AEC). The AEC is then limited,  $C_e \cdot \text{AEC}$ , and applied to advance the solution to the next time step using Eq. (11), where  $0 \leq C_e \leq 1$  is the limit coefficient for a 45 given element  $e$ .

### 3.4. The limiting procedure

The limiting procedure to compute  $C_e$  closely follows [1]. The description here is given in terms of how the algorithm is broken up into computational tasks to prepare for the discussion on task-parallelism later.

50 **Task LHS**

According to Eq. (6) the left hand side is the consistent mass matrix. For simplicity and speed we lump the consistent mass matrix, which is obtained from  $\mathbf{M}_c$  by summing the rows to the diagonals. Though this increases the overall numerical error for a given mesh, it does not change the order of accuracy of the method. If the mesh does not move and its topology does not change, the left hand side does not need to be recomputed for a new time step.

55 **Task RHS**

The high-order right hand side is computed by the two-step procedure given by Eq. (9). Since the two steps are staggered, the gather takes information from nodes to cell centers, followed by a scatter, moving 60 information from cells to nodes, both steps are contained within a single right hand side calculation within a time step. Within the two steps there is no need for parallel communication as an element always resides on a given mesh partition and only mesh nodes are shared between processing elements (PEs).

**Task DIF**

A step of the limiting procedure is to compute the mass diffusion term in Eq. (13). Using linear tetrahedra, this is given for each element by

$$-\left[c_\tau(\mathbf{M}_l - \mathbf{M}_c)\right]_e = -\frac{c_\tau J_e}{120} \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ -1 & -1 & -1 & 3 \end{bmatrix}, \quad (16)$$

where  $J_e = \overrightarrow{BA} \cdot (\overrightarrow{CA} \times \overrightarrow{DA})$  is the element Jacobian, computed from the triple product of the edge vectors 65 of the tetrahedron given by vertices  $A, B, C$ , and  $D$ .

**Task AEC**

Another step is to compute the anti-diffusive element contributions, given in Eq. (15), for each element. For this  $\mathbf{M}_l - \mathbf{M}_c$  is given in Eq. (16), and the inverse of the lumped mass matrix,  $\mathbf{M}_l^{-1}$  is obtained from the nodal cell volumes, computed by summing the quarter of each tetrahedron element volume to nodes. Once the AECs are computed for each element, the next immediate step is to sum all positive (negative) anti-diffusive element contributions to node  $i$

$$P_i^\pm = \sum_e \left\{ \begin{array}{l} \max \\ \min \end{array} \right\} (0, \text{AEC}_e) \quad (17)$$

**Task ALW**

The limiting procedure requires the maximum and minimum nodal values of the low-order solution,  $\mathbf{U}^l$ , and the previous solution,  $\mathbf{U}^n$ ,

$$\mathbf{U}_i^* = \left\{ \begin{array}{l} \max \\ \min \end{array} \right\} (\mathbf{U}_i^l, \mathbf{U}^n). \quad (18)$$

Another alternative is to only consider the low order solution,  $\mathbf{U}_l$ , when computing the allowed solution bounds, which leads to the so-called 'clipping limiter' in place of Eq. (18):  $\mathbf{U}_i^* = \mathbf{U}_i^l$ . This is followed by computing the maximum and minimum nodal values of all elements,

$$\mathbf{U}_e^* = \left\{ \begin{array}{l} \max \\ \min \end{array} \right\} (\mathbf{U}_A^*, \mathbf{U}_B^*, \mathbf{U}_C^*, \mathbf{U}_D^*), \quad (19)$$

then computing the maximum and minimum unknowns of the elements surrounding each node,

$$\mathbf{U}_i^{\max} = \left\{ \begin{array}{l} \max \\ \min \end{array} \right\} (\mathbf{U}_1^*, \mathbf{U}_2^*, \dots, \mathbf{U}_m^*). \quad (20)$$

The limit coefficients will be computed (see below) based on  $P_i^\pm$  and the maximum and minimum increments and decrements the nodal solution values are allowed to achieve,

$$Q_i^\pm = \mathbf{U}_i^{\max} - \mathbf{U}^l. \quad (21)$$

#### Task LIM

Defining the ratios of positive and negative element contributions for each node  $i$  that ensure monotonicity as

$$R_i^\pm = \left\{ \begin{array}{ll} \min(1, Q_i^\pm / P_i^\pm) & P_i^+ > 0 > P_i^- \\ 1 & \text{otherwise} \end{array} \right., \quad (22)$$

the limit coefficient for each element is taken as the most conservative ratio

$$C_e = \min_{i \in \Omega_e} \left\{ \begin{array}{ll} R_i^+ & \text{AEC} > 0 \\ R_i^- & \text{AEC} < 0 \end{array} \right.. \quad (23)$$

The limited AEC is then scatter-added to nodes

$$A_i = \sum_{i \in \Omega_e} C_e \cdot \text{AEC}. \quad (24)$$

#### Task APPLY

The limited AEC is applied to the low-order solution according to Eq. (11),

$$U_i^{n+1} = U_i^l + A_i. \quad (25)$$

The above procedure is general, works on the numerical solution (instead on fluxes or slopes), and written as the same procedure for each scalar for a system of equations. This works well for independent scalars, but for coupled system of equations additional techniques have been developed to reflect the coupled nature of the equations in the limiting procedure. See also [1, 2, 7]. In particular, we have experimented with Löhner's 'indicator variable' approach, which designates a physical variable, e.g., density, whose limit coefficient other variables inherit in a cell, as well as applying a minimum of the limit coefficient for all or some of the conserved variables. These techniques tend to produce better results for some problems while worse for others – in any case, such a priori knowledge of the problem computed is beneficial towards obtaining improved numerical solutions.

## 4. Software design and implementation

This section discusses various software implementation details of the flow solver described in Sec. 3. The implementation uses Charm++ as the parallel programming paradigm. The entire problem, in this case the computational mesh, is broken up into multiple *chares*. Chares can be thought of as data and associated functions that operate on them, as in any object-oriented programming paradigm. In code, a chare is represented by a C++ class. Since each chare holds a different partition of the mesh and associated data, this is simply the realization of the divide and conquer paradigm. Chares communicate with each other via message passing. Up to this point there is nothing new or special compared to any other code relying on a distributed-memory paradigm, e.g., the message passing interface (MPI).

In the following sections we give a brief overview of the major features of the Charm++ runtime system, followed by discussion on those specific features, used by the flow solver, that significantly differentiate this from other distributed-memory implementations. Finally, the main steps of the algorithm and its implementation are outlined.

#### 4.1. An overview of Charm++

Charm++ is a general-purpose production-grade many-tasking programming framework, runtime system, and ecosystem of libraries for modern high-performance computing systems. It offers high productivity and performance portability through features such as multicore and accelerator support, dynamic load balancing, fault tolerance, latency hiding, interoperability with MPI and OpenMP, and online job-resizing. Charm++ has been developed and maintained by the Parallel Programming Laboratory at the University of Illinois at Urbana-Champaign for 20+ years. It is similar to MPI in the sense that it allows writing high-performance applications for the largest distributed-memory computers. The important differences and main advantages of Charm++ over MPI are

- Fully asynchronous execution model, allowing arbitrary overlap of computation and communication
- Task-parallelism, scheduling parts of an algorithm independently of or dependent on each other
- Automatic load balancing, redistributing computational load by migration of work units based on real-time CPU load measurement
- Automatic checkpointing and fault tolerance, allowing restart with differing number of work units and resizing jobs on the fly instead of interruption
- Support for power-, and energy-aware computation, by intelligent work redistribution based on, e.g., CPU frequency scaling or temperature
- Interoperability with MPI, OpenMP, CUDA, from C++, Fortran, and Python

For more information on Charm++ see <http://charmplusplus.org> and for publications see <http://charm.cs.illinois.edu/papers>.

#### 4.2. Communication with message passing in Charm++

In a Charm++ program, data and work-units (chares) interact via asynchronous function calls. Communication via message passing is realized via special class member functions, called *entry methods*. Entry methods differ from ordinary member functions in that they can be called remotely, by using the chare array handle addressed by the chare id and its member function to call. Function arguments to such entry methods can be of any standard or custom type. If the target of the chare happens to reside on a remote compute node, the runtime system serializes the function arguments into a byte stream, sends it across the network, unpacks it on the other side, and puts the function call into the local queue of the target processor. Different functionality (with associated data) is divided into a chare array of such objects, thus an algorithm is realized by a dynamically interacting arrays of chares. As an example, Listing 1 compares asynchronous message passing in MPI and Charm++, displaying code used to send and receive arrays whose size is only known at runtime and can be different on different chares/ranks, characteristic of unstructured-mesh solvers. In Charm++ point-to-point communication is done via such asynchronous entry-method function calls.

The implementation (declaration and definition) of entry-method member functions, such as `fn(...)` in Listing 1, do not differ in any way from usual C++ non-entry methods. Such functions are, however, labeled as entry methods in Charm++ *interface* files, commonly with extension `.ci`, as shown in Listing 1. While the examples listed here send and receive a simple standard C++ vector (a contiguous-storage, dynamically-resizable, array), Charm++ also allows asynchronously sending and receiving arbitrarily complex custom-built data structures this way as well without measurable performance penalties, compared to floating-point calculations, as will be shown. Since the underlying programming paradigm of Charm++ is higher level than MPI+X (where X stands for some form of threading, e.g., OpenMP or CUDA), in Charm++ distributed-memory sends and receives may operate on (safe) custom data structures (instead of low level byte-streams), e.g., containers built on C++ standard library vectors, trees, or hash maps, or arbitrary user-defined data types. In Sec. 6 we demonstrate that communication using such high level data structures yields an insignificant performance hit compared to computation.

#### 4.3. Problem decomposition in Charm++

135 It is important to appreciate that in Charm++ programs one usually does *not* address hardware elements, e.g., CPUs or threads. Instead, chares arrays elements have an ID, an integer by default. This makes sense, since the runtime system may migrate chares (and their data) from one compute node to another during computation to perform load balancing. Such migration is based on real-time load measurement and is transparent to the application. The runtime system dynamically and automatically adapts the  
 140 computational load monitoring load-imbalance and can also migrate data and work-units if it notices that a compute node failed or is about to fail, enabling fault tolerance.

Similar to other distributed-memory solvers for unstructured meshes, after reading the mesh, it is partitioned, usually by a coordinate-, or a graph-based partitioner. However, the number of mesh partitions does not have to equal the number of PEs. Experience shows that such *overdecomposition* or *virtualization*,  
 145 i.e., significantly more work units (mesh partitions) than the available number of PEs, can be beneficial, especially with load balancing. While overdecomposition increases communication costs, it also comes with benefits: (1) useful computation happens on smaller mesh partitions which may improve cache utilization, and (2) finer-grain work units may be load-balanced more efficiently, as will be shown in Sec. 6.

Listing 1: Non-blocking send and receive of a variable-length array with MPI and Charm++.

```
// Non-blocking send and receive with MPI, source: https://stackoverflow.com/a/14051503
if (rank == 0) {
    MPI_Request req;
    // Send a message to rank 1
    MPI_Isend(arr1, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &req);
    // Do not forget to complete the request!
    MPI_Wait(&req, MPI_STATUS_IGNORE);
}
else if (rank == 1) {
    MPI_Status status;
    // Wait for a message from rank 0 with tag 0
    MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
    // Find out the number of elements in the message -> size goes to "n"
    MPI_Get_count(&status, MPI_DOUBLE, &n);
    // Allocate memory
    arr1 = malloc(n*sizeof(double));
    // Receive the message. ignore the status
    MPI_Recv(arr1, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

// -----
// Non-blocking send and receive with Charm++
using namespace std;
// on rank 0: create & fill array a
vector<double> a{...};
// send array a
object[1].fn(a);
// on rank 1: receive array
void Object::fn( const vector<double>& a ) { // use array a }

// Member function labeled as a Charm++ entry method for receiving a
// variable-size array in a Charm++ interface file, Object.ci
array [1D] Object {
    entry void fn( const vector<double>& a );
};
```

#### 4.4. MPI interoperability and processor-aware collections

The Charm++ paradigm is the single abstraction our implementation uses. We also rely on a number of third-party libraries some of which require MPI. This is not a problem with Charm++ as Charm++-code can seamlessly interoperate with MPI-only libraries. There are various ways native Charm++ and MPI code can interact within a single application. We invoke library calls that contain direct MPI calls from Charm++ chare *groups*. Chare groups are similar to chare arrays with the important differences that there is a guaranteed single group element per physical processor and group elements never migrate. This enables invoking MPI calls by libraries from groups that otherwise know nothing about their environment.

In particular, the flow solver uses Zoltan2 [8] for initial mesh partitioning (which supports multiple types of partitioners in parallel as well as overdecomposition), and ExodusII [9] (used for mesh and field solution output in parallel). Using existing libraries thus allows building on the specific expertise of their authors.

Besides groups, another type of processor-aware chare collection in Charm++ is a *nodegroup*. Nodegroups are similar to groups (e.g., they do not migrate) but a node group is a collection of chares with one chare per *process* or *logical (e.g., compute) node*. Thus node groups allow writing code that must be executed on a compute-node basis. An example is  $N$ -to- $M$  I/O, which allows, for example, reading in a large mesh on  $N \ll M$  (instead of  $M$ ) compute nodes. This can be used to optimize I/O performance if  $N$  is the number of I/O nodes and  $M$  is the number of compute nodes on a large cluster. This is how our flow algorithm reads in its input computational mesh, which is followed by mesh partitioning and cell/node redistribution according to the output of the mesh partitioner.

#### 4.5. Task-parallelism

Task-parallelism is a parallel programming technique used to express parts of an algorithm as a graph of connected or independent tasks. Besides the ubiquitous divide-and-conquer (data-parallelism), task-parallelism has been recently gaining popularity, complementing data-parallelism, to increase the degree of overlap among various tasks of an algorithm.

In Charm++ task-parallelism is expressed via its *structured DAG* (SDAG) functionality, where *DAG* stands for direct acyclic graph. DAGs are used to identify dependencies among individual tasks of an algorithm, represented by graph nodes. The DAG in Charm++ is expressed by a simple language extension, specified in the Charm++ interface (.ci) file, that allows expressing the logic of interdependence among tasks. There may be a different DAG within each Charm++ chare array (class) definition. Each array element represents a different C++ class instance, holding its own data, e.g., a partition of the computational mesh and the solution. Execution of the algorithm within a single chare object instance is generally independent of execution within another chare array element.

An example of a simple DAG may consist of tasks A, B, and C. Then if task A and B must both finish before C can be started, a DAG that expresses this logic is given in Figure 1. Besides separating `*_complete()` member functions by a comma, SDAG constructs also support `for`, `while`, and `case` statements. Using such simple Charm++ SDAG code fairly complex application logic can be expressed.

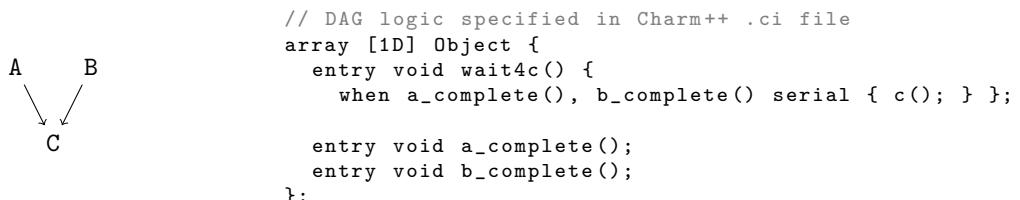


Figure 1: *Left:* Structured DAG expressing a simple logic of task A and B must finish before task C can start. *Right:* Charm++ SDAG code expressing the logic on the left. Here `a_complete()` and `b_complete()` are member functions of `Object` that are implemented by the runtime system, used to trigger the completion of task A and B, respectively. The runtime system monitors the completion of tasks A and B, and only when both are complete will it call member function `c()`.

185 4.6. Flow algorithm software design using Charm++: setup

We are now in a position to describe the main steps of the flow algorithm in Sec. 3. The setup consists of the following main steps:

1. Read user input (from command-line arguments and an input text file)
  2. Read the mesh in parallel (one linear chunk per compute node from a single file)
  3. Pass the linearly-distributed mesh to a mesh partitioner allowing overdecomposition
  4. Redistribute the mesh (cell connectivity and coordinates) based on the distribution from the partitioner
  5. Compute nodal communication maps in parallel
  6. Create migratable chare arrays holding a mesh partition (connectivity, coordinates, nodal communication map, and solution arrays)
- 190 195 Except for Step 1 above, all steps are parallel and no assumption is made on the size of the input mesh, e.g., that it must fit into memory of a single compute node. All steps above are common in distributed-memory implementations of unstructured-mesh solvers, except our Charm++ implementation is fully asynchronous and puts data in migratable chare arrays. In the following we describe Step 5 in some detail, since a scalable distributed-memory implementation of computing the nodal communication maps appears less than 200 straightforward with the constraint that the mesh cannot be loaded to a single compute node's memory.

#### 4.7. Computing the nodal communication maps in parallel

Until Step 5 during setup, only the mesh connectivity graph and the node coordinates are known on each chare. From this information our task in Step 5 is to compute the nodal communication map in a scalable fashion in parallel. No information is available on which chare might store the list of nodes that other chares share and we cannot assume that the entire mesh can fit into the memory of a single compute node. Any approach that singles out a master communicating with slaves will not scale. Additionally, we would like minimize the amount of data that must be sent across the network during setting up the node communication map. A nodal communication map on each partition (chare) is required for any distributed-memory unstructured-grid solver that stores the solution unknowns at mesh nodes. On each chare such a map associates a list of node IDs to a chare ID a given chare shares nodes with. The map must store a (different) list of nodes for each of potentially multiple other chares it shares at least a single node with. Using the standard library in C++ such a map can be implemented using a map of vectors:

```
// storage for nodal communication map for each chare (mesh partition)
std::unordered_map< int, std::vector<int> > nodeCommMap;
```

- 215 At this point, the node communication map, `nodeCommMap` is a dynamic but empty *unordered* (hash) map. Unordered maps are standard associative containers in C++ that contain key-value pairs with unique keys. In the map the keys are the chare IDs the given chare shares a list of nodes with and associate to each key (the value) is a vector of integers (a standard, continuous-storage, resizable array that knows its size). Search, insertion, and removal of elements of unordered maps have guaranteed average constant-time algorithmic complexity. Internally to such a hash map, the elements are not sorted in any particular order, but organized 220 into buckets. Which bucket an element is placed into depends entirely on the hash of its key. This allows fast access to individual elements, since once the hash is computed, it refers to the exact bucket the element is placed into. This is a good data structure for dynamically and frequently changing data.

##### 4.7.1. Step 1: Query

225 One way to accomplish computing the nodal communication map in a distributed-memory setting that is scalable and performant is to build the map in two steps: (1) *query* and (2) *response*. Before the query each chare collects a list of boundary nodes from its mesh partition. A node is on a chare boundary if it belongs to a face of a tetrahedron that has no neighbor element at one of its faces. The list of boundary nodes can be computed using an algorithm that relies on standard data structures, e.g., elements surrounding points, derived from unstructured meshes [2, 10]. Once the boundary nodes are computed, they are categorized into bins that will be sent to different chares to build the node communication map across all chares. The 230

binning is determined by the node IDs and assigned to chares in a linear fashion. The nodes in each bin are then sent to the chare a given bin is assigned to. Note that the nodes in the bins are independent of their final destination: the target chare will merely be responsible for assembling the communication map  
235 for the list of nodes in the bins it is responsible for. Sending and receiving the node lists in bins avoid expensive collectives that would involve all chares and only entail point-to-point communications with data involving only a subset of the mesh boundary nodes. Since the boundary nodes are uniquely assigned to bins, no boundary node is sent to multiple chares, which keeps communicated data to a minimum. The chares receiving the boundary nodes in bins store them in two maps that are the inverse of each other:

```
240 // Node->chare map
241 std::unordered_map< int, std::vector<int> > nodech;
242 // Chare->node map, the inverse of nodech
243 std::unordered_map< int, std::vector<int> > chnode;
```

where `nodech` associates chares (value) to nodes (key), while `chnode` associates nodes (value) to chares (key). These data structures are small, temporary, and will be thrown away after the *response* step.  
245

#### 4.7.2. Step 2: Response

Once all chares have received their assigned bins, the *response* step starts by computing the communication map for each boundary node. Since `nodech` and `chnode` only contain data on queried nodes and this data originates from source chares that have these nodes, these inverse maps already contain the communication maps that we need. Thus we only have to categorize the maps by source chare. The *response* step is given in Listing 2. The algorithm categorizes the nodal communication maps by the querying source chare and prepares the hash map `exp` for sending the maps back. The syntax uses C++ structured bindings and range-based for loops for readability; note that this is actual code lifted from the source. The last step in Listing 2 is to send the categorized maps back to the source chares. The object `thisProxy` is a Charm++ handle to the Charm++ chare array collection this code is implemented in. Using array-like indexing into  
255

Listing 2: *Response* step following the *query* step in building the nodal communication maps. Besides showing the algorithm used to categorize the nodal communication maps by target and neighbor chares, this listing is also an example of how sending a complex data structure using Charm++ looks like in verbatim, i.e., *not pseudo*, code: The last two lines perform a `for` loop to send, in a point-to-point fashion, an integer (the sender chare ID, `thisIndex`) and the nodal communication map needed by `targetchare`, `bndnodes`, a `std::map` whose key is an integer and value is a hash set. Here `std::map` is a standard *sorted* associative container that contains key-value pairs with unique keys. Keys are sorted by using a comparison function. Search, removal, and insertion operations in such maps have logarithmic complexity as maps are usually implemented as red-black trees. The `std::unordered_set` (or hash set) is also a standard C++ associative container that contains a set of unique objects of type of its key. Search, insertion, and removal in such sets have average constant-time complexity. Internally, in unordered sets the elements are *not sorted* in any particular order, but organized into buckets. Which bucket an element is placed into depends entirely on the hash of its value. This allows fast access to individual elements, since once a hash is computed, it refers to the exact bucket the element is placed into.

```
// Storage for sending back categorized communication maps. Outer key: target
// chare, inner key: chare neighboring the target chare, value: unique set of
// nodes shared with neighbor chare.
std::unordered_map< int, std::map< int, std::unordered_set<int> > > exp;

// Compute node communication map to be sent back to chares
for (const auto& [neighborchare, bndnodes] : chnode) {
    auto& e = exp[neighborchare];           // insert neighbor chare ID as outer key
    for (auto n : bndnodes)                 // for all boundary nodes of neighbor chare
        // find and loop over all boundary nodes in node->chare map
        for (auto d : find_value_of(m_nodech, n))
            if (d != neighborchare)
                e[d].insert(n);
}

// Send node communication maps to chares that issued a query to this chare
for (const auto& [targetchare, bndnodes] : exp)
    thisProxy[ targetchare ].bnd( thisIndex, bndnodes );
```

the collection itself, `thisProxy[targetchare].bnd(...)` makes a call to the target chare array element and invokes its member function `bnd()`, an entry method, packing its arguments and sending them across the network if the `targetchare` happens to be on another compute node. Here `thisIndex` stands for the chare ID of the sender, so that the receiver knows where the contribution came from. The communication step in the *response* step is similar to that of in *query*, point-to-point and sending minimal data for boundary nodes. Node communication maps are computed only for those chares that queried the map from the given responding chare. The boundary nodes can also be thought of as a distributed table and each chare only works on a chunk of it. Note that a chare only sends data back to those chares that have queried the chare.

#### 4.8. Flow algorithm software design using Charm++: time stepping

The last step of the setup phase, enumerated in Sec. 4.6, is to create the Charm++ chare arrays, designed to perform the time stepping. These chare arrays are migratable and *bound*. Bound arrays in Charm++ always migrate together while allowing separating different functionality and associated data. The important chare classes that interoperate during time stepping are

- `Transporter` (single chare, driver),
- `Discretization` (chare array, generic unstructured-grid solver base class),
- `DiagCG` (chare array, child class to `Discretization`, specialized to the node-centered continuous Galerkin finite element discretization scheme with a lumped-mass left-hand side and flux-corrected transport combined with Lax-Wendroff-like explicit time stepping scheme, discussed in this paper).
- `DistFCT` (chare array, performs distributed-memory flux-corrected transport, used by `DiagCG`)

The base `Discretization` and child `DiagCG` classes enable the well-known object-oriented design feature, runtime polymorphism. This enables code reuse and helps code generic to all types of discretizations stay uniform and code specific to a given discretization stay modular. Such a design is particularly useful when more than a single type of hydrodynamics schemes are implemented in the single code, as is our case, since we have multiple co-existing continuous (node-centered) as well as discontinuous (cell-centered) Galerkin finite element methods (discussed elsewhere). The base class Charm++ chare array, `Discretization`, encapsulates data and member functions that are generic to all unstructured-grid-based discretization schemes. Its array elements store, in a distributed fashion, the mesh connectivity and node coordinates. As `Discretization` is a chare array, its elements are distributed across the whole problem on all available compute nodes and PEs, and can also migrate for load balancing. The child class `DiagCG` is also a chare array, whose elements store the unknown solution at mesh nodes and implements the particular discretization scheme, discussed in Sec. 3. The elements of `DiagCG` are bound to those of `Discretization`. This enables accessing data associated to the same problem partition, e.g., the mesh, by the child from the base using a raw pointer, since bound elements always migrate together during load balancing. Additionally, `DistFCT` is also bound to `Discretization` and `DiagCG`.

With the above and Sec. 3 in mind, time stepping is preceded by (1) setting initial conditions and (2) computing the left hand side (task `LHS`). As discussed in Sec. 3.4, if the mesh does not change the left hand side does not have to be recomputed during time stepping. Then a single time step consists of the following steps:

1. Compute the size of the next time step,  $\Delta t$
2. Communicate  $\Delta t$  to all chares, performing a parallel all-to-all `min()` operation
3. Compute local contributions to RHS and  $\mathbf{d}$ , spawning tasks `RHS` and `DIF`, see Eqs. (9) and (16)
4. Communicate the RHS and  $\mathbf{d}$ , summing to boundary nodes
5. When all contributions of RHS and  $\mathbf{d}$  are ready on a chare, solve both the high-, and low-order system
6. Compute the nodal  $\mathbf{P}^\pm$  and  $\mathbf{Q}^\pm$ , spawning tasks `AEC` and `ALW`
7. Communicate contributions of  $\mathbf{P}^\pm$  and  $\mathbf{Q}^\pm$  across boundary nodes

8. When all contributions to  $\mathbf{P}^\pm$  and  $\mathbf{Q}^\pm$  are ready on a chare, spawning task LIM, compute the limit coefficients,  $C_e$ , scatter-adding the limited AEC to nodes  
305 9. Communicate the limited AEC in nodes on the boundary  
10. When all contributions of the limited AEC are ready on a chare, apply the limited AEC on the low  
order solution, Eq. (25)  
310 11. Evaluate load imbalance and migrate if necessary  
12. Continue with Step 1, starting a new time step

Steps 6–9 and computing  $\mathbf{d}$  in Step 3 are performed by the bound `DistFCT` array elements while the rest by `DiagCG`. It is important to appreciate that during all steps, every chare array element holds and works on its own mesh partition and also has its own task state, which is different from the state of another chare.  
315 Chares communicate, in a point-to-point fashion, with their handful of neighbors, those that they share chare-  
boundary nodes with, see Sec. 4.7. As a result, no synchronization is necessary beyond what is prescribed  
by the task graphs (discussed in detail below), except computing the new time step size, which is the single  
320 collective within a time step. This provides considerable freedom for the runtime system to schedule tasks  
and overlap computation and communication, constantly monitoring real-time CPU-load and communication  
patterns. The system can also dynamically adapt to external factors influencing computation, e.g., CPU  
frequency scaling due to temperature deviations and temporarily (or notoriously) underperforming compute  
nodes, usually out of control of both developers and users.

*Steps 3–5:* Figure 2 shows the DAG and Charm++ code that expresses the logic behind Steps 3–5, the first computation/communication steps in a time step. The DAG is simple and prescribes that only when both own and contributions to the boundary nodes are complete on a given chare should the runtime system  
325 call the member function `solve()`, which computes both high-, and low-order (unlimited) solutions,  $\mathbf{U}^l$  and  $\Delta\mathbf{U}^h$ . As the DAG does not assume a particular ordering between the tasks `ownrhs` and `comrhs`, either can  
execute first or second and this order can also differ on different chares, working on different mesh partitions.  
As a result, the different tasks cannot assume being the first writing to a particular data array they operate  
330 on. The own and communicated portions of the RHS are buffered into arrays that are different from ones  
the owned portions are stored. The own and communicated contributions are then combined in `solve()` at  
different times when a chare is ready to call this function, independent of other chares.

*Steps 6–10:* Figure 3 shows the DAG and Charm++ code that expresses the logic behind Steps 6–10, the limiting step. The individual tasks of the limiting procedure are discussed in Sec. 3.4. These are LHS, RHS,  
335 DIF, AEC, ALW, LIM, and APPLY. Tasks RHS and DIF have been already performed simultaneously in Steps 3–5  
using the DAG in Figure 2. Thus the limiting procedure involves the remaining tasks. Three intermediate  
nodal vectors are required for computing the limited high-order solution: the sum of all positive and negative  
AEC to nodes,  $\mathbf{P}^\pm$ , Eq. (17), the maximum and minimum increments and decrements the nodal solutions

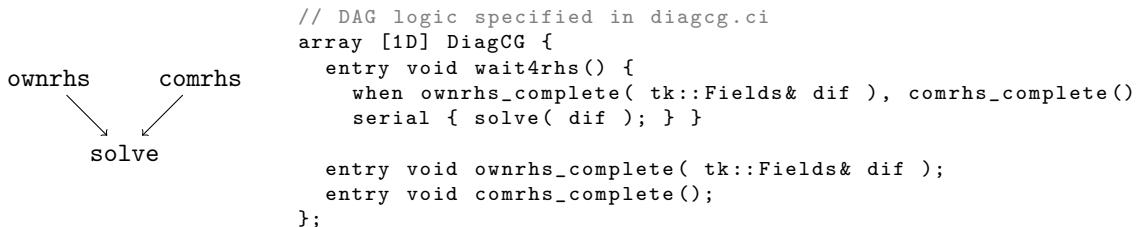


Figure 2: *Left:* Structured DAG expressing the logic of computing and communicating the RHS and the mass diffusion term. Both `ownrhs` and `comrhs` must finish before `solve` can start. *Right:* Charm++ SDAG code expressing the logic on the left. Only when both own and contributions to the boundary nodes are complete on a given chare will the runtime system call the member function `solve()`. Note the function argument passed to `ownrhs_complete()`. This is allowed by Charm++ even though the runtime system implements the body of this function. This is used here to pass through the diffusion term `dif` directly to the `DiagCG::solve()`, which avoids having to store this data in `DiagCG`'s state, reducing its size, which is advantageous for migration as well as checkpoint/restart.

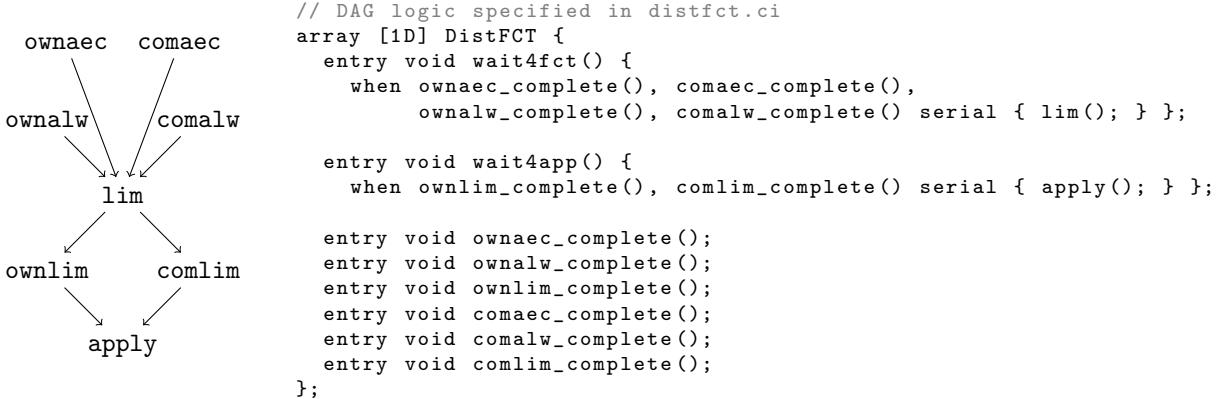


Figure 3: *Left:* Structured DAG expressing the logic of computing and communicating the nodal vectors,  $P^\pm$ ,  $Q^\pm$ ,  $A$ , required in the flux-corrected transport algorithm. As the top portion of the DAG instructs, *all* of `ownaec`, `comaec`, `ownalw`, and `comalw` must finish before `lim` can start. Task `lim` then spawns two tasks, `ownlim` and `comlim`, which *both* must complete before `apply` can be executed. *Right:* Charm++ SDAG code expressing the logic on the left. Only when both own and contributions to the boundary nodes for both  $P^\pm$  and  $Q^\pm$  are complete on a given chare will the runtime system call the member function `lim()`. Only when both own and contributions to the boundary nodes for  $A$  are complete on a given chare will the runtime system call the member function `apply()`.

values are allowed to achieve,  $Q^\pm$ , Eq. (21), and the limited AEC applied to nodes,  $A$ , Eq.(24). Besides the slightly more complex task logic, there is nothing new in Figure 3 not encountered before: ordering among the edges of the graph is not assumed and thus the tasks must operate on disjoint data, which then are combined before used. `DistFCT::apply()` then calls back to `DiagCG` which applies the now combined limited AEC, updates the high-order solution, and continues by telling the runtime system to evaluating the degree of load imbalance across the whole problem and which performs migration if necessary. A number of different load balancing strategies are available in Charm++, which can be configured simply by passing a different command line argument to the executable. A crucial point is that we simply turn on load balancing and the runtime system performs CPU load measurement and object migration if necessary. When load balancing is finished, a member function is called on each chare, continuing to the next time step.

## 5. Solution verification

This section presents test problems used to verify the accuracy and the correctness of the numerical implementation of the method. As expected, the method is 2nd-order accurate for smooth problems.

We adopt a number of test problems that have been derived in [11, 12] using the method of manufactured solutions for the Euler equations that yield smooth solutions. Since the analytical solution for these problems are known, the numerical errors can be quantified. The errors are also used to establish the order of accuracy of the method. Convergence studies for each problem were performed with a series of finer meshes over a unit cube centered at the origin. Note that these are the same meshes that were used in [12]. The meshes were constructed from uniform hexahedra that were further subdivided into approximately uniform tetrahedra. Table 1 summarizes the three meshes used including the number of points, number of tetrahedra, and average edge length,  $h$ .

Mesh	Points	Tetrahedra	$h$
0	132651	750K	0.02
1	1030301	6M	0.01
2	8120601	48M	0.005

Table 1: Mesh sizes for convergence studies. Here K stands for thousand, M for million, and  $h$  is the average edge length.

The global  $L_2$  error in each field variable was calculated as

$$\|\varepsilon\|_2 = \frac{\sum_{v=1}^n M_l^v (\hat{U}^v - U^v)^2}{\sum_{v=1}^n M_l^v} \quad (26)$$

where  $n$  is the number of points,  $\hat{U}^v$  and  $U^v$  are the exact and computed solutions at mesh point  $v$ , and  $M_l^v$  is the lumped mass matrix, i.e., volume, at mesh point  $v$ . The convergence rate  $p$  was then calculated as

$$p = \frac{\log \|\varepsilon\|_2^{m+1} - \log \|\varepsilon\|_2^m}{\log h^{m+1} - \log h^m} \quad (27)$$

where  $m$  is a mesh index.

### 5.1. Vortical flow

The purpose of this problem is to test velocity errors generated by spatial operators in the presence of 3D vorticity and in particular the superposition of planar and vortical flows, analogous to vorticity stretching. The derivation of this test problem is given in [12]. The solution is

$$\begin{aligned} \rho &= 1 \\ u_i(x_i) &= \begin{pmatrix} \alpha x - \beta y \\ \beta x + \alpha y \\ -2\alpha z \end{pmatrix} \\ p(z) &= p_0 - 2\rho\alpha^2 z^2 \\ e &= p(z)\rho(\gamma - 1), \end{aligned} \quad (28)$$

and the source terms, in Eq. (1), are

$$\begin{aligned} S_\rho &= 0 \\ S_{u,i} &= \begin{pmatrix} \rho(\alpha^2 - \beta^2)x - 2\rho\alpha\beta y \\ \rho(\alpha^2 - \beta^2)y + 2\rho\alpha\beta x \\ 0 \end{pmatrix} \\ S_E &= u_i S_{u,i} + \frac{8\rho\alpha^3 z^2}{\gamma - 1}. \end{aligned} \quad (29)$$

The above solution was evolved for a single time unit using  $\gamma = 5/3$ ,  $p_0 = 10$ , and  $\rho = 1$  with three sets of parameters:

1.  $\alpha = 0.1$ ,  $\beta = 1$ . This case corresponds to a predominantly vortical flow with a small amount of flow to and from the origin.
2.  $\alpha = 1$ ,  $\beta = 0.1$ . This case corresponds to a weak vortex with strong flow away from the origin in the  $x$ - $y$  plane and toward the origin in the  $z$ -direction.
3.  $\alpha = \beta = 1$ . This case is effectively a superposition of the previous two, with vortical flow comparable in strength to the flow to and from the origin.

Figure 4 shows the initial and final velocity fields for each variant on center planes through the origin, which confirms the steady state nature of the problem. Also shown are the steady state pressure and energy fields. For the third case, Table 2 gives the  $L_2$  errors in the computed conserved quantities along with the measured convergence rates. A time step of 0.002 was used for the coarsest mesh and successively halved for the finer meshes. Figure 5 shows the log of the  $L_2$  errors as a function of the log of the average edge length. The measured convergence rate in all variables is approximately 2.0, as expected.

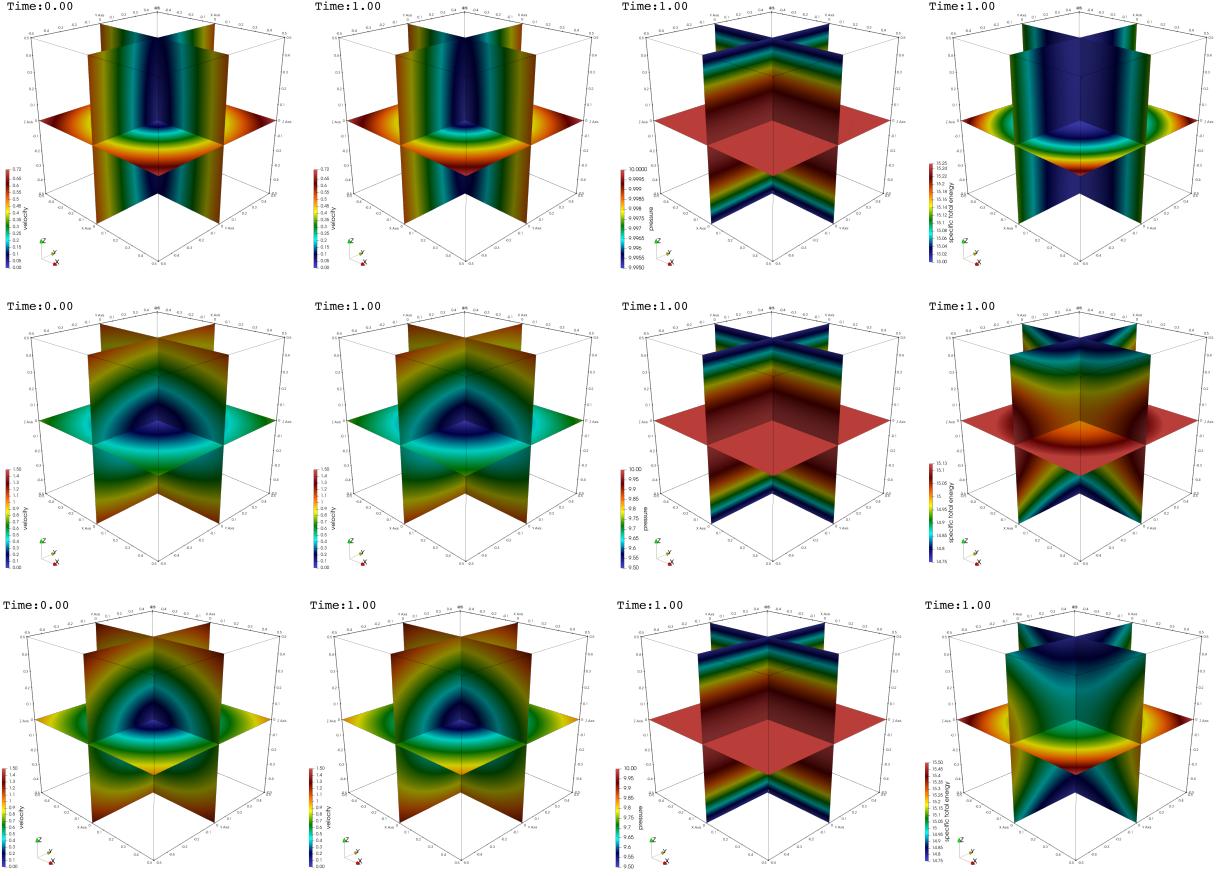


Figure 4: Initial (first column) and final (second column) velocity, pressure (third column), and total energy distributions (fourth column), for  $\alpha = 0.1$  and  $\beta = 1.0$  (top row),  $\alpha = 1.0$  and  $\beta = 0.1$  (middle row), and  $\alpha = \beta = 1.0$  (bottom row) for the vortical flow problem.

### 5.2. Nonlinear energy growth

The purpose of this problem is to test nonlinear, time dependent energy growth and the subsequent development of pressure gradients due to coupling between the internal energy and the equation of state. The derivation of this test problem is given in [12]. The solution is

$$\begin{aligned} \rho(x_i, t) &= \rho_0 + \exp(-\alpha t)g(x_i) \\ u_i(x_i) &= 0 \\ p &= [\rho_0 + \exp(-\alpha t)g] (\gamma - 1) (-3c_e - 3kh^2t)^{-1/3} \\ e &= (-3c_e - 3kh^2t)^{-1/3} \end{aligned} \tag{30}$$

with

$$\begin{aligned} g(x_i) &= 1 - (x^2 + y^2 + z^2) \\ h(x_i) &= \cos(\beta_x \pi x) \cos(\beta_y \pi y) \cos(\beta_z \pi z), \end{aligned} \tag{31}$$

Mesh	$L_2(\rho)$	$L_2(\rho u_1)$	$L_2(\rho u_2)$	$L_2(\rho u_3)$	$L_2(\rho E)$
750K	$1.31 \times 10^{-5}$	$2.91 \times 10^{-5}$	$1.24 \times 10^{-5}$	$1.52 \times 10^{-4}$	$6.29 \times 10^{-5}$
6M	$3.23 \times 10^{-6}$	$6.80 \times 10^{-6}$	$2.67 \times 10^{-6}$	$3.58 \times 10^{-5}$	$1.67 \times 10^{-5}$
48M	$8.00 \times 10^{-7}$	$1.63 \times 10^{-6}$	$6.30 \times 10^{-7}$	$8.05 \times 10^{-6}$	$4.18 \times 10^{-6}$
$\mathbf{p}$	2.01	2.06	2.08	2.15	2.00

Table 2:  $L_2$  errors and convergence rates for the vortical flow problem with  $\alpha = \beta = 1$ .

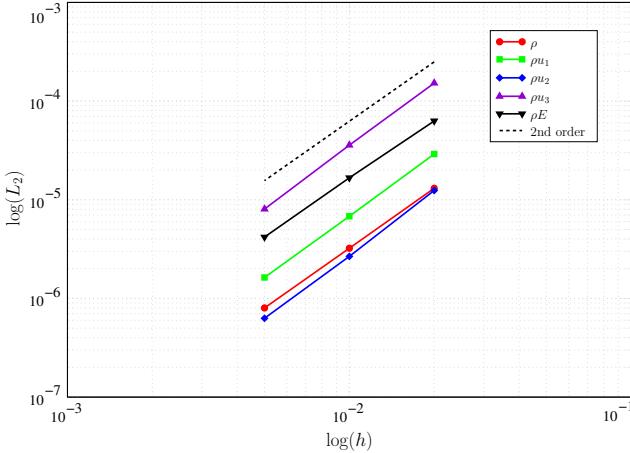


Figure 5:  $L_2$  errors as a function of mesh resolution for the vortical flow problem.

and the source terms, in Eq. (1), are

$$\begin{aligned}
S_\rho &= -\alpha \exp(-\alpha t) g \\
S_{u,i} &= 2kht [\rho_0 + \exp(-\alpha t)g] (\gamma - 1) (-3c_e - 3kh^2t)^{-4/3} \frac{\partial h}{\partial x_i} \\
&\quad + (-3c_e - 3kh^2t)^{-1/3} (\gamma - 1) \exp(-\alpha t) \frac{\partial g}{\partial x_i} \\
S_E &= \rho kh^2 e^4 + eS_\rho.
\end{aligned} \tag{32}$$

Simulations of this problem were performed with the following parameters:

$$\rho_0 = 2; \alpha = 0.25; \kappa = 0.8; \beta_i = (1.0, 0.75, 0.5); c_e = -1 \tag{33}$$

The solution was evolved over a single time unit with  $\gamma = 5/3$ . A time step of 0.001 was used for the coarsest mesh and successively halved for the finer meshes. Figure 6 shows the density, energy, and pressure distributions at four different times on center planes through the origin. Table 3 gives the  $L_2$  errors in the conserved quantities computed along with the measured convergence rates. The measured convergence rate in all variables is approximately 2.0, consistent with expectations.

Mesh	$L_2(\rho)$	$L_2(\rho u_1)$	$L_2(\rho u_2)$	$L_2(\rho u_3)$	$L_2(\rho E)$
750K	$7.67 \times 10^{-4}$	$1.61 \times 10^{-4}$	$1.08 \times 10^{-4}$	$6.98 \times 10^{-5}$	$3.39 \times 10^{-4}$
6M	$2.01 \times 10^{-4}$	$3.84 \times 10^{-5}$	$2.67 \times 10^{-5}$	$1.63 \times 10^{-5}$	$7.97 \times 10^{-5}$
48M	$5.57 \times 10^{-5}$	$9.38 \times 10^{-6}$	$6.59 \times 10^{-6}$	$3.90 \times 10^{-6}$	$1.97 \times 10^{-5}$
$\mathbf{p}$	1.85	2.03	2.02	2.06	2.02

Table 3:  $L_2$  errors and convergence rates for the nonlinear energy growth problem.

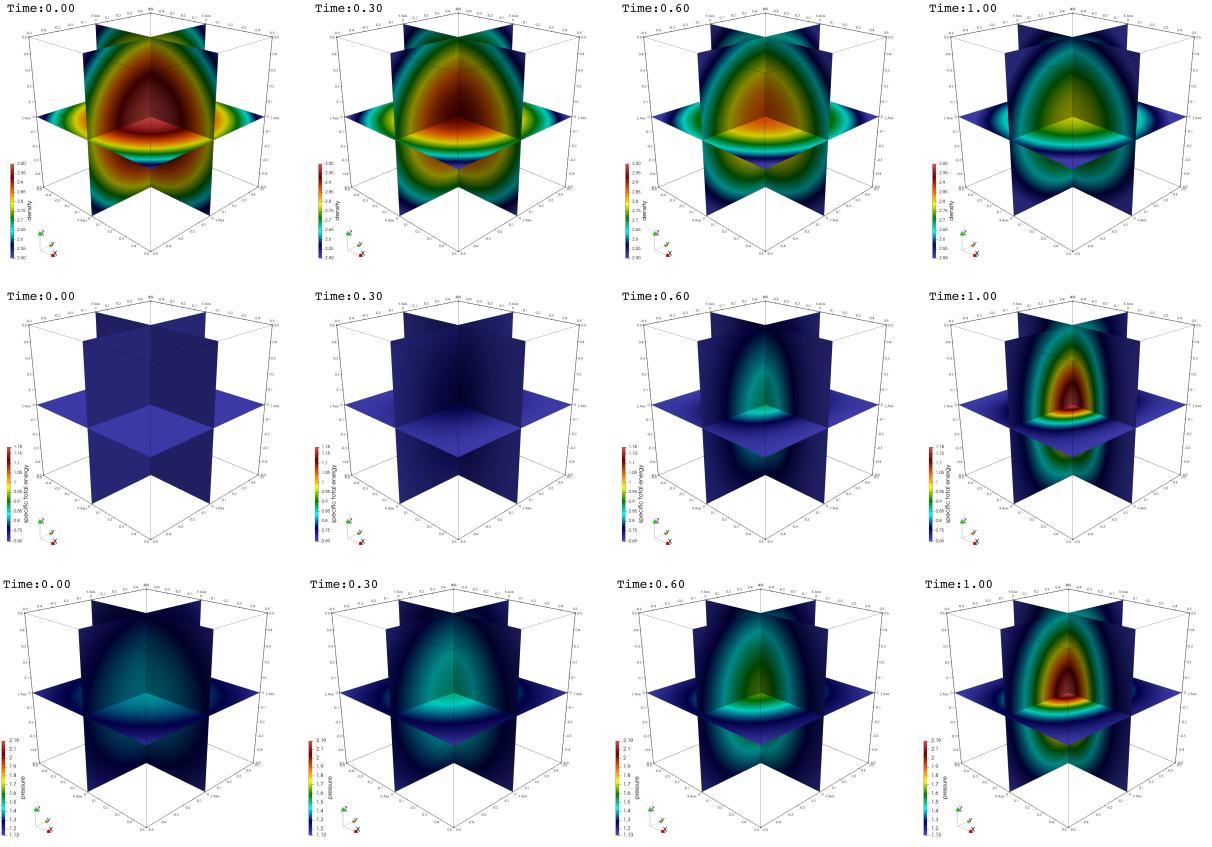


Figure 6: Density (top row), energy (middle row), and pressure (bottom row) distributions on center planes at four different simulation times (columns) for the nonlinear energy growth problem.

### 5.3. Rayleigh-Taylor unstable configuration

The purpose of this test case is to assess time dependent fluid motion in the presence of Rayleigh-Taylor unstable conditions, i.e., opposing density and pressure gradients. The derivation of this test problem is given in [12]. The solution is

$$\begin{aligned}
 \rho(x) &= \rho_0 - (\beta_x x^2 + \beta_y y^2 + \beta_z z^2) \\
 u_i(x_i, t) &= f(t)g(x_i) = \cos(k\pi t) \begin{pmatrix} z \sin(\pi x) \\ z \cos(\pi y) \\ -\frac{1}{2}\pi z^2 [\cos(\pi x) - \sin(\pi y)] \end{pmatrix} \\
 p(x) &= p_0 + \alpha (\beta_x x^2 + \beta_y y^2 + \beta_z z^2) \\
 E &= \frac{p}{\rho(\gamma - 1)} + \frac{1}{2} f^2 g_i g_i
 \end{aligned} \tag{34}$$

and the source terms, in Eq. (1), are

$$\begin{aligned} S_\rho &= u_i \frac{\partial \rho}{\partial x_i} \\ S_{u,i} &= \rho g_i \frac{\partial f}{\partial t} + f g_i S_\rho + \rho f^2 g_j \frac{\partial g_i}{\partial x_j} + \frac{\partial p}{\partial x_i} \\ S_E &= \rho g_i g_i f \frac{\partial f}{\partial t} + \left[ \frac{p}{\rho(\gamma - 1)} + \frac{1}{2} f^2 g_i g_i \right] S_\rho \\ &\quad + \rho f g_i \left[ \frac{1}{\rho(\gamma - 1)} \frac{\partial p}{\partial x_i} - \frac{p}{\rho^2(\gamma - 1)} \frac{\partial \rho}{\partial x_i} + f^2 g_j \frac{\partial g_j}{\partial x_i} \right] + f g_i \frac{\partial p}{\partial x_i}. \end{aligned}$$

395 The solution was evolved over a single time unit with  $\gamma = 5/3$ . A time step of 0.001 was used for the coarsest mesh and successively halved for the finer meshes. Figure 7 depicts the velocity field at the surface at different simulation times, showing the reversal of the velocity components in time, indicating the spatial and temporal dynamics of the velocity field. Figure 8 density, pressure, energy, and velocity on the surface at the final simulation time. Table 4 gives the numerical  $L_2$  errors in the conserved quantities computed.

400 The order of convergence approaches the expected value of 2.0.

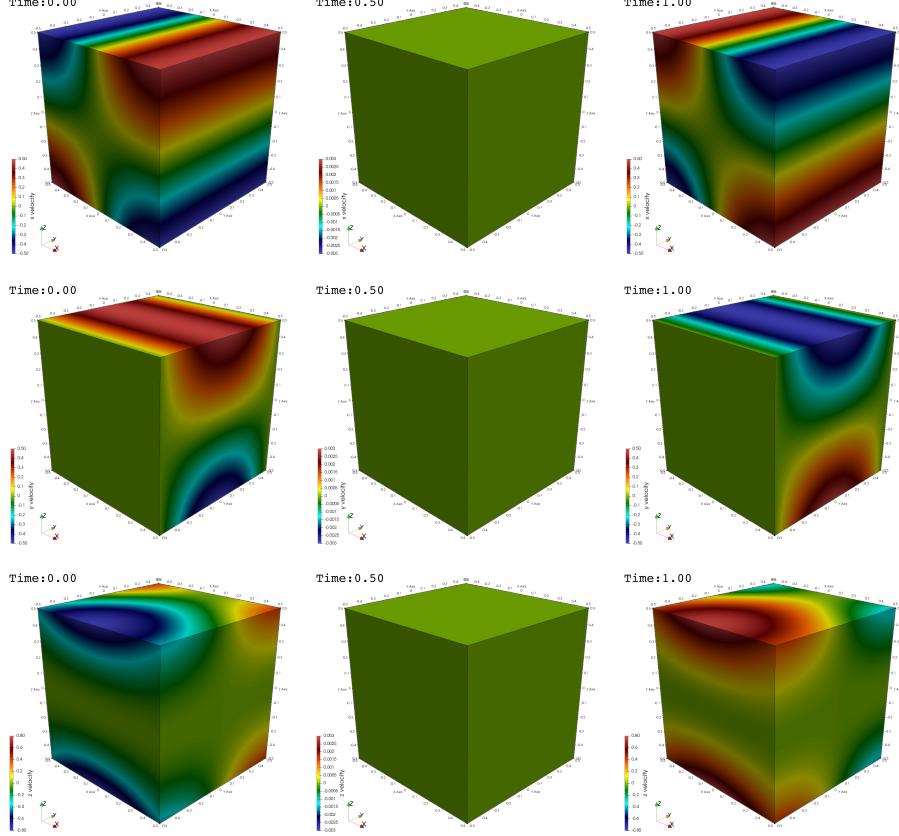


Figure 7: Velocity components on the surface at three different simulation times ( $t = 0, 0.5$ , and  $1.0$ ) for the Rayleigh-Taylor unstable configuration. The velocity reverses direction over the indicated time interval.

#### 5.4. Taylor-Green vortex

3D simulations of the 2D Taylor-Green vortex, see e.g., [13], were performed using the same meshes as for the other test problems described above. A similar configuration was also computed in [14] on a different,

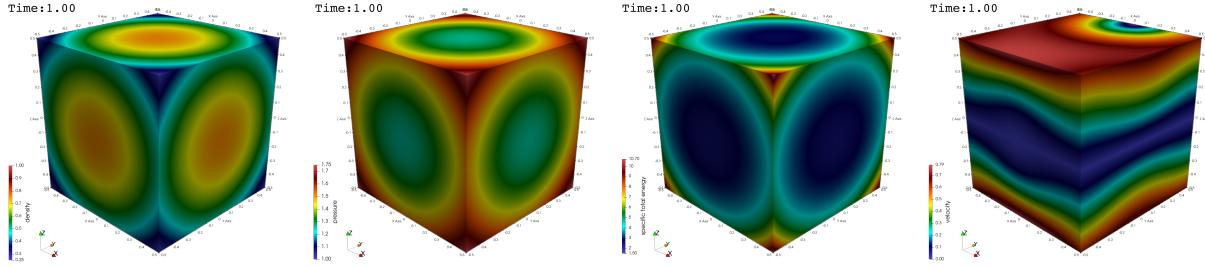


Figure 8: Density, pressure, energy, and velocity on the surface at the final simulation time for the Rayleigh-Taylor unstable configuration.

Mesh	$L_2(\rho)$	$L_2(\rho u_1)$	$L_2(\rho u_2)$	$L_2(\rho u_3)$	$L_2(\rho E)$
750K	$4.50 \times 10^{-4}$	$6.04 \times 10^{-4}$	$6.46 \times 10^{-4}$	$4.22 \times 10^{-5}$	$3.35 \times 10^{-4}$
6M	$1.38 \times 10^{-4}$	$1.84 \times 10^{-4}$	$1.92 \times 10^{-4}$	$1.28 \times 10^{-5}$	$8.29 \times 10^{-5}$
48M	$4.22 \times 10^{-5}$	$5.58 \times 10^{-5}$	$5.49 \times 10^{-5}$	$3.98 \times 10^{-5}$	$2.12 \times 10^{-5}$
$\mathbf{p}$	1.71	1.72	1.81	1.69	1.97

Table 4:  $L_2$  errors and convergence rates for the Rayleigh-Taylor problem.

practically 2D, domain. In this test case, the following solution

$$\begin{aligned} \rho(x) &= 1 \\ u_i(x_i, t) &= \begin{pmatrix} \sin \pi x \cos \pi y \\ -\cos \pi x \sin \pi y \\ 0 \end{pmatrix} \\ p(x) &= 10 + \frac{1}{4}\rho (\cos 2\pi x + \cos 2\pi y) \\ e &= \frac{p(x)}{\rho(\gamma - 1)} \end{aligned} \tag{35}$$

is maintained in a steady state with the energy source term

$$\begin{aligned} S_\rho &= 0 \\ S_{u,i} &= 0 \\ S_E &= \frac{3\pi}{8} (\cos \pi x \cos 3\pi y - \cos 3\pi x \cos y). \end{aligned} \tag{36}$$

- 405 This test problem was also run for a single time unit with  $\gamma = 5/3$ . A time step of 0.002 was used for the coarsest mesh and successively halved for the finer meshes. Figure 9 shows the velocity magnitudes across a center plane at different simulation times. Table 5 shows the numerical  $L_2$  errors of the conserved quantities computed. As expected, the approximate order of convergence is 2.0.

Mesh	$L_2(\rho)$	$L_2(\rho u_1)$	$L_2(\rho u_2)$	$L_2(\rho u_3)$	$L_2(\rho E)$
750K	$1.14 \times 10^{-5}$	$8.56 \times 10^{-5}$	$4.59 \times 10^{-4}$	$2.64 \times 10^{-5}$	$2.49 \times 10^{-4}$
6M	$2.34 \times 10^{-6}$	$1.62 \times 10^{-5}$	$8.72 \times 10^{-5}$	$4.40 \times 10^{-6}$	$5.84 \times 10^{-5}$
48M	$5.65 \times 10^{-7}$	$3.17 \times 10^{-6}$	$1.77 \times 10^{-5}$	$9.10 \times 10^{-7}$	$1.42 \times 10^{-5}$
$\mathbf{p}$	2.05	2.35	2.30	2.27	2.04

Table 5:  $L_2$  errors and convergence rates for the Taylor-Green problem.

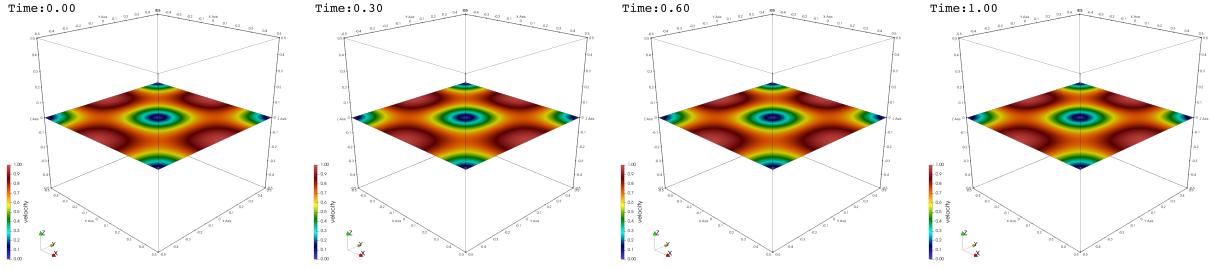


Figure 9: Velocity magnitudes at a center plane at different simulation times for the Taylor-Green test problem.

Mesh	Points	Tetrahedra	$dx$
0	7294	34181	0.840
1	51794	273448	0.414
2	389139	2187584	0.206
3	2014277	17500672	0.103

Table 6: Mesh sizes for the Sod shock tube problem. Here  $dx$  is the average edge length along the length of tube.

### 5.5. Sod's shock tube

The 1D Riemann problem, see e.g., [13], is modeled as a 3D tube with a circular cross section of unit length with a diameter of 0.05, with  $\gamma = 1.4$ , and the following initial conditions:

$$u_i(x_i) = 0.0 \quad (37)$$

$$\rho(x_i) = \begin{cases} 1.0 & x < 0.5 \\ 0.125 & x \geq 0.5 \end{cases} \quad (38)$$

$$p(x_i) = \begin{cases} 1.0 & x < 0.5 \\ 0.1 & x \geq 0.5 \end{cases} \quad (39)$$

where  $x$  is the distance along the length of the tube. The density field is plotted on the surface of Mesh 1 at  $t = 0$ , and at  $t = 0.2$  in Figure 10. The numerical solution was also extracted at  $t = 0.2$  along a line through the center of the tube. The density, velocity, total energy, and pressure are depicted in Fig. 11 for the successively finer meshes given in Table 6 together with the analytical solution.

### 5.6. Sedov's blast wave

Another test problem with a discontinuous solution is the Sedov problem [15], where a source of energy is defined to produce a shock in a single computational cell at  $t = 0$ . We used three successively finer meshes, given in Table 7, to compute the numerical solution in 3D with  $\gamma = 5/3$  and the following initial conditions:

$$u_i(x_i) = 0.0 \quad (40)$$

$$\rho(x_i) = 1.0 \quad (41)$$

$$e(x_i) = \begin{cases} 1.0 \times 10^{-4} & x_i \neq 0.0 \\ e_s & x_i = 0.0 \end{cases} \quad (42)$$

where  $e_s$  and  $p_s$  are mesh-dependent source values summarized in Table 7 along with the corresponding volume  $V_s$  of the source region. Note that energy is in units of Mbar – cm<sup>3</sup>/g, pressure is in units of Mbar, and source volume is in units of cm<sup>3</sup>. These initial conditions correspond to pressure ratios of  $\mathcal{O}(10^8)$ – $\mathcal{O}(10^{10})$  across a single element.

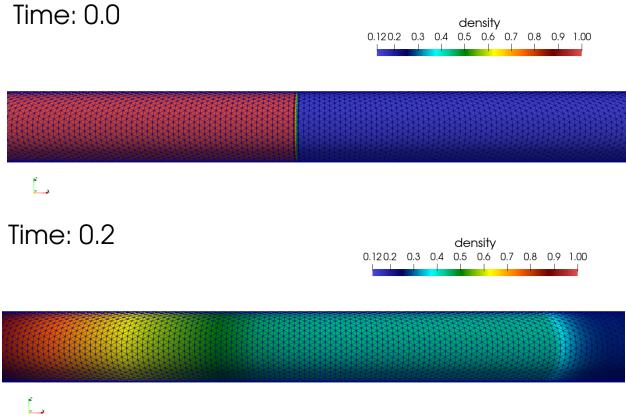


Figure 10: Surface mesh colored by density at  $t=0$  and  $t=0.2$  on mesh 1 for the Sod shock tube problem.

Mesh	Points	Tetrahedra	$h$	$V_s$	$e_s$
1	65958	362363	0.029	$1.14 \times 10^{-6}$	$5.42 \times 10^4$
2	505724	2898904	0.014	$1.78 \times 10^{-7}$	$3.46 \times 10^5$
3	3956135	23191232	0.007	$2.23 \times 10^{-8}$	$2.77 \times 10^6$

Table 7: Mesh sizes and source parameters for the Sedov problem. Here  $h$  is the average edge length.

The numerical solution for the pressure and density at two different instants in time are given in Figure 12. The numerical solution for the density was also extracted along a line through the  $x$  axis using the three meshes. This is plotted in Figure 13 together with a semi-exact solution at  $t=1.0\mu s$ .

## 6. Parallel performance

This section discusses parallel performance, including strong and weak scaling, the profile of a time step, and we demonstrate automatic load balancing using various strategies available in Charm++.

### 6.1. Strong scaling

Using increasing number of compute cores with the same problem measures strong scaling, characteristic of the algorithm and its parallel implementation. Strong scalability helps answer questions, such as *How much faster can one obtain a given result (at a given level of numerical error) if larger computational resources were available*. To measure strong scaling we ran the vortical flow problem using a 794M-cell mesh on varying number of PEs for 100 time steps and measured the wall-clock time for the 100 steps. To reduce the effects of statistical variability of the machine and its communication network, the time for a given data point was computed as the median of the times, measured at every 10th step multiplied by 10.

Figure 14 depicts wall-clock times of 100 time steps using a 794M-cell mesh. The figure shows that strong scaling is close to ideal up to approximately 30K cores after which communication overwhelms computation. This is an important data point: the recommended minimum load per core for this algorithm on this machine, is about 28K elements per PE; throwing more resources at this size problem is not economical and, conversely, using less than about 28K elements per core will likely yield suboptimal performance.

Figure 14 also shows the difference in performance of the algorithm between Charm++’s SMP (symmetric multi-processing) and non-SMP mode. SMP, compared to non-SMP, mode refers to a number of Charm++ implementation details that allow the runtime system to treat a *logical* compute node as a shared-memory machine, which then allows various optimizations, such as pointer passing instead of communication if both the sender and receiver reside on the same compute node. SMP mode also allows configuring the runtime system to use a designated compute core (or cores) for every logical node responsible solely for communication

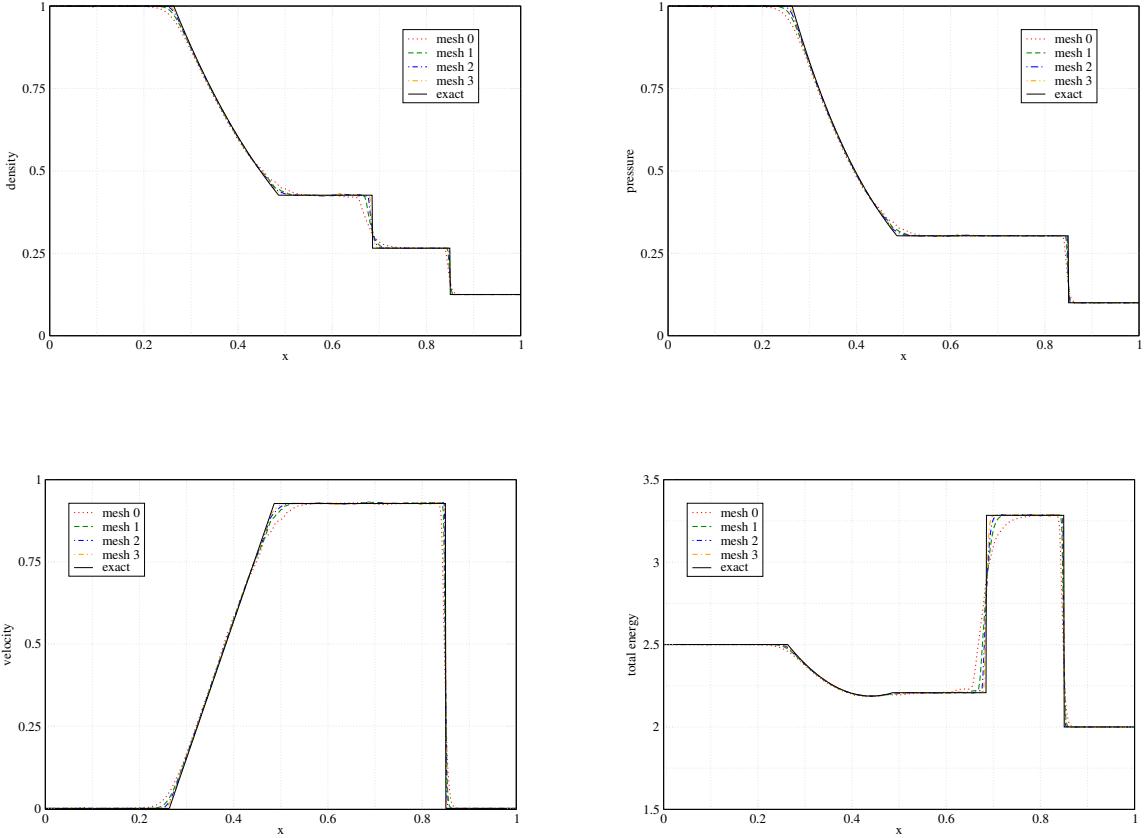


Figure 11: Density, velocity, pressure, and total energy with their exact solutions along the center of the tube for Sod shock tube problem for 4 different meshes.

and allowing the rest of the cores to only perform computation. Such freedom in configuration allow tailoring the runtime system specific to the hardware by, e.g., taking into account of caches shared by specific compute cores, and in general, allow a more optimal utilization of the multi-level memory hierarchy with non-uniform access costs at each level that is increasingly found in current supercomputing hardware. Simply put, Charm++’s SMP mode can be thought of as MPI+X, where X stands for some programming paradigm for threading and on-node parallelism, e.g., OpenMP, POSIX threads, or Kokkos. In contrast, Charm++’s non-SMP mode can be thought of as “*MPI everywhere*”, where every compute core is treated as independent with its own memory space. Note that there are no source code changes, additional code, or duplicate kernels that would be required to utilize Charm++ in SMP vs. non-SMP mode. The code uses a single abstraction for parallel programming and that is Charm++. Selecting SMP or non-SMP mode amounts to *building* Charm++ in SMP or non-SMP mode and linking the solver the same way to the desired Charm++ instance as a library.

We ran the same code running the same mesh and problem in both SMP and non-SMP mode and performed the strong scaling study for both. Wall-clock times, measuring time-stepping only, as well as total runtimes are given in Table 8, and the former are also plotted in Figure 14. For all runs, the non-SMP setup corresponded to using all 36 cores of a compute node uniformly, while in SMP mode we designated a core for communication per each of the 2 sockets (18 cores) of a compute node. (This means that in SMP mode, of a total of 14,400 cores, 400 compute nodes, only 13600 were used for actual computation.) Figure

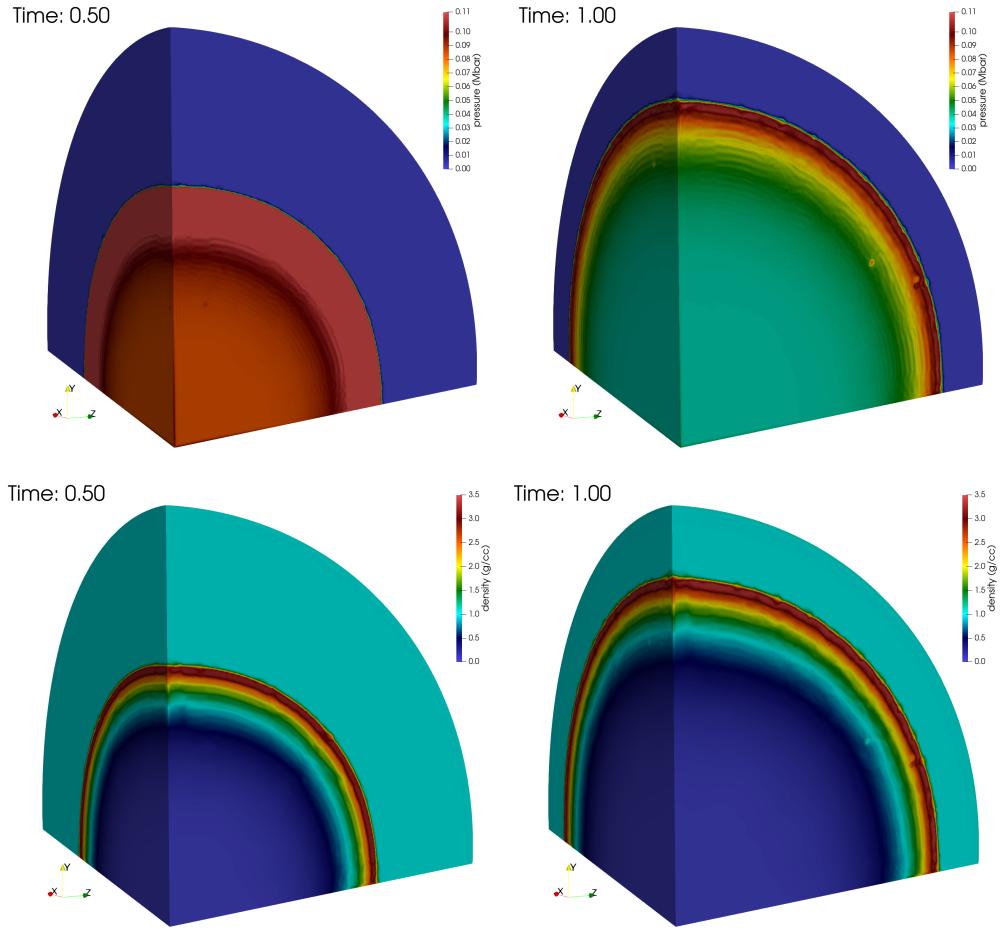


Figure 12: Surface pressure (top) and density (bottom) for the Sedov problem at  $t=0.5\mu s$  and  $t=1.0\mu s$  on the Mesh 3.

14 shows that, in general, SMP mode is approximately 2x faster than non-SMP mode, c.f., the corresponding middle columns in Table 8. The due to the benefits of SMP mode described above.

Table 8 also shows the wall-clock measurements of the total runtimes. It is apparent that at the lower core counts (360–3600) non-SMP mode outperforms SMP mode when measured by total runtime. This is the effect of the unequal performance of reading the mesh followed by mesh data redistribution and the computation of mesh node communication maps in SMP and non-SMP modes. Both modes perform these tasks in parallel, but SMP mode uses 2 parallel streams per compute node, while non-SMP mode uses 36. Thus non-SMP mode will perform better, at lower core counts. The data in the tables also show that this performance figure turns around between 3600 and 7200 PEs, where both SMP and non-SMP mode yield approximately equal total runtimes. At larger core counts ( $\approx 5K$ ) and beyond, SMP mode clearly outperforms non-SMP mode. This is expected, considering the limited parallelism of the underlying parallel file system. As expected, at larger core counts it is more beneficial to use SMP mode, e.g., at 36K cores, the total runtime of non-SMP mode is almost 6 times that of SMP mode.

#### 480 6.2. Weak scaling

Figure 15 depicts wall-clock times of 100 time steps keeping the load per compute node the same. We start with a 48M-cell mesh and perform an initial uniform mesh refinement before time stepping that replaces every tetrahedron in the computational mesh with 8 smaller ones and thus yields exactly 384M

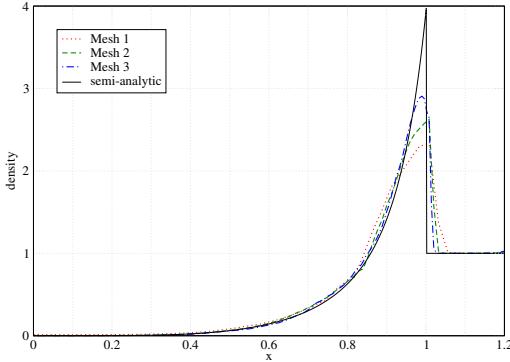


Figure 13: Density profiles for the Sedov problem with increasing mesh resolution at  $t=1.0\mu s$ .

Number of PEs	100 time steps, sec, non-SMP	Total time, min:sec, non-SMP	100 time steps, sec, SMP	Total time, min:sec, SMP
360	94599	20:35	80007	38:56
720	60966	14:09	37470	22:25
900	50117	12:03	28114	16:04
1800	28868	8:00	13562	9:38
3600	15676	6:10	6730	5:11
7200	8416	5:12	3681	4:35
14400	4227	6:48	1998	2:55
28800	2130	15:30	1577	3:33

Table 8: Wall-clock times for time stepping only (measuring 100 time steps) and total time (including mesh load and setup) in Charm++’s non-SMP and SMP modes. The time stepping times are plotted in Figure 14.

cells and run this on 6 computes nodes. The next points were obtained by doing two and three refinement steps, respectively, yielding over 3B and almost 25B cells, respectively. Increasing the problem size in exact proportion with the computational resources establishes weak scalability, characteristic of the algorithm and its implementation. Weak scaling enables answering questions, such as *How effective can one use the available (largest) compute resources if ever decreasing numerical errors, e.g., larger resolutions, are required*. Figure 15 shows that the weak scaling of the algorithm is not ideal: there is an approximate 14% increase of extra computational cost compared to ideal weak scaling. This quantifies the ”cost” of the algorithm and its implementation when used at ever larger scales (both larger problems and larger compute resources).

### 6.3. Computational cost profile

Figure 16 depicts the relative CPU utilization of the various computational tasks during 100 time steps in SMP mode. This run was done on 25 physical compute nodes, using 2 logical nodes per compute nodes, designating 2 PEs per each compute node for communication only, which leaves 850 worker PEs. The figure shows that the right hand side computation (gather+scatter) takes about 40% of a time step: this step consists of both tasks RHS and DIF, computing the right hand side of Eq. 1 including the flux and source terms for both the low and high order solutions (including mass diffusion). Another significant portion, 30%, is aec + a1w, computing the anti-diffusive element contributions and the minimum and maximum increments of allowed solution values in mesh nodes, respectively, prerequisites of the FCT limiting procedure. Computing the limit coefficients, task lim computing Eqs. (21–24) in each node, costs a little over 10%. About 2-5%

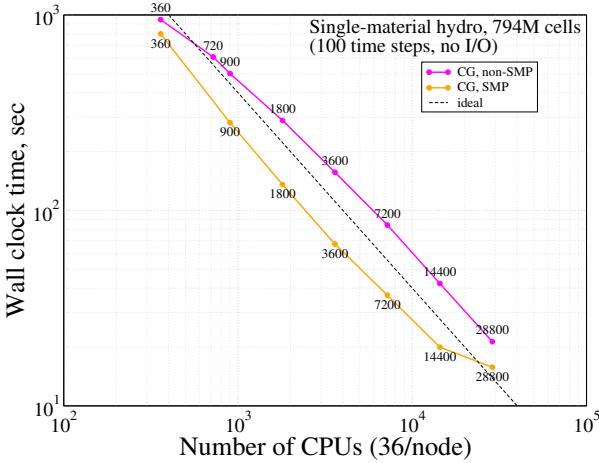


Figure 14: Strong scaling at  $\mathcal{O}(10^4)$  compute cores of the using two different modes of Charm++: non-SMP and SMP.

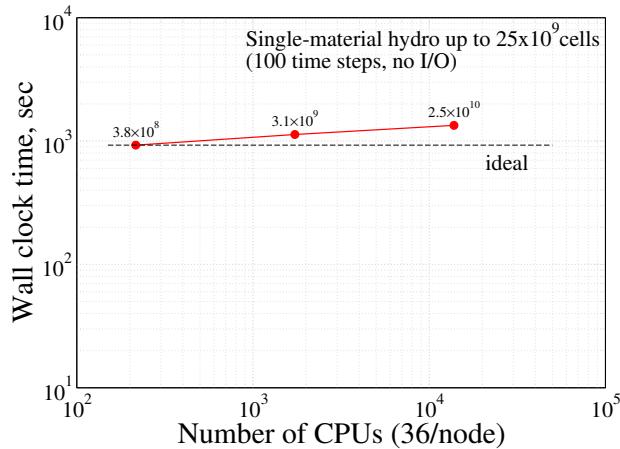


Figure 15: Weak scaling up to 25 billion cells.

of the cost, purple in Figure 16, is the cost of computing the size of the next time step size. The figure also shows that 10-15% of the total runtime is spent idle, when CPUs are waiting for messages without doing any useful work. Finally, the tiny black gaps between the white `idle` and the top of the figure denoting 100% is the cost of the runtime system: scheduling, sending and receiving messages, etc.

The pie-chart in Figure 16, helps to appreciate the communication costs. On the pie-chart, the tiny dark red section on the left depicts the cost of the communication (0.7% of a time step) of the right hand side (the rhs and diffusion). The cost of communication tasks, `comaec`, `comalw`, and `comlim`, see also Figure 3, are depicted as almost vertical sections on the right between the light blue `gather` and the yellow `1im` slices. Their combined cost is  $0.38\% + 0.44\% + 0.36\% \approx 1.2\%$ . This puts the total relative cost of communication to just below 2% of the total runtime of time stepping. The dark blue section between the yellow `1im` and other light blue `scatter` (0.7%) depicts the total cost of combining the own and communicated portions of the limited AECs, Eq. (24), and of applying the limited solution, Eq. (25).

Figure 17 gives an idea of the CPU utilization by the different tasks in time and their overlapped execution due to asynchrony. The figure depicts 5 consecutive time steps on a single compute node, horizontally

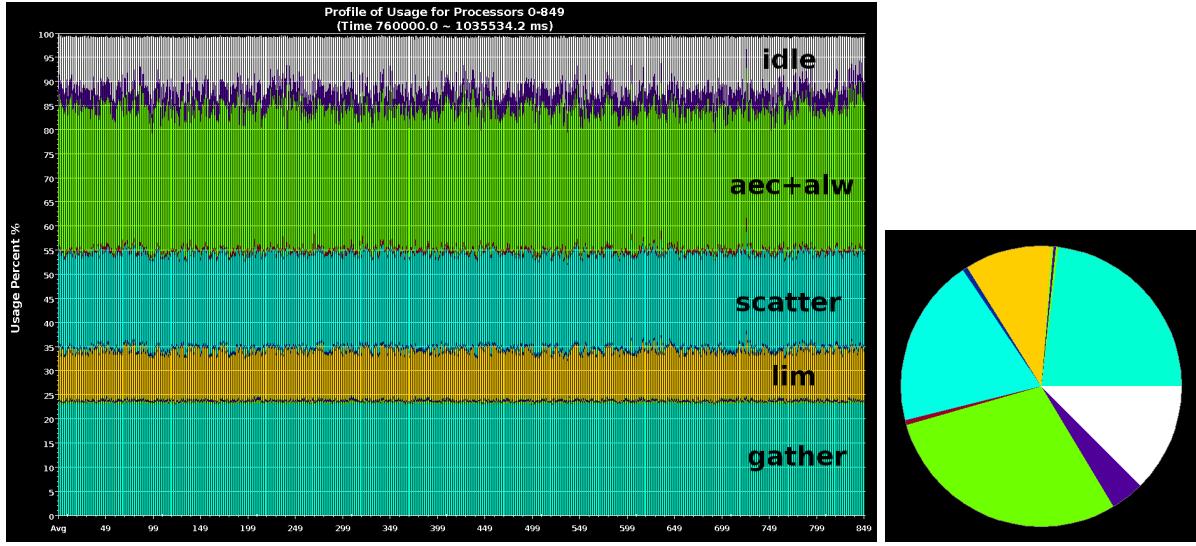


Figure 16: Average relative CPU utilization across 100 time steps and 850 worker CPUs.

displaying 36 PEs. Based on the minimum, maximum, and average number of elements across the 850 mesh partitions, respectively, 934152, 934153, and 934152, one would think that CPU utilization should be very well-balanced. However, the timeline shows that even this extremely homogeneous load distribution results in appreciable idle times at the end of time steps, denoted by black between the blue gather and scatter regions. This adds up to the 10-15% average idle time, discussed above. This is likely caused by the uneven load due to the unequal number of boundary points among charles: mesh partitions in a corner of the cube have many boundary points compared to internal mesh partitions without any boundary points.

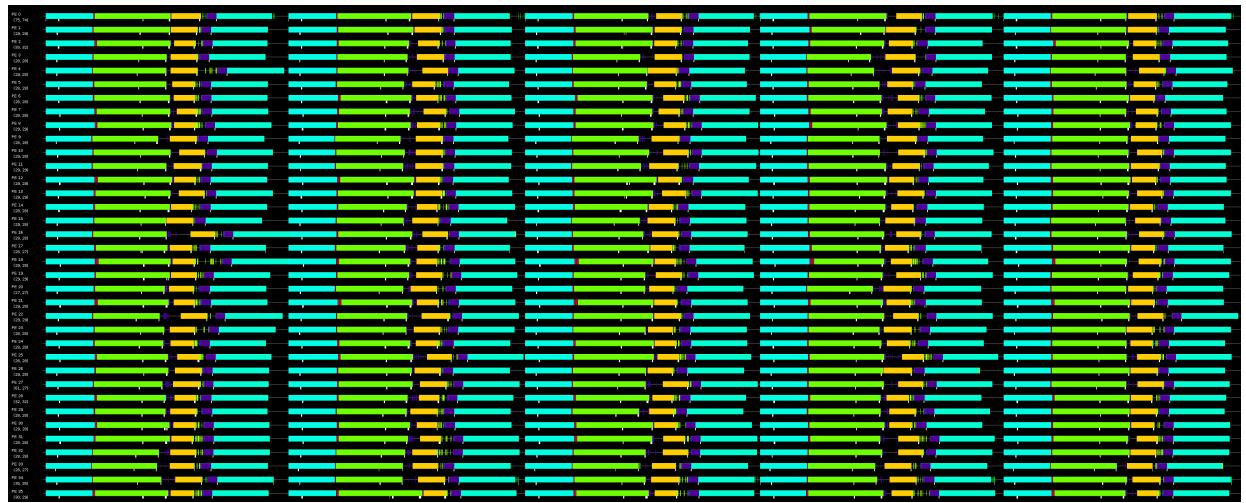


Figure 17: CPU utilization across 5 time steps on a single compute node (36PEs) from a 900-core run on 25 compute nodes. The task coloring corresponds to that of Figure 16 and the small white vertical lines denote message sends. Note that the right hand side calculation, see Eq. (9), is broken up into two parts, since the gather step can be computed before the new  $\Delta t$  is available, followed by the scatter step, allowing overlap of computation and communication.

While improvements to the above CPU utilization (e.g., to reduce the idle time and increase overlap) are likely possible, there are three important conclusions we can draw from the above discussion:

- 525 1. The overhead of the runtime system is negligible compared to computation and communication.

2. The communication costs ( $\approx 2\%$ ) are acceptable, especially considering the higher level (and safer) abstraction allowed by Charm++ compared to MPI-style programming, see Listing 1.
3. Even well-balanced distributed problems (measured by the number of computational cells) may lead to appreciable idle times.

530 *6.4. Load balancing*

To exercise the built-in load balancers in Charm++, and to demonstrate them for an unstructured-grid solver, we have computed the Sedov problem, adding some extra computational load to those cells whose density exceeds the value of 1.5. This mimics, e.g., combustion, non-trivial material equations of state, etc., and induces realistic load imbalance, characteristic of multi-physics simulations.

535 Figure 19 depicts the simulation time during time stepping, measured in ms/step, for a number of shared-memory runs on a 3M-cell mesh, taking 500 time steps. One can see that the density peak exceeds the value of 1.5 around the 130th time step, inducing load imbalance and this persists until the end of the simulation. The extra load corresponds to the cells depicted in Figure 18, displayed at two different times. It is clear from the Figure 19 that multiple load balancing strategies can successfully and effectively homogenize the uneven, 540 dynamically generated, a priori unknown computational load. The common requirement for effective load balancing is fine-grained work units, ensured by increasing the degree of overdecomposition (virtualization) from 1 to 100x, which yields 100 $\times$  the mesh partitions compared to the number of CPUs the problem is running on. As expected, increasing the number of partitions beyond the number of CPUs has a cost due to increased communication cost, c.f., black and red lines. Table 9 shows some timings without and with 545 load balancing. For this setup the GreedyCommLB load balancer performs the best, yielding 8.2% speedup compared to using no load balancing.

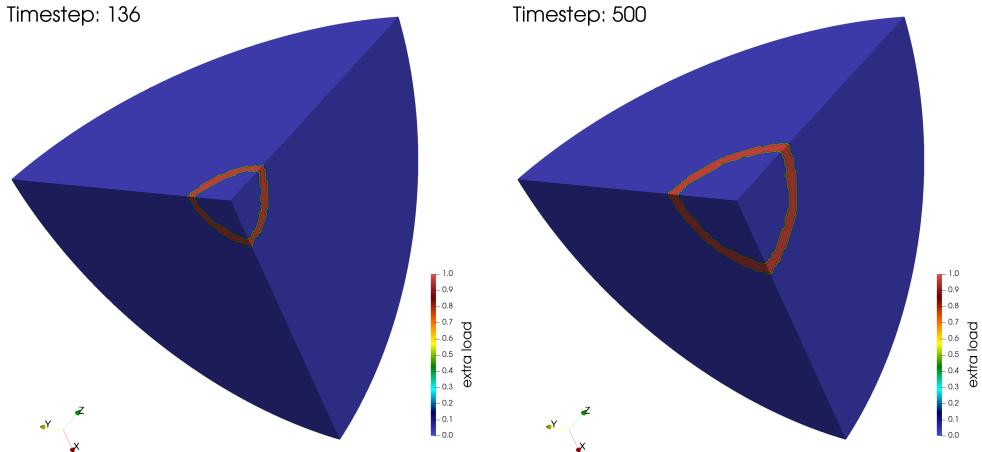


Figure 18: Spatial distributions of the extra load, corresponding to the fluid density exceeding the value 1.5, during the time evolution of the Sedov problem on the 3M-cell mesh: (left) shortly after the onset of load imbalance, (right) at end of the simulation.

Figure 20 shows simulation times for a larger, 30M-cell mesh, run on 10 compute nodes of a distributed-memory cluster with 36 CPUs/node. Except the larger mesh run on a cluster, everything else is the same as before. For the distributed-memory computations, besides centralized load balancers (GreedyCommLB, 550 GreedyRefineLB, RefineLB, RefineKLB), we have also used distributed load balancers, e.g., DistributedLB and NeighborLB, available in Charm++. As their name suggests, centralized load balancers require synchronization but their decision is more accurate due to their full knowledge of the entire simulation. Distributed load balancers only exchange neighbor information thus they are less accurate but cost less. This is consistent with our findings in Figure 20 as well as the timings in Table 10. For this problem, the best performance 555 is achieved by NeighborLB with a speedup over 10 $\times$  compared to no load balancing.

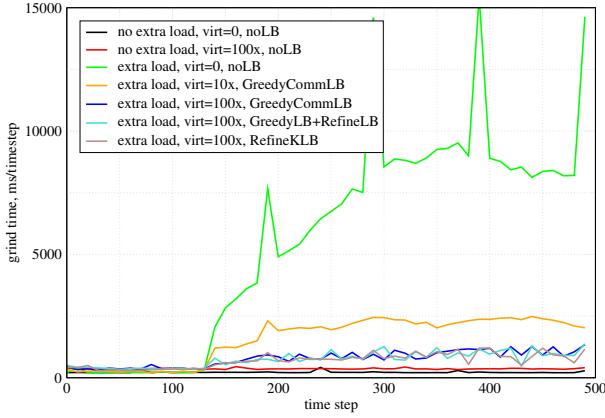


Figure 19: Grind-time during time stepping computing a Sedov problem without and with extra load in cells whose fluid density exceed 1.5, inducing load imbalance, without and with load balancing. These simulations were run on a shared-memory workstation, in SMP mode, using 30 worker threads + 2 communication threads, on a 3M-cell mesh. Load balancing was invoked at every 10th time step for all load balancers.

Case	Extra load	Virtualization	Charm++ load balancer	Median grind-time, ms/step	Total time, s	Speedup relative to case 3
1	no	—	—	215	93	—
2	no	100×	—	362	148	—
3	yes	—	—	6898	2567	1.0×
4	yes	10×	GreedyCommLB	2027	656	3.9×
5	yes	100×	GreedyCommLB	788	312	8.2×
6	yes	100×	GreedyLB+RefineLB	752	315	8.1×
7	yes	100×	RefineKLB	732	310	8.2×

Table 9: Timings for the Sedov problem, using a 3M-cell mesh, without and with extra load without and with load balancing on a shared-memory machine.

As shown, excellent performance can be obtained on both shared-, and distributed-memory machines, using the built-in automatic load balancers of Charm++. We emphasize that we have written no load balancing code: we simply ensure overdecomposition and turn on load balancing; the runtime system measures CPU load during simulation and automatically performs object migration to homogenize the load. This is particularly beneficial for applications with *a priori* unknown and/or dynamically changing load distribution, characteristic of multiphysics and/or heterogeneous performance of a large data center. The data also demonstrates that the cost of load balancing is negligible compared to the savings over the unbalanced problem without load balancing.

## 7. Summary

This paper discussed the implementation, verification, and parallel performance of a node-centered continuous Galerkin finite element flux-corrected transport algorithm for the simulation of high-speed compressible fluid dynamics on large 3D unstructured meshes using the Charm++ asynchronous tasking runtime system. We have also demonstrated the benefits of automatic load balancing in Charm++. The source code with documentation is available at <https://quinoacomputing.org>.

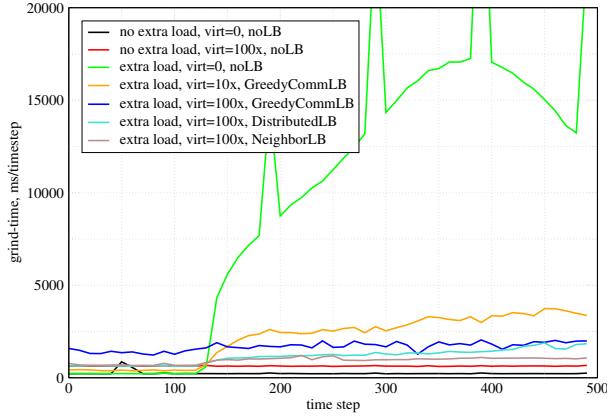


Figure 20: Grind-time during time stepping computing a Sedov problem with extra load in cells whose fluid density exceed the threshold of 1.5, inducing load imbalance. These simulations were run on a distributed-memory cluster on 10 compute nodes, in SMP mode, using 34 worker threads + 2 communication threads per node, on a 30M-cell mesh. Load balancing was invoked at every 10th time step for all load balancers.

Case	Extra load	Virtualization	Charm++ load balancer	Median grind-time, ms/step	Total time, s	Speedup relative to case 3
1	no	—	—	222	93	—
2	no	100×	—	630	359	—
3	yes	—	—	11562	4618	1.0×
4	yes	10×	GreedyCommLB	2530	1231	3.8×
5	yes	100×	GreedyCommLB	1685	1058	4.4×
6	yes	100×	DistributedLB	1209	514	9.0×
7	yes	100×	NeighborLB	995	430	10.7×

Table 10: Timings for the Sedov problem, using a 30M-cell mesh, with extra load with and without load balancing on 10 compute nodes of a distributed-memory cluster.

## 570 8. Acknowledgements

The authors thank Charmworks, Inc., for their helpful suggestions regarding load balancing, Projections, and programming with Charm++ in general. The authors would also like to thank Neil Carlson of Los Alamos National Laboratory for suggesting the algorithm used for computing the nodal communication maps in parallel and for the discussions on solving partial differential equations on unstructured meshes and software development in general. The work presented in this paper was supported by the Laboratory Directed Research and Development program of Los Alamos National Laboratory under project number LDRD-20170127-ER.

## References

- [1] R. Löhner, K. Morgan, J. Peraire, M. Vahdati, [Finite element flux-corrected transport \(FEM-FCT\) for the Euler and Navier-Stokes equations](#), Int. J. Numer. Meth. Fl. 7 (10) (1987) 1093–1109. doi:[10.1002/fld.1650071007](https://doi.org/10.1002/fld.1650071007). URL <http://dx.doi.org/10.1002/fld.1650071007>
- [2] R. Löhner, [Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods](#), Wiley, 2008. URL <https://onlinelibrary.wiley.com/doi/book/10.1002/9780470989746>

- 585 [3] P. Lax, B. Wendroff, *Systems of conservation laws*, Communications on Pure and Applied Mathematics 13 (2) (1960) 217–237. [arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpa.3160130205](https://arxiv.org/abs/https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpa.3160130205), doi:10.1002/cpa.3160130205.  
URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpa.3160130205>
- 590 [4] J. Donea, A Taylor-Galerkin method for convective transport problems, Int. J. Numer. Meth. Eng. 20 (1984) 101–111.
- [5] R. Löhner, K. Morgan, O. C. Zienkiewicz, *The solution of non-linear hyperbolic equation systems by the finite element method*, International Journal for Numerical Methods in Fluids 4 (11) (1984) 1043–1063. [arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/fld.1650041105](https://arxiv.org/abs/https://onlinelibrary.wiley.com/doi/pdf/10.1002/fld.1650041105), doi:10.1002/fld.1650041105.  
URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.1650041105>
- [6] S. K. Godunov, Eine Differenzenmethode für die Näherungsberechnung unstetiger Lösungen der hydrodynamischen Gleichungen., Mat. Sb., Nov. Ser. 47 (1959) 271–306.
- 595 [7] S. T. Zalesak, *The Design of Flux-Corrected Transport (FCT) Algorithms For Structured Grids*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 29–78. doi:10.1007/3-540-27206-2\_2.  
URL [https://doi.org/10.1007/3-540-27206-2\\_2](https://doi.org/10.1007/3-540-27206-2_2)
- [8] *Zoltan: Parallel Partitioning and Load Balancing*, <http://www.cs.sandia.gov/Zoltan>.
- [9] *ExodusII*: The Sandia Engineering Analysis Code Access System (SEACAS) is a suite of preprocessing, postprocessing, translation, and utility applications supporting finite element analysis software using the Exodus database file format, <https://github.com/gsjaardema/seacas>.
- [10] J. Waltz, *Derived data structure algorithms for unstructured finite element meshes*, Int. J. Numer. Meth. Eng. 54 (7) (2002) 945–963. doi:10.1002/nme.453.  
URL <http://dx.doi.org/10.1002/nme.453>
- 605 [11] J. Waltz, T. Canfield, N. Morgan, L. Risinger, J. Wohlbier, *Verification of a three-dimensional unstructured finite element method using analytic and manufactured solutions*, Computers & Fluids 81 (2013) 57 – 67. doi:<https://doi.org/10.1016/j.compfluid.2013.03.025>.  
URL <http://www.sciencedirect.com/science/article/pii/S0045793013001333>
- 610 [12] J. Waltz, T. Canfield, N. Morgan, L. Risinger, J. Wohlbier, *Manufactured solutions for the three-dimensional Euler equations with relevance to Inertial Confinement Fusion*, J. Comp. Phys. 267 (2014) 196 – 209. doi:<https://doi.org/10.1016/j.jcp.2014.02.040>.  
URL <http://www.sciencedirect.com/science/article/pii/S0021999114001661>
- [13] J. Kamm, J. Brock, S. Brandon, D. Cotrell, B. Johnson, P. Knupp, W. Rider, T. Trucano, V. Weirs, Enhanced verification test suite for physics simulation codes, Tech. Rep. LA-14379 ; SAND2008-7813, Los Alamos National Laboratory (2008).
- 615 [14] J. Waltz, Personal communication (2013).
- [15] L. Sedov, *Similarity and Dimensional Methods in Mechanics, Tenth Edition*, Taylor & Francis, 1993.  
URL <https://books.google.com/books?id=xXSg388Su38C>