ParFUM manuscript No.
(will be inserted by the editor)

ParFUM: A Parallel Framework for Unstructured Meshes for Scalable Dynamic Physics Applications

Orion S. Lawlor[1], Sayantan Chakravorty[2], Terry L. Wilmarth[2], Nilesh Choudhury[2], Isaac Dooley[2], Gengbin Zheng[2] and Laxmikant V. Kalé[2]

[1] *Department of Computer Science*
University of Alaska at Fairbanks
`olawlor@acm.org`

[2] *Department of Computer Science*
University of Illinois at Urbana-Champaign
`{schkrvrt, wilmarth, nchoudh2, idooley2, gzheng, kale}@cs.uiuc.edu`

The date of receipt and acceptance will be inserted by the editor

Abstract Unstructured meshes are used in many engineering applications with irregular domains, from elastic deformation problems to crack propagation to fluid flow. Because of their complexity and dynamic behavior, the development of scalable parallel software for these applications is challenging. The CHARM++ Parallel Framework for Unstructured Meshes allows one to write parallel programs that operate on unstructured meshes with only minimal knowledge of parallel computing, while making it possible to achieve excellent scalability even for complex applications. CHARM++'s message-driven model enables computation/communication overlap, while its run-time load balancing capabilities make it possible to react to the changes in computational load that occur in dynamic physics applications. The framework is highly flexible and has been enhanced with numerous capabilities for the manipulation of unstructured meshes, such as parallel mesh adaptivity and collision detection.[1]

1 Introduction

For the past forty years, a significant fraction of all computing cycles have been spent solving discretized differential equations on grids. Mechanical engineers use the stress-strain relationship to simulate the static and dynamic loading, impact and failure response of buildings, cars, airplanes, rockets and every other human artifact. Electrical engineers use Maxwell's equations to simulate electric and magnetic fields in silicon chips, motors, light bulbs, radio antennae and space satellites. Oceanographers use shallow-water wave equations to simulate the phases and amplitudes of tides and tsunamis in the world's oceans. Theoretical astronomers use Einstein's general relativity to simulate space-time gravity waves emanating from merging neutron stars.

Each of these applications demands considerable computing power and hence necessitates parallel computation. Each has its own dynamic or idiosyncratic performance aspects and hence is difficult to tune and load balance. Each must be developed by application scientists, not computer scientists. But each hour spent developing these applications is an hour the scientist cannot spend experimenting, analyzing and understanding—in short, time spent writing code is valuable time lost.

The UIUC Parallel Programming Laboratory's overarching goal is to reduce the amount of time application scientists spend writing high-performance parallel software. To do this, we have developed a foundation of software infrastructure called CHARM++[1]. CHARM++ consists of a variety of broadly applicable high-performance tools integrated in a single run-time system. Virtualization techniques are employed for hiding latency via message-driven execution[2], automatic application-independent load balancing[3], automatic communication optimization[4], check-pointing[5], fault tolerance[6, 7], and performance visualization and analysis[8]. All of these tools help make a parallel code run better, but even with CHARM++, developing a new parallel program still requires many hours of effort. By providing domain-specific support, in this case for solving problems on un-

[1] This work was supported in part by the National Science Foundation (DMR 0121695 and NGS 0103645) and the Department of Energy (B341494).

structured grids, we hope to make it easier for application scientists to develop new parallel programs.

A preliminary finite element framework[9] was developed as part of this effort. The framework effectively separated the parallelization process from the problem-domain modeling and numerics, enabling applications scientists to focus on the problem, and computer scientists to focus on the parallel implementation. The preliminary framework enabled automatic load-balancing which proved very useful, especially for complex and dynamic applications.

This paper details ParFUM, a Parallel Framework for Unstructured Meshes, that like the original CHARM++ FEM framework[9] allows unstructured-mesh applications to be easily parallelized. But ParFUM goes beyond the original framework with the following new functionality.

- ParFUM supports mesh ghost elements, multiple element types, detailed boundary conditions, parallel partitioning, and mesh adaptivity. Each of these will be described in detail in the following sections.
- ParFUM uses a library-like control structure, where the user code controls the outer loop (often the time or iteration loop) and makes occasional calls into the framework to perform communication. Previously, the framework ran the time loop and called user code at well-defined points, which was slightly more straightforward for explicit classical structural dynamics computations, but was much less flexible. Allowing the user to control the sequence of operations expands the scope of the framework to allow steady-state computations, implicit (matrix-based) solution methods, predictor-corrector timesteps, special start-up and shutdown phases, and all the other idiosyncrasies needed by complex applications.
- ParFUM now allows the user code to mix in arbitrary parallel communication calls in MPI with mesh computations. Both plain native MPI (for portability) and CHARM++ Adaptive MPI[10] (for load balancing, check-pointing, etc.) are supported. Previously, the framework performed all communication, and whatever communication primitives provided were inevitably insufficient for a few applications.
- ParFUM supports user programs written in C or FORTRAN 90, in addition to C++. Users in the scientific community found the older framework's heavily templated C++ code confusing, and a poor fit to their existing dense-array-based numerical codes.

2 CHARM++ and AMPI

ParFUM is built upon the CHARM++[11] infrastructure for parallel programming. It therefore inherits support for capabilities such as dynamic load balancing[12], automatic check-pointing, communication optimization and processor virtualization[13]. CHARM++'s highly portable nature also means ParFUM can run on a wide variety of computer architectures and operating systems.

2.1 Processor Virtualization

Processor virtualization[13] is the core idea behind CHARM++. The programmer partitions her computation into a large number of objects, or *virtual processors*, without concern about the number of physical processors available. The user views the application in terms of these virtual processors(VPs) and their interactions. The CHARM++ run-time system allows the VPs to interact amongst each other through asynchronous method invocation. Figure 1 shows the user's view of an application and one way that the CHARM++ run-time system might map the application to physical processors.

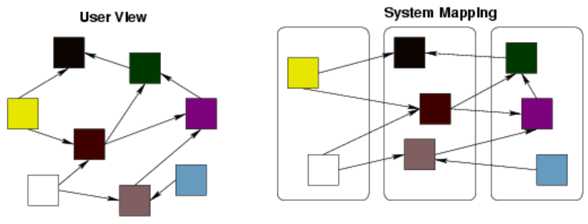


Fig. 1 Virtualization in CHARM++

The CHARM++ run-time system maps virtual processors to physical processors, allowing user code to be written without knowledge of the location of virtual processors. This gives the run-time system the ability to change the mapping of virtual processors to physical processors in the middle of an execution. It is capable of migrating virtual processors between physical processors at run-time. The CHARM++ run-time system allows for message delivery and collective operations such as reductions and broadcasts in the presence of these migrations. The run-time system can migrate objects during execution to adapt to the changing load characteristics of an application. Thus processor virtualization enables us to perform measurement-based run-time load balancing. Distributed and centralized load balancers can be developed to remap virtual processors to physical processors. In the virtualization context, load balancing has been used to scale molecular dynamics simulations to thousands of processors[8]. We discuss load balancing further in Section 2.4.

Another major benefit of processor virtualization is the automatic adaptive overlap between computation and communication. If one VP on a processor is waiting for a message, another VP can run on the same physical processor in the interim.

2.2 Adaptive MPI

Adaptive MPI (AMPI)[10] is an implementation of the Message Passing Interface (MPI)[14,15] in CHARM++.

Each MPI process is a user-level thread bound to a CHARM++ virtual processor. The MPI communication primitives are implemented as communication between the CHARM++ objects associated with each MPI process. Traditional MPI codes can be used with AMPI with slight modification, making it possible for these codes to take advantage of automatic load balancing and adaptive overlap of communication and computation.

2.3 Multiphase Shared Arrays

Another feature of CHARM++ used in ParFUM is the Multiphase Shared Array (MSA)[16]. MSA is used extensively in the parallel partitioning (Section 3.4) component of ParFUM. MSA is a distributed shared memory model in which data is accessed in phases. In each phase all the data elements in a particular array are accessed in the same mode by all participant threads. The mode of an array can be changed between phases. The different modes supported by MSA are read-only, write and accumulate. In the *read-only* mode participants can read as many elements as they want. The *write* mode allows only one participant to write to any particular data element. In the *accumulate* mode, a commutative-associative operation is used to accumulate data contributed by different participants to a single element. In the accumulate mode the final result at a data element depends only on the values accumulated and not the sequence in which they are accumulated.

MSA data elements can be load balanced at run time on the basis of computation and communication load so that data elements move to the processors where most of their accesses originate. MSA can also be used to read in a huge amount of data on one processor but store it simply and efficiently on several processors.

2.4 Load Balancing

Many ParFUM applications involve simulations with dynamic geometry, and use adaptive techniques to solve highly irregular problems. In these applications, load balancing is one of the key factors for achieving high performance on large parallel machines. Load balancing is especially useful for applications with refinement where the amount of computation on a particular chunk can change significantly as the number of elements comprising the chunk varies. It is also useful in applications where the computational load for subsets of elements varies over the duration of the simulation.

Built on top of the CHARM++ load balancing framework[17] and AMPI, ParFUM supports automatic measurement based dynamic load balancing by migrating a chunk along with its AMPI thread. During the execution of a ParFUM application, the load balancing framework collects workload information and object-communication pattern on each processor. The load balancing decision module uses this information to redistribute the workload, migrating the chunks from overloaded processors to underloaded ones. This approach relies on a principle of persistence[13] that holds for most physical simulations: computational load and communication structure of (even dynamic) applications tends to persist over time.

We provide a rich set of load balancing strategies and also allow users to implement their own. The load balancing problem is a multi-dimensional optimization problem, as it involves minimizing both the communication times and load-imbalances. Since it is an NP-hard problem, producing an optimal solution is not feasible. However, we have developed a rich set of heuristic strategies such as:

- Greedy Strategy: This simple strategy organizes all the objects in decreasing order of their computation times. The algorithm repeatedly selects the heaviest un-assigned object, and assigns it to the least loaded processor. This algorithm may lead to a large number of migrations. However, this simple strategy works effectively in most cases.
- Refinement Strategy: The refinement strategy is an algorithm which improves the load balance by incrementally adjusting the existing object distribution, especially on highly loaded processors. The computational cost of this algorithm is low because only some processors are examined. Further, this algorithm results in only a few objects being migrated, which makes it suitable for fine-tuning the load balance.
- Metis-based Strategy: This strategy uses the METIS graph partitioning library[18] to partition the object-communication graph. The objective of this strategy is to find a reasonable load balance, while minimizing the communication among processors.

3 ParFUM

This section describes the basic infrastructure of ParFUM. We lay the groundwork for ParFUM by first describing the terminology used throughout ParFUM and this article, describing how communication works, how ghost layers assist in maintaining an up-to-date mesh and finally how an initial serial mesh is partitioned.

3.1 ParFUM Concepts and Terminology

The terminology used by ParFUM is as follows.

- *Domain*: The space in which the user is trying to solve a problem. For example, when simulating cracks in a motor housing, the domain is the housing. Reality is 3-d plus time, but problems are often solved in 2-d for simplicity and efficiency, and occasionally time itself is considered just another axis of a static

4-d solution domain. Real systems are also entirely unbounded, but problems are almost always solved inside a bounded or repeating domain.

- *Node*: An individual point in the problem domain. Nodes always have at least coordinate data associated with them, and often include various solution data as well.
- *Element*: A small piece of the problem domain, defined by the nodes surrounding it. For example, a 2-d simulation might use 3-node triangular or 4-node quadrilateral elements, while a 3-d simulation might use 4-node tetrahedra or 8-node hexahedra (bricks). A typical simulation has a few million elements.
- *Element Type*: A particular kind of element. ParFUM supports partitioning and communication for mixed meshes consisting of different element types. Different element types have proven useful for crack propagation, where special *cohesive* elements join the faces of adjacent tetrahedra, and for fluid flow simulations, where interior and boundary elements have different numerical requirements and hence different types. ParFUM allows any fixed number of nodes per element, and hence supports arbitrarily high order elements or unique application-defined element types.
- *Element Connectivity*: The list of nodes surrounding each element. This list is what actually defines the element. In ParFUM, element connectivity is simply stored as a large 2-d array of node numbers. Nodes and elements are normally numbered starting from 0 in C, while the first node or element is normally numbered 1 in FORTRAN. Internally the framework uses 0-based numbers, but the interface routines can convert to and from either numbering based on a symbolic constant passed by the user.
- *Solution Data*: Numbers that represent some step in the problem solution process. For example, a structural dynamics application might represent the deformation of a bending bridge by storing the deformation (or displacement) at each node of the mesh. In ParFUM, solution data is represented as an *attribute* of the corresponding *entity*, as described below.
- *Boundary Conditions*: Some representation of how the outside world influences the problem domain. Boundary conditions can be used to immobilize or force motion into parts of mechanical simulations, inject and drain fluid from the boundaries of fluid dynamics simulations, and even couple simulations that utilize different meshes and materials. For some applications, the processors along partition boundaries can be treated as a special sort of boundary condition. A particularly interesting type of boundary condition is a rotational or translational periodicity, for which ParFUM has some minimal support.
- *Mesh*: A group of nodes, elements, solution data, and boundary conditions that together represent a problem domain. In the field of mesh generation, by contrast, a mesh normally does not include solution data, and occasionally does not even include boundary conditions. A ParFUM mesh normally includes both solution and boundary data, in addition to communication data used to knit the mesh across processors. In ParFUM, a mesh is represented as an opaque mesh handle, and manipulated by making ParFUM function calls. ParFUM directly supports a variety of mesh operations, with single function calls to insert or extract the element connectivity, solution data and boundary conditions; read or write a mesh to a disk file; partition a mesh into pieces for parallel execution; reassemble a partitioned mesh into a single piece; or perform a deep copy of an entire mesh.
- *Chunk*: One partition of a mesh. Normally each MPI process has exactly one chunk of the mesh, although in AMPI, multiple MPI processes normally coexist on each processor.
- *Sparse Element*: An element used to represent boundary conditions. The difference between normal elements and sparse elements is in partitioning—ignoring ghosts, each normal element is placed on exactly one chunk/processor; but a separate copy of a sparse element is placed in every mesh chunk that contains the relevant nodes. For example, a 2-d computation might define a sparse element type consisting of 2-node lines that lie around the exterior boundary of the domain, and represent the exact boundary condition there using an integer attribute.
- *Entity*: A generic term for anything in a mesh that can hold data: a node, an element, or a sparse element.
- *Attribute*: An array of solution data associated with an entity. For example, a structural dynamics simulation might keep a displacement for each node. To store this displacement in ParFUM, a dense array of displacements for each node could be registered as an attribute of the node entity.
- *Shape Function*: Stores the weight of a node's solution value across space. Solution data, e.g. stored at the nodes, normally bleeds out into the immediately surrounding elements according to a weighted average of the surrounding nodes. Typical shape functions are constant, for nearest-neighbor look-up; first-order, for linear value interpolation; second-order, for quadratic interpolation; and so on. Though critically important to a problem's numerics, shape functions have no bearing on mesh partitioning or parallel communication, and are hence ignored in plain ParFUM. Because most of ParFUM's functionality does not need or use shape functions, it immediately works with arbitrary novel or unique element types.
- *Numerics*: A catch-all term for "the things ParFUM doesn't help with". Specifically, ParFUM ignores time-stepping (e.g., Euler, Runge-Kutta), solution iterations (e.g., conjugate gradient, BiCG), element types (e.g., first-order linear tetrahedra), and most importantly element equations/physics (e.g., linear

elastic or nonlinear viscoplastic deformation for mechanics, DNS or RaNS for fluid dynamics, etc.). The things ParFUM does help with are listed in detail in the next subsection.

A ParFUM application looks very much like a typical MPI application. At start-up, the mesh is read in. Users have their choice of reading their existing serial mesh on processor 0 and partitioning on the fly, or partitioning the mesh beforehand and reading the mesh partitions in parallel in ParFUM format. Once the mesh is set up, the program executes some sort of solution loop over time or iterations (or other work) for the local chunk of the mesh, occasionally communicating with other chunks as described below.

3.2 ParFUM Communication

The most important feature to understand when considering ParFUM for a new application is the application's parallel communication needs. All communication in ParFUM is performed on a partitioned mesh, although the type of communication needed affects the mesh partitioning. ParFUM supports two types of communication: ghosts and shared nodes.

Most applications perform some sort of calculation in which each node or element requires solution data from its neighboring elements. This means that elements on the boundary of a chunk require data from elements on another chunk. ParFUM can solve this problem by adding *ghosts* to each boundary of a chunk. A ghost entity (element or node) is a local, read-only copy of a *real* entity, a non-ghost, that exists on another chunk. ParFUM provides a single collective call to update the read-only ghost nodes or elements with the actual value stored in the original. For many applications, this allows nodes or elements at a chunk boundary to seamlessly access data on another chunk. The exact definition of "neighboring element" for ghosts varies from application to application, but ParFUM supports a variety of adjacencies as described in Section 3.3.

For example, a time-dependent finite element structural dynamics program with explicit time-stepping might perform a calculation for each timestep as in Figure 2.

```
1. Zero the net force at each node
2. for each element
3.    Compute stress from node deformations
4.    Compute node forces due to element stress
5.    Add element force to surrounding nodes
6. for each node
7.    Use node's net force to deform node
8. Apply node-based boundary conditions
```

Fig. 2 Pseudocode for a typical serial structural dynamics program

An element computes its state based on the surrounding nodes' deformations in step 3 and contributes forces to surrounding nodes in step 5. Then the nodes are deformed (displaced) based on their total force in step 7. In a serial program, these loops run over the entire mesh.

In a mesh partitioned for parallel execution by ParFUM, separate copies of the nodes on the partition boundary are created. In the simple 1-d mesh of Figure 3, the original node e becomes $e1$ on chunk 1 and $e2$ on chunk 2. The problem with boundary nodes is that they only receive forces from the elements on that processor; so node $e1$ will only receive forces from element M, and $e2$ will only receive forces from element L. Thus in step 7, the nodes $e1$ and $e2$ will move incorrectly, since they have an incomplete net force.

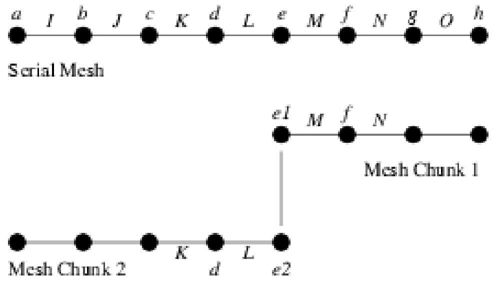


Fig. 3 A 1-d mesh with 8 nodes and 7 elements, partitioned into two chunks. Note that boundary node e is duplicated into copies $e1$ and $e2$.

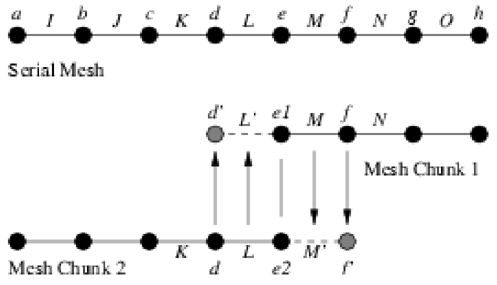


Fig. 4 The same mesh, with ghost elements L' and M' and ghost nodes d' and f'.

To parallelize this application using ghosts, we set up a ghost layer containing ghost elements adjacent to each of our real nodes, as shown in Figure 4. We can then compute the element stresses for all real elements, send those stresses to the ghost copies of the elements (in this case, L' and M'), and then use those stresses to compute the net force on all our real nodes, as shown in Figure 5. The shared nodes $e1$ and $e2$ will have the same net force, because $e1 = L' + M = L + M' = e2$.[2] Note that the new ghost nodes d' and f' have an incomplete set of

[2] Caveat: Since floating-point arithmetic is not associative, shared nodes with more than two neighbors may not receive exactly the same value up to round-off if evaluated in different orders.

element neighbors, and hence will receive an incomplete net force, but we only care about real nodes, and in fact in this case the ghost nodes can be omitted entirely. This transformation assumes only the fact that elements first read a value from their neighboring nodes (in this case, deformation in step 3), compute a temporary variable (stress) that can easily be sent across processors, then write to their neighboring nodes (in step 5)—nothing else, including linearity or separability of any operation, is assumed.

```
1. Zero the net force at each node
2. for each real element
3.    Compute stress from node deformations
3a.Communicate element stresses to ghost elements
3b.for each real or ghost element
4.    Compute node forces due to element stress
5.    Add element force to surrounding nodes
6. for each real node
7.    Use node's net force to deform node
8. Apply node-based boundary conditions
```

Fig. 5 Parallelized program that exchanges ghost element stresses. Ghost nodes are not needed.

Another way to parallelize the same program is to begin by exchanging ghost node deformation vectors. Then all elements, real and ghost, have the inputs they need to compute a valid stress and net force on their neighbors. The complete listing is shown in Figure 6. Note that now all the computations at each ghost element are duplicated, and we now need ghost nodes. However, it may be less expensive or less invasive to send node deformations rather than element stresses. This transformation relies only the fact that all the node deformations are read (in step 3) before they are modified (in step 7), and is hence the most general parallelization approach.

```
1. Zero the net force at each node
1a.Communicate node deformations to ghost nodes
2. for each real or ghost element
3.    Compute stress from node deformations
4.    Compute node forces due to element stress
5.    Add element force to surrounding nodes
6. for each real node
7.    Use node's net force to deform node
8. Apply node-based boundary conditions
```

Fig. 6 Program parallelized using ghost nodes.

The final way to parallelize this program is to note that each element's step 5 computes a partial force, but the forces are simply added together—that is, the effect of each element combines at the nodes in a linear fashion. Anytime this is the case, we can do away with ghosts entirely, and simply compute a partial force sum on the shared boundary nodes, then sum up the total force across processors—in Figure 3, $e1$ gets the force from M, $e2$ gets the force from L, and their sum $L + M$ is the correct net force on both nodes. ParFUM provides direct single-call support for this linear shared-node summation operation. This approach shown in Figure 7 is the simplest, as no ghosts are involved, but only works with a linear operation at the force-combining phase in step 5. Neither the element response in steps 3 and 4 nor the node response in step 7 need be linear, and forces always combine linearly, and hence summation over shared nodes is common for structural mechanics. But other sorts of interactions, such as the inter-element fluxes in fluid dynamics, often combine non-linearly (e.g., with flux-limiter methods) so ghost nodes and elements are often used in other fields.

```
1. Zero the net force at each node
2. for each real element
3.    Compute stress from node deformations
4.    Compute node forces due to element stress
5.    Add element force to surrounding nodes
5a.Sum node forces across chunks (for shared nodes)
6. for each real node
7.    Use node's net force to deform node
8. Apply node-based boundary conditions
```

Fig. 7 Program parallelized using a sum over shared nodes.

In many matrix-based linear or linearized computations, one matrix-vector product is computed using steps similar to 1-5, and can be parallelized using any of these same three techniques. Initially, a vector of data, here deformations, is distributed across the nodes. Each element can be seen as a small "stiffness matrix" that converts its node displacements into a set of node forces. Together, all the elements can be thought of as a global stiffness matrix that convert a vector of node displacements into a vector of node forces. In parallel, the entire input and output node-data vectors do not exist on any processor, but instead each processor owns its corresponding piece of each vector as determined during mesh partitioning. The shared node parallelization approach of Figure 7 in effect divides up the columns of the stiffness matrix—each processor uses its entries of the input vector to compute a partial result for some of the rows of the output vector, and then sums its partial results with those of other processors to obtain the correct output values for its entries of the output vector. The ghost node parallelization approach of Figure 6 divides up the rows of the stiffness matrix—each processor first obtains all the input vector entries (including both owned entries and ghosts) it needs, then independently computes its output vector entries. The ghost element parallelization approach of Figure 5 factorizes the "force from deformation" stiffness matrix into two non-square "node force from element stress" and "element stress from node deformation" matrices, both of which can be imagined to be divided along rows; between applying these two matrices, we need a ghost communication step to obtain stresses for our ghost elements.

In summary ParFUM's communication mechanisms and techniques for handling ghost layers apply well to matrix-vector product methods and hence most matrix-based applications, including those using linear or nonlinear iterative matrix solvers (e.g., conjugate gradient, BiCG) that are built on matrix-vector products and vector operations. ParFUM's methodology encourages such mesh-free implicit representations because there is no need to explicitly store any of these matrices—the "stiffness matrix" representation can just be a list of elements in the mesh and a loop, and no dense or sparse representation is ever needed or assumed by ParFUM.

As in these examples, there are often several choices for how to parallelize a given program. For some problems, the memory overhead of ghost nodes and elements is significant. But the choice is rarely particularly critical—ParFUM directly supports communication over shared nodes, ghost nodes, and ghost elements equally well; the number of messages in all cases is exactly equal to the number of neighboring mesh chunks; and the amount of data exchanged is often approximately equal as well. Hence users can parallelize their program in whatever way fits best with their existing code.

3.3 Ghost Layers

As shown in Section 3.2, ParFUM adds ghost elements and nodes to allow the elements along a chunk boundary to access data from neighboring elements on another chunk in a seamless manner. ParFUM allows the user to specify what *neighboring* exactly means for a particular calculation. In a particular application two tetrahedra that share an edge might be considered neighbors, whereas in another case only tetrahedra that share faces might be considered neighbors. The user can also have multiple layers of ghosts for applications that need neighbors of neighbors. The user describes the neighboring relationship by specifying a *face*. An element will be added as a ghost to your chunk if it shares a face with at least one of your elements. The user specifies the number of nodes that make up a face, the number of faces per element and the list of faces in an element. The user might choose to create ghost nodes for nodes of ghost elements that do not already exist on a chunk.

Figure 8 shows a mesh broken up into two chunks. Figure 9 shows each chunk with ghosts added for boundary elements that share an edge. Since the shared face in this case is an edge, each face has 2 nodes and each element has three faces. Figure 10 shows the same chunks with ghosts added for boundary elements that share a node. In this case the shared face is a single node and each element has 3 faces that they might share. As expected, the number of ghost elements is significantly higher when the shared face is a node instead of an edge.

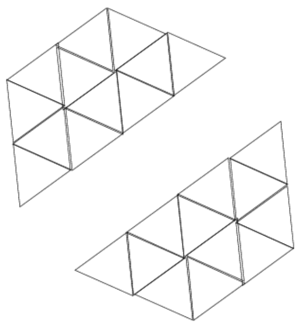
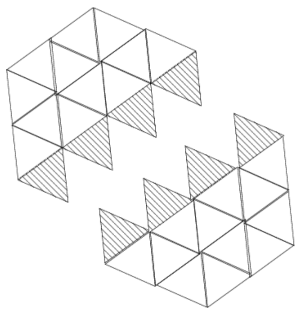
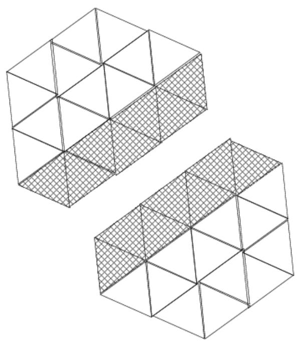


Fig. 8 A mesh with two chunks to which we shall add ghosts



```
/* 2 nodes per face: edge adjacency.
Add ghost nodes as well*/
FEM_Add_ghost_layer(2,1);
/*The faces(edges) in an element */
const static int tri2edge[]={0,1, 1,2, 2,0};
/*Triangles are surrounded by 3 edges */
FEM_Add_ghost_elem(0,3,tri2edge);
```

Fig. 9 The mesh in Figure 8 with ghosts added for boundary elements that share edges. The shaded elements are the ghosts. It also has the code segment for specifying this layer of ghosts



```
/* 1 node per face: node adjacency.*/
FEM_Add_ghost_layer(1,1);
/*The faces(nodes) in an element */
const static int tri2edge[]={0,1,2};
/*Triangles are surrounded by 3 edges */
FEM_Add_ghost_elem(0,3,tri2edge);
```

Fig. 10 The mesh in Figure 8 with ghosts added for boundary elements that share nodes. The shaded elements are the ghosts

ParFUM allows the user to create arbitrary ghost layers by adding multiple layers of ghosts. For example a user might add one layer of edge-based ghosts and another layer of node-based ghosts around that. ParFUM can create ghosts not only of things that are on other processors, but also for various problem symmetries, like mirror reflection, and various types of periodicities. The interface for these ghosts is simple—you ask for the symmetries to be created, then you will get extra ghosts along each symmetry boundary. The symmetry ghosts are updated properly during any communication, even if the symmetry ghosts are ghosts of real local elements from the same chunk.

ParFUM can generate ghosts in serial as well as parallel. The parallel ghost generation is extremely useful for the case where a large mesh has been partitioned in parallel as described in Section 3.4. We do not want to bring all the partitioned chunks back to one processor to create the ghosts. This consumes too much memory on processor 0 and fails for large meshes. Parallel ghost generation helps us avoid this bottleneck and lets ParFUM solve large meshes for problems that need ghosts.

3.4 ParFUM Mesh Partitioning

ParFUM can partition an application's serial mesh into a large number of smaller parts or *chunks*. During partitioning nodes and elements are given new local numbers. Each element is assigned to one particular chunk. Nodes shared between elements all on a particular chunk belong to that chunk. Nodes that are shared between elements on different chunks are duplicated in those chunks. We refer to such nodes as *shared nodes*.

Figure 11 shows an example of an original mesh with 5 nodes and 3 elements. The connectivity of the elements in the original mesh is shown in Table 1. ParFUM partitions that mesh into two chunks A and B in Figure 12. in a separate MPI process. ParFUM provides each chunk with the connectivity and user data associated with its elements and nodes. The connectivity of the elements of the two chunks is shown in Table 2.

ParFUM also sets up a communication mapping between different copies of a shared node. Each chunk keeps lists of the local numbers of shared nodes and ghost nodes and elements to send and receive–these lists of indices are used by the communication routines of ParFUM. The lists are sorted such that if two chunks share a node (or ghost, etc.), that node's local number is stored in the same entry of each chunks' list—that is, corresponding entries in the communication lists refer to copies of the same shared node. The lists of shared nodes for the chunks A and B in Figure 12 are specified in Table 3.

ParFUM has both serial and parallel methods of partitioning the original mesh. The serial version reads in the entire input mesh on processor 0 and then partitions it into multiple chunks, using the graph partitioning tool METIS [19,20,21] to obtain a mapping from elements to chunks. Since partitioning is fairly memory intensive, it is not always possible to partition a big mesh into a large number of chunks using the serial partitioner. Moreover building all the chunks on processor 0 means that at some instant processor 0 not only stores the entire input mesh, but also the chunks for all the other processors.

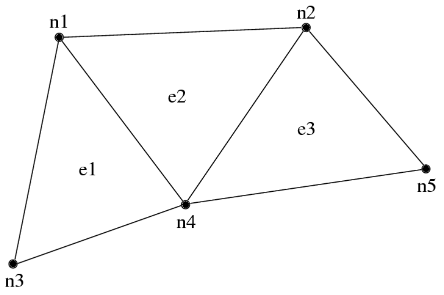


Fig. 11 A 2-d mesh with 3 triangular elements and 5 nodes

Element	Adjacent Nodes		
e1	n1	n3	n4
e2	n1	n2	n4
e3	n2	n4	n5

Table 1 Connectivity table for the mesh in Figure 11

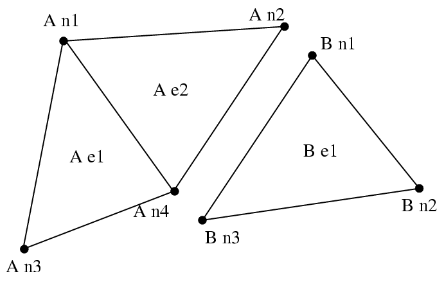


Fig. 12 Partition of the mesh in Figure 11 into 2 parts A and B

Connectivity data of Chunk A

Element	Adjacent Nodes		
e1	n1	n3	n4
e2	n1	n2	n4

Connectivity data of Chunk B

Element	Adjacent Nodes		
e1	n1	n2	n3

Table 2 Connectivity table for chunks A and B of the partitioned mesh in Figure 12

Shared Nodes on A	Shared Nodes on B
n2	n1
n4	n3

Table 3 Shared node lists for the chunks A and B of the partitioned mesh in Figure 12

This increases the memory consumption on the already loaded processor 0 even more.

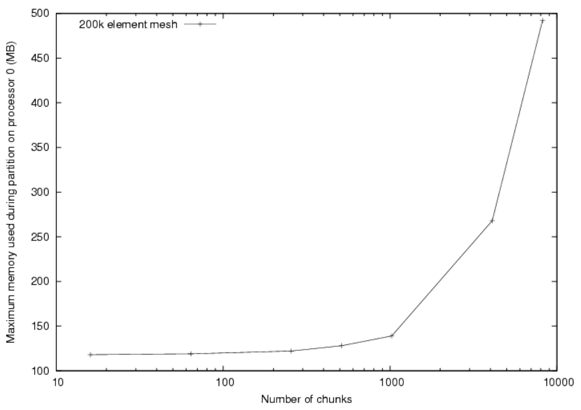


Fig. 13 Maximum memory used by the serial partitioner while partitioning a 200k element mesh into varying numbers of chunks

Figure 13 shows the memory consumption on processor 0 while dividing a mesh of 200 thousand elements into varying numbers of chunks. We can see that the memory required increases when the number of chunks increases. We could not even partition a large 2 million element mesh into 12000 pieces due to insufficient memory. Thus the serial partitioner suffers from a memory bottleneck on processor 0 and stops us from solving problems that we otherwise could have solved.

The parallel partitioner[22] eliminates the memory bottleneck on one processor. Though we use ParMETIS [23] to calculate the mapping from elements to chunks, we parallelize the process of building chunks: sending node and element data to the corresponding chunk, figuring out the shared nodes and setting up the communication lists for the shared nodes.

The parallel partitioning algorithm can be broken up into three parts.

1. Calculate a mapping of elements to chunk that tries to minimize the number of elements on the boundaries between different chunks
2. Use the mapping from the previous step to create chunks that contain all the data for the elements and nodes mapped to them.
3. Find the nodes that are shared between different chunks and set up communication lists.

In order to distribute the computation load more or less equally among the processors, ParMETIS should be called with the connectivity data for equal numbers of elements on each processor.

We do this by using a trivial strategy to initially break up the input mesh into chunks that contain the same number of elements and nodes. Each process is deemed to be *responsible* for the chunk that it gets during the trivial partition. After the ParMETIS call, each processor knows the mapping of each element that it is responsible for.

The next step in the partitioning algorithm requires us to find which nodes and elements belong to each chunk. Each processor knows the chunk to which each of the elements it is responsible for is mapped (henceforth referred to as the *owner* of that element). Each processor sends all the elements on it to their respective owners.

However sending the nodes to their owners is complicated by two factors: each node might be owned by multiple chunks and after the call to ParMETIS each processor knows the owner for the elements but not the nodes it is responsible for. As a first step to sending the nodes to their owners, each processor finds the owners of the nodes that are present in the connectivity of the elements it is responsible for. The owner of an element is one of the owners of all the nodes in the connectivity of that element. This data is then collected over all processors. From this each processor extracts the ownership information for all the nodes that it is responsible for. It then sends the node to all the chunks that own it.

Now each processor needs to set up the communication lists for shared nodes as mentioned earlier. For this it uses the global node ownership information calculated in the previous step. A chunk finds all the other chunks that also own a node owned by it. Then it finds all the nodes that a chunk shares with another chunk and uses this information to set up the communication lists.

A number of steps in the parallel partitioning require us to collect data scattered across multiple processors. As an example, after the call to ParMETIS all the elements owned by a particular processor are scattered across different processors and need to be brought to that processor. An operation like this seems well suited for a distributed shared memory model.

We used MSA [16] because it not only simplified the implementation, but also provided additional features such as collecting the results locally before sending them out to the destination processor. An MPI implementation would have been far more complicated.

We evaluate whether our implementation removes the bottleneck on processor 0. A 200 thousand element mesh is partitioned into 256 chunks on varying number of processors. Figure 14 shows the memory consumption on processor 0 for varying numbers of total physical processors. On a single physical processor the parallel partitioning algorithm consumes nearly 4 times the serial partition algorithm on a single processor. However the memory consumption on processor 0 decreases with increasing number of physical processors. The break even point for this data set is around 3 processors, after which the parallel implementation consumes less memory on a given processor than the sequential one would. This means that the parallel partitioning algorithm should be used with this mesh for runs on more than 3 processors. The memory consumption flattens out for higher number of processors as the decrease due to fewer chunks per processor is offset by more replication in MSA because of the higher number of physical processors.

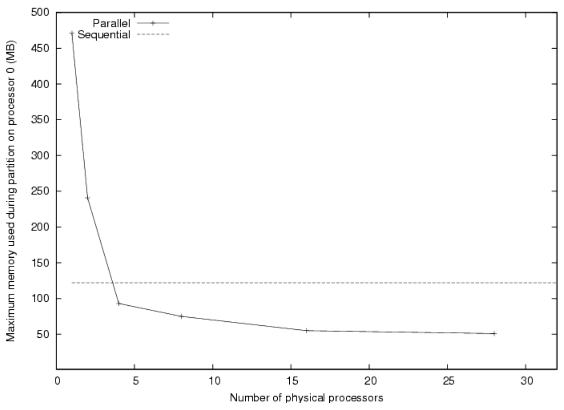


Fig. 14 Maximum memory used on processor 0 while partitioning a 200k element mesh into 256 chunks in parallel on varying numbers of physical processors

A major problem with the serial partitioning algorithm was that memory consumption on it increased as the same mesh was broken into increasing number of chunks. So ParFUM's serial partitioner would limit the scalability to large problems. This defeated the primary purpose of the framework, which is to allow applications to scale to large number of processors. We partitioned the same mesh as above on a different number of processors into as many chunks as there are physical processors. Figure 15 shows that memory consumption for both the serial and parallel partitioners. As expected, the memory used decreases with increasing number of processors in the parallel case, whereas it increases in the serial case.

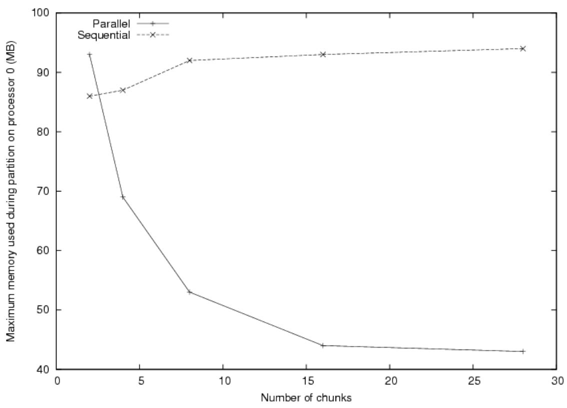


Fig. 15 Maximum memory used on processor 0 while partitioning a 200k element mesh into different number of chunks. The number of physical processors and chunks is the same

The parallel partition algorithm relieves the memory bottleneck at processor 0. Although the parallel partitioning algorithm consumes more memory than the sequential one when used on one processor, its memory consumption falls below the sequential partitioner when a sufficient number of processors are used. Thus runs with large meshes, which would have been impossible to partition with the sequential partitioner, become possible to partition with the parallel partitioner.

4 Extensions to the Framework

The basic structure and functionality of ParFUM described above already allows for the development of a wide variety of applications. However, certain applications will benefit from additional extensions to the basic functionality. These extensions include methods for generating and accessing additional topological information for the mesh, as well as various mesh modification and adaptivity functions. This extended functionality is described in the following subsections. None of these extensions existed in the CHARM++ FEM framework, the predecessor to ParFUM.

4.1 Topological Entity Relationships

By default ParFUM only stores *element-to-node* connectivity information. The connectivity tables list the nodes comprising each element in the mesh. In many applications it is desirable to access other topological information directly. For example, element centered data might be used to compute a force vector which will be applied to a node. Thus, each node would need to sum the force vector contributions from all of the adjacent elements. To get these contributions from the adjacent elements, an efficient way to determine *node-to-element* adjacencies is needed. Similarly useful are the *node-to-node* and *element-to-element* adjacencies. Thus it is desirable for the application to have easy and efficient access to these mesh adjacency information.

ParFUM provides these topological adjacency relationships by deriving them from the *element-to-node* connectivity data structure which defines the mesh itself. The *node-to-element* adjacencies are the set of all elements a node is shared with. The *node-to-node* adjacencies are computed such that two nodes are adjacent if and only if their *node-to-element* adjacencies have at least one element in common. Building the *element-to-element* adjacencies is more complicated than the others. It uses the same approach to define what a *neighboring* element is as is used for ghost generation in Section 3.3. The user can define two elements to be neighboring if they share a node, an edge, or face.

Adjacency information refers to not only elements and nodes on the local chunk but also includes a ghost layer on the neighboring chunks. A remote entity that is adjacent to a local entity is identified by the index of its ghost copy on the local chunk. This allows users to access topological data seamlessly across processors.

The *element-to-node* connectivity data structure is stored as an attribute associated with each element type that exists in the mesh. It is a single, resizable, $m \times n$ 2-d array, where m is the number of elements and n is the number of nodes per element. The *element-to-element* adjacency table is similarly stored with the element types as an $m \times p$ array, where m is the number of elements and p is the number of tuples per element. The *node-to-element* and *node-to-node* adjacencies are not simple 2-d arrays as are the element-based adjacencies. Each node may be adjacent to any number of nodes or elements. Thus these nodal adjacencies are stored as jagged arrays, i.e. arrays of variable length vectors. The jagged array is an attribute associated with the nodes. Each vector in the array can be resized without needing to resize the entire array.

ParFUM provides two ways to access adjacency data. The first is by using the standard ParFUM functions which provide access to arbitrary attributes. The second way to access the adjacency information is to use ParFUM's adjacency accessor functions. One accessor function returns all elements or nodes adjacent to a given entity. Another can return just a single adjacent entity which is on a given edge or face of the given element. A final accessor can return which face of an element is shared with a given element. A higher level of iterators can also be easily created using these and other lower level functions in ParFUM. The application described in Section 5.1 uses its own adjacency iterators built upon these accessors.

ParFUM also provides methods to modify the adjacency information of elements and nodes. These are used mostly by the mesh modification extensions in ParFUM described in Section 4.2.

4.2 Modification of Parallel Meshes

Adaptivity of meshes is an important feature for a robust mesh simulation framework. Adaptivity can provide better numerical accuracy in simulations with minimal increases in computation time. Providing support for adaptivity of meshes in parallel is not as simple as it would be for a serial mesh on a single processor. Moreover, some applications need to perform incremental and unsynchronized modifications to their parallel meshes. One application that needs parallel mesh modification is the adaptive space-time meshing algorithm[24, 25], described in Section 5.1. In order to support different forms of mesh modification in ParFUM, we decided to create a simple but robust abstraction that would support a wide range of application needs. The modifications included in this ParFUM extension are not limited to subdivision of existing elements, as is the case in many meshing packages that support adaptivity. The mesh modification functions in a ParFUM application can be performed at any time, without any global synchronization.

ParFUM's mesh modification abstraction contains four mesh modification primitives: add an element, remove an element, add a node, and remove a node. Each primitive performs a simple operation, but maintains the consistency of the mesh across all processors. Maintaining this consistency is non-trivial, and it includes modifying ghost layers and changing adjacency information. A consistent mesh must have a single ghost layer in which elements are considered to be neighbors if they share a single node like in the example shown in Figure 10 in Section 3.3. We hope to relax this requirement to allow different types of ghost layers in the future. The mesh must contain no hanging nodes, if the elements around the hanging node are to be considered adjacent. The abstraction does, however, allow creation of *element-to-element* adjacencies in a flexible user-prescribed manner. A user may not remove any node which is still adjacent to one or more elements, nor create an element with non-existent nodes. A final assumption for our abstraction is that all adjacency tables are consistent and accurate. For example, it should never be the case that two nodes are adjacent in some table if they are both not adjacent to a common element.

Adding and removing nodes are simpler operations than the other primitives. However, these operations are still non-trivial since newly added nodes might be inserted along a boundary between chunks and are thus shared nodes. When adding a node, an optional list of nodes that are adjacent to the given node can be provided. This parameter is used to determine whether to create the new node as a local or shared node. If all nodes in the given list are shared with the same chunk, the new node will also be shared. This allows the insertion of a node *between* a set of nodes, such as on an edge or face of an existing element. Although the list is given, it is only used for determining whether the node should be shared. The nodal adjacencies for the new node will only be created when an element connected to that node is created.

Adding an element is one of the primitives which can create either a local element or create an element in the local ghost layer. The default behavior creates a new element locally and updates neighboring ghost layers if it is a ghost on some other chunk. Adding an element in the ghost layer is quite similar, except we have to compute the chunk where the actual element is to exist and communicate to that chunk that it should create the element. When the element is created on that chunk, ghosts are created and updated on the neighboring chunks. Supporting both operations allows for clarity of adaptivity algorithms that operate in regions along mesh chunk boundaries.

Although conceptually simple, the implementation in parallel is complicated. After the new element is added on any chunk, that chunk must update all of its adjacency tables. A further difficulty for the implementation of this primitive is maintaining the local to remote entity index mappings. Since multiple operations may occur on a single chunk simultaneously, care must be taken to ensure that no race conditions occur when updating these distributed data structures.

To remove an element, we first remove the local copy of the element, along with references to the element in adjacency tables. A message is then sent to all chunks which contain that element as a ghost. The remote chunks will delete their ghost copies of the element. After the element is removed, the ghost layers on the originating chunk and all chunks which had the element as a ghost must be updated. Updating the ghost layers is a complicated operation since elements may need to be removed from multiple chunks.

We show in Section 4.3 that our abstraction is powerful enough to easily build important mesh adaptivity functions.

4.3 Parallel Mesh Adaptivity

The ability to adapt unstructured meshes is important to dynamic physics simulations. Providing this functionality in ParFUM expands the capabilities of the framework to handle a wide variety of challenging simulations. This functionality proves its usefulness in several ways:

- Poor quality elements (thin triangles, "sliver" tetrahedra) can result in a loss of accuracy and numerical stability in physics simulations. Smoothing and local mesh repair operations can alleviate some of these problems.
- Solution accuracy is also affected by the granularity of the mesh in terms of element size. A finer mesh will capture a solution more accurately. However, maintaining a fine mesh is costly. Refinement and coarsening operations make it possible to refine a mesh where more activity is occurring and coarsen it in less dynamic regions.
- Simulations may exhibit dramatic behavior differences amongst the elements (see Section 5.3). This can lead to mesh partitions that take much more time to compute than others, resulting in load imbalance. While Charm++'s virtualization and load balancing abilities can take care of some of this, it may be necessary to do dynamic repartitioning to obtain a fully balanced state.
- Similarly, refinement and coarsening may result in mesh partitions with drastically differing numbers of elements, also causing load imbalances. It is conceivable that a partition could refine so much that its processor is overloaded even after all other partitions have been migrated away. Dynamic repartitioning is also beneficial in such situations.

Thus we see that what we mean by the ability to *adapt* a mesh consists of the ability to smooth, repair, refine and coarsen a mesh, as well as to adjust the partitioning of the mesh at run-time. Which of these methods applies is largely governed by the physics being simulated.

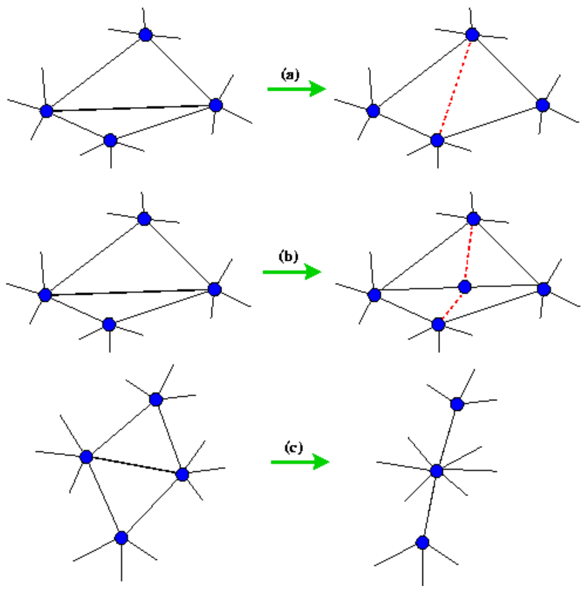


Fig. 16 Three primitive mesh adaptivity operations: (a) flip edge; (b) bisect edge; (c) contract edge

Given ParFUM's parallel mesh modification capabilities and adjacency information, we have implemented several primitive mesh repair, refinement and coarsening operations. These operations can be used on their own, or by user applications for special purpose mesh adaptivity operations. They are also used for refinement and coarsening algorithms provided to the application developer by the framework.

The basic 2-d operations provided are shown in Figure 16. A *flip_edge* operation modifies two elements that share an edge by removing and reinserting the edge between the two nodes of the elements that are opposite the original edge. A *bisect_edge* operation inserts a node on an edge and adds edges between the new node and the two nodes opposite the original edge. A *contract_edge* operation removes the one or two elements adjacent to an edge, contracting the two end nodes to a single node. In addition to these operations, we have a *remove_node* operation which acts as the inverse of *bisect_edge* by removing a degree-4 node and two adjacent elements. We also have a *split_node* operation that acts as the inverse of *contract_edge* by adding a new node and an edge between the old and new nodes along with two elements adjacent to the new edge.

In 3-d, our implementation includes Delaunay 2-3 and 3-2 flip operations. The 2-3 flip involves transforming two elements that share a face into three elements that share an edge. The 3-2 flip is the inverse operation of the 2-3 flip. We have a *bisect_edge* operation that bisects an edge, adding a node along the edge and splitting in two an arbitrary number of elements adjacent to that edge. Two additional refinement operations involve inserting a new node into the volume of a tetrahedra or onto a face. Inserting a node on a face with the *split_face* operation splits each of the two neighboring elements into three elements. The *split_element* operation splits the element into four tetrahedra with the insertion of a single node.

Figure 17 presents an example mesh of a 2-d rectangular bar, running on 4 processors. In an experiment, a long vertical strip along the center of the bar is refined for a couple of steps using refinement primitives. A smoothing algorithm is also applied in tandem with the refinement. Then we apply coarsening primitives to the lower 40% of the bar for a couple of steps. It demonstrates the capability to selectively refine and coarsen parts of any mesh as a function of time.

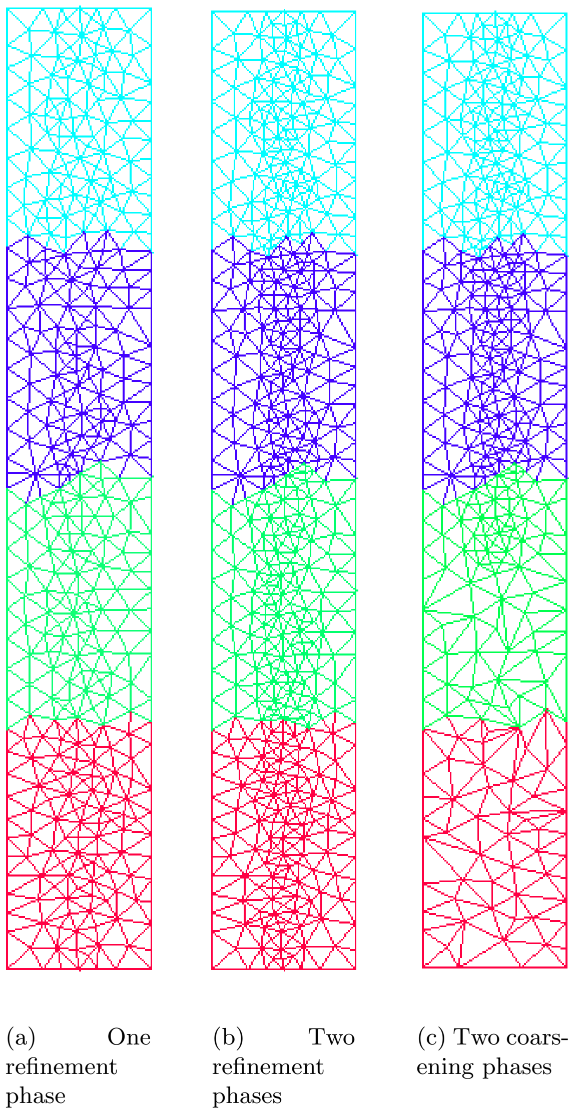


Fig. 17 A mesh representing a bar with a narrow portion in the middle refined for two steps and then the lower 40% is coarsened for two steps. The simulation was run on 4 processors.

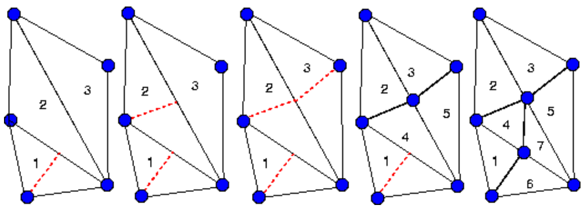


Fig. 18 Longest edge bisection in 2-d

4.3.1 Refinement and Coarsening in Parallel We have implemented a number of more elaborate refinement and coarsening algorithms for use within ParFUM. Algorithms for 2-d refinement, coarsening, repair and gradation of triangle meshes have been fully integrated into ParFUM using the new parallel mesh modification primitives. The 3-d implementations are under development. Our parallel implementations of refinement for 2-d and 3-d unstructured meshes are based on the longest edge bisection algorithms first introduced by Rivara[26]. This approach recursively applies the *bisect_edge* operation and propagates to satisfy the longest edge requirement. For example, the algorithm is typically initiated on an element's longest edge. The edge is bisected with the primitive operation if it is the longest edge of the adjacent neighbor element. If this does not hold, the longest edge bisection is applied to the neighbor element first. This process of propagation is illustrated in Figure 18. These higher-level refinement and coarsening algorithms make use of element quality criteria for decisions on which elements to adapt first, and can be applied to entire regions of the mesh with user-specified criteria.

4.3.2 Data Transfer for Mesh Modification Operations We have implemented a small module to perform basic default solution transfer behavior. The module provides for the linear interpolation of nodes during refinement, coarsening and smoothing operations. It also handles basic copying of element data for bisected elements. The module provides a clean interface by which the user can provide their own desired data transfer operations for use during local mesh modification.

ParFUM also has a parallel data transfer module for performing data transfer for both node- and cell-centered data over an entire mesh. This library makes use of collision detection (Section 4.4) and has been used to transfer solution data during remeshing in rocket simulations. We discuss this application of parallel data transfer further in Section 5.2.

4.4 Parallel Collision Detection

ParFUM and Charm++ include a parallel collision detection library [27,28] which provides an efficient means for determining intersections of mesh pieces or other objects in a 3-d space. Each processor contributes a set of bounding boxes to the library. After intersecting these

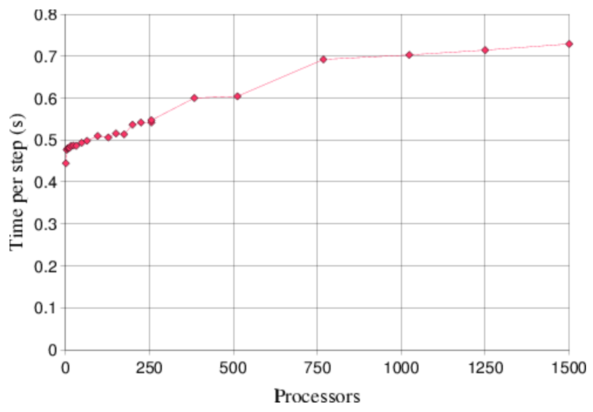


Fig. 19 Collision detection library performance with a fixed number of objects per processor.

sets of bounding boxes, the library returns to each processor a list of its local boxes that overlap with boxes from other processors. The user code can then appropriately handle any collisions.

Collision detection is currently used in real world applications. An obvious use of collision detection is in structural dynamics simulations. It is desired in such applications to determine if two physical objects, possibly deformable or rigid bodies, collide. If the physical objects collide, then some contact physics will be applied to keep the two objects from interpenetrating. A second less obvious but equally important use of collision detection is to transfer a solution from an old mesh to a new mesh during a remeshing operation. When a mesh is modified, data for a given element in the old mesh may be used to compute the element data for any element in the new mesh which overlaps the same physical space as the element in the old mesh. The process of determining whether two elements overlap is the same as detecting if the two elements collide. The use of this library for solution transfer is described in Section 5.2.

Collision detection can be time consuming and thus an efficient parallel method is needed. Furthermore, a single processor may not have enough memory to hold all bounding boxes from all processors, so it may be impossible to use any serial collision detection library on a large problem. The performance of the library depends upon the problem, but takes $O(n/p)$ (where n represents mesh size in number of elements and p represents the number of processors) time under reasonable assumptions for most problems. In a practical test, the library exhibits speedups of 915 on 1,500 processors, a parallel efficiency of 60% as displayed in Figure 19[28].

5 Applications of ParFUM

ParFUM and its extensions are used in a wide variety of applications. We discuss four examples that use a number of the features of the basic framework and the extensions. ParFUM greatly simplified the parallelization of all these applications and reduced the amount of effort involved on behalf of the application programmer.

5.1 Spacetime Discontinuous Galerkin

The Spacetime Discontinuous Galerkin (SDG) method [24,25] provides a powerful way to analyze phenomena such as shock wave propagation in solids, evolution equations for state variables in inelastic constitutive models and Hamilton-Jacobi level set models for interface kinetics.

The SDG method uses discrete basis functions in space and time over partitions of the spacetime domain. The current implementation of the SDG method employs unstructured spacetime meshes satisfying a special causality constraint. For such meshes, the SDG method can be implemented as an advancing front solution technique which interleaves the generation of a patch of a small number of elements and the solution procedure in the patch. The space domain to be analyzed is represented by an unstructured mesh referred to as the space mesh.

The SDG method has two main variants: adaptive and non-adaptive. In the adaptive version, the space mesh is refined or coarsened to obtain a more accurate solution at points of interest without paying a high cost all over the domain. The non-adaptive version does not change the space mesh.

The SDG algorithm is very amenable to parallelization since the computation-intensive solution procedure for a patch is independent of all other patches. One of the first attempts at parallelizing the SDG algorithm involved a master-slave design. The space mesh was stored on one master processor, which created the patches and handed them over to other slave processors to solve. However it was found that the master processor soon became a communication and computational bottleneck. It was realized that to scale to very large numbers of processors, the space mesh would have to be distributed among all the processors.

ParFUM seemed a suitable platform for this new implementation of parallel SDG. Instead of rewriting the whole application for the parallel version, it was decided to provide an interface that could be used by the existing serial advancing front code. All issues of locking, synchronization and data transfer are handled by this interface. The interface is built on top of ParFUM. Ghost layers described in Section 3.3 are used to provide a uniform method for accessing nodes and elements on local and remote chunks. Topological relationships between nodes and elements described in Section 4.1 are used by the advancing front code. ParFUM's capability for parallel mesh adaptivity described in Section 4.3 is very useful for the adaptive version. In the adaptive code chunks with refined elements would have more computation than those with coarser elements. Section 2.4

described how load balancing can be very useful in a situation like this.

Some additional CHARM++ code was developed to provide functionality specific to this application such as special-purpose locking. The implementation of the non-adaptive version of the parallel SDG has been completed whereas the adaptive version is under development. We evaluated the non-adaptive version on an elastodynamics code running on a 2D space mesh with only 140,000 elements. We tested it on the Turing[3] cluster. Figure 20 shows the number of patches that are solved in a second on varying number of processors. The number of patches is a measure of the amount of work completed. We see that even for a relatively small mesh it scales reasonably well till 256 processors. The performance does suffer a little for large numbers of processors. Load imbalance among different chunks is the primary reason for it. We intend to use the dynamic runtime load balancing features of CHARM++ in the future to improve the performance of small meshes on large numbers of processors.

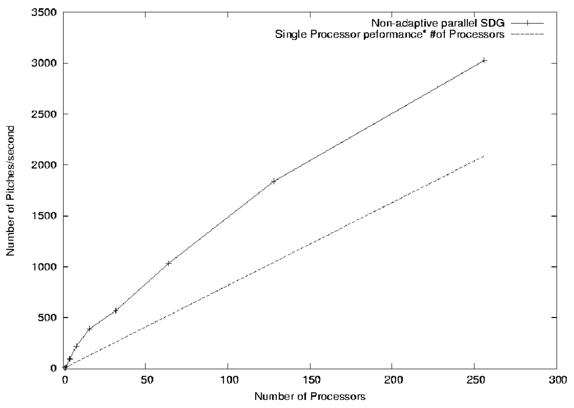


Fig. 20 Number of Patches solved per second for different numbers of processors

5.2 Parallel Solution Transfer

As a simulation progresses, deformation can cause the shape of a mesh to change, sometimes distorting elements severely. These low quality elements can result in numerical instability and a loss of accuracy in physics simulations. Smoothing and local mesh repair operations can provide some improvement, but sometimes a mesh is beyond repair. In this case, we may choose to create an entirely new mesh from a model of the existing mesh. This can be a complicated process because the old mesh will have data that must be transferred to the new mesh, but the elements and nodes of the old mesh will not correspond in any way to those of the new mesh.

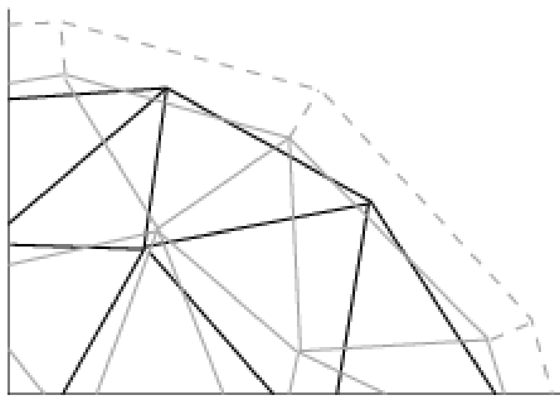


Fig. 21 Extruding boundaries of old mesh (grey) to cover new mesh (black)

In this section we describe *Rocrem*, a utility written for the remeshing of rocket meshes. Rocrem uses a special module of ParFUM to perform parallel data transfer from a deformed mesh consisting of both surface and volume components to the new surface and volume meshes via the following steps:

- The new mesh is read into ParFUM and partitioned into k chunks or virtual processors on n processors
- Each VP reads a partition of the old volume mesh and associated solution data
- The boundaries of the old mesh are extruded (see Figure 21) so that curved surfaces of the new mesh are guaranteed coverage
- Perform parallel data transfer:
  - Use ParFUM's collision detection library (discussed in Section 4.4) to match up old mesh to new mesh
  - Transfer data from old mesh components to overlapping components in the new mesh
  - Perform linear interpolation of node data
  - Perform volume-weighted average to transfer data to elements

We have tested the parallel data transfer utility on the Turing cluster with a small mesh containing approximately 200,000 elements and obtained the speedups shown in Figure 22. Due to the small size of the rocket, there are only sixteen partitions. Thus, the experiments in the figure were always run with sixteen virtual processors dispersed amongst the one through 16 physical processors used for the experiments. The single processor time was 429 seconds for this mesh. A considerably larger mesh of 3.8 million elements took 10 minutes for solution transfer on 32 physical processors.

5.3 Crack Propagation Simulation

Fractography2d and Fractography3d are dynamic crack propagation simulations that simulate pressure-driven

[3] *Turing* is a cluster of 640 Apple Xserves connected by Myrinet. Each node has dual 2 GHz G5 processors and 4 GB of RAM.

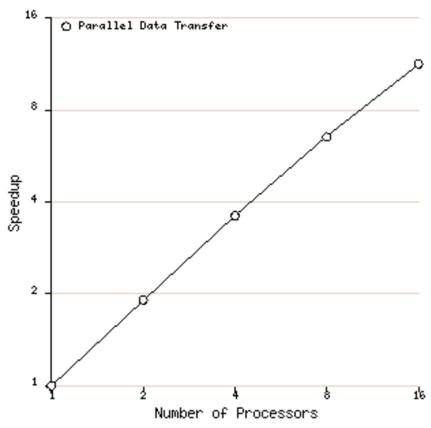
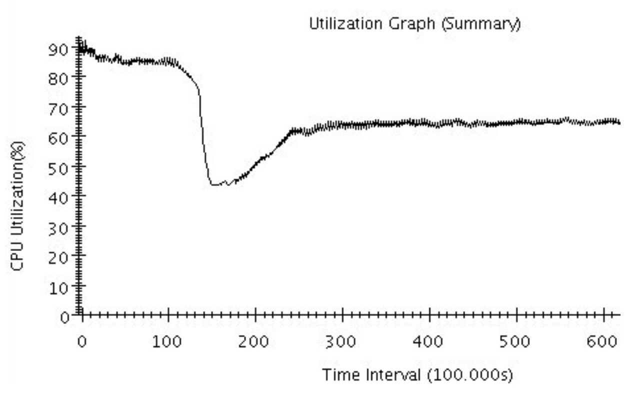
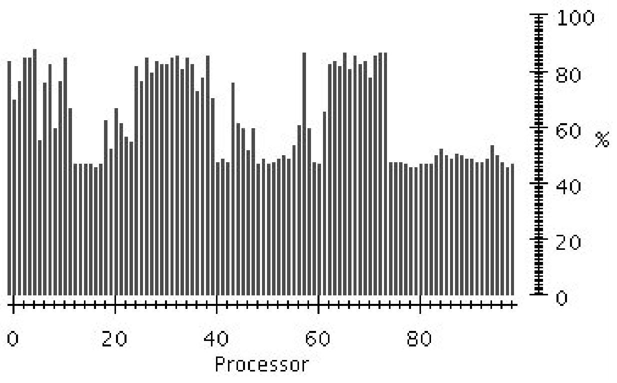


Fig. 22 Speedup of parallel data transfer for a small rocket mesh



(a) CPU utilization over time intervals



(b) CPU utilization across processors

Fig. 23 Performance of Fractography3d without load balancing. (a) Note the drop in CPU utilization at the transition to plasticity. (b) After the onset of plasticity, CPUs with only elastic elements are underloaded.

crack propagation in structures. These codes were developed by the research group of Dr. Philippe Geubelle in collaboration with the Parallel Programming Laboratory.

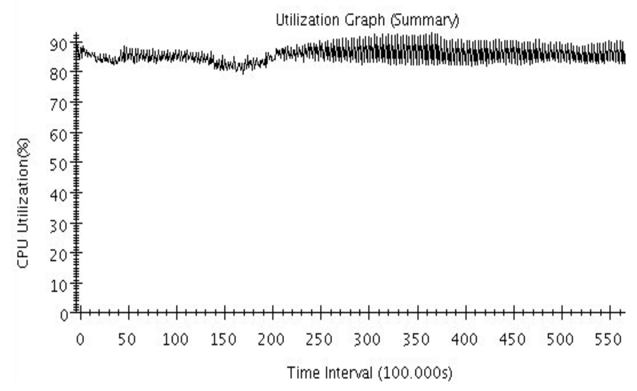
In this ParFUM application, the physical domain is discretized into tetrahedral elements. In each iteration, displacements are calculated on the nodes from forces contributed by surrounding elements. Typically, the number of elements is very large, and they are grouped in a number of chunks distributed across processors. Forces on boundary nodes across partitions are communicated across chunks, combined and new displacements calculated.

When an external force is applied to the material under study, the initially elastic response of the material may change to plastic as stress increases, resulting in much more expensive computation in that region. This in turn causes some of the mesh partitions to spend more time on computation per timestep than other partitions, resulting in load imbalance. ParFUM inherits a rich set of load balancers from CHARM++, the underlying runtime system, and these proved extremely useful in this situation [12].

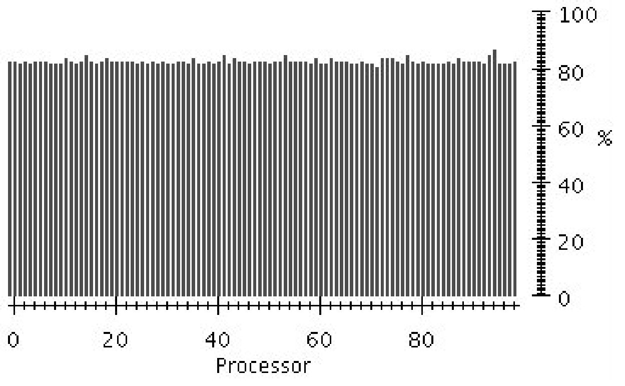
To demonstrate the use of load balancing, we first examined a synthetic problem with an elastic bar. As force is exerted from the front of the bar, the wave propagates to the back of the bar and bounces back. During this process, the elastic bar transforms into a plastic state along the wave of the force. This simulation was run on 32 processors of the SGI Altix at NCSA and used 160 mesh chunks. Without a load balancer, the duration of the simulation was 207 seconds. Using a simple synchronous greedy load balancing strategy, the duration of execution comes down to 198 seconds. A more sophisticated asynchronous load balancing further improved the simulation time to 187 seconds.

Given these promising results, we used a larger crack propagation simulation with 1000 AMPI processes on 100 processors of the Turing cluster. Without load balancing, the simulation took 24 hours, while using greedy load balancing helped finish the simulation in 18.5 hours. Figure 23(a) shows the CPU utilization graph versus time. Around time interval 120, the CPU utilization has already dropped to 42%. This is due to portions of the domain that have transformed to the plastic state, causing the load imbalance. The imbalanced workload on the processors can be seen in Figure 23(b). While some processors have a CPU utilization as high as 90%, some are lower than 50%. Figure 24(a) shows the same utilization graph with load balancing. The load has been effectively balanced and Figure 24(b) shows all processors having almost equal work.

Another collaboration with Dr. Geubelle's research group has prompted the ongoing development of mesh adaptivity in ParFUM. In solving a 1-d wave propagation problem, we wish to capture information at the shock front. One end of a 2-d bar is fixed and the other end is pulled with constant velocity. This results in a wave front that travels from pulled end. To best capture



(a) CPU utilization vs. time



(b) CPU utilization vs. processor number

Fig. 24 Performance of Fractography3d with load balancing. The CPU utilization is uniformly higher

the shock-wave front a very fine mesh is desired. A coarse mesh is less accurate. However, starting with the entire mesh very refined is very slow, so adaptivity is used to refine the mesh at the moving wave front and maintain a coarsened mesh elsewhere. A normalized velocity gradient is used as the criteria to identify the shock and modify the mesh size accordingly.

Preliminary results indicate that the regionally refined and coarsened mesh exhibits similar accuracy to an initially fine mesh for modeling the shock-wave front. Once parallel coarsening is complete, we are confident that using adaptivity will result in better performance. Since the region of refinement is constantly changing in this simulation, it should be a challenging problem for CHARM++'s load balancing algorithms to tackle.

5.4 Dendritic Growth Simulation

An older but influential ParFUM application was developed as a collaboration with Jeong, Goldenfeld and Dantzig[29]. This application simulates metal solidification, with the goal of better understanding the details of dendrite growth, which strongly affect the microstructure of cast metals. The problem involves coupled phase (liquid or solid), temperature, velocity and pressure fields on an adaptive grid that tracks the moving solid/fluid interface.

As simulation begins, the application code generates a new mesh in serial using an octree decomposition. A typical mesh consists of a half-million nodes and slightly fewer elements. ParFUM then partitions this mesh and begins parallel execution. At each timestep, a bidirectional conjugate gradient solver iterates to find the coupled solution data that solves the nonlinear governing equations. This is implemented in parallel using a simple shared node ParFUM communication step during each solver iteration. After a few iterations, the solve converges and we can begin the next timestep. After many timesteps, the solid/fluid interface has moved significantly and the adaptive mesh is becoming out of date. At this point, the application generates a new mesh to better track the new solid/fluid interface and the entire process repeats.

This ParFUM application was the first to use nonlinear solvers. Because the solver outer loop is nested inside the time loop, this application could not be written using the older ParFUM-calls-user control style and required the inverted (user calls ParFUM) control structure that is now standard.

6 Related Work

The need for domain-specific frameworks in scientific computing has been recognized for some time. However, most of the existing frameworks are either meant for structured grids or are not parallel. Two types of much simpler frameworks do exist, non-parallel and structured grid. Simple data structures allow a serial meshing framework to be built without the many complications involved with a dynamic physics application in parallel. Creating a structured grid framework involves a subset of the problems that occur when implementing an unstructured mesh framework. Currently, only a small number of parallel frameworks for unstructured mesh-based simulation exist. We discuss some of the existing software for manipulating parallel unstructured meshes.

One framework for mesh-based simulations is SIERRA[30] from Sandia National Laboratories. While it is not yet publicly available, it promises a large feature set. It is designed for large simulations on some of the world's largest parallel computers.

Simmetrix is a commercial software package which targets the entire mesh-based simulation workflow. It contains components for mesh generation and mesh adaptivity, and supports a variety of input models. Simmetrix also includes a simulation design environment. Simmetrix is currently adding parallel support to the software, so we do not evaluate how well it performs in parallel. In comparison, ParFUM is designed for parallel performance from the ground up.

Additional efforts in parallel mesh adaptivity are ongoing at RPI[31]. ParFUM uses similar refinement and coarsening algorithms.

One MPI based framework for unstructured mesh simulations is PYRAMID[32]. This framework developed by the Jet Propulsion Laboratory provides standard adaptive mesh refinement(AMR) operations in parallel, as well as partitioning via ParMetis.

The framework named Roccom is an "Object-Oriented, Data-Centric Software Integration Framework for Multiphysics Simulations" [33,34]. It unifies many separate components or applications involved in a single simulation. It primarily handles converting multiple mesh formats, as well as providing an orchestration mechanism to allow high level control over many components and applications.

An academic product of the CFDLab at the University of Texas at Austin is a C++-based meshing framework called libMesh. LibMesh works in parallel, making use of MPI and utilizes existing numerical solver packages.

One library for finite element codes was specifically designed for object oriented programming in C++. This library, deal.II, supports some of its operations in parallel on an SMP computer via shared memory or on a cluster via MPI. It supports some types of local grid refinements, but is limited to using only hexahedral elements. The limits on supported mesh types may be a problem for application developers wanting to use the library.

PREMA[35] is a runtime system specifically created for AMR applications. Some of its dynamic runtime features are similar to those that have been available in Charm++ for some time.

ParFUM is a general purpose unstructured mesh framework. ParFUM has always been built specifically for high performance on large parallel machines, so it is free from many problems that would arise when trying to convert a serial framework to work in parallel. Because ParFUM is built on the robust and highly adaptive Charm++ framework, ParFUM inherits a wide range of adaptive features that other frameworks, especially those built on MPI, will not achieve without major changes.

7 Conclusions and Future Research

We have developed a software infrastructure called ParFUM to simplify the creation and optimization of parallel applications that use unstructured meshes. The framework automates tasks such as mesh partitioning, maintenance of ghost layers and shared nodes, and communication of user-selected attributes to them. Preliminary support for adaptivity is also included in the framework, which is now ready to accommodate more sophisticated computational geometry algorithms for mesh adaptivity. Since it is built using Charm++/AMPI, ParFUM also supports automatic overlap of communication and computation, dynamic load balancing, and checkpoint-restart. ParFUM also includes capabilities for collision detection, parallel partitioning and specific refinement and coarsening algorithms.

ParFUM has already proved useful in several applications such as space-time meshing, crack propagation and rocket simulation. It has been shown to be scalable to a large number of processors and to present a low overhead.

We plan to extend the capabilities of ParFUM with more sophisticated support for adaptivity, dynamic repartitioning via coalescing and splitting existing partitions, and so on. We also are working on grid-capable versions with load balancing strategies which allow an application to spread across multiple clusters [2]. We hope that more application scientists and engineers will use the framework and provide us with feedback on how to further enhance its usefulness and improve programmer productivity while attaining high parallel performance.

Acknowledgments

The authors wish to thank Milind Bhandarkar, the developer of ParFUM's predecessor, the Charm++ FEM Framework, as well as Aaron Becker and Dima Ofman for their contributions to the new ParFUM codes. We would also like to thank our collaborators on the various applications that make use of ParFUM: Philippe Geubelle, Scot Breitenfeld, Sandhya Mangala and Robert Haber.

References

1. Kalé L (2004) Performance and productivity in parallel programming via processor virtualization. In Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10), Madrid, Spain.
2. Koenig G, Kale L (2005) Using message-driven objects to mask latency in grid computing applications. In 19th IEEE International Parallel and Distributed Processing Symposium.
3. Kale L, Bhandarkar M, Brunner R (2000) Run-time Support for Adaptive Load Balancing. In Rolim J, ed. Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP). Cancun - Mexico, Volume 1800: 1152–1159.
4. Kale L, Kumar S, Vardarajan K (2003) A Framework for Collective Personalized Communication. In Proceedings of IPDPS'03, Nice, France, p. 69.
5. Huang C (2004) System support for checkpoint and restart of charm++ and ampi applications. Master's thesis, Dept. of Computer Science, University of Illinois.

6. Chakravorty S, Kale L (2004) A fault tolerant protocol for massively parallel machines. In FTPDS Workshop for IPDPS 2004, IEEE Press.
7. Zheng G, Shi L, Kalé L (2004) Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In 2004 IEEE International Conference on Cluster Computing, San Dieago, CA.
8. Kalé L, Kumar S, Zheng G, Lee C (2003) Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS), Melbourne, Australia, pp. 23–32.
9. Bhandarkar M, Kalé L (2000) A Parallel Framework for Explicit FEM. In Valero M, Prasanna V. K, Vajpeyam S, eds. Proceedings of the International Conference on High Performance Computing (HiPC 2000), Lecture Notes in Computer Science. Springer Verlag, Vol. 1970:385–395.
10. Huang C, Lawlor O, Kalé L (2003) Adaptive MPI. In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03), College Station, Texas, pp. 306–322.
11. Kale L, Krishnan S (1996) Charm++: Parallel Programming with Message-Driven Objects. In Wilson G, Lu P, eds. Parallel Programming using C++. MIT Press, pp. 175–213.
12. Zheng G (2005) Achieving High Performance on Extremely Large Parallel Machines. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.
13. Kalé L (2002) The virtualization model of parallel programming : Runtime optimizations and the state of art. In LACSI 2002, Albuquerque, NM.
14. Message Passing Interface Forum (1993) MPI: A Message Passing Interface. In Proceedings of Supercomputing '93, IEEE Computer Society Press, pp. 878–883.
15. MPI Forum (1997) MPI-2: Extensions to the message-passing interface http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.
16. DeSouza J, Kalé L (2004) MSA: Multiphase specifically shared arrays. In Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing, West Lafayette, IN.
17. Bhandarkar M, Kale L, Sturler E, Hoeflinger J (2001) Object-Based Adaptive Load Balancing for MPI Programs. In Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074, pp. 108–117.
18. Karypis G, Kumar V (1996) Parallel multilevel k-way partitioning scheme for irregular graphs. In Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM), p. 35.
19. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**:359–392.
20. Karypis G, Kumar V (1998) Multilevel k-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed Computing **48**:96 – 129.
21. Karypis G, Kumar V (1998) Multilevel algorithms for multi-constraint graph partitioning. In Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM), Washington, DC, USA, IEEE Computer Society, pp. 1–13.
22. Chakravorty S (2005) Implementation of parallel mesh partition and ghost generation for the finite element mesh framework. Master's thesis, Dept. of Computer Science, University of Illinois.
23. Karypis G, Kumar V (1996) Parallel multilevel k-way partitioning scheme for irregular graphs. In Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing, p. 35.
24. Abedi R, Chung S.-H, Erickson J, Fan Y, Garland M, Guoy D, Haber R, Sullivan J, Thite S, Zhou Y (2004) Spacetime meshing with adaptive refinement and coarsening. In SCG '04: Proceedings of the twentieth annual symposium on Computational geometry, New York, NY, USA, ACM Press, pp. 300–309.
25. Kale L, Haber R, Booth J, Thite S, Palaniappan J (2003) An efficient parallel implementation of the spacetime discontinuous galerkin method using charm++. In Proceedings of the 4th Symposium on Trends in Unstructured Mesh Generation at the 7th US National Congress on Computational Mechanics.
26. Rivara M (1984) Algorithms for refining triangular grid suitable for adaptive and multigrid techniques. International Journal for Numerical Methods in Engineering **20**:745–756.
27. Lawlor O (2001) A grid-based parallel collision detection algorithm. Master's thesis, Dept. of Computer Science, University of Illinois.
28. Lawlor O, Kalé L (2002) A voxel-based parallel collision detection algorithm. In Proceedings of the International Conference in Supercomputing, ACM Press, pp. 285–293.
29. Jeong J-H, Goldenfeld N, Dantzig J (2001) Phase field model for three-dimensional dendritic growth with fluid flow. Physical Review E., **64**:1–14.
30. Stewart J, Edwards H (2004) A framework approach for developing parallel adaptive multiphysics applications. Finite Elements in Analysis and Design, **40**:1599–1617.
31. DeCougny H, Shephard M (1999) Parallel refinement and coarsening of tetrahedral meshes. International Journal for Numerical Methods in Engineering, **46**:1101–1125.
32. Norton C, Lou J, Cwik T (2001) Status and directions for the pyramid parallel unstructured amr library. In IPDPS, IEEE Computer Society, p. 120.
33. Jiao X, Campbell M, Heath M (2003) Roccom: an object-oriented, data-centric software integration framework for multiphysics simulations. In ICS '03: Proceedings of the 17th annual international conference on Supercomputing, New York, NY, USA, ACM Press, pp. 358–368.
34. Jiao X, Zheng G, Lawlor O, Alexander P, Campbell M, Heath M, Fiedler R (2005) An integration framework for simulations of solid rocket motors. In 41st AIAA/ASME/SAE/ASEE Joint Propulsion Conference, Tucson, Arizona.
35. Barker K, Chernikov A, Chrisochoides N, Pingali K (2004) A load balancing framework for adaptive and asynchronous applications. IEEE Trans. Parallel Distrib. Syst., **15**:183–192.