A Parallel Framework for Explicit FEM

Milind A. Bhandarkar

Laxmikant V. Kalé

Department of Computer Science

University of Illinois at Urbana-Champaign

{milind,kale}@cs.uiuc.edu

Abstract

As a part of an ongoing effort to develop a "standard library" for scientific and engineering parallel applications, we have developed a preliminary finite element framework. This framework allows an application scientist interested in modeling structural properties of materials, including dynamic behavior such as crack propagation, to develop codes that embody their modeling techniques without having to pay attention to the parallelization process. The resultant code modularly separates parallel implementation techniques from numerical algorithms. As the framework builds upon an object-based load balancing framework, it allows the resultant application to automatically adapt to load imbalances resulting from the application or the environment (e.g. timeshared clusters). This paper presents results from the first version of the framework, and demonstrates results on a crack propagation application.

1 Introduction

When a parallel computation encounters dynamic behavior, severe performance problems often result. The dynamic behavior may arise from intrinsic or external causes: intrinsic causes include dynamic evolution of the physical system being simulated over time, and algorithmic factors such as adaptive mesh refinement. External causes include interference from other users' jobs on a time shared cluster, for example. Significant programming efforts are required to make an application adapt to such dynamic behavior.

We are developing a broad approach that facilitates development of applications that automatically adapt to dynamic behavior. One of the key strategies used in our approach is multi-partition decomposition: the application program is decomposed into a very large number of relatively small components, implemented as objects. The underlying object-based parallel system (Charm++) supports multiple partitions on individual processors, and their execution in the data driven manner. The runtime system controls the mapping and re-mapping of these objects to processors, and consequent automatic forwarding of messages. Automatic instrumentation is used to collect detailed performance data, which is used by the load balancing strategies to adaptively respond to dynamic behavior by migrating objects between processors. This load balancing framework has been demonstrated to lead to high-performance even on large parallel machines, and challenging applications, without significant effort on part of the application programmer [3].

In order to promote programmer productivity in parallel programming, it is desirable to automate as many parallel programming tasks as possible, as long as the automation is as effective as humans. The load balancing framework accomplishes such automation for the broad categories of parallel applications. However, to use this framework, one must write a program in a relatively new, and unfamiliar parallel programming language (in this case Charm++). In contrast, an application scientist/engineer is typically interested in modeling physical behavior with mathematical techniques; from their point of view, parallel implementation is a necessary evil. So, the effort required to develop a parallel application is clearly detrimental to their productivity, even when adaptive strategies are not called for.

We are continuing this drive towards automation by considering special classes of applications that can be further automated. In this paper, we describe the beginnings of our effort to make such a special class: computations based on the finite element method (FEM). The framework whose design and implementation we describe in this paper allows an application programmer to write sequential code fragments specifying their mathematical model, while leaving the details of the finite element computation, including the details of parallelization, to the "system" — the finite element framework, along with the underlying load balancing framework. The code written in this fashion is succinct and clear, because of the clear and modular separation between its mathematical and parallelization components. Further, any application developed using this framework is automatically able to adapt to dynamic variations in extrinsic or intrinsic load imbalances.

Other FEM frameworks such as Sierra (Sandia National Laboratories) exist. However, a detailed review of related work is not possible here due to space limitations. We note that most other frameworks are not able to deal with dynamic behavior as ours does.

In the next section, we describe the overall approach to dynamic computations that we're pursuing. Section 3 provides a background for finite element computations, while section

4 describes our parallel FEM framework. Our experiences in porting an existing crack propagation application (developed by P. Guebelle et al at the University of Illinois), and the relevant performance data are presented in section 5. This framework is only a first step in our effort to facilitate development of complex FEM codes. Some directions for future research are identified in the last section.

2 Parallelizing Dynamic Computations

In *dynamic* applications, the components of a program change their behavior over time. Such changes tend to affect performance negatively, especially on a large number of processors. Not only does the performance of such applications tend to be poor, the amount of effort required in developing them is also inordinate.

We are developing a broad approach for parallelization of such problems, based on *multi-partition decomposition* and data-driven execution. The application programmer decomposes the problem into a large number of "objects." The number of objects is chosen independently of the number of processors, and is typically much larger than the number of processors. From the programmer's point of view, all the communication is between the objects, and not the processors. The runtime system is then free to map and re-map these objects across processors. The system may do so in response to internal or external load imbalances, or due to commands from a time shared parallel system.

As multiple chunks are mapped to each processor, some form of local scheduling is necessary. A data-driven system such as Charm++ can provide this effectively. On each processor, there is a collection of objects waiting for data. Method invocations (messages) are sent from object to object. All method invocations for objects on a processor are maintained in a prioritized scheduler's queue. The scheduler repeatedly picks the next available message, and invokes the indicated method on the indicated object with the message parameters.

Scientific and engineering computations tend to be iterative in nature, and so the computation times and the communication patterns exhibited by its objects tend to persist over time. We call this "the principle of persistence", on par with "the principle of locality" exhibited by sequential programs. Even the dynamic CSE applications tend to either change their pattern abruptly but infrequently (as in adaptive refinement) or continuously but slowly. The relatively rare case of continuous and large changes can still be handled by our paradigm, using more dynamic and localized re-mapping strategies. However, for the common case, a runtime system can employ a "measurement based" approach: it can measure the object computation and communication patterns over a period of time, and base its object re-mapping decisions on these measurements. We have shown [3] that such measurement based load balancing leads to accurate load predictions, and coupled with good object re-mapping strategies, to high-performance for such applications.

Based on these principles, we have developed a load balancing framework within the Charm++ parallel programming system [2]. The framework automatically instruments all Charm++ objects, collects their timing and communication data at runtime (in a "database"), and provides a standard interface to different load balancing strategies (the job of a *strategy* is to decide on a new mapping of objects to processors). The framework is sufficiently general to apply beyond the Charm++ context, and it has been implemented in the Converse [5] portability layer. As a result, several other languages on top of Converse (including threaded

MPI) can also use the load balancing functionality.

Overall, our approach tries to automate what “runtime systems” can do best, while leaving those tasks best done by humans (deciding what to do in parallel), to them. A further step in this direction is therefore to “automate” commonly used algorithms in reusable libraries and components. The parallel FEM framework is an attempt to automate the task of writing parallel FEM codes that also adapt to dynamic load conditions automatically, while allowing the application developer to focus solely on mathematical modeling techniques, as they would if they were writing sequential programs.

3 Finite element computations

The Finite Element Method is a commonly used modeling and analysis tool for predicting behavior of real-world objects with respect to mechanical stresses, vibrations, heat conduction etc. It works by modeling a structure with a complex geometry into a number of small “elements” connected at “nodes”. This process is called “meshing”. Meshes can be structured or unstructured. Structured meshes have equal connectivity for each node, while unstructured meshes have different connectivity. Unstructured meshes are more commonly used for FEM.

The FEM solver solves a set of matrix equations that approximate the physical phenomena under study. For example, in stress analysis, the system of equations is:

$$F = kx$$

where F is the stress, k is the stiffness matrix, and x is the displacement. The stiffness matrix is formed by superposing such equations for all the elements. When using FEM for dynamic analysis, these equations are solved in every iteration, advancing time by a small "timestep". In explicit FEM, forces on elements are calculated from displacements of nodes (strain) in the previous iteration, and they in turn cause displacements of nodes in the next iteration.

In order to reduce errors of these approximations, the element size is very small, resulting in a large number of elements (typically hundreds of thousands). The sheer amount of computations lends this problem to parallel computing. Also, since connectivity of elements is small, this results in good computation to communication ratios. However, the arbitrary connectivity of unstructured meshes, alongwith dynamic physical phenomena introduce irregularity and dynamic behavior, which are difficult to handle using traditional parallel computing methods.

4 The Parallel FEM framework

Our FEM framework treats the FEM mesh as a bipartite graph between nodes and elements. In every iteration of the framework, elements compute attributes (say, stresses) based on the attributes (displacements) of surrounding nodes, and the nodes then update their attributes based on elements they surround. Nodes are shared between elements, and each element is surrounded by a (typically fixed) number of nodes.

We partition the mesh into several "chunks". In order to reduce communication, we take locality into account by using spatial partitioning. We divide the elements into chunks. Nodes shared by multiple elements within the same chunk belong to that chunk. Nodes shared between elements that belong to different chunks are duplicated in those chunks. In order to identify association of a node with its mirrors, we represent each node with a global index. However, for the purpose of traversing the list of nodes in a chunk, we also have an internal local index for each node. Each chunk also maintains a table that maps its shared nodes to a (chunk-index, local-index) pair.

The FEM Framework implements all parallelization features, such as a chare array of chunks, a data-driven driver for performing simulation timesteps, messages for communicating nodal attributes etc. It makes use of generic programming features of C++, and consists of several Charm++ class "templates", with the user-defined datatypes as template parameters. Another approach for our framework could have been to provide base classes for each of these asking the user to write derived classes that provide the application-specific methods. However, this approach adds overhead associated with virtual method invocation. It also misses on several optimization at compile time as can be done with our template-based approach.

User code and the FEM framework have interfaces at various levels. The FEM framework provides parallelization, communication data structures, as well as a library for reading the file formats it recognizes. The application-specific code provides the data structures, function callbacks, and data that controls the execution of the FEM framework.

Figure 1 shows the algorithm used by this framework.

```
Chunk 0:
  Initialize Configuration Data [FEM_Init_Conf_Data]
  Broadcast Configuration data to all chunks.

All Chunks:
  Initialize Nodes [FEM_Init_Nodes]
  Initialize Elements [FEM_Init_Elements]
  for i = 0 to FEM_Get_Total_Iter
    for all elements
      FEM_Update_Element(Element, Node[], ChunkData)
    for all nodes
      FEM_Update_Node(Node, ChunkData)
    FEM_Update_Chunk(ChunkData)
    Communicate shared NodeData to neighboring chunks
  end
```

Figure 1: Pseudocode for the Data-Driven Driver of FEM Framework

4.1 Application-specific Datatypes

All FEM applications need to maintain connectivity information to represent the bipartite graph among nodes and elements. This information is maintained by the FEM framework in its data structures associated with each chunk. In addition, each application will have different attributes for nodes, elements etc. Only these need to be provided by the user code.

Node: This user-defined datatype contains nodal attributes. It also provides instance methods called `getData` and `update`. `getData` returns the `NodeData` (see below), and `update` combines the `NodeData` received from other chunks with the node instance.

NodeData: `NodeData` is a portion of `Node`, which is transmitted to the mirror of that node on another partition. It consists of nodal attributes that are needed to compute element attributes in the next iteration. In stress analysis, these typically include force vectors. The `Node` type may contain other nodal attributes that may be fixed at initialization, so one does not need to communicate them every iteration, and thus can be excluded from `NodeData`.

Element: This datatype should contain attributes of each element, such as material, density, various moduli etc. These properties usually do not change, and maybe put in the `ChunkData`(see below), however, separating it into a different datatype makes the code much cleaner.

ChunkData: Most of the information about a chunk such as number of nodes, and elements, and communication data, is maintained by the framework itself, and user-code need not worry about these. Application data that is not an attribute of a node or an element, but has to be updated across iterations belongs to `ChunkData`. A typical usage is to store the error estimate of a chunk that needs to be used for testing convergence.

Config: This read-only data structure is initialized only on one processor and broadcast by the framework after the initialization phase. This may include various material properties, filenames for output files, number of timesteps, and other parameters. If this data structure contains pointers and other heap-allocated structures, the application-specific code has to include methods for serialization.

4.2 Application-specific Callbacks

In addition to above datatypes, user needs to provide only the following function callbacks. These functions will be called at various stages during a simulation.

At the initialization stage, chunk 0 calls `FEM_Init_Conf_Data` function to initialize the application-specific configuration data, and broadcasts it to every other chunk. Each chunk then initializes the application-specific node and element data structures by calling `FEM_Init_Nodes` and `FEM_Init_Elements` callbacks. `FEM_Get_Total_Iter` callback needs to return the number of iterations to perform.

The core computations are performed by the callback functions `FEM_Update_Element`, `FEM_Update_Node`, and `FEM_Update_Chunk`. These functions are called by the framework on all elements and nodes on every chunk once per iteration. Typically, an element is updated by calculating the stresses using the nodal attributes of surrounding nodes. In more complicated applications, such as the crack-propagation, it calculates the progress of the crack. A node is updated by calculating its velocity and acceleration. This is where the simulation spends most of its time, so care should be taken to optimize these functions.

4.3 Mesh File Format

Currently, the FEM framework recognizes only one file format for reading in the FEM mesh. This file format is described below. However, in order to support more file formats, and to make it easier for converting existing applications, we have provided a library of routines that can be called to specify the FEM mesh.

The FEM framework reads in the mesh information stored in one file per chunk. Every such file contains three sections: Nodes, Elements, and Communication. The Nodes section contains the list of global node numbers that have been assigned to a chunk. Elements section contains the list of elements assigned to a chunk, alongwith their connectivity with surrounding nodes. The connectivity is specified using local node numbers rather than global. Communication section specifies which nodes' data needs to be sent to which chunks. Our file format is optimized in order to reduce the initialization time. We have provided "offline" programs to convert output of popular partitioning programs such as Metis [4] to our file format.

5 Crack Propagation

We have ported an initial prototype of an application that simulates pressure-driven crack propagation in structures (such as the casing of a solid booster rocket.) This is an inherently dynamic application. The structure to be simulated is discretized into an FEM mesh with triangular elements[1]. The simulation starts out with initial forces on boundary elements, calculating displacements of the nodes, and stresses on the elements. Depending on the geometry of the structure and pressure on boundary elements, a crack begins to develop. As simulated time progresses, the crack begins to propagate through the structure. the rate of propagation and the direction depends on several factors such as the geometry of the computational domain, material properties, and direction of initial forces. In order to detect the presence of a crack, several "zero-volume" elements are added near the tip(s) of the crack. These elements are called "cohesive" elements. In the current discretization, these cohesive elements are rectangular, and share nodes with regular (triangular) elements.

In order to parallelize this application, we may start out with a load-balanced mesh partitioning, produced with Metis. However, as the crack propagates, cohesive elements are added to partitions that contain tip(s) of the propagating crack, increasing computational load of those partitions, and thus causing severe load imbalance. A partial solution to this problem is to insert cohesive elements everywhere in the mesh initially. This tries to reduce load imbalance by introducing redundant computations and memory requirements (increasing the number of elements). Since our framework supports dynamic load balancing for medium-grained partitions, one can eliminate such redundancy.

[1]This initial prototype simulates a 2-D structure. Final versions will simulate 3-D structures with tetrahedral meshes. No changes will have to be made to the framework code and interfaces between framework and application code.

5.1 Programming Effort

We started out with a sequential program of 1900 lines that simulated crack propagation on a 2-D structure, and converted it to our framework. Since the parallelization aspects were handled by the framework, all that was required to convert this program was to locate node-specific and element-specific computations, separating them out into user-defined data types, and supplying them to the framework. The resultant code was only 1200 lines of C++ (since the connectivity information between nodes and elements is also handled by our framework, code for reading it in, and forming a bi-partite graph, and iterating on this graph through indirection, was completely eliminated.) The framework itself has about 500 lines of C++ code, including dynamic load balancing mechanisms. Even when we add facilities for adaptive refinements, and oct-tree based meshes, reduction in application-specific code is expected to be similar.

5.2 Performance

Since many FEM codes are written using Fortran (with better optimization techniques for scientific programs), we compared the performance of our C++ code for crack-propagation with similarly partitioned Fortran code. C++ code is about 10% slower. We expect this difference to become smaller with advances in C++ compiler technology.

We divide the original mesh into a number of small chunks; many more than the available number of processors, mapping multiple chunks on the same processor. Therefore, one may be concerned about the overhead of scheduling these chunks. However, as Figure 2 shows, mapping multiple chunks to each processor does not add significant overhead. In fact, as number of chunks per processor increases, performance may improve, as the data of a smaller chunk is more likely to fit in cache! This also eliminates the difference between C++ and Fortran versions of our codes because the advantages of Fortran to perform cache-specific optimization are no longer relevant.

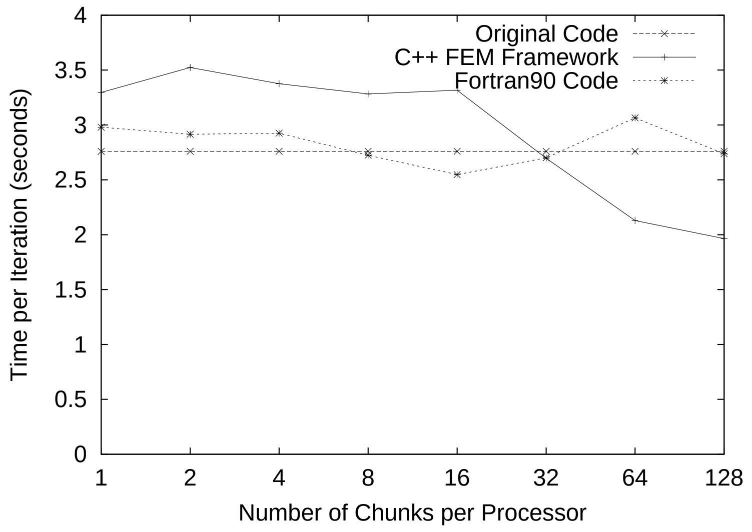

Figure 2: Effects of partitioning the mesh into chunks more than number of processors.

5.3 Adaptive Response

As described earlier, the pressure-driven crack propagation application exhibits dynamic behavior that affects load balance as simulation progresses. In order to judge the effectiveness of our framework in handling such imbalance, we ran this application on 8 processors of NCSA Origin2000, splitting the input mesh of 183K nodes into 32 chunks, i.e. 4 chunks per processor. Figure 3 shows the results where performance (indicated by number of iterations per second) is sampled at various times. As the simulation finishes 10 timesteps, a crack begins to appear[2] in one of the partitions, increasing the computational load of that partition dramatically. This demonstrates the impact of load imbalance in such an application. At timestep 25 and 35, the load balancer is activated. It instructs the FEM framework of the new mapping, which it carries out by migrating the requested chunks. Improvement in performance is evident immediately.

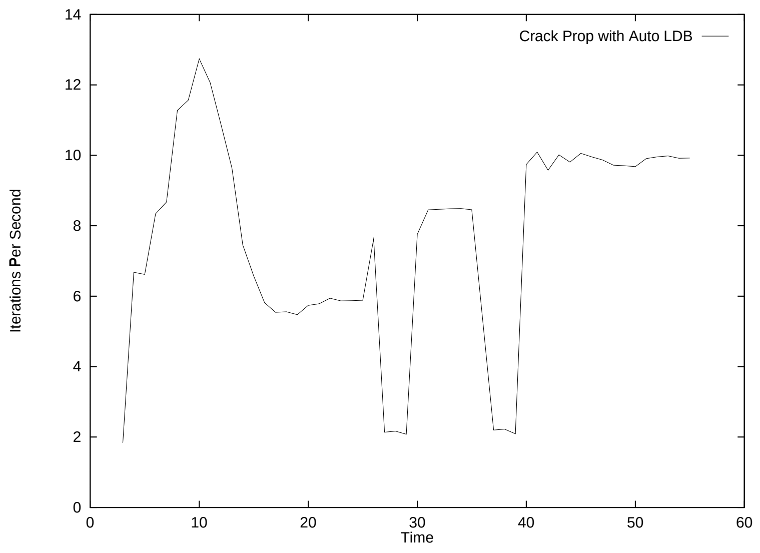

Figure 3: Crack Propagation with Automatic Load Balancing

6 Conclusion and Future Work

We described a C++ based framework for developing explicit finite element applications. Application programmers do not need to write any parallel code, yet the application developed will run portably on any parallel machine supported by the underlying Charm++

[2]In actual simulations, the crack may appear much later in time, however in order to expedite the process, we emulated the effects of crack propagation.

parallel programming system (which currently includes almost all available parallel machines, including clusters of workstations). Further, the application will handle dynamic application-induced imbalances, variations in processor speeds, interference from external jobs on a timeshared clusters, evacuation of machines when demanded by their owners, and so on.

The final paper will also include further performance data on the application for midsize workstation clusters (with 32 or 64 processors).

In our future work, we aim at complex FEM applications. Specifically, we first plan to support addition and deletions of elements, adaptive refinement of FEM meshes, oct-tree and quad-tree based meshes. Further extension to this work can be in the form of support for implicit solvers, multi-grid solvers and interface to matrix-based solvers.

References

[1] Neil Hurley Darach Golden and Sean McGrath. Parallel adaptive mesh refinement for large eddy simulation using the finite element methods. In *PARA*, pages 172–181, 1998.

[2] L. V. Kale, Milind Bhandarkar, and Robert Brunner. Run-time Support for Adaptive Load Balancing. In *Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, March 2000.

[3] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten.

NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[4] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. TR 95-035, Computer Science Department, University of Minnesota, Minneapolis, MN 55414, May 1995.

[5] Robert Brunner L. V. Kale, Milind Bhandarkar and Joshua Yelon. Multiparadigm, Multilingual Interoperability: Experience with Converse. In *Proceedings of 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP) Orlando, Florida - USA*, Lecture Notes in Computer Science, March 1998.

[6] J.B. Weissman, A.S. Grimshaw, and R. Ferraro. Parallel Object-Oriented Computation Applied to a Finite Element Problem. *Scientific Computing*, 2(4):133–144, February 1994.