The TeraScale Computational Framework

TeraScale, LLC
P.O. Box 1396
Cedar Crest, NM 87008
www.terascale.net

Executive Summary

There is a quiet revolution taking place that will profoundly alter the evolution of finite element application programs. The vortex of this revolution is the widespread availability of large clusters of inexpensive computer processors. Even relatively small firms can now afford to assemble a cluster of 100 3-GHz PCs running Linux. In order to make efficient use of a large cluster of processors, it is necessary to perform the requisite finite element calculations in parallel in a scaleable manner. There are subsets of the available finite element algorithms (e.g. explicit dynamics, matrix-free conjugate gradient) that can be parallelized in a straightforward manner. Even those algorithms become challenging to parallelize when algorithms such as contact or multi-point constraints are included. The vast majority of finite element applications are based upon implicit algorithms in which case the linear algebra solver becomes the single most important parallel challenge. They are inherently very difficult to scale and can run inefficiently on even a small number of processors.

Converting an existing feature-rich finite element program to run in a scaleable manner can be a daunting task. The TeraScale computational framework can be used to support and simplify this effort. The framework provides the programming foundation upon which the core finite element algorithms can run in parallel using much of its existing code. The framework frees the finite element code developer from dealing with the computer science aspects of parallel computing as well as providing many “common services” necessary to such computation. These include: I/O Services; Memory Management: Scattering & Gathering of Parallel Processes; Mathematical Libraries; Algorithmic Control; and Linear Algebra Solution Services.

Why use Frameworks?

A framework represents a collection of software components for building applications. By collecting these into a single toolkit, a framework enables the application developer to leverage these components into many different applications. Consequently, the amount of work (and code) required developing and maintaining an application is greatly reduced.

Historically, the finite element applications developer has been an engineer trained in some specific field of mechanics. Typically, they spend an inordinate portion of their software development time dealing with computer science details rather than focusing on algorithms and mechanics. The TeraScale finite element framework insulates the engineer from the computer science details and lets the engineer concentrate on the computational mechanics aspects of the application.

The realm of high performance, parallel, finite element applications provides a rich set of common abstractions upon which to build a computational framework. Examples of common services or tasks found in finite element applications include:

- IO services
- Memory management
- Parallel gather/scatter operations and global reductions
- Mathematical libraries
- Algorithmic control
- Linear algebra solution services

The common thread that runs through all these services is that they are essentially computer science/mathematical exercises that are not dependent upon physics equations or formulations. It is precisely these services that require the most attention when porting scientific applications between different hardware platforms. Generally, the scientific or physics parts of any application compile, link and run correctly on disparate hardware platforms with little porting effort. The vast majority of the porting effort goes into dealing with the highly system dependent intricacies of such tasks as IO services, memory management and locating/linking the proper support libraries. By placing these services into a common framework, all finite element applications built using the framework leverage the porting efforts required to move between new hardware platforms. Of course, they also leverage the considerable development time required to write and debug them.

Parallel does not have to mean Difficult

The TeraScale framework architecture is designed to minimize the amount of specialized parallel coding that the finite element applications developer must understand and code. Ideally, the engineer simply writes code in a serial mode with no regard for parallel issues. However, the application developer cannot be completely isolated from all parallel issues. The computational framework provides the application developer a set of interfaces that isolate the parallel coding to a few simple interfaces.

The TeraScale framework is based upon a “SPMD Model” (Single Program Multiple Processor). The SPMD model is based upon the concept that the finite element mesh is decomposed (i.e., partitioned) into a set of sub-meshes that are assigned to each processor and spread onto the individual processors. Once the mesh is spread, each processor executes a copy of the same application on the piece of the mesh that it has been assigned. A fundamental aspect of the partitioning process is to embed into the partitioned/spread sub-meshes the information about mesh entities that are shared by multiple processors. These parallel data structures are discussed below in the descriptions of mesh partitioning and the Parallel Mesh Object (PMO).

Other parallel programming paradigms can be used (e.g., threaded models), but TeraScale believes the SPMD model provides the best balance for ease of use and for scalability.

Today, it is possible to purchase a 64 processor Linux cluster (with whatever is the current “fastest” PC chip) for less than $100K. These Linux clusters will continue to grow in popularity simply because they are cheap and powerful.

The major task required for computing in a SPMD model is to move data back and forth among processors. Within distributed memory architectures, this is done via message passing using MPI, PVM, or other communication libraries. The physics algorithms based on the finite element method and boundary element method have inherent points of synchronization - places in the algorithms where multiple objects must write to some other object. Hence, the algorithm cannot proceed until every processor finishes sending/receiving messages. These points of synchronization are well characterized in the finite element method and are generally referred to as gather/scatter/assemble operations, or as global reduction operations. The framework has abstractions (interfaces) for these operations (i.e., the developer does not make calls to the MPI library directly). Furthermore, these interfaces insulate the application from the particular message passing library that is used by the framework. Today, the TeraScale framework uses MPI as the message passing library. However, finite element applications have a very long life and perhaps in the next decade some new, better, faster, etc., message passing library will become available. Should that occur, TeraScale will re-implement the methods that serve the applications parallel needs and the mechanics application will not have to change any lines of code.

The TeraScale framework hides these parallel aspects behind well-defined interfaces that are familiar to the finite element applications developer. Think of the framework as a highly efficient data delivery system. The framework delivers the data to application when and where it is required. This description may sound as if there is very little going on in the framework. In reality, there is a tremendous level of complexity involved in delivering this data in a reliable and scalable manner. The services that the framework provides are inherently platform dependent. The framework allows this porting effort to be performed once, and then leverages it across all applications

One important guideline that is inherent in the design of the finite element framework is that there be no performance penalty for running (the parallel framework code) on a single processor. In general, applications built upon the framework should run seamlessly on any of:

- A single processor CPU.
- Parallel hardware based upon shared memory architecture.
- Distributed memory architectures.
- Clusters of shared memory machines.

No parallel application scales perfectly. However the framework should scale according to the limit of the underlying algorithms, as long as the model is large enough to warrant using more processors. A good rule of thumb is that each processor should have at least 20K degrees of freedom. The actual parallel performance is tied to the ratio of the number of degrees of freedom in the problem to the number of shared degrees of freedom between processors. Inter-processor communications is typically three orders of magnitude slower that memory bandwidth speeds on a single processor. The application must amortize the cost of inter-processor communications over a large number of floating point operations between communication calls.

What about my Legacy Fortran Code?

When we refer to the “framework,” we are really describing a collection of sub-systems that provide the complete toolkit (i.e., a set of components) necessary to build complex parallel applications. The TeraScale framework is written primarily in the C++

programming language. There are large bodies of the TeraScale libraries (e.g., math library, element library, etc.) written in Fortran and TeraScale has made it easy to call Fortran from the C++ classes. These libraries also provide a Fortran binding so that they can be called directly from other Fortran subroutines. There is a huge body of well-used Fortran legacy applications that simply cannot be set aside and re-written in C++. In fact, it is difficult to achieve the performance levels in C++ that can be achieved in Fortran since the Fortran language is inherently designed for computations with multi-dimensional arrays. The memory management, found in the framework and the Parallel Mesh Object, is carefully crafted to lay out the fields as blocked Fortran-like arrays so that they are easy to pass to Fortran.

To "re-write" an existing application using our framework is an exercise in picking and choosing which pieces to keep. Think of a legacy Fortran application as a jigsaw puzzle where each piece is a Fortran subroutine. Throw the pieces out on the table and then begin plugging them back into the new framework-based application in the appropriate spots.

The Framework Procedural Base Classes

The concept of writing "object oriented" code is sometimes taken to extremes. Certainly, the TeraScale framework is object oriented, but if taken too far, the concept looses all hope of achieving high performance. TeraScale strikes a balance by recognizing that the finite element method is inherently procedural in nature.

The TeraScale framework provides a pre-defined set of abstractions for the procedural flow of algorithms in the finite element application. These are in the form of a set of C++ abstract base classes. An abstract base class is one that cannot be used directly – specific concrete derived classes provide the implementation of some required methods in the class with specific interfaces. The derived class inherits a large amount of functionality from the base class. That is why the abstraction exists in the first place – there is some large body of common functionality.

The set of abstract base classes in the framework contains:

- ***Procedure*** – a defined set of physics (physics objects). The procedure object's main job is to manage the time marching algorithm. This includes advancing the state of the fields and global reductions and reading/writing the state for the model. Loose coupling between multiple physics is orchestrated by the procedure.
- ***Physics*** – a single subset of physics. Generally, this represents a single physics but it could hold tightly coupled multi-physics where the set of physics is coupled through the solver.

  The Physics' main job is to advance the solution one increment in "time" at the request of the Procedure. The physics layer of the framework maps cleanly onto the traditional notion of a finite element code. This is where the concept of a mesh first appears. The physics object registers the nodal fields required for the computations and often registers fields on node, edge, and face sets. It also constructs and owns a set of the element blocks based on the attributes of the mesh and the physics to be performed.
- ***Element Block*** – a collection of finite elements that have the same topology and section/subsection/material hierarchy. An element block object is derived to hold

a particular element and physics formulation. An element block uses one or more element objects to compute the element formulation.

The element block has a collection of member elements, each of which is processed through some element object. It is important to realize that an element object holds methods, while the member elements of the element block hold data. The member elements are pumped through those methods to compute the values of their data.

The element block object is responsible for registering all the element fields required for the computation.

- ***Section*** – the section object holds the physical representation of the element at each integration station of the element. The section object also holds algorithms to integrate the section. For solid elements, the section only holds the material object (see below). For a complicated element, such as a layered shell, the section holds descriptions for the geometric lay-up of the shell layers, the integration rules for integrating through the layers, and the material objects for each of the layers. A section object may hold a collection of section objects (i.e., subsections) and/or a material. Usually, it holds one or the other. When it does hold subsections, it also holds the algorithm to assemble each subsection's contribution to the section.
- ***Material*** – a material object integrates the material response through time at a set of material points in the model. It holds a collection of algorithms for accomplishing this task.

What does the derived class have to implement? The minimum requirement is to implement two methods: `initialize(…)` and `execute(…)`. In these methods the application registers and initializes any of the required fields and provides the particular mechanics algorithms required by the application.

In most applications the procedural flow is invoked once for the `initialize(…)` method and then many times for the `execute(…)` method. The specific application is free to implement any other methods as part of the derived class and call any Fortran methods that may be available.

The flow of the application proceeds downward through all derived classes arranged in a tree. The main program will invoke the `execute(…)` method of its Procedure classes in the appropriate order. Each Procedure class calls the `execute(…)` method of its Physics classes using its particular algorithm. The Physics classes execute their element block execution methods. The element block classes execute their Section execution methods, which execute their Material execution methods. In each "level" of the tree, the program control is determined by the particular implementation of the `initialize(…)` and `execute(…)` methods.

Partitioning the Model for a SPMD Framework

In the Single Program Multiple Processor (SPMD) parallel programming model, the finite element mesh must be partitioned across the set of processors. There are numerous partitioning algorithms that can be used. The two most common are topological partitioning and geometrical partitioning. In the topological case, the partitioner application performs a graph analysis of the connection of the mesh to minimize the number of shared nodes across processor boundaries. A geometrical partitioner uses

some algorithm to slice up the mesh is space and is typically much faster than a topological partitioner. In general, applications will run slightly faster with a good topological partitioning than with a simple geometric partitioning. However, there are numerous finite element algorithms that require a geometric partitioning (e.g., contact between two bodies) in order to achieve parallel performance. TeraScale provides a geometric partitioner application, which is based upon the recursive coordinate bisection algorithm. The TeraScale partitioner is designed to be able to accommodate new algorithms in the future should they be deemed necessary or desirable.

Figure 1 shows a simple mesh and its partitioning across 4 processors. The mesh contains nodes, edges, faces, and elements, referred to as "mesh entities." The nodes, edges, and faces that reside on the inter-processor boundaries (shown in red) are shared between multiple processors (in certain cases, elements can be shared as well). TeraScale's implementation of the SPMD model requires that one processor own the mesh entity while the other processors simply have a copy of the mesh entity. Also, our SPMD model requires every mesh entity to have a unique global ID (i.e., unique across all processors).

The partitioner application reads in the original mesh and spreads it into pieces for each processor. Figure 1, shows the mesh partitioned into four pieces, each of which resides in its own mesh file after partitioning.

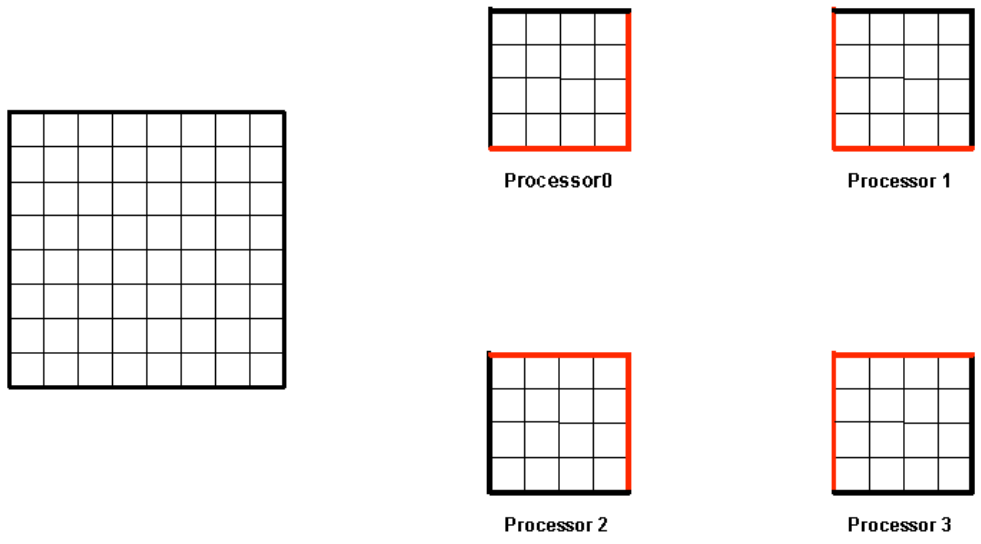

Figure 1. A four way partitioning of a simple mesh.

While not formally a part of the computational framework, the partitioning services are a fundamental infrastructure requirement for deploying parallel finite element applications. Furthermore, the SPMD model usually takes advantage of independent parallel IO. That is, each processor can write output to its own independent disk (hence there is no contention for the disk amongst processors). As a consequence, upon completion of the

analysis, there may be a set of files that must be recombined (e.g., results files, restart files, history files). This calls for a "departitioner" service that puts them back together again. The partitioner and departitioner services are provided along with the TeraScale framework.

The Parallel Mesh Object (PMO)

The finite element mesh must have a representation both in memory and on permanent storage (i.e., file). Above, in the discussion of the partitioning services, the mesh is spread into a set of sub-mesh files; one for each processor. The actual mesh file is written in a neutral file format. Most commercially available file formats do a good job of representing a mesh on a single processor in that they support the generality of numerous materials defined on a collection of common element types. They support boundary conditions and loads through a general notion of sets (e.g., node sets, edge sets, face sets, element sets). However, when viewed in the context of partitioned meshes as in Figure 1, they often lack the data structures to define the shared entities (nodes, edges, faces, and elements). Furthermore, they often lack the concept of a unique global ID for each and every mesh entity. Such global IDs are a requirement for parallel computations.

The TeraScale framework makes use of the TeraScale Parallel Mesh Object (PMO) to represent the mesh. The PMO is a stand-alone library that can be deployed independent of the TeraScale framework. The PMO provides a coherent array of services for computing on a mesh spread over some set of processors. Think of the PMO as a virtual mesh object that is cognizant of where all its pieces reside across the parallel platform. The concept of general subsetting mechanisms for nodes, edges, faces, and edges is supported, with full capabilities to create, query, and access the mesh data.

The PMO contains the abstraction of mesh readers and mesh writers. These define a set of interfaces for reading and writing finite element data to permanent storage. The concept is quite simple; the application accesses data through the set of interfaces defined upon the mesh object. The mesh object performs read/write on demand (i.e., it does not read/write anything from file unless requested to do so). The particular flavor of mesh reader or mesh writer that is given to the mesh object can be changed at run time. Mesh readers and writers for alternate mesh formats can be derived with minimal effort. The advantage is that any finite element application can use a new mesh reader without changing any code in the finite element application. If the finite element application reads and writes its mesh through the PMO, then it can instantly leverage any new file formats available through the library of mesh readers and mesh writers.

One of the fundamental reasons for developing a computational framework is to provide a mechanism for developing multiple applications that leverage an existing body of code. The environment provided by the framework creates a set of conventions that enable coupling multiple applications to solve more complex multi-field problems. To this end, the TeraScale framework and the PMO provide a set of specifications of how mathematical fields are defined and represented. A mathematical field has a designated math type of scalar, vector, tensor, or quaternion. The math type is further characterized by special cases of vectors, tensors and quaternion depending on the particular tensor or vector space required by the field. (Quaternions are a mathematical representation of finite rotations). The math type provides more information than just the number of components in the field. It provides a convention for the ordering of those components in the array-based representation of the field within the computer. For example a symmetric tensor in three dimensions has six components. The framework specifies

those components are defined as an array of length six in the order of components: 11, 22, 33, 12, 23, and 31. A three dimensional quaternion is defined as an array of length four in the order of components: X, Y, Z, and Q.

The PMO also supports field management. The application can register fields with attributes on subsets of the mesh. For example, an application might register a vector valued displacement field over all the nodes in the mesh. The field management functionality is extremely flexible. It provides memory management of the fields as well as the ability to marshal the fields to permanent storage repeatedly during an analysis. For example, the displacement field described above represents a state field that must be saved to permanent storage in order to restart the finite element application. There is one copy of the displacement field in memory, but there may be many copies of that field saved on the restart file under different time stamp values.

The PMO provides the run time parallel IO services required by the application and the partitioner and departioner services. The partitioner and departioner services operate upon PMO mesh objects and the fields defined on them. Hence, it is possible to map from M processors to N processors (e.g., run an analysis on 4 processors and then restart it on 8 processors).

Of considerable importance is the fact that the PMO has parallel services to generate a unique set of global IDs for all the mesh entities. This is a challenging task in parallel and its significance to many finite element applications is enormous. Referring again to Figure 1, typically the initial mesh comes is represented using some neutral file format and is partitioned as shown. The partitioner application uses the PMO and the appropriate mesh reader to read in the mesh, determine the proper decomposition and spread the sub-meshes to the individual processors using a mesh writer. While the usual "plain vanilla" finite element application typically uses nodal degrees of freedom, many finite element applications are dependent upon edge, face and element degrees of freedom. Hence, the partitioned mesh may have all the shared node and element information and unique global node and element IDs, but lacks any information about the unique set of edges and faces. The PMO provides methods to generate the unique set of all edges and faces within a mesh using an order N algorithm where N is the number of elements in the mesh. This is a significant capability in a serial application and very difficult to achieve in parallel. Furthermore, the PMO generates (in parallel) unique global IDs for each and every edge and face in the mesh and establishes all the sharing information and the processor owning information required for the computations to proceed.

Common Mathematical Methods

The framework contains a set of mathematical methods that may be applied to fields of scalars, vectors, tensors and quaternions. The set of methods represents a common set of operations found in almost all finite element applications. They include:

- Create a rotation matrix from a quaternion.
- Create a quaternion from a rotation matrix.
- Incrementally update quaternion from a rotation vector.
- Compute eigenvalues and eigenvectors of a symmetric tensor.
- Rotation of symmetric tensors.

- Rotation of vectors.
- Solve symmetric set of NxN equations at a point (N small, not to be confused with a linear algebra solver over the entire mesh).

Linear Algebra Solution Services

The TeraScale Framework provides a common interface to linear solver services for scalable solution of sparse systems of equations on distributed and shared memory parallel architectures. The interface to linear algebra packages is based upon a finite element view of the process that augments the native solver view. The underlying linear algebra representation of the assembled global element operators and right-hand-sides is hidden from the physics application developer. This abstraction layer allows the interchange or selection of different linear algebra solvers and/or preconditioners without modifying the physics implementation.

TeraScale provides a parallel direct solver package as well as a library of Krylov based iterative solvers with various preconditioners.

The solver interface provides direct support for node, edge, face and element degrees of freedom.

Acknowledgments

The TeraScale framework and supporting libraries were funded in part by:

- Engineering Sciences Directorate of Lawrence Livermore National Laboratories under contract No. B503619.
- Sandia National Laboratories Livermore under contract No. 13024.
- National Science Foundation under Grant No. CMS-0112950