An Object-Oriented Programming Framework for Parallel Finite Element Analysis with Application: Liquid Composite Molding

Brian J. Henz and Dale R. Shires
U.S. Army Research Laboratory
Attn: AMSRL-CI-HC
Aberdeen Proving Ground, MD 21005
bhenz@arl.army.mil
dshires@arl.army.mil

Abstract

The use of object-oriented programming techniques in development of parallel, finite element analysis software enhances software reuse and makes application development more efficient. In this paper, an object-oriented programming framework for developing parallel finite element software is described. All required steps, from data file parsing and equation solving to post processing and graphical user interfaces, are discussed. After development of the framework, a sample parallel finite element code, namely COMPOSE, is taken from its original functional programming paradigm and implemented in the new framework. Besides ease of development, the use of generic visualization and interface tools for software utilizing the framework speeds delivery of research codes to end users.

Keywords: Object-Oriented Programming, Parallel Processing, Finite Element Method, Resin Transfer Molding, Software Development

1. Introduction

With parallel computers becoming easier to obtain, often in the form of clusters and multiprocessor personal computers (PCs), the development of software to take advantage of this impressive computational power is essential. One of the areas in scientific computing that has shown the ability to utilize parallel processing techniques on shared and distributed memory architectures is the finite element method (FEM). The FEM has been utilized for some time in serial and parallel codes to solve engineering problems in various disciplines. By developing an object-oriented programming (OOP) framework for parallel FEM software, these existing codes can be parallelized and easily expanded when more complex physical models are available or required.

There are many issues to overcome when parallelizing a serial FEM code. Since the software is typically implemented with structured functional programming techniques in a language such as Fortran or C, reusing the code can be difficult to impossible. If the parallelized code is also implemented in a functional manner, then future extensions or reuse by other developers is neither simple nor economical. However, with these same codes parallelized by implementation within an OOP framework, much of the generic FEM work does not need to be repeated and extensions are less difficult. Besides these advantages, it is also possible to create a generic graphical user interface (GUI) that visualizes the geometric data and computational results with an architecture-independent tool. Two examples have been developed by these authors [6] and by Clarke et al. [4]. The availability of an easy-to-use GUI facilitates the transfer of FEM codes in the research domain to end users in industry.

The use of OOP for finite element developments is not new and appears in the literature [5, 11, 21]. These authors demonstrate the advantages of using OOP practices for FEM software, which are the basis for our developments. More recently, some work has gone into extending the OOP framework for FEM software for parallel developments [12, 16]. In this work, we expand upon the idea of the parallel OOP framework by introducing the use of a data format developed for the Interdisciplinary Computing Environment (ICE), the utilization of cross-platform GUI tools, and a matrix class with interfaces to multiple equation solver packages. The new framework for developing FEM software is called SPOOCEFEM (Simple Parallel Object-Oriented Computing Environment for the Finite Element Method), pronounced spöő’kéfêm.

The application described in this paper is the Composite Manufacturing Process Simulation Environment (COM-
POSE) developed at the U.S. Army Research Laboratory (ARL). This FEM software models the manufacturing process, namely resin transfer molding (RTM), that is used in the manufacture of many components for Department of Defense (DoD) combat systems. COMPOSE was previously implemented and parallelized with functional programming techniques utilizing Fortran 90/95 and is reimplemented here with SPOOCEFEM to demonstrate the positive attributes of developing and parallelizing FEM software with SPOOCEFEM.

2. OOP developments for serial or parallel

This section includes discussions of the data format utilized in SPOOCEFEM, some of the classes available, and file converters for NASTRAN and PATRAN neutral files. The work discussed in this section is applicable to serial or parallel FEM software development and is the framework around which new applications are built.

2.1. The FEM data file format

The data format used in SPOOCEFEM is the eXtensible Data Model and Format (XDMF) developed by Clarke and Namburu [3] for communication between interdisciplinary applications in the solution of multi-physics problems. This data format works well for the OOP approach since it is extensible and designed for storing data used/generated by computational software. Although the XDMF format is defined for many types of computational applications, SPOOCEFEM only utilizes a subset for unstructured grid problems as encountered in the FEM.

The data required by computational software is stored in XDMF as either “light” or “heavy” data. “Light” data is typically descriptive in nature and is used to store such information as the number of nodes in a mesh, the number of each type of element, and other small sets of information such as boundary conditions (BC) and material data. “Light” data is stored in eXtensible Markup Language (XML) files for ease of parsing, writing, and dataset modifications. “Heavy” data is typically large datasets required by FEM software such as element connectivity, node locations, and simulation results. “Heavy” data is stored in Hierarchical Data Format version 5 (HDF5) files for access across architectures. Having “heavy” data stored in binary files has many advantages including providing for random access to data when only a subset of a large dataset is required.

An advantage of utilizing the file format from ICE is that there are many GUI tools already available for visualizing the data. These include the standard GUI in ICE and its converters to EnSight and OpenDX formats. By wrapping the SPOOCEFEM libraries with The Simplified Wrapper and Interface Generator (SWIG), the Java interface developed by these authors is also available for visualization of geometric and simulation results data. Another advantage of using XDMF is that there are a growing number of scientific applications that utilize XDMF for input and output, making the problem solving capabilities of these codes more accessible. This capability is especially important when developing multi-physics software that requires modules from various developers with unique input/output requirements.

The following snippet from a SPOOCEFEM data file is used as an illustrative example of the file format. Notice the COMPOSE tag in the XML text. Everything between this tag and the FEM tag is application specific (at this time the only possible tags are TIMESTEP and VISCOSITY for isothermal simulations). TIMESTEP is the time step used in the transient simulation and is constant during the mold filling process. The viscosity is also a constant for the isothermal mold filling process.

```xml
<?xml version="1.0" ?>
</COMPOSE>
<TIMESTEP Value="0.5" />
</FEM>
</BCS>
</E2DTRI3 Number="2560">
   <BC ID="1" Location="5" TypeName="IC_FILLFRAC" Value="1.0" />
   <BC ID="0" Location="5" TypeName="BC_PRESSURE" Value="69000.0" />
</ELEMENTS>
</MATERIALS>
</MATERIAL ID="0" NumberProperties="4">
   <PROPERTY Name="IAXIS" Type="INTEGER" Value="1" />
   <PROPERTY Name="KYY" Type="REAL" Value="44.0e-08" />
   <PROPERTY Name="KXX" Type="REAL" Value="44.0e-08" />
   <PROPERTY Name="IAXIS" Type="INTEGER" Value="1" />
   <PROPERTY Name="VOC" Type="REAL" Value="0.195" />
</MATERIALS>
</MATERIAL>
</FEM>
</COMPOSE>
```

The pressure BCs and the fill fraction initial conditions (IC) are listed inside of the BC tag. ICs share many attributes with the BC class. Accordingly, it was decided that for parsing and data storage it was simpler to consider ICs as BCs and have unique methods of application and other functionality. For illustration, consider the fill fraction IC that requires a node location and real value for data storage. This is exactly the same data required for the pressure BC and therefore except for the method of application, the IC and BC can be stored in the same way.

The NODES tag is parsed and understood by the FEMPARSER, whereas the ELEMENTS tag contains some application specific data that is ignored by the FEMPARSER. The NODES tag contains the global number of nodes, the location in the HDF5 file of the XYZ values, and the location of the global indices (also stored in an HDF5 file). The ELEMENTS tag contains the global number of elements of all types with the number of the particular type specified in each tag below. For example, this mesh is made entirely...
of 3-noded triangular elements so the number of E2DTRI3 elements is the same as the total number of ELEMENTS. Each unique type of element contains the location of the elemental connectivities, global indices, and material numbers. The elemental thickness values are later read by the ComposeFEMparser and stored in application specific element types. The MATERIALS tag is parsed by the FEMparser and individual properties are stored by name. All properties listed with type REAL are stored as real values and all properties with type INTEGER are stored as integers by the use of a union in C++. This is a powerful method because all materials for any application are already parsed by the FEMparser and stored according to their application-specific name. In this file the permeability tensor is a 2-by-2 symmetric tensor but each term is stored as a material property and accessed by the application by name (e.g., KXX is the upper left term in the permeability tensor for COMPOSE).

2.2. NASTRAN and PATRAN neutral format parsers/converters

Parsers for reading geometric data from NASTRAN and PATRAN neutral files are included in the FEM class. Currently, the converters support reading of geometric information and material numbers for many element types including triangles, quadrilaterals, and tetrahedrons. For access to more information that may be contained in NASTRAN and PATRAN neutral files, such as BCs, it is simply a matter of creating a converter inside of the FEM class. Having converters available for standard data formats and implemented inside of the FEM class provides for quick development of robust preprocessing routines that are sometimes neglected in favor of other routines in research applications.

2.3. The finite element class

The hierarchy of the FEM class utilized is shown in Figure 1. This class contains all of the generic finite element functionality required for developing a complete application. Some of the data includes BCs, material data, geometric information, and data file parsing utilities. The FEM class itself is a container class and is used to hold all of the FEM data and to perform utility functions for the stored data.

There are many types of elements available in SPOOCE-FEM and all inherit from the base Element class. This class contains the lowest common data structures and functionality required of all elements. The data stored in this class includes a pointer to the specified material and an Array with the connectivity list. Inheriting from the Element class are the Element2d and Element3d classes, these are for 2- and 3-dimensional elements, respectively.

At the moment these classes do not contain any data of their own but are useful for differentiation between element types. Inheriting from the Element2d class are the Element2dTri3 and the Element2dQuad4 classes for 3-noded triangles and 4-noded quadrilaterals, respectively. These classes contain more specific information such as the number of nodes in each element and the local coordinate computation function. At this time the only 3 dimensional element type is the Element3dTet4 class which is for 4-noded tetrahedrons. These elements do not require local coordinate computations and therefore are only currently used for data storage.

Boundary and initial conditions both inherit from the BC class and are stored in an FEM object as a linked list. This storage allows for easy addition and removal of any IC or BC and is useful for preprocessing in a GUI. The BC class itself only stores an integer that indicates the location of a BC or IC and all functionality thereafter is defined by the specific type of BC or IC. In the PressureBC class, the pressure value is stored as a real and is interpreted as pressure on a node for COMPOSE but could just as easily be pressure on an element edge or face for 2- and 3-dimensional stress analysis respectively. The actual application of the BC is left to the application designer at this time, as it would be difficult to assume how a BC is used in each case.

2.4. The helper classes

There are many common data structures and algorithms that are useful in the FEM. Some of these are included as separate classes and templates that are accessible to all objects. These helper classes include an adjacency list, a list template, an array template, a binary tree, and an XML writer. The adjacency list is especially useful because it is used to compute the connectivity of the finite element mesh and then the sparsity pattern of the stiffness matrices. The adjacency list is also useful for computing external nodes, or those nodes that are on the outside of a geometry, which is in turn used in the Reverse Cuthill-McKee (RCM)
optimization pass. The XML writer provides functionality for creating indented, correct, and easy to read XML files. Arrays are useful containers that enable run time bounds checking for developmental and debug code revisions. Binary trees are used in several cases to provide $O(\log n)$ fast searches and for priority queue construction.

3. Parallel developments

In this section the software developments that pertain only to parallel finite element analysis are discussed. The goal for SPOOCEFEM is to remove many of the hurdles and concerns for a developer of parallel finite element software. To this end the following classes and templates were created to alleviate the burden on the application developer.

3.1. The finite element class

The FEM class contains the functionality for partitioning an unstructured finite element mesh by nodes or elements by utilizing the Metis [9] graph partitioning tool. The partitioning is accomplished at run-time so that any number of processors can be utilized with a simple command line option. One advantage of this method over storing partition information in the data file is that a simulation can be executed in parallel without having to worry about whether or not the proper partition information is available. Once the partition numbers are known by the FEM object, the shared or ghost nodes are computed. All of this is taken care of by the FEM object and therefore the application developer is relieved of this cumbersome task. Generation of partitioning datasets and maintenance of the bookkeeping required for parallel computation are rather tedious and time-consuming operations. This integration of parallel bookkeeping into the FEM class allows for other optimizations, such as the minimization of communication between distant processors [19], to be applicable to all applications. Optimization of the communication routines and the bookkeeping operations themselves are possible since they have been encapsulated in the class.

3.2. The vector and matrix classes

Typically, much of the time spent in FEM software is in the solution of the system of equations. This is the case in isothermal COMPOSE where 80% to 90% of the run time is spent in the solution of the associated linear equations. For this reason, great care has been taken to improve efficiency of the vector and matrix routines. One of the novel approaches in SPOOCEFEM is the inclusion of multiple equation solver packages. The Matrix and Vector class interfaces provided by SPOOCEFEM ensure a consistent and transparent interface to all solvers. These interfaces are implemented to provide maximum flexibility to the developer in choosing an optimal solution method.

It has been shown in COMPOSE that for typical problem sizes ($< 1 \cdot 10^6$ nodes) a direct solver is often the preferred solution method (see Figure 2). The linear system solved for the bar chart in Figure 2 is taken from the COMPOSE software simulation of a part when the resin has infused 95% of the mold. At this point the problem is much more difficult to solve because as the simulation progresses the stiffness matrix contains more nonzero terms throughout. In Figure 2, PTC refers to solvers from PETSc [2], PSP is the PSPASES [8] direct solver, COMPOSE [10] is the current application solver, METIS refers to matrix partitioning performed by Metis, $O3$ and $Ofast$ are compiler optimization levels defined by the SGI MIPS compiler, and “cg” or “cr” refers to conjugate gradient (CG) and conjugate residual solvers, respectively. For example, the PTC.Metis.O3.cg bar refers to the system solution computed by PETSc, partitioned by Metis, compiled with $O3$ optimizations, and utilizing the CG solver.

For various problem sizes it would be optimal to take advantage of the best solver for the situation, whether in a parallel or serial environment. Therefore, the Matrix and Vector classes can store data in multiple formats including those used by PETSc, PSPASES, SPOOLES [1], the Vectorized Sparse Solver (VSS) from the NASA Langley Research Center, and the solver currently used in COMPOSE. The interface to all matrix objects is consistent, and the user is only required to decide during object instantiation which solver type(s) need to be accessible to the matrix and vector object. For example, in the non-isothermal COMPOSE software where symmetric and non-symmetric systems of equations are to be solved, the stiffness matrix associated with the flow simulation may require
PSPASES for the solution of the symmetric system of equations, whereas the thermal problem is non-symmetric and therefore may have a stiffness matrix stored in the PETSc format for access to the Generalized Minimum Residual (GMRES) solver. Once created, all of the interfaces to the matrices are consistent and the storage details are hidden in the application. There is some overhead associated with accessing the matrix data, but since so much time is required by the solution routines this overhead is viewed as minimal [17].

The matrix data for COMPOSE is stored in the Yale or Compressed Sparse Row (CSR) format. The stiffness matrix is partitioned by rows so that each process has a contiguous set of data. This is the only information the user is required to be aware of for the generation of a parallel FEM software in the SPOOCEFEM framework. Once the problem is read and partitions generated with Metis the elemental stiffness matrices are generated and assembled into the global stiffness matrix stored according to the partition information. The problem is then solved in parallel using the requested solver. This description is for partitioning based on nodal domain decomposition.

4. Application development with the SPOOCEFEM framework

In this section, we describe the re-implementation of the FEM software COMPOSE in the SPOOCEFEM framework to improve software reuse and maintenance and to allow for the use of direct solvers. COMPOSE was originally developed using a top-down functional design decomposition coded in Fortran 90. Of primary concern is the ultimate performance of the SPOOCEFEM version of COMPOSE compared with that of the original version. The problem is first described mathematically and shown in discretized form. Next, the application-specific code required for COMPOSE is discussed, and finally, performance of the parallel COMPOSE versions are compared.

4.1. Problem description

The liquid composite molding process, namely the RTM process, is utilized in many U.S. DoD applications for the manufacture of composite components. Originally developed by Mohan et al. [13, 15], the implicit pure finite element mold filling formulation tracks the transient flow front of the resin in the fibrous preform for the manufacture of composite components. A brief discussion of the governing equations follows. Starting from the continuity equation and using the methods described by Mohan, the following representation is derived:

$$\int_\Omega \frac{\partial \Psi}{\partial t} d\Omega + \int_\Omega \Psi \nabla \cdot \mathbf{u} \, d\Omega = 0. \quad (1)$$

where $\Psi$ is the fill fraction and $\mathbf{u}$ is the resin velocity. Using Darcy’s law, given as

$$\mathbf{u} = -\frac{\mathbf{K}}{\mu} \cdot \nabla P; \quad (2)$$

to substitute for $\mathbf{u}$ in equation 1 the following is obtained

$$\int_\Omega \frac{\partial \Psi}{\partial t} d\Omega = \int_\Omega \Psi \nabla \cdot \left( \frac{\mathbf{K}}{\mu} \cdot \nabla P \right) \, d\Omega, \quad (3)$$

where $\mathbf{K}$ is the permeability tensor of the fibrous preform, $\mu$ is the resin viscosity, and $P$ is pressure. After applying the FEM the semi-discretized equation is given as

$$C\dot{\Psi} + \mathbf{K} \mathbf{P} = \mathbf{q}. \quad (4)$$

where

$$C = \int_\Omega \mathbf{N}^T \mathbf{N} \, d\Omega$$

$$\mathbf{K} = \int_\Omega \left( \frac{\mathbf{K}}{\mu} \right) \mathbf{B} \, d\Omega$$

$$\mathbf{q} = \int_\Gamma \mathbf{N}^T \left( \frac{\mathbf{K}}{\mu} \cdot \nabla P \cdot \mathbf{n} \right) \, d\Gamma$$

$$\dot{\Psi} = \frac{\Psi_{n+1} - \Psi_n}{\Delta t}.$$

In equation 5, $\mathbf{N}$ is the elemental shape function and $\mathbf{B} = \nabla \mathbf{N}$. Equation 4 is solved in the structured FEM software COMPOSE utilizing element-based domain decomposition. The same equation is solved with SPOOCEFEM with node-based domain decomposition, otherwise everything else including element types (3-noded triangle, 4-noded quadrilateral, and 4-noded tetrahedral), the stiffness matrix storage format (CSR), and iterative CG based solvers remain the same.

4.2. Application specific FEM code

Some application specific code for COMPOSE is required in order to use the SPOOCEFEM framework. The bulk of the application specific code is included in the various element types used in COMPOSE. For the 2.5-dimensional simulation code there are two types of elements, namely 3-noded triangles and 4-noded quadrilaterals. The 3-noded triangular elements will be described...
in detail with the assumption that all developments are equally applicable to the 4-noded quadrilateral element type. The COMPOSE 3-noded triangular element class is called ComposeElement2dTri3 and inherits from the Element2dTri3 class, Figure 3. From this inheritance the ComposeElement2dTri3 class has access to storage of material data, connectivity, calculation of local coordinates, and reading and writing to XML and HDF5 files. The ComposeElement2dTri3 class expands this functionality with calculation of the elemental stiffness matrix for resin flow in the RTM process. Some other functionality was found to be the same for all 2 dimensional element types in COMPOSE and so a ComposeElement2d class was created to handle calculation of the local permeabilities, the lumped mass matrix, and storage of the elemental thickness. The storage and retrieval of viscosity information is required of 2- and 3-dimensional element types in COMPOSE and so one more class was created for this purpose; ComposeElement. The inheritance diagram for the ComposeElement2dTri3 class is shown in Figure 3 and shows visually the multiple inheritance required for this level of abstraction.

The BCs required for an isothermal COMPOSE simulation are either pressure at a node (Dirichlet) or flow rate at a node (Neumann). These two BC types have already been included in the SPOOCEFEM framework and therefore no extension is required. The application of Dirichlet BCs is handled by the MatrixCSR class for serial and parallel matrices. The Neumann BC is set by the application by modification of the right hand side vector value. The XML parser for COMPOSE is simply a reimplementation of the FEM XML parser with handles for the new tags including elemental thickness, simulation time step, and resin viscosity. By utilizing the FEM XML parser functions for standard finite element data, such as elemental connectivity, nodal XYZ data, BCs, and material properties, very little new parsing is required in COMPOSE.

The final class created for COMPOSE is the ComposeFEM class that contains all of the routines required to execute a COMPOSE simulation. These routines include the time-stepping and mass convergence routines. Two advantages of putting all of these routines into the ComposeFEM class is that the algorithm used for 2- and 3-dimensional analysis and serial and parallel simulations is the same, only the data storage is different. The addition of new element types and performance enhancements are mostly external to this part of the application and so upgrades are simple and accomplished by replacing the old shared libraries with new ones or recompiling and linking with static ones. The addition of other enhancements, such as the “One Shot” [22] fill time computation routine and sensitivity analysis [7], are accomplished for all isothermal COMPOSE simulations with one version of the algorithm. In the past the “One Shot” algorithm required modification to the serial and parallel versions of the COMPOSE code base, but with the object-oriented code-base the new routines work for all types of analysis possible within COMPOSE.

### 4.3. Performance comparison between structured and OOP-based COMPOSE software

Extensive performance testing and code enhancements have been performed on the structured COMPOSE code [14, 18, 20]. It is against this software that the current OOP developments are compared. Simulation results are presented here for the SPOOCEFEM-based COMPOSE software with the direct PSPASES solver and the iterative CG solver from PETSc. The simulation time is the time required for COMPOSE to simulate complete mold filling in a complex geometry with a large number of nodes. The speedup results are scaled to assume perfect speedup with 8 processors. Empirical evidence, based on numerous simulation runs, seems to indicate that at this number of processors the problem decomposition is of a size that maps better to cache and local memory. The times for the 425156 node simulation are presented for 8 processors also, but the problem is somewhat larger and superlinear speedup is observed between 8 and 16 processors.

#### Table 1. Simulation time in seconds for 105722 node RTM simulation.

<table>
<thead>
<tr>
<th>No. CPUs</th>
<th>PSPASES</th>
<th>PETSc</th>
<th>COMPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7070</td>
<td>97973</td>
<td>14945</td>
</tr>
<tr>
<td>16</td>
<td>5362</td>
<td>38261</td>
<td>8944</td>
</tr>
<tr>
<td>32</td>
<td>4439</td>
<td>19143</td>
<td>5742</td>
</tr>
<tr>
<td>64</td>
<td>4891</td>
<td>11841</td>
<td>N/A</td>
</tr>
</tbody>
</table>

In a complete filling simulation it is observed that for 8 processors the PSPASES-based software out performs the
Table 2. Speedup results for 105722 node RTM simulation, assuming perfect speedup at 8 processors.

<table>
<thead>
<tr>
<th>No. CPUs</th>
<th>PSPASES</th>
<th>PETSc</th>
<th>COMPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8.000</td>
<td>8.000</td>
<td>8.000</td>
</tr>
<tr>
<td>16</td>
<td>10.548</td>
<td>20.485</td>
<td>13.368</td>
</tr>
<tr>
<td>32</td>
<td>12.743</td>
<td>40.944</td>
<td>20.822</td>
</tr>
<tr>
<td>64</td>
<td>11.565</td>
<td>66.192</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 3. Simulation time in seconds for 425156 node RTM simulation.

<table>
<thead>
<tr>
<th>No. CPUs</th>
<th>PSPASES</th>
<th>PETSc</th>
<th>COMPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>25405</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>16</td>
<td>22578</td>
<td>N/A</td>
<td>59865</td>
</tr>
<tr>
<td>32</td>
<td>17300</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>64</td>
<td>11554</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>128</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 4. Speedup results for 425156 node RTM simulation, assuming perfect speedup at 8 processors.

<table>
<thead>
<tr>
<th>No. CPUs</th>
<th>PSPASES</th>
<th>PETSc</th>
<th>COMPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8.000</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>16</td>
<td>9.002</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>32</td>
<td>11.748</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>64</td>
<td>17.591</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>128</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

original COMPOSE software by a factor of about 2. This increase in performance is encouraging and performance enhancements in SPOOCEFEM will continue to increase this lead. The PETSc-based code is about 6 times slower than the original COMPOSE software but this can be partially explained by the overhead incurred in the various routines within COMPOSE, besides the linear solver. During a time step of mold filling the stiffness matrix is modified many times, and this modification has a large amount of overhead associated with it in PETSc. This point is being worked on diligently and will hopefully be alleviated in future work on SPOOCEFEM. One point to make here is that the solutions obtained from the PSPASES- and PETSc-based parallel simulations match the serial solution with VSS to machine accuracy, but the original solver in COMPOSE has a noticeable error associated with it. This would mean that the results obtained from the SPOOCEFEM version of COMPOSE are much more reliable and therefore a small loss in performance is not an issue if results are more accurate.

Some quick notes about implementation are required at this point. First and foremost it must be noted that the structured version of COMPOSE utilizes element-based domain decomposition, whereas SPOOCEFEM currently only supports node-based domain decomposition. This is important because in the formation of the stiffness matrices within SPOOCEFEM COMPOSE for the individual processes, some redundant computation is required for elements contained in multiple partitions. Second, the preconditioned conjugate gradient (PCG) solver used by the structured code computes the residual slightly differently than the OOP-based code so that the number of iterations changes as the number of domains changes in the structured code but stays approximately constant in the OOP-based code.

5. Conclusions

The SPOOCEFEM framework discussed in this paper has demonstrated its usefulness in the development of parallel FEM software. The ease of development, software reuse, and the GUI tools are all advantages of utilizing SPOOCEFEM. As with other OOP designs, the SPOOCEFEM framework required a higher investment in time and thought vs. functional programming approaches. After SPOOCEFEM was developed though, implementation of the application COMPOSE was swift and required little additional code. Much of the functionality available in C++ is utilized to make developing FEM applications simpler, but making the base FEM development complex. An example of the extensive OOP work is the use of multiple inheritance by COMPOSE in the development of the element classes. Another example is the use of polymorphism in the element and BC classes, making extensions by individual applications efficient. The greatest complexity comes from the Vector and Matrix classes, where various combinations of data and parallel algorithms are located, but accessed through a consistent interface. This complexity is warranted though, as now the applications developer is relieved of the task of changing solver calls and data storage to accommodate various new packages or solvers.

6. Future development

The original impetus to develop SPOOCEFEM was the current work to extend COMPOSE to model the non-isothermal resin flow and cure kinetics in RTM with parallel computers. Since much work would have gone into parallelizing the code with little usable source code for future projects it was decided to create a reusable framework for
FEM software development. Now that the isothermal version of COMPOSE is implemented in the SPOOCEFEM framework, the job of creating the parallel non-isothermal COMPOSE software is much easier and is usable by other projects yet to come. Other sparse matrix formats besides CSR will be added to SPOOCEFEM in the future along with the associated equation solvers. Element-based domain decomposition requires refinement in SPOOCEFEM so that applications can take advantage of it.

7. Acknowledgments

This research was made possible by a grant of computer time and resources by the Department of Defense High Performance Computing Modernization Program (HPCMP). Parallel versions of COMPOSE were developed through support from the Common HPC Software Support Initiative (CHSSI) project.

References