DFEMwork: a parallel computing framework for material processing

Audet, Martin; Hétu, Jean-Francois; Ilinca, Florin

5th NRC Symposium on Computational Fluid Dynamics and Multi-scale Modeling [Proceedings], pp. 1-10, 2008

DFEMwork: A parallel computing framework for material processing

Martin Audet[1], Jean-François Hétu[1], Florin Ilinca[1]

[1] *Industrials Materials Institute, National Research Council of Canada, 75 de Mortagne, Boucherville, Qc, J4B 6Y4, Canada*
e-mail: jean-francois.hetu@cnrc-nrc.gc.ca, martin.audet@cnrc-nrc.gc.ca, florin.ilinca@cnrc-nrc.gc.ca

Abstract This paper presents a description of dFEMwork - a parallel, fully distributed, and multi physics finite element framework for the resolution of material processing problems. The framework constitutes an infrastructure for scalable resolution of various non-linear coupled PDEs. Its object based structure allows for high level abstractions while hiding most of the complexity of parallelism from the users. The presented work addresses parallelization of the system components. All parallel objects are distributed using the concept of ghost nodes and elements and communicate through message passing paradigm. Particular attention is also given to low level data structures, coding practices and low complexity algorithm and data structures were chosen to insure computational performance. Numerical results are presented for free surface flows requiring dynamic domain redistribution and load balancing. Parallel efficiency of the system, both computational and memory wise, are presented.

Key words: finite element method, molding simulation, parallel computation

1. INTRODUCTION

Full 3D finite element simulation of fabrication processes such as injection molding for real industrial parts is computationally challenging. Many sources of difficulties have to be managed. These problems are transient, non-linear, multiphysics and involve geometrically complex computational domains. Furthermore, to achieve acceptable accuracy, elements should be concentrated in regions where large changes of the solution occur, leading to large number of elements and, depending on the problem discretization, to a large amount of time steps. In the end, parallel computing is impossible to circumvent if one wishes to address such realistic industrial problems within reasonable time.

In principle, finite element methods can be parallelized and their data structures can be distributed quite naturally. Also, the conceptual parallelization of the iterative algebraic systems is achieved although somewhat more challenging. However, the development of an efficient and fully scalable implementation is much more challenging when one has to consider 1) the limited amount of memory available on each processor, 2) the work load dynamically changes and should be well balanced between processors and 3) no single process has complete view of the problem since data is distributed between computers.

Developing programs that can efficiently use large parallel clusters is a difficult endeavor. Several parallel computing frameworks, such as POOMA[1,2], SIERRA[3] and PetSC[4], have been developed during the last decade. These frameworks were essentially developed for two reasons: first, to harness the full computational power of large distributed-memory clusters, and second, to facilitate the development of scalable applications for code developers and researchers untrained in parallel computing. These frameworks are generally targeted toward specific fields of applications, designed for specific numerical methods, or developed to investigate a specific field of computer science. Overall, computational frameworks have proven to be successful within their field of applications, allowing researchers in this specific field to develop parallel applications without to much concern about parallel issues.

DFEMwork is a computational framework that has been developed to facilitate the development of such parallel applications. This framework is an integrated software system that is used for the development of solvers. The motivating multiphysics applications for the framework described in this paper is an ongoing project at NRC-Industrial Materials Institute. The ultimate objective is to develop an integrated software system, DFEMwork, for detailed simulation of injection molding of polymers and casting of metals. The software system is nevertheless sufficiently flexible and modular and has been applied to systems beyond polymer processing, such as the flow of rheologically complex fluids, and viscoelastic stress analysis. We briefly overview the methodology and the software components of this system.

The remaining of this paper is organized as follows. Section 2 presents motivating applications, in particular governing equations, finite element formulations for the simulation of the filling phase of an injection molding process. Section 3 presents the framework architecture, overall design philosophy, data structure design, data distribution concepts and algorithms as well as objects related to finite element computations. Finally Section 4 presents parallel performance results for different types of problems.

2. MOTIVATING APPLICATIONS

This work is motivated by the need to solve complex industrial mold filling applications. Such problems involve the transient solution of free-surface flows coupled with heat transfer. In this section we present a typical injection molding application. The aim is to illustrate and measure the parallel efficiency for the solution of multi-physics, coupled PDEs.

The equations governing the incompressible flows are the Navier-Stokes and continuity equations:

$$\rho\left(\frac{\partial u_i}{\partial t}+u_j\frac{\partial u_i}{\partial x_j}\right)=-\frac{\partial p}{\partial x_i}+\frac{\partial}{\partial x_j}\left[\eta\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right)\right],\ \text{and}\ \ \frac{\partial u_i}{\partial x_i}=0 \qquad (1)$$

Heat transfer is modeled by the energy equation:

$$\rho c_p\left(\frac{\partial T}{\partial t}+u_i\frac{\partial T}{\partial x_i}\right)=\frac{\partial}{\partial x_i}\left(k\frac{\partial T}{\partial x_i}\right)+\frac{1}{2}\eta\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right):\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right). \qquad (2)$$

In the above equations, t, u, p, T, ρ, η, c_p and k denote time, velocity, pressure, temperature, density, viscosity, specific heat, and thermal conductivity, respectively.

For mold filling applications, in addition to solving for the flow equations we have to track in time the position of the interface between the filling material and the air/void inside the cavity. Front tracking is done using a level-set method [5]. The boundary conditions complete the statement of the problem. On the inlet section, both the velocity and inlet melt temperature are imposed. Non-slip boundary conditions are generally imposed on the filled cavity walls and a zero shear stress condition is specified on the free surface.

The application presented in the present work involves the injection molding of a semi-solid metal, that is a metal alloy at a temperature such that we found both the solid and liquid phases. The segregation of particles forming the solid phase is modeled by the diffusive flux model of Phillips et al. [6]. The solid fraction is therefore obtained by solving the transient advective-diffusive equation

$$\frac{\partial \phi}{\partial t}+u_i\frac{\partial \phi}{\partial x_i}=-\frac{\partial N_i}{\partial x_i} \qquad (4)$$

where the diffusive flux $\mathbf{N}$ is given by

$$\mathbf{N}=\mathbf{N}_c+\mathbf{N}_\eta, \qquad (5)$$

with $\mathbf{N}_c$ describing the interaction caused by varying collision frequency and $\mathbf{N}_\eta$ describing the interaction caused by spatially varying viscosity:

$$N_{ci} = -a^2 \phi K_c \frac{\partial(\dot{\gamma}\phi)}{\partial x_i}, \; N_{\eta i} = -a^2 \phi^2 \dot{\gamma} K_\eta \frac{\partial(\ln \eta)}{\partial x_i}. \tag{6}$$

In the above equations a represents the radius of solid particles in the suspension, $\dot{\gamma} = \sqrt{2\varepsilon_{ij}\varepsilon_{ij}}$ is the shear rate, and K_c, K_η are model constants. In the standard form proposed by Phillips et al. [6] they take on the values K_c=0.41, K_η=0.62.

Model equations are discretized in time using a first order implicit Euler scheme. At each time step, the global system of equations is solved in a partly segregated manner as described in refs [5,7-10]. The incompressible Navier-Stokes equations (1) are solved using a Galerkin Least-Squares method [7], the energy equation is solved by a combined SUPG/GGLS (Streamline Upwind Petrov-Galerkin / Galerkin Gradient Least-Squares) method [7], and the front tracking equation is discretized by a SUPG method. The solution algorithm was previously used by the authors to solve a variety of molding applications, as die casting [5], polymer injection molding [8], gas-assisted injection molding [9], co-injection [7] and injection of metal powders [10]. This body of work has shown the ability of the solution algorithm to treat a large spectrum of flow regimes ranging from low velocity creeping flow to high velocity turbulent flows, including heat transfer, free surface and multi-phase modeling.

3. FRAMEWORK CONCEPTS, ARCHITECTURE AND SERVICES

Design Philosophy

DFEMwork design results from an object oriented approach. The rationale for this is based on the belief that an object oriented approach when used judiciously can lead to efficient code (speed and memory) while at the same time delivering the usual benefits associated with object oriented programming that is an increased separation of concerns, a lower level of redundancy, an increased cohesion and reduced coupling between software units which in the end result in higher reusability and reliability. These benefits are the consequences of the higher encapsulation and abstraction level provided by the object oriented approach.

Concurrently, the framework has been designed to hide most of the parallel complexity from the application so that the code developer implementing new PDEs solver is little concerned about parallel computing issues. DFEMwork is implemented in C++ and based on a single-program-multiple-data (SPMD) parallelism model using the MPI message passing standard.

Framework organization

The computational framework is organized into a *Core* library, an *Analysis* library and a *Algebraic System Solver* library. The *Core* library includes all distributed objects and parallel data management functionalities. Its primary function is to provide basic distributed objects for data such as *mesh graphs, Field vectors* and *boundary conditions*. It also provides functionalities specific to distributed objects such as dynamic data remapping, interprocess data transfer. The *Analysis* library includes all objects related specifically to the resolution of PDEs using the Finite Element Method. It includes all the classes needed for the data and algorithmic management of coupled multiphysics simulations. Finally, the *Algebraic System Solver* library provides a collection of parallel preconditioned iterative solvers.

1. *Core* Library

The *Core* library implements basic classes representing large amount of data such as *Field vectors* and *mesh graphs* that must be distributed properly to allow effective parallel operations. This library tries to hide the parallelism and the distributed nature of data by bringing on each process the appropriate information (communications) and performing the necessary index renumbering so that on each process the stored data appears almost as an ordinary "sequential" data structure.

A key concept: Distribution

The *Distribution* class describe how data is shared between processes. It is a key concept in the *Core* library. It assumes a special relationship between the set of data components C and the set of processes P expressed as follows: $P(c) \in S(c)$ for every component $c \in C$ where $P(\cdot)$ is the ownership function associating a process to every component and where $S(\cdot)$ is the storage function associating to every component a non empty subset of processes on which the component is stored. The relation imply that each component is stored on at least one process, the process which own the component. A component can however be stored on other processes.

These two functions allows classifying components into numerous category notably the set of *active* or *local* components representing respectively the subset of components owned or stored on a given process. In addition, the set of *interfacial* (often called ghost) components for a given process is the set of local components which are also local on another process. This later component subset allows to define the *neighbor* subset for a given process as the subset of other processes with which local components are shared. It is with its neighbors that a process communicate normally most.

A *Distribution* class therefore describe on each process the number of components in each category in addition to identifying precisely the interfacial components shared with each neighbor. *Distributions* moreover impose a storage order for the component on each process for simplicity and efficiency: the *isolated* components first (i.e. the components stored only on this process) followed by the interfacial, the active interfacial components being stored first.

With the *Core* library, it is important to note that a component set represents any data vector whose entry type is representable by MPI (ex: non pointer scalar or POD struct). Nodal or elementary data vectors are common example of vector distributed according to a *Distribution*.

Main classes:

In addition to *Distribution*, the *Core* library implements a few basic classes using an object based approach (e.g. little inheritance and no polymorphism):

- *Vector<T>*: a generic distributed vector of components of type T (template) distributed according to a Distribution object. Its main methods allow exchanging interfacial components with its neighbors (in both directions). Often used to store nodal or elemental data.
- *VectorProfile*: a special distributed vector of integers used to hold component indices referring to another distribution. Can be used to store node indices or element indices (referring to the node or element distribution respectively) to represent for example a boundary condition.
- *SparseVector<T>*: a generic distributed vector of type T corresponding and distributed identically to a *VectorProfile*. It is used to store information associated with the *VectorProfile* indices, the value of a boundary condition for example.
- *MeshGraph* : contains the connectivity tables for the elements and maintain consistent distributions for the nodes and elements. The distribution for nodes and elements are interdependent: local nodes are adjacent to local elements and. local elements are adjacent to active nodes.
- *Remap*: establish bijective relationship between every (or subsets of) components of two distributions. This class can be used to keep a connection between components before and after components are redistributed and guide information transfer between the corresponding sets of vectors.

The *Core* library also implements the following important functionalities involving those classes:

- Vector components transfer (in both directions) between two distributed vectors with D_1, D_2 distributions according to a compatible *Remap* object $R_{12} : D_1 \leftrightarrow D_2$.
- Composition of two *Remap* objects: lets D_1, D_2 and D_3 be three distributions and $R_{12} : D_1 \leftrightarrow D_2$, $R_{23} : D_2 \leftrightarrow D_3$ two *Remap* objects between them, the composition *Remap* $R_{12} \circ R_{23} = R_{13} : D_1 \leftrightarrow D_3$ associates components directly from distribution 1 and 3.
- From a *VectorProfile* distributed according to D_1^* containing component indices in distribution D_1 and a *Remap* $R_{12} : D_1 \leftrightarrow D_2$ object, compute a new *VectorProfile* object (including its new distribution D_2^*) containing the indices of the same components (if any) in distribution D_2. Also compute the *Remap* object for the *VectorProfile* object itself $R_{12}^* : D_1^* \leftrightarrow D_2^*$. Note that this new *Remap* object can be used to create a new *SparseVector* corresponding to the new *VectorProfile*. This functionality is necessary to transfer boundary conditions after a domain redistribution.
- From a *MeshGraph* and partition vector indicating on which process each active node should now be active, redistribute the *MeshGraph* and compute the corresponding *Remap* objects for nodes and elements.

When used with a graph partitioner, the *Core* library allows to relatively easily create basic distributed data structures for finite element computations and to easily redistribute them to better equilibrate computations. In this work we used the ParMetis [11] parallel graph partitioner to compute partitions.

Computational efficiency considerations

Data management classes of the *Core* Library have been designed to maximize computational efficiency and are implemented using large vectors of POD similarly to what is done in more traditional languages such as Fortran. The design is primarily an object based approach into which classes have little inheritance and no polymorphism. This design choice, by opposition to arrays of small objects, allowed us to:

- Minimize dynamic memory management overhead (time/space).
- Minimize the number of free memory blocks in the heap. This maintains the speed of each individual dynamic memory operation since these operations takes a time proportional to the number n of free blocks (ex: the common "best fit" algorithm have an O(log(n)) complexity).
- Generate more continuous memory access pattern.
- Keep the overhead of the object oriented approach low and outside of the main loops.
- Allows the compiler to perform more optimization.

In addition, it was decided to relax the data encapsulation principle and give access to the pointers on the vectors. This allowed computational and data processing routines to manipulate data using low level (and efficient) pointers constructs while allowing the classes themselves to concentrate on communication, distributed data management and essential methods and leaving the other functionalities to utility functions or user code. Moreover, since plain vectors are the common implementation of all C++ vector classes and are understood by all languages, working those data structures allows interoperability with other languages (ex: C and Fortran) and makes it easier to interact with other C++ libraries (ex: STL).

Operations on *Core* objects tend to do all transformations and information exchange in a single pass to allow lower complexity algorithm (often linear) and exchanging a few large messages (much more efficient) instead of many small ones. Non blocking communications where send/receive are done early and waits done late with as much work as possible between the twos is also done. This allows hiding of latencies caused by the imperfect process synchronization and overlapping communications with computations when allowed by the underlying message passing layer (hardware / middleware).

2. *Analysis* Library

The *Analysis* Library provides all classes and functions required for the resolution of PDEs using the Finite Element Method. It is based on object oriented concepts relying on inheritance, polymorphism, C++ templates and design patterns to get a few orthogonal sets of classes with clearly defined roles:

- *Domain*: contains a finite element mesh, including shape functions, element connectivity (*Core::MeshGraph)*, node coordinates (*Core::Vector*). A *Domain* instance is essentially a finite element mesh container.
- *Fields*: Nodal or element distributed vectors containing either dense or sparse data associated to a *Domain* object.
- *Equations*: discretized PDEs. These are essentially operators that are used to compute a finite element solution to its PDE. These objects are stateless, meaning that they do not manage solution fields.
- *Analysis*: orchestration classes that manage time integration, solution *Fields*, coupling between different equations, post-processing activation, etc.
- *PropertyEvaluators:* these are classes used for material models and evaluation of material properties on elements.
- *Utility:* other services such as non-linear solvers, post-processing of data, etc.

These classes, their relationships and interactions will be explained in further details for the case of a mold filling analysis. Mold filling analysis typically involves the resolution of a free surface flow problem coupled with an energy equation. Hence, we need to solve the Navier-Stokes equation in the filled portion of the domain, the energy equation in the complete domain and a level-set equation to track the free surface.

Analysis classes

These classes are responsible to manage computational domains and solution *Fields*, manage coarse level time integration as well as the coupling between different solution *Fields*. They are also responsible to provide their results for post-processing and output. *Analysis* classes are either autonomous objects or composed through inheritance. For example, the *MoldFillingAnalysis* class implements the basic mold filling analysis. The *MoldFillingAnalysis* uses two levels of computational domain: one domain which is the complete finite element domain representing the complete mold cavity, and, a series of sub domains, created dynamically to fit to the filled portion of the cavity. These subdomains are created and redistributed when requested by the analysis. Solution *Fields* defined on the complete domain are reduced and transferred to smaller *Fields* defined on the subdomain using the *Core* library remapping functions. Remap functions are bidirectional and are also used to transfer solutions computed in subdomains back to the complete domain.

The *MoldFillingAnalysis* also provides some hook methods for its derived classes. Classes deriving from it can easily implement additional equations and couple them back into the base class using these hooks. (See Figure 1).

Equation classes

Equation classes are responsible to solve a particular PDE using given initial solution *Fields*, material properties and time step. These are operators that are used by the different *Analyses* when they need to compute a solution for a given PDE. The framework implements the resolution of equations (1) to (4) into *NavierStokesEquation, EnergyEquation* and *FrontTrackingEquation*. These classes are responsible for the finite element discretization to use, choice of linearization technique (Picard, Newton, …), computation, assembly and resolution of algebraic systems. These classes use the *Algebraic System*

Solver library to solve their algebraic systems. Furthermore, additional capabilities can be dynamically and transparently added to these equations using decorators.

PropertyEvaluators

Equations are usually coupled through source terms and material properties. In order to make *Equation* classes independent, a general property evaluation mechanism has been developed. These objects, called *PropertyEvaluators*, are responsible to evaluate material properties on each element, while hiding their dependencies with respect to other solution *Fields*. In practice, this means that the *NavierStokesEquation* is not aware that the viscosity model it uses also depends on external *Fields* such as the temperature or solid fraction.

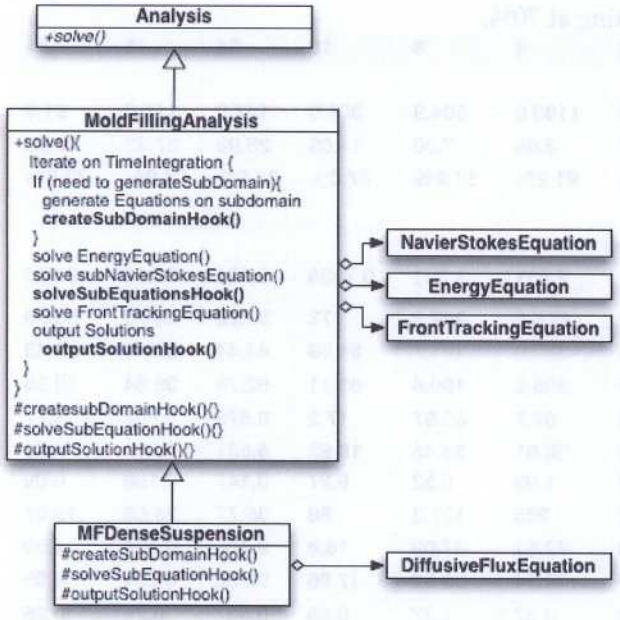


Figure 1: Class design (Template Pattern[12]).

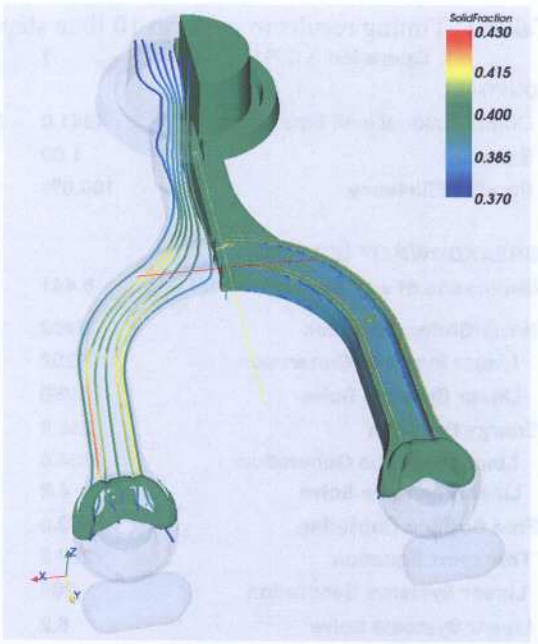


Figure 2: Mold filling simulation.

3. PERFORMANCE RESULTS

Parallel acceleration tests were performed on a 64-node Beowulf cluster. Each node consists of an Intel motherboard with two Intel Xeon 3.4Ghz (FSB-800) and 2GB of RAM. Computational nodes are connected with Myrinet 2000 (PCIXD) cards and switches and MPICH-GM is used.

The parallel efficiency and scalability of the framework is evaluated by performing mold filling analyses with a dense suspension fluid flow model on an industrial U-shaped part. First a complete analysis is performed. This computation provides initial solutions used in parallel efficiency measurements which start from a partially filled cavity. The test problem is shown in Figure 2. The finite element mesh is composed of 1,918,364 tetrahedral elements and has 359,714 nodes. At each node we have four unknowns when solving the momentum-continuity equations (three components of velocity and pressure) and one unknown when solving the energy, front tracking and diffusive flux equations. Ten time steps were executed using 1, 2, 3, 4, 8, 12, 16, 18, 24, 32, 48 and 64 computational nodes (1 MPI process per node). The initial solution corresponds to a state where 70% of the part volume is filled. This level of filling was selected in order to use at most 90% of the available RAM memory on one processor, thus minimizing the occurrence of page swapping. Each equation was solved in corrections using a Newton linearization method with convergence criteria of 10^{-4} on corrections and on residuals (l_2

and l_∞ relative norms). The number of iterations was monitored to ensure that the number of Newton iterations remains the same for all computations.

Figure 2 illustrates, on one side of the part, the filled portion of the mold with the mold cavity shown in transparency. The colors in the filled region represent the solid fraction ϕ, the blue color indicating the lower concentration regions found in the high shear rate zones. Streamlines are shown on the other side of the U-shaped part.

The execution times (wall times) for the generation of Jacobian matrices and resolution of algebraic systems were measured for all equations. Timings needed to create the sub-domains were also monitored. Execution times are shown in Table 1.

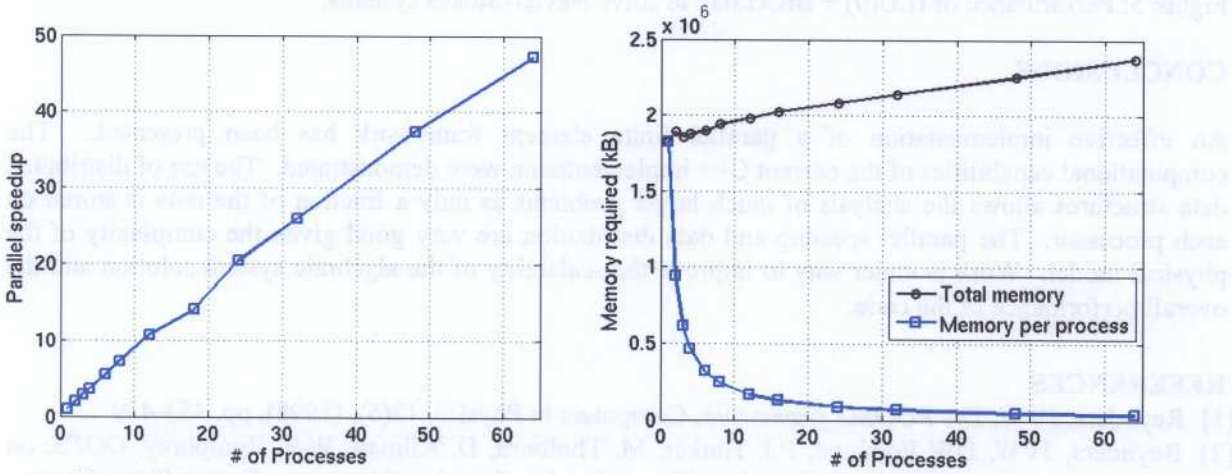
Table 1: Timing results to perform 10 time steps starting at 70%.

Operation \ CPU	1	2	4	8	16	32	48	64
OVERALL								
Computational wall time (s)	4341.0	2260.0	1190.0	594.9	309.0	167.0	116.0	91.9
Speedup	1.00	1.92	3.65	7.30	14.05	25.99	37.42	47.23
Parallel Efficiency	100.0%	96.0%	91.2%	91.2%	87.8%	81.2%	78.0%	73.8%
BREAKDOWN BY EQUATION								
Generation of subDomain	6.441	3.557	2.431	1.301	0.7238	0.421	0.348	0.313
NavierStokes Equation	2462	1292	686.6	342.3	178	97.22	66.75	53.79
Linear Systems Generation	1202	627.6	320.8	161.7	84.89	44.44	30.11	23.43
Linear Systems Solve	1260	664.4	365.8	180.6	93.11	52.78	36.64	30.36
Energy Equation	258.8	131.3	67.7	33.97	17.2	8.678	5.807	4.77
Linear Systems Generation	254.6	129.3	66.61	33.45	16.93	8.531	5.701	4.68
Linear Systems Solve	4.2	2	1.09	0.52	0.27	0.147	0.106	0.09
Free Surface Capturing	939.5	482.2	255	127.2	66	35.77	25.66	19.07
Transport Equation	272.2	140.4	72.51	37.09	18.6	9.489	6.574	4.90
Linear Systems Generation	264	135.9	70.14	35.82	17.95	9.102	6.294	4.65
Linear Systems Solve	8.2	4.5	2.37	1.27	0.65	0.387	0.28	0.25
Diffusive Flux Equation	646.2	336.1	170.2	86.18	45.06	23.82	16.61	13.28

The overall parallel acceleration is shown in Figure 3(a). The parallel speedup is very good given the complexity of the physical model and the distributed nature of the data structures. For example, the computation carried out on 64 processors runs 47 times faster than on one processor. The fact that the data structures are distributed between processors results in an important reduction of the memory used on each processor. Figure 3(b) shows the total RAM memory used (black symbols) and the corresponding amount used on each processor (blue symbols). When using 64 processors, the memory space used on each processor is 49 times smaller that when running on one processor. This means that the present code using distributed data structures can handle much larger problems as only a fraction of the data is stored on each processor.

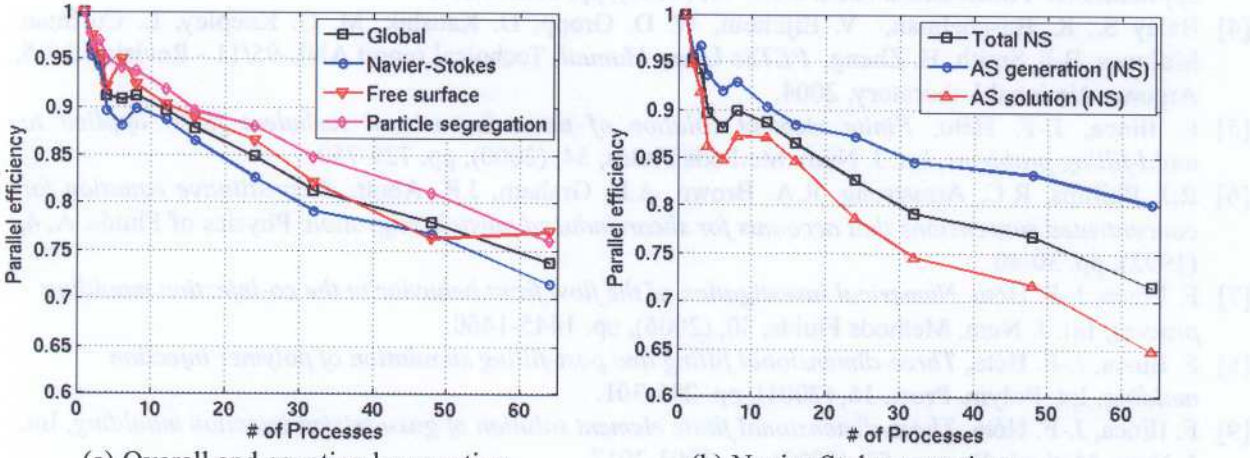
The parallel efficiency of the overall computation and of its segments is shown in Figure 4(a). The monitored time indicates that, as expected, the parallel efficiency decreases as the number of processes increases. Almost the same behavior is recovered for all systems of equations, namely for the Navier-Stokes solution, the free surface equation and the particle segregation equation. The lowest parallel efficiency is observed for the Navier-Stokes equations. This is determined by the fact that the global system of equations resulting form the discretization of the momentum-continuity equations is harder to solve than those arising from the scalar transport equations. This can also be seen in Figure 4(b) which indicates that the parallel efficiency of the generation and assembly of the algebraic system (AS) is larger that that of the AS resolution. For example, on 64 processors the matrix generation phase has a 80% parallel efficiency, whereas the corresponding value for the system solution phase stands at

65% only. This behavior has two main causes. First, as the number of processors increases, the efficiency of the ILU(0) preconditioner decreases as it is constructed separately on distributed matrix blocks and hence more terms are neglected. This is illustrated in Figure 5(a) which shows the total number of iterations needed to solve the algebraic systems of the Navier-Stokes equations (corresponding to the solution of 30 linear systems). The number of iterations increases faster with the number of processors when using up to 4 processors and then the effect is somehow less important. When running on 64 processors the total number of iterations is 42% higher than when using one processor. However, this degradation in performance could be compensated by the use of more efficient preconditioners needing more memory to compute, possibly unavailable at lower process count. The second effect is determined by the unit cost of one iteration as shown in Figure 5(b). As more processors are used, we have on one hand a smaller number of terms in the preconditioner and hence fewer operations to perform, and on the other hand a larger amount of inter-processor communications. The combined effect results in the present case in a slight deterioration of the parallel efficiency per iteration when more processors are used.



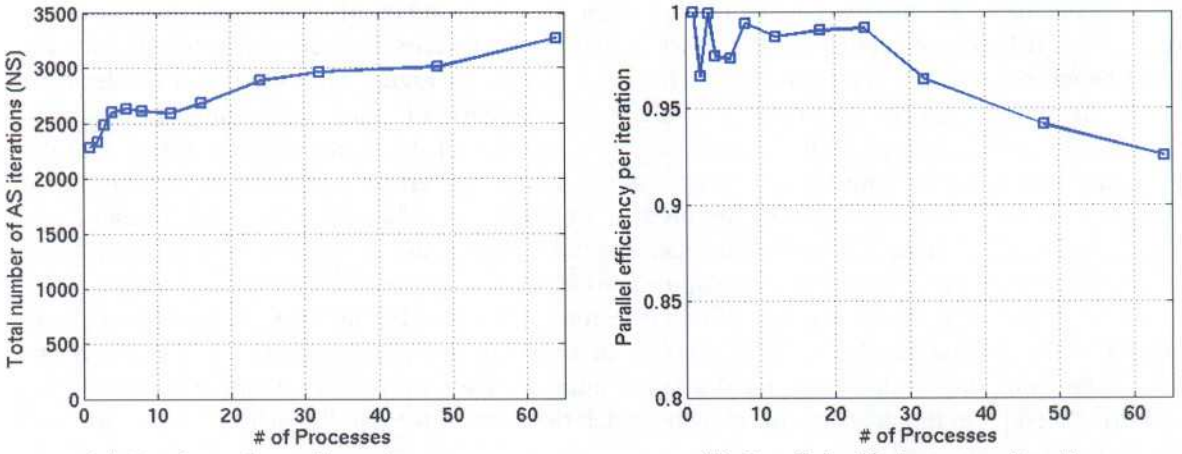
(a) Overall parallel acceleration (b) RAM memory requirements

Figure 3: Parallel performance and memory management



(a) Overall and equation by equation (b) Navier-Stokes equations

Figure 4: Parallel efficiencies



(a) Total number of iterations (b) Parallel efficiency per iteration

Figure 5: Performance of ILU(0) + BiCGStab to solve Navier-Stokes systems.

CONCLUSIONS

An effective implementation of a parallel finite element framework has been presented. The computational capabilities of the current C++ implementation were demonstrated. The use of distributed data structures allows the analysis of much larger problems as only a fraction of the data is stored on each processor. The parallel speedup and data distribution are very good given the complexity of the physical model. Work is under way to improve the scalability of the algebraic system solution and the overall performance of the code.

REFERENCES

[1] Reynders, JVW, *The POOMA framework*, Computers in Physics, 12(5), (1998), pp. 453-459

[2] Reynders, JVW, DW Forslund, P.J. Hinker, M. Tholburn, D. Kilman, W.F. Humphrey, *OOPS: an object-oriented particle simulation class library for distributed architectures*, Comp. Phys. Comm., 87(1-2), (1995), pp. 212-224.

[3] Stewart, J.R., H.C. Edward, *A framework approach for developing parallel adaptive multiphysics applications*. Finite. Elem. Anal. Des., 40, (2004), pp. 1599-1617.

[4] Balay S., K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. Curfman McInnes, B.F. Smith, H. Zhang, *PETSc Users Manual*, Technical report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[5] F. Ilinca, J.-F. Hétu, *Finite element solution of three-dimensional turbulent flows applied to mold-filling problems*, Int. J. Num. Methods Fluids, 34, (2000), pp. 729-750.

[6] R.J. Phillips, R.C. Armstrong, R.A. Brown, A.L. Graham, J.R. Abott, *A constitutive equation for concentrated suspensions that accounts for shear-induced particle migration,* Physics of Fluids, A, 4, (1992), pp. 30-40.

[7] F. Ilinca, J.-F. Hétu, *Numerical investigation of the flow front behavior in the co-injection moulding process,* Int. J. Num. Methods Fluids, 50, (2006), pp. 1445-1460.

[8] F. Ilinca, J.-F. Hétu, *Three-dimensional filling and post-filling simulation of polymer injection molding*, Int. Polym. Proc., 16, (2001), pp. 291-301.

[9] F. Ilinca, J.-F. Hétu, *Three-dimensional finite element solution of gas-assisted injection moulding*, Int. J. Num. Methods Engng., 53, (2002), pp. 2003-2017.

[10] F. Ilinca, J.-F. Hetu, A. Derdouri, J. Stevenson, *Metal Injection Molding: 3D Modeling of Non-isothermal Filling*, Polymer Engng. Science, 42(4), (2002), pp. 760-770.

[11] K. Schloegel, G. Karypis, V. Kumar, *Parallel static and dynamic multi-constraint graph partitioning*, Concurrency and Computation: Practice and Experience. 14(3), (2002), pp. 219-240.

[12] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of reusable object-oriented software*, Addison-Wesley, Boston, (1995), 395p.