A Parallel Solid Mechanics Solver for Multi-physics Finite Element Problems

X. Sáez, E. Casoni, G. Houzeaux, M. Vázquez
Dept. of Computer Applications in Science and Engineering, Barcelona Supercomputing Center (BSC-CNS), 08034 Barcelona, Spain

Abstract

While solid mechanics codes are now proven tools both in the industry and research sectors, the increasingly more exigent requirements of both sectors are fuelling the need for more computational power and more advanced algorithms. While commercial codes are widely used in industry during the design process, they often lag behind academic codes in terms of computational efficiency. In fact, the commercial codes are usually general purpose and include millions of lines of codes. Massively parallel computers appeared only recently, and the adaptation of these codes is going slowly. In the meantime, academy adapted very quickly to the new computer architectures and now offers an attractive alternative: not so much modeling but more accuracy.

Alya is a computational mechanics code developed at Barcelona Supercomputing Center (BCS-CNS) that solves Partial Differential Equations (PDEs) on non-structured meshes. To address the lack of an efficient parallel solid mechanics code, and motivated by the demand coming from industrial partners, Alya-Solidz, the specific Alya module for solving computational solid mechanics problems, has been enhanced to treat large complex problems involving solid deformations and fracture. Some of these developments have been carried out in the framework of PRACE-2IP European project.

In this article a solid mechanics simulation strategy for parallel supercomputers based on a hybrid approach is presented. A hybrid parallelization approach combining MPI tasks with OpenMP threads is proposed in order to exploit the different levels of parallelism of actual multicore architectures. This paper describes the strategy programmed in Alya and shows nearly optimal scalability results for some solid mechanical problems.

1. Introduction

Fluid mechanics simulation codes are generally more advanced in terms of parallel efficiency than solid mechanics codes. One of the reasons is that fluid problems have traditionally required larger meshes than solid mechanics problems. With the increasing need for more advanced modeling techniques involving non-local approaches or multiphysics interactions, Finite Element (FE) solid mechanics codes requirements have traditionally slowed down the parallel efforts aimed at increasing the computational scalability of such codes.

The Alya System [6], which was developed at the Barcelona Supercomputing Center (BSC-CNS), is a Computation Mechanics(CM) code with two main features. Firstly, it is specially designed to run with the highest efficiency standards in large-scale supercomputing facilities. Secondly, it is capable of solving different physics problems, each one with its own modeling characteristics, in a coupled way. These two main features are intimately related, which means that any complex coupled problems solved by Alya will still be solved efficiently.
Alya is organized in modules that solve different physical problems. During the last five years, the main efforts have been dedicated to the fluid mechanics module, covering a wide range of physical problems, such as compressible and incompressible flows, thermal flows, but also excitable media or quantum mechanics for transient molecular dynamics, see [2,4,5,10]. These modules are now being used in production and scalability has been proven on 20000 CPUs using meshes with billions of elements. Only recently the solid mechanics module has received special attention to fulfill the industry new requirements and challenges.

The Alya code has been present in the PRACE benchmark suite since the start of the PRACE projects. The code has been thoroughly tested and proven to run efficiently on many supercomputers such as the BlueGene/Q (JUGENE at JSC, Germany), the Bull Cluster (Curie at CEA, France), and the PowerPC cluster (Marenstrum at BSC-CNS, Spain). As a result of these efforts, last year the Alya team was allocated more than 20 million CPU hours for a project to solve a biomechanics problem on the Fermi supercomputer at CINECA (Italy) through a PRACE regular call.

The development team is currently investing heavily in enhancing the performance of Alya, mainly in the context of PRACE work. In this process, we are mainly addressing algorithmic issues – namely iterative solvers, mesh multiplication, and parallel I/O [3,1,11,12] – along with some OpenMP implementation concerns [13]. This article discusses aspects of implementation in Alya and presents some speedup results obtained in the context of the PRACE-2IP project.

In particular, the present work introduces the solid mechanics module of Alya that has been developed to treat large complex problems involving solid deformations. Using the large-scale PDE solver capability of Alya’s kernel, the solid module was implemented so that any future development of more complex FE techniques should conserve the high scalability of the code. These developments have been carried out along two parallel strategies: first, the Message Passing Interface (MPI) implementation, where the parallelization is based on a sub-structuring technique and uses a Master-Slave strategy [4]. Secondly, the OpenMP implementation: in order to treat many-core processors, the OpenMP paradigm has been implemented; the parallel model is therefore a hybrid MPI/OpenMP model. Some scalability studies will be presented. For implicit methods, the algebraic solver is the key point to obtain a scalable code. In fact, if the work is well balanced, the matrix and right hand side (RHS) assemblies scale perfectly. Iterative solvers are used to solve the momentum equation [15,16]. For symmetric problems (such as the ones studied here), a Conjugate Gradient (CG) method with diagonal preconditioning is used. However, Deflated CG can also be used in order to reduce the number of iterations [3]. Both methods usually perform very well in terms of rate of convergence for solving these equations. For non-symmetric matrices, Alya has also standard iterative solvers, such as GMRES or Bi-CGSTAB method [7], which also scale well.

Parallelization in Alya is mainly done externally via MPI. A domain decomposition strategy has been implemented. It only uses parallelism at task level, which is provided by MPI. For that reason, a hybrid code is developed in order to take advantage of all the levels of parallelism that a multicore architecture offers and also to enable one MPI task accessing all the memory of a node. To exploit the thread-level parallelism of multicore architecture OpenMP has been used in the most time-consuming routines of solid. Additionally, as the communications between threads are via shared memory, a reduction of the communication cost between processors is expected. On the one hand, mesh partitioning is automatically done using METIS [8], creating a set of subdomains that communicate via MPI with a typical Master-Slave strategy. On the other hand, internal loops are parallelized using OpenMP to take profit from memory sharing in multicore clusters.

The article is structured as follows. In Section 2 the governing equations are introduced and the numerical scheme is presented briefly. Section 3 is devoted to describe the structure of Alya, with special attention on the parallel strategy and the linear solver. Section 4 presents the code implementation for the hybrid model, with special emphasis on scalability results and efficiency with respect to purely message passing parallelization. The hybrid implementation of MPI/OpenMP is validated using a three-dimensional test structural test with a highly complex mesh. Some scalability and speedup results are presented. Finally, some concluding remarks are highlighted.
2. Problem statement

The computational solid mechanics problem is solved using a standard Galerkin method for a large deformation framework using a generalized Newmark time integration scheme. Since standard numerical approximations in the field of solid mechanical problems are used and the aim of this paper is not to describe the numerical problem, this framework is only briefly summarized below, for more details see [9].

2.1 The solid mechanical problem

Solid mechanical problems consists on the simulation of a deformable body, which is subjected to some external forces or imposed displacements. For instance, consider the body shown in Fig. 1. The initial configuration, also called the undeformed or reference configuration is shown on the left figure, denoted by $B_0$. In the total Lagrangian formulation all the equations are referred to this configuration. The current or deformed configuration is shown on the right of the figure, denoted by $B$.

Formally, let $\varphi : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ be a function that maps a material point $X \in B_0$ in the reference configuration, to point $x = \varphi(X) \in B$ in the current configuration (see Figure 1). The principle of virtual works, or equation of balance of momentum with respect to the reference configuration, describes the deformation of the body when it is subjected to boundary conditions, either of Dirichlet type (by prescribing displacements $u$) or Newman type (by prescribing external forces in the normal direction, i.e, tractions $P \cdot n_0 = t_0$). It is written as

$$\text{Div } P + b_0 = \rho_0 \ddot{u}, \quad \forall X \in B_0$$

where $u$ are the displacements with respect to the reference configuration (that is, $u = x - X$), $\rho_0$ is the mass density and Div is the divergence operator with respect to the reference configuration, such that $\text{Div } P = \nabla_0 \cdot P$. Tensor $P$ and vector $b_0$ stand for, respectively, the first Piola-Kirchoff stress and the distributed body force on the undeformed configuration.
2.2 Numerical approximation

The finite element method for the spatial discretization is used (see [9,14]). The method consists on partitioning the computational domain into several disjoint elements.

By multiplying by a test function $w$ and integrating over the element, the standard weak form of the balance of momentum equation becomes,

$$\int_{B_0} \text{Div} \mathbf{P} \cdot w \, dV + \int_{B_0} \mathbf{b}_0 \cdot w \, dV = \int_{B_0} \rho_0 \dot{\mathbf{u}} \cdot w \, dV$$

After discretization, we end up with the following problem: find $\mathbf{u}_h$ such that

$$\mathbf{M} \cdot \mathbf{\ddot{u}}_h + \mathbf{f}_\text{int} = \mathbf{f}_\text{ext}$$

where $\mathbf{M}$, $\mathbf{f}_\text{int}$ and $\mathbf{f}_\text{ext}$ are, respectively, the mass matrix, the vector of internal and external forces. These quantities are constructed and assembled from the corresponding element quantities, as classically done in the FE method.

As far as the time discretization is concerned, the generalized Newmark formulation is used. The Newmark scheme depends on three parameters, whose combinations of values lead to implicit or explicit schemes. In the case of an implicit scheme, an iterative Newton-Raphson algorithm is considered in order to solve the governing equations for the displacements, velocities and acceleration at the next time step. For more details, see [9].

3. Alya Large-scale computational framework

Alya’s architecture is modular, grouping the different tasks into kernel, modules and services. The kernel, the core of Alya, contains all the facilities required to solve any set of discretized PDEs (e.g., the solver, the I/O, the coupling, the elements database, the geometrical information, etc.). Each module describes the physical description of a given problem (e.g., the discretized terms of the PDE, the meaning of the boundary and initial conditions, etc.). Finally, the services contain the toolboxes providing several independent procedures to be called by modules and kernel (e.g., parallelization or optimization).

Parallelization

Full details about the code parallelization can be found in [1,11,12]. Briefly speaking, the parallelization is based on a master-slave strategy for distributed memory supercomputers, using MPI as the message-passing library. The master reads the mesh and performs the partition of the mesh into submeshes, or subdomains, using METIS (an automatic graph partitioner). Each process will then be in charge of a subdomain. These subdomains are the slaves. The slaves build the local element matrices and the local right-hand sides, and are in charge of solving the resulting system solution in parallel.

In the assembling tasks, no communication is needed between the slaves, and the scalability depends only on the load balancing. In the iterative solvers, the scalability depends on the size of the interfaces and on the communication scheduling.

As mentioned previously, the momentum and continuity equations are solved with unsymmetric and symmetric iterative solvers respectively. During the execution of the iterative solvers, two main types of communications are required:

- Global communications via MPI_AllReduce, which are used to compute residual norms and scalar products, and
- Point-to-point communications via MPI_SendRecv, which are used when sparse matrix-vector products are calculated.

All solvers need both these types of communications, but, when using complex solvers like the DCG, additional operations may be required, such as the MPI_AllGather functions explained in [3]. In the current implementation of Alya, the solution obtained in parallel is, up to round-off errors, the same as the sequential one all the way through the computation. This is because the mesh partition is only used for distributing work without in any way altering the actual sequential algorithm. This would not be the case if one considered more complex solvers, like the primal/dual Schur complement solvers, or more complex preconditioners, like linelet or block LU (which are implemented as well).
Fig 2. is a schematic flowchart for the execution of a simulation using Alya. The tasks that the master process is responsible for are shown on the left side of the figure with a grey background. The master process performs the first steps of the execution, namely reading the file and partitioning the mesh. Afterwards, the master sends the corresponding subdomain information to each slave process; then the master and the slaves enter the time and linearization loops, represented as one single loop.

Fig. 2.: MPI functions within an Alya iteration
Solid mechanics module

This section briefly describes the solid mechanics module (solidz) presented in this work. The framework presented in Section 2 was implemented in this new module. A large database of element types and integration schemes of different orders is available from previous work in other modules. Well-known constitutive models (e.g., neo-Hookean, Holzapfe-Ogden, Mooney-Rivlin, etc. as well as bridges with usual commercial FE subroutines such as VUMAT) are also implemented.

Fig. 3: solidz module in Alya

Fig 3. shows a flowchart of the solidz module. All the geometrical and physical data of the problem are introduced as input files. Once read, Alya initializes the computation within the solidz module, either in serial or parallel mode. The parallel service must be specified in the input files.

4. Hybrid strategy for parallelization

As pointed out, Alya is parallelized using a domain decomposition method. The elements are distributed between tasks and each task is assigned to a different processor. Due to the private memory access of the processors, the data communication and the synchronization between tasks are done at parallelism task level using MPI. Nowadays, the most important trend in contemporary computer architecture is leaning towards processors with multiple cores per socket, in which the cores have access to the same memory bank. This approach allows running at lower frequencies with better performance than a single core processor. The parallelization strategy for this architecture is multithreading: a Master thread forks a number of slave threads and then the computation is divided among them. The data communication and the synchronization between threads are done using the shared memory inside the multiprocessor. This strategy is known as parallelism at thread level and OpenMP is the standard interface for this model.

In order to take advantage of both parallelism levels, a hybrid version of Alya with OpenMP/MPI is proposed. The main feature of this multicore architecture relies on the fact that both, the shared memory inside the multiprocessor and the message passing between nodes are used. Basically, the structure consists of two steps: firstly, it assigns a task to each node and secondly, the
node creates a thread per core, see Fig 4. Hence, the task is able to access all the memory of the node. As a consequence, there will be a reduction of the communication cost between processors.

Fig. 4: Multicore architecture for a hybrid OpenMP/MPI parallelism

The OpenMP parallelism

To exploit the thread-level parallelism of a multicore architecture, OpenMP directives are added to the MPI parallel code. The first step consists in selecting the most-time consuming routines of the code in total execution time. Among these routines not all of them are susceptible to be parallelized at a thread-level: they must contain a loop structure with independent iterations. It is important that the loop contains a large code body, since a considerable amount of computational work hides the overhead of thread-management.

Table 1 depicts the most time-consuming routines in terms of the total execution time when running an Alya-solidz simulation. In the finite element matrix and right hand side (RHS) assembly routine, sld_elmope, the matrix to be solved in the discrete system is formed by looping over all the elements of the mesh and adding the contributions from that element to the global matrix/RHS. This routine is expensive for a number of reasons, including: significant loop nesting, many matrices have to be assembled and several calls to other subroutines (e.g., constitutive models).
Table 1: Total time of the function with respect to the total execution time.

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Routine name</th>
<th>Description of the routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>93.6</td>
<td>sld_doiter</td>
<td>Controlling the internal loop of the equations</td>
</tr>
<tr>
<td>93.6</td>
<td>sld_solexp</td>
<td>Explicit time stepping scheme</td>
</tr>
<tr>
<td>93.5</td>
<td>sld_solite</td>
<td>Solving an iteration of the equations</td>
</tr>
<tr>
<td>93.5</td>
<td>sld_matrix</td>
<td>Computing the elemental matrix and RHS</td>
</tr>
<tr>
<td>93.5</td>
<td>sld_elmope</td>
<td>Assembling the matrix and RHS</td>
</tr>
<tr>
<td>93.4</td>
<td>elmca2</td>
<td>Computing derivatives at gauss points</td>
</tr>
</tbody>
</table>

In order to introduce thread parallelism in the assembly loop, two main OpenMP directives are used to tell the compiler how to parallelize it:

- **Guided scheduling**: since the workload during each iteration is not the same, iterations are assigned to threads in blocks. In a guided scheduling the block size decreases during each iteration (in contrast with the dynamic scheduling), therefore obtaining a better relation between the thread management time and the balanced workload.

- **Data scope**: the variables that are shared among the iterations are visible to all threads, while the ones that have an independent value among iterations have a different copy per thread.

A critical point on the thread parallelism relies on the so-called race conditions, which might lead to nondeterministic results. Basically, race conditions arise when different threads try to update the same state or array. Operations upon shared states are known as critical regions. It is clear that both, the operation to assemble an element into a local matrix, and the addition of that local matrix into the global matrix, must be thread safe. This data has to be protected by introducing critical regions. Despite the fact that these regions cannot be avoided, it is important to minimize them, because their excessive use can serialize the execution, hence loosing the optimal parallel architecture of the code. One possible solution is to specify different regions names, combined with the use of atomic clauses for a single memory location.

In order to evaluate the gain in terms of time execution that OpenMP architectures provides, a reduced case has been computed. Fig 5. depicts the speedup analysis of the assembly routine for an explicit and an implicit scheme. The results are encouraging: firstly, the performance with one thread is comparable to the sequential version and, secondly, the thread version for both schemes is scaling nearly optimal.
In order to demonstrate the performance and the good scalability of the proposed hybrid OpenMP/MPI strategy, a mechanical test involving a large data structure is considered. It consists of a fusion reactor, which is assumed to have a homogeneous isotropic material behavior, described by the Young modulus $E=210\text{GPa}$ and Poisson $\nu=0.3$. The density of the material is chosen to be $\rho=7958.75\text{ kg/m}^3$.

In the test case, the mesh is initially subdivided into 491415 elements, with an hybrid mesh composed of terahedrals, pentahedrals and hexahedrals, with a complicated geometry, see details in Fig 7. An implicit Newmark scheme is used for the computation. The time step is given by $\Delta t = \xi \frac{l}{c}$, where $l$ is the minimum length of the linear elements, $c$ the speed of sound and $\xi$ is a fixed safety factor, set to 1000. Due to the symmetry of the problem a standard conjugate gradient solver with diagonal preconditioner is used. Fig 8. depicts the displacements obtained after 2000 time steps.
The computation has been performed on Marenostrum III (BSC), which is a supercomputer based on Intel SandyBridge processors that uses a Linux OS. The machine consists of 3028 nodes, two 8-core processors (Intel Xeon E5-2670 cores at 2.6 GHz) per node and 32 GBytes of memory per node. The total Peak Performance is 0.7 PetaFlops. The scalability has been studied as follows: the size of the problem has been maintained fixed while the number of cores used in the simulation has been increased (hard scaling). Given the architecture of Marenostrum III, the number of threads per MPI task is 16, since each node consists of two 8-core processors, and the number of MPI tasks will be the number cores divided by 4. Scalability results of the hybrid version are presented in Fig 8. The hybrid method shows optimal performance but also profiting the advantages that a hybrid OpenMP/MPI parallelization provides.
6. Conclusions

This paper has presented the solid mechanics module of Alya code for solving linear and nonlinear continua and structures problems with large deformations. The main contribution has been the implementation of a new hybrid version of the code with nearly identical behavior of the original Alya code. To enhance the massively parallel performance OpenMP directives have been added to the current MPI Alya code. The technique has been tested successfully with a three-dimensional structural test with different solvers, showing optimal scalability results.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU's 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493. The work is achieved using the PRACE Research Infrastructure resources Marenostrum III (BSC-CNS), Barcelona.

Bibliography

6. Alya system. URL http://www.bsc.es/computer-applications/alya-system
8. Metis, family of multilevel partitioning algorithms. URL http://glaros.dtc.umn.edu/gkhome/views/metis