A parallel computing model for the acceleration of a finite element software

Serge Moto Mpong, Pierre de Montleau, André Godinas, Anne Marie Habraken
Université de Liège, Département M&S, Chemin des Chevreuils, 1, 4000 Liège, Belgique
Tel : (32) 4 366 9332
Fax : (32) 4 366 9192

Abstract

This paper presents the parallelization model used for the non-linear finite element software LAGAMINE. The proposed model is based on the use of a coarse grain approach for the parallelization of the assembly of the stiffness matrix. For the solver of the generated linear problem, we used a parallel direct solver of Gauss type. Finally, we present a discussion of the results of our approach on an example of deep drawing with description of the texture evolution, and on an example of upsetting of an elastic cube. For the deep drawing simulation, the computational time for the assembly of the stiffness matrix is more important than the time spent by the solver for the solution of the generated linear problem. Therefore this example is important and had determined our strategy of parallelization.

Keywords: parallel computing; finite element; metal forming simulation.

1. Introduction

The Finite Element code LAGAMINE is developed by the MSM department of the University of Liège since 1982, and has been adapted to numerous finite elements and constitutive laws. Its modular architecture has been efficient to allow improvements towards applications very far from the initial goal: rolling simulation, soil simulation. The present researches on micro-macro modelling increase the requirement of CPU time. If large size simulations with 100 000 degrees of freedom are not presently aimed, even with 5000 DOF, the CPU time is now a strong limitation because of the complexity of the constitutive laws.

Nowadays, parallelization of software is often divided into two main approaches: the coarse grain approach and the fine grain approach. For Finite Element software, the coarse grain method is usually characterized by the use of the mesh partitioning. In this case, the mesh is partitioned in a number of sub-meshes generally corresponding to number of processors on which the computation is to be done. Then, the same program is executed on each part of the mesh by one processor. This approach is the SPMD (Single Program Multiple Data) one [7]. In the fine grain approach, there is no partition of the mesh. The parallel program is made by the parallelization of instructions issued of the analysis of dependency between the instructions and in the loops. The efficiency of all these approaches depends on the organization of the memory and on the used exchange protocol (MPI, PVM, or OpenMP) [5]. Our approach is intermediate between these two main categories of parallelization. It is characterized by the parallelization of the assembly of the stiffness matrix using a coarse grain approach. Because of the architecture of the available parallel computers, which are share memory, we decided to use OpenMP protocol.

On the following lines, we will summarize the different modifications of the code and their effects on CPU time for a deep drawing and an upsetting of a cube examples.

2. Constitutive equations

2.1. Equilibrium equations

Relation (1) presents the equilibrium equation in the simple case, when we have neither volume and nor external forces. Solid metallic alloys are assumed deformable bodies. Their governing equations consist in the elasto-plastic constitutive equations (1-3), defining the Cauchy stress tensor $\underline{\sigma}$, with respect to the strain rate tensor $\underline{\dot{\varepsilon}}$, with $\mathbf{v}$ as the velocity field, $\bar{\varepsilon}$ as the strain field, K_0, p_1, p_2, p_3, p_4 as temperature dependant parameters. The

Von Mises equivalent stress, strain and strain rate are identified by $\overline{\sigma}$, $\overline{\varepsilon}$, $\dot{\overline{\varepsilon}}$.

$$\nabla.\underline{\sigma}=0 \quad (1)$$

$$\overline{\sigma} = K_0 \overline{\varepsilon}^{p_4} \cdot exp\left(-p_1 \overline{\varepsilon}\right) p_2 \cdot \sqrt{3}\left(\sqrt{3} \cdot \dot{\overline{\varepsilon}}\right)^{p_3} \quad (2)$$

$$\underline{\dot{\varepsilon}}=\frac{1}{2}\left(\nabla v+(\nabla v)_T\right) \quad (3)$$

2.2. *Time discretisation*

For the computation of the solution of the non-linear system of equations, an incremental method is used. The strategy used is the subdivision of the load in time steps and its application step by step. If F is the external load applied, we apply $\lambda_1 F$ with $0<\lambda_1<1$. On each step, the constitutive equations are integrated in function of the strain rate. The iterations are done on the choice of the displacement until the convergence of the Newton-Raphson algorithm used for the solution of the equation system. Then the load is increased to $\lambda_2 F$ with $\lambda_1<\lambda_2\leq 1$ until $\lambda_n=1$. The equilibrium is then realised when the internal nodal loads are equal to the nodal applied loads.

2.3. *Weak form of the mechanical equations*

The equilibrium equations are integrated using the principle of virtual power. The solution of the equilibrium equations is equal to the solution for which the total energy of the system is minimised. That solution is also equal to the solution that verifies the principle of virtual power.
According to the above-mentioned time integration scheme, at each time step, equations (1) to (3) have to be solved for $x^{t+\Delta t}$ on the updated geometry $\Omega^{t+\Delta t}$.

Taking into account the previous time discretisation, the application of the principle of virtual power method to the above mentioned momentum equation (1), to which the external and the volumic loads have been added, leads to the following weak form:

$$\forall \underline{\delta u} \in \vartheta_0$$

$$\int_{\Omega^{t+\Delta t}} \underline{\sigma}^T \underline{\delta\varepsilon}\, dV = \int_{\Omega^{t+\Delta t}} \rho\, \underline{f^T \delta\, u}\, dV + \int_{\partial\Omega^{t+\Delta t}} \rho\, \underline{t}^T \underline{\delta\, u} dS$$

(4)

With $\underline{\delta u}$ *the virtual displacement,*
$\underline{\sigma}$ *the Cauchy stress vector,*
$\underline{\delta\varepsilon}$ *virtual strain compatible with the virtual displacement,*
$\underline{\rho}$ *density,*
$\underline{f}$ *external volumic load,*
$\underline{t}$ *external surface load,*
V and S respectively the current volume and the current surface.

3. Parallelization of the assembly loop

3.1. *Different kinds of parallelism used in the bibliography for finite element software*

There are many kinds of classification for the different models of parallelization [5].
In function of the granularity, which characterises the parallelization finest, we have two different classes of parallelism: the fine grain parallelization in which the elementary unit parallelisable is an instruction and the coarse grain parallelization in which instructions are packed into tasks or processes which are the elementary parallelisable unit. Models of parallelization and parallel programming languages usually use one of those two models.
We can also classify parallel programming models by the exchange of communication model between the different tasks or processes. Then, we have explicit exchange models of communication and implicit exchange models of communication. In both cases, the program is written using a classical programming language, and processes execute the same program or parts of the same program. For programs using an explicit exchange of communication model, all the variables are of private type (that means that they are not shared by the different tasks) and remain in the cache memory of each process. Data are then exchanged between processes using an explicit

call of particular subroutines. It is the case of parallelization models using MPI (Message Passing Instruction) or PVM (Parallel Virtual Machine). Meanwhile, for programs using an implicit exchange model, there is no call of subroutine for exchanging data. Variables can be private and remain in the cache memory of each process or shared and are packed in the heap memory or in a part of the memory accessible by all the processes. Communications between processes are then done implicitly by a direct access to these memory zones. This is the case for OpenMP model of parallelization.

For scientific calculus software such as finite element codes, we usually have two main approaches of parallelization, in function of the division or not of the data:

- The first approach is the SPMD (Single Program Multiple Data), which can be called division of data method (usually it is the mesh which is divided) in nearly equal packs usually equal to the number of processors of the parallel machine. The same program is then executed on each processor on a part of the data. The exchange of information between the processes is then made using either an explicit exchange model (MPI or PVM) or an implicit exchange model (OpenMP). The parallelism is here in the partition of the data, and the protocol is used here for the exchange of data between the processes. This approach is the one generally used for finite element software [2]. It requires few modifications in the original program, except in the linear system resolution algorithm. The parallelism effort is here concentrated on the partition of the mesh and on the system resolution algorithms. The partition of the mesh is done so that the boundary regions between adjacent submeshes are regular and have few nodes, in order to limit the number of communication at the interfaces [7]. For the solution of the linear system, there are two different approaches: the first one is a global approaches, which consists in the resolution of the global system either by a distribute direct algorithm, or by a distribute iterative algorithm [7]; the second approach, which is used in domain decomposition algorithms, consists in the solution of a restricted part of the global linear system on each domain and the solution of a continuity problem at the interface so that the solution of restricted problems converges to the solution of the global problem [1][8]. In those two main approaches, there is the necessity of explicit or implicit communication either for matrix/vector operations of the global system, or for the solution of the continuity problem at the interfaces.
- The second method is generally characterised by the use of parallel languages. It consists in the use of the parallelism of the language to execute simultaneously independent instruction of the program. The goal here is to change the sequential order of the program. It requires a dependency analysis of the program, in order to exhibit independent instructions and to preserve its semantic. In general this approach of parallelism is a fine grain method. It can also be done by using an exchange of communication protocol such as OpenMP, which creates tasks at the beginning of a parallel region. The sequential program is then divided into tasks. The communication model is generally implicit. These tasks can then be executed in parallel, each on a processor, if their number is lower or equal to the number of processors. This model of parallelism is generally used on MIMD architecture machine and on shared memory architecture parallel machine.

3.2. Choice of a method for the parallelization

For the parallelization of our software, the second approach has been chosen. The first one requires indeed an important effort for the partitioning of the mesh and for the parallelization of the direct solver. As our parallel machines are shared memory machines, we have decided to use OpenMP protocol for the communication exchange. The

strategy of parallelisation is then to exhibit independent instructions and to execute them simultaneously, using OpenMP directives.

3.3. Steps of the parallelization

The first thing done was an analysis of the computational time on our examples which require a lot of time. According to that analysis, the areas of the program which use most computational time for our complex applications are the constitution of the assembly matrix for the deep drawing simulation with texture evolution and the resolution of the linear system for other applications like casting. First, we examined the assembly loop of the stiffness matrix in order to parallelise its different iterations. Thus, we analyse the data flow, which permits to determine the order of the definition and the use of a variable. That determines the dependency between different instructions. Simultaneously, the analysis of the control flow had been done, in order to transform control instruction of the program so that it alloys its parallelization.

3.3.1. Different types of dependency

There are three different types of dependencies:

- The "read after write" dependency for which it is necessary to wait that a variable is assigned before being used in another instruction. This induces a sequentiallity in the program, so the necessity to execute the first instruction (the assign) before the second one (the use).
- The "write after write" dependency which is the dependency between two consecutive instructions, which assign a value to a same variable. The problem here is the memory management, because it is not possible to write simultaneously at the same place of the memory.
- The "write after read" dependency which is the dependency between two instructions when in the first one the value of the variable is used and in the second one there is an assign of the same variable. So it is not possible to modify the value of this variable while its previous value is being used.

From the previous dependencies, the "read after write" dependency is the only one which induces a sequentiality in the program. For it, the assign instruction must necessary be done before the use of its value. For the other dependencies, also called artificial dependencies, it is possible to reorganise the program so that those dependencies are removed.

3.3.2. The OpenMP protocol

It is an industrial standard based on an API (Application Programmer Interface) for shared memory or virtually shared memory machines. It is possible to use it either for coarse grain parallelism (like domain decomposition) or for fine grain parallelism (loops or instructions). Nowadays, this standard is highly implemented on Fortran90, C and C++ compilers of many constructors of parallel shared memory machines. There is only one process which is created and, parallel zones are executed by creation of tasks. Parallel zones are either different iterations of a loop packed to be executing simultaneously and shared between the different tasks, or the execution of several occurrence of a subroutine. Parallel zones can also be parts of the program executed simultaneously. The sharing of the work between the tasks can be done statically by the programmer, or dynamically by the system. The program is sequential at the beginning, and this until the first parallel zone. Then, it creates tasks, which are executed in parallel. At the end of parallel zones, only the principal task pursues its execution. The model of communication is implicit. The programmer can do the introduction of OpenMP directives in the program. They are view as comments if the compilation is done without the use of OpenMP flags or by a compilator which does not support this standard [3].

In a parallel zone, the programmer defines the attribute of all the variables.

- **The « SHARED » attribute**

The variables having this attribute are shared between all the tasks. That means that they are accessible by all the tasks. In Fortran, variables defined in common blocks or in modulus have this status by default. Dynamic arrays allocated outside of parallel zones have also this attribute by default.

- **The « PRIVATE » attribute**

Variables having these attributes are copied on each task memory zone. So, they are different from a task to another. At the beginning of a parallel zone, the value of a private variable is not defined, even if it had been assigned before. At the end of a parallel zone, the value of the variable is not transferred to the following sequential zone. Local variables and automatic arrays have this attribute by default.

- **"FIRSTPRIVATE" and "LASTPRIVATE" attributes**

The variable must have the private attribute. In the case of "FISRTPRIVATE" attribute, each private instant of the variable is initialised by the value of this variable in the previously sequential zone. In the case of "LASTPRIVATE" attribute, only use for "DO" and "PARALLEL DO" directive, the last value of the variable is transferred to the following sequential zone.

3.3.3. Dependency Analysis of the assembly loop

The assembly loop is a loop over the elements of the mesh. For each element, the associated law is called and the constitutive equations are integrated to compute the internal forces and the tangent matrix. The parallelization method is simply the build of a PARALLEL DO loop in order to share between the available processors the different iterations of the loop. The objective of the analysis of dependency is the determination of the order of definition and of use of the variables. They are then used for the solution of the system of dependency encountered between the instructions. Here, we will focus on variables with "SHARED" attribute because there are shared by the different parallel tasks working on the iterations of the assembly loop of the stiffness matrix.

- **Solutions of the « read after write » dependencies**

Normally, this type of dependency indicates sequential instructions in the program. So, we have been obliged to remove those variables and to modify the programming of the software in order to have parallelisable instructions. It is what we have done for variables used as counter in the assembly loop and declared outside this loop. For variables updated as counter in the assembly loop and only used after this loop, we use the fact that the addition is transitive. So there were not suppressed and, at the end of the loop, these variables contain exactly the truth result, if the different parallel instructions are done at once and the same time. Therefore we use an OpenMP critical instruction, the "ATOMIC" directive, to oblige the system to authorize only one task to update at once that variable. Thus those instructions are not executed at the same time by different tasks. For arrays in which there were components modified in the loop, we create new variables for those components and we assign them the private attribute.

- **Solutions of the other system of dependency**

For the other systems of dependency, new variables were created and used, so that the two instructions could be executed simultaneously.

4. Solver of the linear system of equations

For the solution of the linear system of equations generated, we decided to adapt a parallel gauss type solver developed at the Universitat Politénica de Cataluña in Spain.

4.1. *Storage of the matrix in the memory*

The stiffness matrix needs an important storage space in the random access memory. The control of this memory must be studied for the optimisation of the computer work, because the access time to the data depends on the area of the memory in which they are stored. Two type of storage exist for direct methods:

- In the first group, there is a reservation of the memory space necessary for the storage of the matrix and for the additional resolution term of the factorisation. The skyline method in this approach is usually used. The former resolution method of our software uses this type of storage.
- In the second group of methods, a memory zone containing only non zero elements of the matrix is defined. This approach is generally called Morse storage. Thus, the necessary storage space is less important, but the addressee of the elements is more complicated (indirect addressee). The

factorisation phase needs the reservation of new memory zone. One of the methods used nowadays is the symbolic factorisation, which determines the non-zero element of the profile of the matrix. This method has been implemented in our software.

For the iterative methods, the problem of data storage is less important, as the operations are easily parallelised. The terms of the stiffness matrix can be lead in the memory in assembly form by Morse storage like in direct method, or conserved element by element. In this last case, there is a more important use of memory, but there is less access conflict when several processes are working together, each of them using a part of the finite elements.

4.2. Description of our parallel solver

Our new parallel solver is a parallel gauss type solver which has been developed in the CEPBA department of the Universitat Politénica de Cataluña, and adapted to the solution of our problem. This parallel direct solver can be describe by the following points:

- the Morse storage is used, so that only non zero elements are stored. Thus, the necessary storage space is reduced;
- A METIS software is used for the renumbering of the equations. That renumbering optimises as less as possible the number of operations that will be performed during the factorisation [6];
- The symbolic factorisation is performed. This operation gives the structure of the L and U matrix that will be obtained during the factorisation and permit to foresee their storage.

These three steps are performed only once at the beginning of the simulation.

5. Applications

5.1. Deep drawing with evolution of the structure

We present here the results obtained for the deep drawing simulation example, realised as part of the present PhD thesis of Laurent Duchêne in our department. The material is steel with a small elasticity limit. The mesh has 4020 finite elements nodes and 1504 volumic elements, for a total of 7000 degrees of freedom. The simulation is a three dimension mechanic computation, which consists in the deformation of the initial steel part by a punch until the final shape. The material is characterised by the elasto-plastic Minty-law describe in [4]. The computation, after 10 days on only one processor was not finish on a Silicon Graphix SG3800 machine, which has 64 processors. On 8 processors of this same machine, it has required 2 days of computation with a speed-up of six. The efficiency was about 93% on two processors, 84% on four processors and 73% on eight processors, as shown on Table1.

Table1: Results obtained for the deep drawing case on a SG3800 machine

Number of processors used	1	2	4	8
Speed-up	1	1.86	3.36	5.91
Efficiency (%)	100	93	84	73

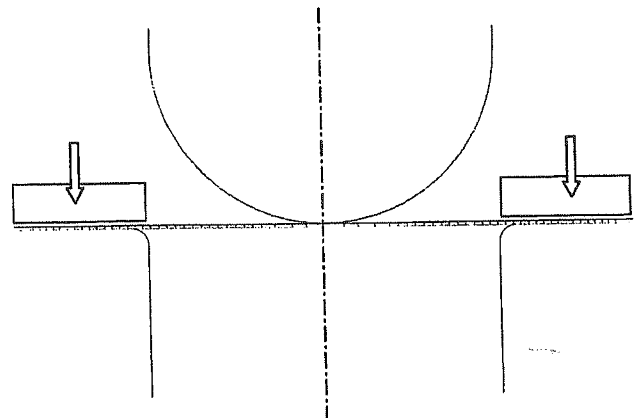

Figure1: Experimental set-up of the deep drawing

5.2. Upsetting of an elastic cube

The second example is an upsetting of a cube. The mesh of this cube has 4096 finite elements nodes and 3375 volumic elements, for a total of 11500 degrees of freedom. The behaviour of the material is characterised by a linear elastic law. The computation has been done on one processor, two processors, 4 processors and

eight processors of a Silicon Graphix SG3800 computer, which has 64 processors. The efficiency was about 93% on 2 processors, 85% on 4 processors and 75% on 8 processors, as shown on Table2.

Table2: Results obtained for the upsetting of the elastic cube

Number of processors used	1	2	4	8
Speed-up	1	1.82	3.4	6
Efficiency (%)	100	91	85	75

6. Conclusion

We have presented a model of parallelization of a finite element software simple to realise. Our approach was first of all the identification of areas of the software that need a lot of computation time for our big examples. Then, those zones have been parallelised. These zones were the assembly loop of the stiffness matrix and solution of the generated linear system. For the assembly loop of the stiffness matrix, we analysed and we eliminated the dependencies between the different iterations of the assembly loop. As our parallel computer is shared memory, we used OpenMP protocol. For the solution of the linear system generated, we adapted a parallel direct solver of Gauss type developed at the Universitat Politénica de Cataluña in Spain. For the moment, we have result in accordance with the literature, and our simulations can be done within acceptable CPU times.

7. References

[1] J. Bramble and J.-E. Pasciak, *A domain decomposition technique for stokes problems*, App. Num. Math., pp. 251-261 (1990)

[2] G.-F. Carey, *Parallelism in finite element modelling*, APNUM 2, pp. 281-288 (1996)

[3] J. Chergui, P. F. Lavallée, J.-P. Proux, C. Roult, *OpenMP parallélisation multitâche pour machine à mémoire partagée*, IDRIS, Janvier 2001.

[4] L. Duchêne, A. M. Habraken and A. Godinas, *Influence of steel sheet anisotropy during deep-drawing process*, The 4th international ESAFORM conference on Material Forming, pp 461-464, April 23-25 (2001)

[5] C. Farhat, *Which parallel finite element algorithm for which architecture and which problem?*, Eng. Comput., 7:186-195 (September 1990)

[6] George Karypis and Vipin Kumar, *A Software Package for the Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing orderings of sparse Matrices*, Department of computer science, University of Minnesota, September 20th, 1998

[7] S. Marie, *Un modèle de parallélisation SPMD pour la simulation numérique de procédé de mise en forme des matériaux*, Thèse de doctorat de l'Ecole des Mines de Paris (1997)

[8] F.-X. Roux, *Calculateurs massivement parallèles MIMD et méthodes de résoluiton par sous-domaines*, Onéra, division calcul parallèle.

8. Acknowledgement

The authors would like to thank The Belgium Ministère de l'Enseignement Supérieur et de la Recherche, the Belgium Fonds National de Recherche Scientifique and The Région Wallone for their financial support. They are also deeply indebted to the CEPBA Department of the Universitat Politénica de Cataluña for their help in the discussion for the amelioration of the software and to Professor Serge Cescoto of the University of Liège who permits this collaboration.