ELSEVIER

Advances in Engineering Software 36 (2005) 361–373

ADVANCES IN ENGINEERING SOFTWARE

www.elsevier.com/locate/advengsoft

Parallelization of an object-oriented FEM dynamics code: influence of the strategies on the Speedup

Olivier Pantalé*

LGP CMAO, Ecole Nationale d'Ingenieurs, 47 Ave d'Azereix, BP 1629, Tarbes Cedex 65016, France

Received 6 February 2004; received in revised form 26 August 2004; accepted 4 January 2005

Abstract

This paper presents an implementation in C++ of an explicit parallel finite element code dedicated to the simulation of impacts. We first present a brief overview of the kinematics and the explicit integration scheme with details concerning some particular points. Then we present the OpenMP parallelization toolkit used in order to parallelize our FEM code, and we focus on how the parallelization of the DynELA FEM code has been conducted for a shared memory system using OpenMP. Some examples are then presented to demonstrate the efficiency and accuracy of the proposed implementations concerning the Speedup of the code. Finally, an impact simulation application is presented and results are compared with the ones obtained by the commercial Abaqus explicit FEM code.
© 2005 Elsevier Ltd. All rights reserved.

Keywords: Non-linear finite-element; Large deformations; Plasticity; Impact; C++; Object-oriented programming; OpenMP; Parallel computing

1. Introduction

Crash and impact numerical simulations are now becoming widely used engineering tools in the scientific community. Accurate analysis of large deformation inelastic problems occurring in impact simulations is extremely important due to the high amount of plastic flow. Number of computational algorithms have been developed, and their complexity is continuously increasing. Some commercial codes like Abaqus-Explicit [1] can be used in such a field. With the increasing size and complexity of the numerical structural models to solve, the analysis tends to be a very large time and computational resources consuming. Therefore, the growth of the computational cost has out-placed the computational power of a single processor in recent years. As a consequence, supercomputing involving multiprocessors has become interesting to use. Supercomputers have also been replaced by some cheaper microprocessor-based architectures using shared-memory processing (SMP) or distributed-memory processing (DMP). In SMPs, all processors access the same shared memory as shown in Fig. 1, while in DMPs each processor has its own private memory.

The parallelization techniques in FEM codes can be classified into two categories. The first-one concerns DMPs where Message Passing Interface (MPI) is well established as high-performance parallel programming model. Many applications can be found in the literature dealing with parallel dynamics FEM codes using the MPI [2,3]. MPI is a scalable parallel programming paradigm because the user has to rewrite a serial application all at once into a domain decomposed program. Parallelization of codes within SMPs computers is mainly carried out using special compiler directives. Each manufacturer provided their own set of machine specific compiler directives leading to well known problems concerning portability of such codes from one architecture to another. The OpenMP [4] standard was designed to provide a standard interface in Fortran and C/C++ programs for such a parallelization. Hoeflinger et al. [5] explored the cause of poor scalability with OpenMP and pointed out the importance of optimizing cache and memory utilization in numerical applications. The use of OpenMP gives a limited control over the threads compared to the more fundamental Pthreads standard [6]. However, OpenMP is more easy to learn and use than Pthreads leading to a lower development time. Portability and efficiency of OpenMP over Pthreads is also better.

* Tel.: +33 5 6244 2700; fax: +33 5 6244 2708.
E-mail address: Olivier.Pantale@enit.fr.
URL: http://www.enit.fr/recherche/lgp/cmao.

0965-9978/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.advengsoft.2005.01.003

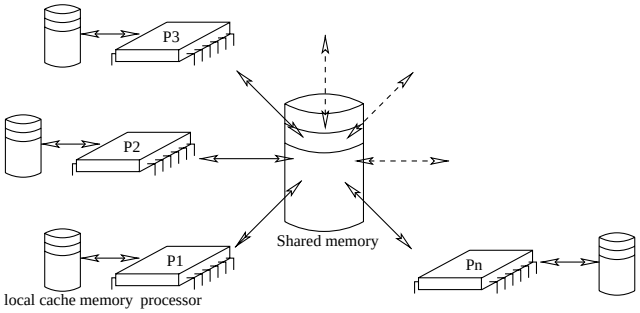


Fig. 1. Shared-memory processing (SMP) architecture.

The most common approach in transient dynamics simulations is to use a Domain Decomposition Method (DDM) [2,3,7]. In this approach, the structure is decomposed into a set of sub domains. The final solution of the problem usually requires local computations over each subdomain (this leads to the parallel problem) and computation of the global interfacial problem using various techniques. Our approach in this paper is quite different since we focused on local parallelization techniques to be applied on some CPU time consuming subroutines inside the explicit integration main loop of the program. In this approach, only the internal force vector and the stable time-step computations are parallelized using some OpenMP parallelization techniques, leading to a more efficient code without the need of DDM.

In this paper, some aspects regarding the parallel implementation of the Object-Oriented explicit FEM dynamics code DynELA [8,9] using OpenMP are presented. In a first part of this paper, an overview of the FEM code is presented with some details concerning the explicit integration scheme, the stable time-step and the internal force vector computations. In a second part we present some of the parallelization techniques used to Speedup the code for a SMPs architecture. A benchmark test is used in this part to compare the performance of the proposed parallelization methods. Finally, the efficiency and accuracy of the retained implementations are investigated using a numerical example relative to impact simulation.

2. Overview of the FEM code

2.1. Basic kinematics

In this work, the conservative and constitutive laws are formulated using an updated Lagrangian formulation in large deformations. In a Lagrangian description, let $\vec{X}$ be the reference coordinates of a material point in the reference configuration $\Omega_X \subset \mathbb{R}^3$ at time $t=0$, and $\vec{x}$ be the current coordinates of the same material point in the current configuration $\Omega_x \subset \mathbb{R}^3$ at time t. The motion of the body is then defined by $\vec{x} = \phi(\vec{X}, t)$. Let $\mathbf{F} = \partial\vec{x}/\partial\vec{X}$ be the deformation gradient with respect to the reference configuration Ω_X. According to the polar decomposition theorem, $\mathbf{F}=\mathbf{RU}=\mathbf{VR}$, $\mathbf{U}$ and $\mathbf{V}$ are the right and left stretch tensors, respectively, and $\mathbf{R}$ is the rotation tensor. The spatial discretization based on FEM of the equation of motion leads to the governing equilibrium equation [10]

$$\mathbf{M}\ddot{\vec{x}} + \overrightarrow{F^{\text{int}}}(\vec{x},\dot{\vec{x}}) - \overrightarrow{F^{\text{ext}}}(\vec{x},\dot{\vec{x}}) = 0 \tag{1}$$

where $\dot{\vec{x}}$ is the vector of the nodal velocities and $\ddot{\vec{x}}$ the vector of the nodal accelerations. $\mathbf{M}$ is the mass matrix, $\overrightarrow{F^{\text{ext}}}$ is the vector of the external forces and $\overrightarrow{F^{\text{int}}}$ the vector of the internal forces. This equation is completed by the following initial conditions at time $t=0$:

$$\vec{x}_0 = \vec{x}(t_0); \quad \dot{\vec{x}}_0 = \dot{\vec{x}}(t_0) \tag{2}$$

If we use the same form φ for the shape and test function (as usually done for a serendipity element), one may obtain the following expressions for the elementary matrices in Eq. (1):

$$\begin{aligned} \mathbf{M} &= \int_{\Omega_x} \rho\varphi^{\mathrm{T}}\varphi \,\mathrm{d}\Omega_x; \quad \overrightarrow{F^{\text{int}}} = \int_{\Omega_x} \nabla\varphi^{\mathrm{T}}\sigma \,\mathrm{d}\Omega_x; \\ \overrightarrow{F^{\text{ext}}} &= \int_{\Omega_x} \rho\varphi^{\mathrm{T}}\vec{b} \,\mathrm{d}\Omega_x + \int_{\Gamma_x} \varphi^{\mathrm{T}}\vec{t} \,\mathrm{d}\Gamma_x \end{aligned} \tag{3}$$

where ∇ is the gradient operator, superscript T is the transpose operator, Γ_x is the surface of the domain Ω_x where traction forces are imposed, ρ is the mass density, σ the Cauchy stress tensor, $\vec{b}$ is the body force vector and $\vec{t}$ is the surface traction force vector.

2.2. Time integration

Solution of the problem expressed by Eq. (1) requires integration through time. In our case, this one is achieved numerically in accordance with an explicit integration scheme. This is the most advocated scheme for integrating in the case of impact problems, i.e. high speed dynamics. For an explicit algorithm, the elements of the solution at time t_{n+1} depend only on the solution of the problem at time t_n without the need of any iteration in each step. Stability imposes the time-step size Δt to be lower than a limit as discussed further. In this work, we are using the generalized-α explicit scheme proposed by Chung and Hulbert [11] who have extended their implicit scheme to an explicit one. The main interest of this scheme resides in its numerical dissipation. The time integration is driven by the following relations:

$$\ddot{\vec{x}}_{n+1} = \frac{\mathbf{M}^{-1}(\overrightarrow{F^{\text{ext}}}_n - \overrightarrow{F^{\text{int}}}_n) - \alpha_{\mathrm{M}}\ddot{\vec{x}}_n}{1-\alpha_{\mathrm{M}}} \tag{4}$$

$$\dot{\vec{x}}_{n+1} = \dot{\vec{x}}_n + \Delta t[(1-\gamma)\ddot{\vec{x}}_n + \gamma\ddot{\vec{x}}_{n+1}] \tag{5}$$

$$\vec{x}_{n+1} = \vec{x}_n + \Delta t\dot{\vec{x}}_n + \Delta t^2\left[\left(\frac{1}{2}-\beta\right)\ddot{\vec{x}}_n + \beta\ddot{\vec{x}}_{n+1}\right] \tag{6}$$

Numerical dissipation is defined in the above system from the spectral radius $\rho_b \in [0.0:1.0]$ conditioning the numerical damping of the high frequency. Setting $\rho_b = 1.0$ leads to a conservative algorithm while $\rho_b < 1.0$ introduces numerical dissipation in the scheme. The three parameters α_M, β and γ are linked to the value of the spectral radius ρ_b by the following relations:

$$\alpha_M = \frac{2\rho_b - 1}{1 + \rho_b}; \quad \beta = \frac{5 - 3\rho_b}{(2 - \rho_b)(1 + \rho_b)^2}; \quad \gamma = \frac{3}{2} - \alpha_M \tag{7}$$

The time-step Δt is limited, it depends on the maximal modal frequency ω_{max} and on the spectral radius ρ_b by the following relation

$$\Delta t = \gamma_s \Delta t_{crit} = \gamma_s \frac{\Omega_s}{\omega_{max}} \tag{8}$$

where γ_s is a safety factor that accounts for the destabilizing effects of the non-linearities of the problem and Ω_s is defined by:

$$\Omega_s = \sqrt{\frac{12(\rho_b - 2)(1 + \rho_b)^3}{\rho_b^4 - \rho_b^3 + \rho_b^2 - 15\rho_b - 10}} \tag{9}$$

The generalized-α explicit integration flowchart is given in Box 1. In this flowchart, the three steps 5b, 5e and 5f are the most CPU intensive ones. We focus now on some theoretical aspects of those three steps before presenting some parallelizing methods to apply.

2.2.1. Internal forces computation

It is generally assumed that, according to the decomposition of the Cauchy stress tensor σ into a deviatoric term $\mathbf{s} = \mathrm{dev}[\sigma]$ and an hydrostatic term p, the hypo-elastic stress/strain relation can be written as follow

$$s^{\nabla} = C : D; \quad \dot{p} = K \,\mathrm{tr}[\mathbf{D}] \tag{10}$$

where s^{∇} is an objective derivative of $\mathbf{s}$, K is the bulk modulus of the material, $\mathbf{C}$ is the fourth-order constitutive tensor and $\mathbf{D}$ (the rate of deformation) is the symmetric part of the spatial velocity gradient $\mathbf{L} = \dot{\mathbf{F}}\mathbf{F}^{-1}$. The symbol ':' denotes the contraction of a pair of repeated indices which appear in the same order, so $\mathbf{A}:\mathbf{B} = A_{ij}B_{ij}$. As the DynELA FEM code is dedicated to large strains simulations, we must ensure the objectivity of the terms in Eq. (10). A procedure that now has become widely used consists in writing the constitutive equation in a co-rotational frame defined by a rotation tensor $\mathbf{w}$ with $\dot{\mathbf{w}} = \omega\mathbf{w}$ and $\mathbf{w}(t=0) = \mathbf{I}$. Defining any quantity ( ) in the rotating referential as co-rotational one denoted by $(\,)^c$, one may obtain:

$$\rho^c = \rho; \quad \sigma^c = \mathbf{w}^T \sigma \mathbf{w}; \quad \mathbf{C}^c = \mathbf{w}^T[\mathbf{w}^T\mathbf{C}\mathrm{w}]\mathbf{w} \tag{11}$$

For details concerning this change of frame, see Ref. [12]. The choice of $\omega = \mathbf{W}$ where $\mathbf{W}$ is the skew-symmetric part of the spatial velocity gradient tensor $\mathbf{L}$ leads to the well known Jaumann rate. Eq. (10) in this co-rotational frame leads to the following form:

$$\dot{\mathbf{s}}^c = \mathbf{C}^c : \mathbf{D}^c; \quad \dot{p} = K \,\mathrm{tr}[\mathbf{D}^c] \tag{12}$$

In order to integrate these equations through time, we adopt the use of elastic-predictor/plastic-corrector (radial-return mapping) strategy, see for example Refs. [10,12,13]. An elastic predictor for the stress tensor is calculated according to the Hooke's law by the following equation

$$p^{tr}_{n+1} = p_n + K \,\mathrm{tr}[\Delta e]; \quad \mathbf{s}^{tr}_{n+1} = \mathbf{s}_n + 2G \,\mathrm{dev}[\Delta e] \tag{13}$$

where G is the Lamé coefficient and $\Delta e = (1/2)\ln[\mathbf{F}^T\mathbf{F}]$ is the co-rotational natural strain increment tensor between increment n and increment $n+1$. At this point of the computation, we introduce the von Mises criterion defined by the following relation:

$$f = \bar{\sigma} - \sigma^v = \sqrt{\frac{3}{2}\mathbf{s}^{tr}_{n+1} : \mathbf{s}^{tr}_{n+1}} - \sigma^v, \tag{14}$$

where σ^v is the current yield stress of the material. If $f \le 0$, then the predicted solution is physically admissible and the whole increment is assumed to be elastic ($\mathbf{s}_{n+1} = \mathbf{s}^{tr}_{n+1}$).

Box 1

Flowchart for generalized-α explicit integration

(1) Internal matrices computation: **N**, **B**, **J**, det[**J**].
(2) Computation of the global mass matrix **M**.
(3) Computation of the vectors $\overrightarrow{F^{int}}$ and $\overrightarrow{F^{ext}}$.
(4) Computation of the stable time-step of the structure.
(5) Main loop until simulation complete.
  (a) Computation of the predicted quantities:

$$\dot{\tilde{x}}_{n+1} = \dot{\vec{x}}_n + (1 - \gamma)\Delta t_{n+1}\ddot{\vec{x}}_n$$

$$\tilde{\vec{x}}_{n+1} = \vec{x}_n + \Delta t_{n+1}\dot{\vec{x}}_n + \left(\frac{1}{2} - \beta\right)\Delta t^2\ddot{\vec{x}}_n$$

  (b) Computation of the vectors $\overrightarrow{F^{int}}$ and $\overrightarrow{F^{ext}}$.
  (c) Explicit solve:

$$\ddot{\vec{x}}_{n+1} = \frac{M^{-1}(\overrightarrow{F^{ext}_n} - \overrightarrow{F^{int}_n}) - \alpha_M\ddot{\vec{x}}_n}{1 - \alpha_M}$$

$$\dot{\vec{x}}_{n+1} = \dot{\tilde{\vec{x}}}_{n+1} + \Delta t_{n+1}\gamma\ddot{\vec{x}}_{n+1}$$

$$\vec{x}_{n+1} = \tilde{\vec{x}}_{n+1} + \Delta t^2_{n+1}\beta\ddot{\vec{x}}_{n+1}$$

  (d) If simulation complete, go to 6.
  (e) Internal matrices computation: **B**, **J**, det[**J**].
  (f) Computation of the stable time-step of the structure.
  (g) Go to 5a.
(6) Output.

Box 2

Radial return algorithm for an isotropic hardening flow law

(1) Compute the hardening coefficient $h_n(\bar{\varepsilon}_n^{vp})$ and the yield stress $\sigma_n^v(\bar{\varepsilon}_n^{vp})$.
(2) Compute the value of the scalar parameter $\Gamma^{(1)}$ given by:

$$\Gamma^{(1)} = \frac{\sqrt{\mathbf{s}_{n+1} : \mathbf{s}_{n+1}} - \sqrt{\frac{2}{3}}\sigma_n^v}{2G\left(1 + \frac{h_n}{3G}\right)}$$

(3) Consistency condition loop from $k=1$.
  (a) Compute $\sigma_{n+1}^v(\bar{\varepsilon}_n^{vp} + \sqrt{2/3}\Gamma^{(k)})$ and $h_{n+1}(\bar{\varepsilon}_n^{vp} + \sqrt{2/3}\Gamma^{(k)})$.
  (b) Compute $f = 2G\sqrt{2/3}\Gamma^{(k)} - \bar{\sigma} + \sigma_{n+1}^v$ and $\mathrm{d}f = 2G\sqrt{3/2} + \sqrt{2/3}h$.
  (c) If $f/\sigma_{n+1}^v <$ tolerance go to 4.
  (d) Update $\Gamma^{(k+1)} = \Gamma^{(k)} - f/\mathrm{d}f$.
  (e) $k \leftarrow k+1$ and go to 3a.
(4) Update the equivalent plastic strain $\bar{\varepsilon}_{n+1}^{vp} = \bar{\varepsilon}_n^{vp} + \sqrt{3/2}\Gamma^{(k)}$.
(5) Update the deviatoric stress tensor $\mathbf{s}_{n+1} = \mathbf{s}_n - 2G\Gamma^{(k)}(\mathbf{s}_n/\sqrt{\mathbf{s}_n : \mathbf{s}_n})$.

If not, the consistency must be restored using the radial return-mapping algorithm reported in Box 2.

2.2.2. Internal matrices computation

The internal matrices computation is done element by element. This computation is totally independent from one element to any other one. This computation consists in the computation of the elementary matrices **N** for the shape functions, $\mathbf{B} = \partial\mathbf{N}/\partial\vec{x}$ for the derivatives of the shape functions and **J** the Jacobian. This computation is done for every quadrature point of each element.

2.2.3. Stable time-step computation

Explicit schemes are conditionally stable. The time-step size must be lower than the critical value depending on the maximum pulsation ω_{max} of the body as shown in Eq. (8). In our application, the value of ω_{max} is evaluated by the power iteration method proposed by Benson [14]. The corresponding algorithm is given in Box 3. Once the evaluation of ω_{max} is done, Eq. (8) gives the stable time-step value for the structure.

Box 3

Computation of the maximal model frequency

(1) Initializations $n=0$; $x_0 = \{1,\ldots, 0,\ldots, -1\}^T$.
(2) Computation of the elementary elastic stiffness matrices $\mathbf{K}^e$.
(3) Loop over n iterative.
  (a) Loop over all elements to evaluate $\hat{x}_n = \mathbf{K}x_n$ on the element level.
    (i) Gather x_n^e from global vector x_n.
    (ii) $\hat{x}_n^e = \mathbf{K}^e x_n^e$.
    (iii) Scatter of $\hat{x}_n^e$ into global vector $\hat{x}_n$.
  (b) Computation of the Rayleigh Quotient $\mathcal{R} = x_n^T\hat{x}_n / x_n^T\mathbf{M}x_n$.
  (c) $\hat{x}_{n+1} = \mathbf{M}^{-1}\hat{x}_n$.
  (d) $f_{max} = \max(\hat{x}_{n+1})$.
  (e) $x_{n+1} = \hat{x}_{n+1}/f_{max}$.
  (f) If $|f_{max} - \mathcal{R}|/(f_{max} + \mathcal{R}) \le$ tolerance go to 4.
  (g) Return to 3a.
(4) Return the maximal model frequency $\omega_{max} = \sqrt{f_{max}}$.

3. Object-oriented design

3.1. Overview of object-oriented programming

Numerical softwares are usually based on the use of a procedural programming language such as Fortran. Over the last few years, the use of object-oriented programming (OOP) techniques has increased and C++ language [15] has become popular for writing FEM codes. Briefly speaking, the use of OOP leads to highly modularized codes through the use of defined classes, i.e. associations of data and methods. The benefits of OOP to implementations of FEM programs has already been explored by several authors [8,16–18].

3.2. Finite element classes

As it can be found in other papers dealing with the implementation of FEM [16–18] we developed some specific classes for this application. The FEM represented by the class `Structure` is mainly composed of the classes `Node`, `Element`, `Material` and `Interface` as shown in Fig. 2. In this application, all quantities are stored into the corresponding object as a consequence of OOP encapsulation. This specificity leads to a difference between a classical FEM programming, where quantities are stored in global vectors declared common, and our approach. This will be very important for the parallelization of the code as we will see later.

- The class `Node` contains nodal data such as nodal number or coordinates. Two instances of the `NodalField` class are linked to each node, the first one contains nodal quantities at time t, the second one at time $t+\Delta t$. At the end of an increment, we swap the two references to transfer quantities from one step to the next one. Boundary conditions through the `BoundaryCondition` class may affect the behavior of each node in particular sub-treatments such as contact conditions. Those conditions are dynamically linked to the nodes,

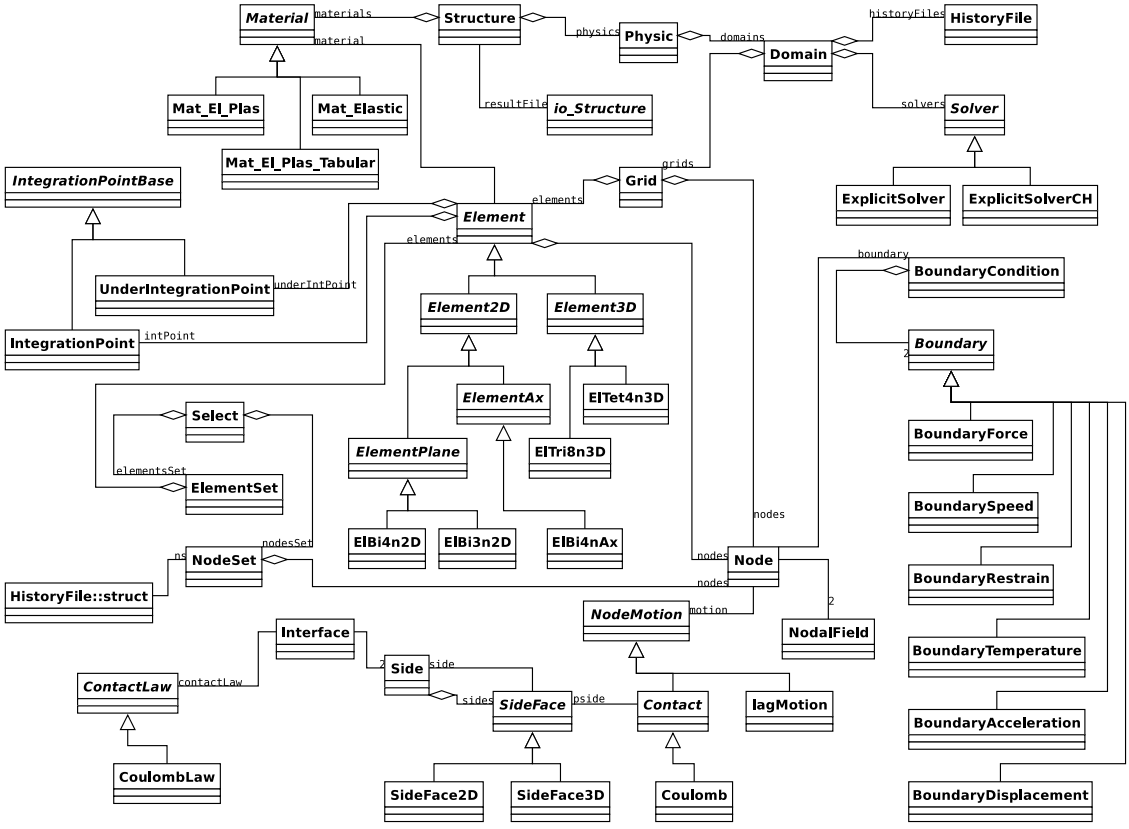


Fig. 2. Simplified UML diagram of the Object oriented framework.

therefore, they can change during the computation. This is important for example for contacting nodes.

- The class `Element` is a virtual class containing the definition of each element of the structure. Many specialized derived classes have been defined depending on the real nature of the element.
- The class `Interface` contains definitions concerning the contact interfaces, the contact law through the `ContactLaw` class and the contact definition through the `Side` class.
- The `Material` class is used for the definition of the materials used in various models.
- The class `Solver` serves as a base class for derived solvers.
- Many other utility classes exist for time-history plot, node and element groups management, data files read/write [19].

4. Parallelization of the code

A Compaq ProLiant 8000 under Linux Redhat 8.0 is used for developing and evaluating the performances of the parallel code. This one is equipped with eight Intel Xeon PIII 550/2Mb processors and 5 GB of system memory. Compilation of the code is done using the Intel C++ 7.1 compiler without any optimization flag in order to compare various implementations without compiler influence. This kind of computer is usually dedicated to web server applications.

The parallelization of our FEM application is based on the use of OpenMP [4]. It is used to specify parallelization on shared memory machines with the use of compiler directives, library routines and environment variables. Communication is implicit as we use a shared memory architecture. Parallelization of the finite element software involves a restructuring of the code for an efficient run on multiprocessor systems by distributing the work among the processors. This task is simplified because the DynELA code is an Object-Oriented one. The type of parallelism used in OpenMP is sometimes called 'fork-join' parallelism because we launch multiple parallel threads (fork) in parallel regions of the code and join them into a single thread (the master one) for serial processing in non-parallel regions as described in Fig. 3. A thread is an instance of the program running on behalf of some user or process.

Parallelization with OpenMP can be done automatically (through compiler flags) or manually (through explicit compiler directives in the code). We tested both methods, and as many other authors [20], found that the automatic parallelization of the code leads to very bad Speedup results. Manual parallelizing of the code is achieved by inserting

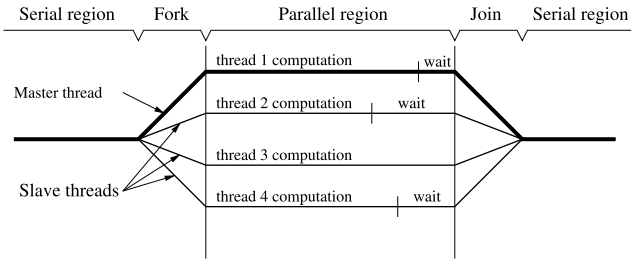


Fig. 3. Fork-join parallelism.

specific `#pragma` directives in C/C++ codes. For example:

```
void buildSystem(List ⟨Elements⟩
elements) {
#pragma omp parallel for
  for (int i=0; i⟨elements.size(); i++
  ) {
    elements(i).computeMatrices();
  }
}
```

In this example, the `#pragma omp parallel for` directive instruct the compiler that the following loop must be forked, and the work must be distributed among multiple processors. All of the threads perform the same computation unless a specific directive is introduced within the parallel region. For parallel processing to work correctly, the iterations must not be dependent on each other and of course, the `computeMatrices` method must be thread-safe. In computer programming, thread-safe describes a program portion or routine that can be called from multiple programming threads without unwanted interaction between the threads. Thread safety is of particular importance in OpenMP programming. By using thread-safe routines, the risk that one thread will interfere and modify data elements of another thread is eliminated by circumventing potential data race situations with coordinated access to shared data.

The user defines parallel region blocks using the `#pragma omp parallel` directive. The parallel code section is executed by all threads including the master thread. Some data environment directives (`shared`, `private`...) are used to control the sharing of program variables that are defined outside the scope of the parallel region. Default value is `shared`. A `private` variable has a separate copy per thread with an undefined value when entering or exiting a parallel region.

The synchronization directives include `barrier` or `critical`. A `barrier` directive causes a thread to wait until all other threads in the parallel region have reached the barrier. An implicit barrier exists at the end of a parallel region block. A `critical` directive is used to restrict access to the enclosed code to only one thread at a time. This is very important point when threads are modifying shared variables.

Of course, this is only a brief overview of the OpenMP directives and we refer to Ref. [4] for further complements about this standard.

4.1. Load balancing

As we presented earlier, we adopt the use of an elastic predictor/plastic corrector strategy in this work. In dynamic computations, CPU time/element may vary from one element to another during the computation of the plastic corrector because plastic flow occurs in restricted regions of the structure. Therefore, as presented in Section 2.2.1, if the elastic predictor is physically admissible, the CPU consuming return-mapping algorithm presented in Box 2 is not executed for the corresponding integration point. Only the evaluation of the criterion (14) allows to know the treatment to apply. This also continuously evolves as the plastic front moves across the structure.

As a consequence, the prediction of the CPU time needed for the computation of the internal force vector $\overrightarrow{F^{\text{int}}}$ is impossible to do here. Concurrent threads may request quite different CPU time to complete, leading to wastes of time, because we must wait for the latest thread to complete before reaching the serial region (see Fig. 3 where thread 2 is the faster one and thread 3 the slower one). To avoid such a situation, we must use a dynamic load balance in order to equilibrate the allocated processors work. The class Jobs (see Fig. 4) is dedicated to this. The class Job contains the list of elements to be computed by one thread. The main differences from the load balancing procedure developed

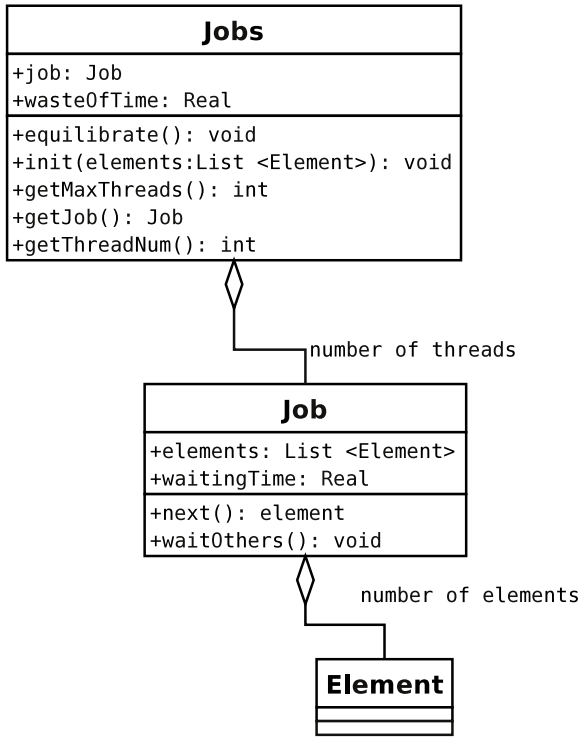


Fig. 4. Jobs class description.

here with other ones usually used in Domain Decomposition Methods (DDM) coming from the literature [21] are summarized here after:

- we use an explicit solver, therefore, the load balancing over-cost must be very small (iterations of the main loop in Box 1 are quite fast within an explicit integration scheme),
- in our approach, the spatial distribution of elements/thread can be any one. There is no need to solve any interfacial problem as in a DDM approach.

For each iteration in the main loop of Box 1, at the end of the requested computation in each of the parallel threads, we measure the waiting time for each thread. We build an indexed list containing the ranking for each thread ranging from the faster one to the slower one. If the waiting time of the faster thread is over a given parameter specified by the user, some elements are transferred from the slower thread to the others in order to equilibrate the allocated processors work.

4.2. Benchmark test used for Speedup measures

4.2.1. Impact of a copper rod

We need a benchmark test in order to compare the efficiency of the various proposed parallelization methods presented further. The impact of a copper rod on a rigid wall is a standard benchmark problem for dynamics computers codes. A comparison of numerical results obtained with the DynELA code and other numerical results has already been presented in Ref. [8]. In this paper we will focus on the Speedup obtained after the parallelization of the code using this benchmark test. The initial dimensions of the rod are $r_0 = 3.2$ mm and $l_0 = 32.4$ mm. The impact is assumed frictionless and the impact velocity is set to $V_i = 227$ m/s. The final configuration is obtained after 80 μs. The constitutive law is elasto-plastic with a linear isotropic hardening, material properties, given in Ref. [22], corresponding to an OHFC copper are reported in Table 1. Only half of the axisymmetric geometry of the rod has been meshed in the model. Two different meshes are used with 1000 (10×100) and 6250 (25×250) elements, respectively. This quite large number of elements has been chosen to increase the computation time. Table 2 reports a comparison for the final length l_f, the footprint radius r_f and the maximum equivalent plastic strain $\bar{\varepsilon}^p_{max}$ obtained with our finite element code and other numerical results such as the one obtained by Liu et al. [22]

Table 1
Material properties of the OHFC copper rod for the Taylor test

Young modulus (GPa)	E	117.0
Poisson ratio	ν	0.35
Density (kg/m^3)	ρ	8930
Initial flow stress (MPa)	σ^0_v	400.0
Linear hardening (MPa)	H	100.0

Table 2
Comparison of numerical results for the Taylor test

FEM code	r_f	l_f	$\bar{\varepsilon}^p_{max}$
DynELA (25×250 elements)	7.11	21.33	3.30
DynELA (10×100 elements)	7.08	21.35	3.27
Abaqus explicit (100×100 elements)	7.08	21.48	3.23
Liu (5×50 elements)	7.15	21.42	–

or the same simulation problem with the Abaqus Explicit program (using the same 10×100 mesh as presented before). The differences between the solutions are reasonable and this benchmark test is retained.

4.2.2. Time measures

In an explicit FEM code CPU times are quite difficult to measure. We developed a specific class called `CPUrecord` for this purpose. CPU measures are usually done using the standard `time` function in C but the problem here is that this one has only a time resolution of $\Delta t = 10$ ms. In this application, we use the Pentium benchmarking instruction Read Time Stamp Counter (RDTSC) that returns the number of clock cycles since the CPU was powered up or reset. On the used computer, this instruction gives a time resolution of about $\Delta t = 1/(550 \times 10^6) \simeq 1.8$ ns.

4.3. Internal forces computation parallelization

In this part, we focus on the parallelization of the internal force vector computation presented in Section 2.2.1. This computation is the most CPU intensive part of the FEM code. To illustrate the use of the OpenMP parallelization techniques we present in this section different ways to parallelize the corresponding block with the influence on the Speedup. This case is a typical application of OpenMP on major loops leading to a coarse grain parallelization. This one gives better results than the classical fine grain parallelization usually done with OpenMP. In fact, fine grain parallelization suffers from the drawback of frequent thread creations, destructions and associated synchronizations. In the following example, the method `computeInternalForce` is applied on each element of the mesh and returns the internal force vector resulting from the integration over the element of Eq. (12). The `gatherFrom` operation will assemble the resulting element internal force vector into the global internal force vector of the structure. A typical C++ fragment of the code is given as follow:

```
Vector Fint;
for (int elm=0; elm ⟨elements.size( );
elm++) {
  Vector FintElm;
  elements (elm).computeInternalForces
  (FintElm);
  Fint.gatherFrom          (FintElm,
  elements(elm));
}
```

```
Vector Fint; // internal force Vector

// parallel loop base on OpenMP pragma directive
#pragma omp parallel for
for (int elm = 0; elm < elements.size (); elm++)
{
  Vector FintElm; // local internal force Vector

  // compute local internal force vector
  elements(elm).computeInternalForces (FintElm);

  // gather operation on global internal force vector
#pragma omp critical
  Fint.gatherFrom (FintElm, elements(elm));
} // end of parallel for loop
```

Fig. 5. Source code for the method (1) variant.

We present here after four different techniques from the simplest one to the most complicated one and compare their efficiency using the 1000 elements mesh.

(1) In this first method, we use a `parallel` for directive for the main loop and share the `Fint` vector among the threads. A critical directive is placed just before the `gatherFrom` operation because `Fint` is a shared variable. See Fig. 5 for the corresponding source code fragment.
(2) In this method, we use a parallel region directive. In this `parallel region`, all threads access a shared list of elements to treat until empty. The `Fint` vector is declared as `private`. Both main operations are treated without the need of any `critical` directive. At the end of the process, all processors are used together to assemble the locals copies of `the Fint` vector into a global one.
(3) This method is similar to the previous one except that each thread has a predetermined equal number of elements to treat. Therefore, we avoid the use of a shared list (as in method 2), each processor operates on a block of elements. A new class `Jobs` is used to manage the dispatching of the elements over the processors. This one will be described further.
(4) This method is similar to the previous one except that we introduce the dynamic load balance operator presented in Section 4.1. See Fig. 6 for the corresponding source code fragment.

Table 3 reports some test results. The Speedup factor s_p is the ratio of the single-processor CPU time (T_s) over the CPU time (T_m) obtained with the multi-processor version of the code. The efficiency e_f is the Speedup ratio over the number of processors used (n):

$$s_p = \frac{T_s}{T_m}; \quad e_f = \frac{s_p}{n} \tag{15}$$

Variation in the number of CPU to use is done by specifying this value from the environment variable `OMP_NUM_THREADS`. Table 3 shows that this ratio can

```
jobs.init(elements); // list of jobs to do (instance of class Jobs)
int threads = jobs.getMaxThreads(); // number of threads
Vector Fint = 0.0; // internal force Vector
Vector FintLocal[threads]; // local internal force vectors

// parallel computation of local internal force vectors
#pragma omp parallel
{
  Element* element;
  Job* job = jobs.getJob(); // get the job for the thread
  int thread = jobs.getThreadNum(); // get the thread Id

  // loop while exists elements to treat
  while (element = job->next())
   {
     Vector FintElm; // element force vector

     // compute local internal force vector
     element->computeInternalForces (FintElm);

     // gather operation on local internal force vector
     FintLocal[thread].gatherFrom (FintElm, element);
   }
  job->waitOthers(); // compute waiting time for the thread
} // end of parallel region

// parallel gather operation
#pragma omp parallel for
for (int row = 0; row < Fint.rows(); row++)
{
  // assemble local vectors into global internal force vector
  for (thread = 0; thread < threads; thread++)
    Fint(row) += FintLocal[thread](row);
} // end of parallel for loop

// equilibrate the sub-domains
jobs.equilibrate();
```

Fig. 6. Source code for the method (4) variant.

be over 100%, this case is usually called Super-linear Speedup. This result comes from the fact that, as a consequence of the dispatching of the work, each processor needs less memory to store the local problem, and cache memory can be used in a more efficient way. In a computer, processor tends to fetch data into cache before it reads it (usually a block of data, not a single element). Next time data are needed, there is a very fast access if it is still in the cache otherwise it will be slow. If the amount of data treated by the processor is not too big, the chance that the needed next data resides in the cache is high, otherwise cache-missing occurs. If we run the same computation test with 6250 elements instead of 1000, we obtain an efficiency value of 90% for eight processors, and always below 100% for 2–8 processors. Cache-missing seems to occur in this case.

Fig. 7 shows a plot of the Speedup versus number of processors. We can see that using method 1 leads to a very bad parallel code especially when the number of processors is greater than 5, while significant improvement comes with methods 3 and 4. In fact, in method 1, the presence of a critical directive in the `gatherFrom` operation leads to a very low Speedup because only one thread can do this quite CPU intensive operation at a time. In the second method, we also need a critical directive to pick an element from the global shared list of elements to treat and it costs CPU time for that. Methods 3 and 4 are the most optimized ones. The dynamic load balance method is the fastest one whereas it needs some extra code to compute and operate this balance.

Table 3
Speedup of the $\overrightarrow{F^{int}}$ computation for various implementations

Method	1 CPU	4 CPU			8 CPU		
	Time	Time	Speedup	Efficiency (%)	Time	Speedup	Efficiency (%)
1	167.30	72.25	2.88	72.2	72.25	2.31	28.9
2	163.97	45.98	3.56	89.1	25.39	6.45	80.7
3	164.52	42.18	3.90	97.5	20.86	7.88	98.5
4	164.25	38.55	4.26	106.5	19.66	8.35	104.4

Of course this extra time is taken into account in the results presented. In fact, the CPU time needed for the `computeInternalForce` operation may differ from one element to an other because of differences in material law or elastic/plastic loading in different parts of the structure, so some threads have to wait, without doing effective computation, until the slower thread completes. Dynamic load balance improves the efficiency by reducing this waiting time.

4.4. Time-step computation parallelization

Concerning the parallelization of the time-step computation we measured the CPU times using the Taylor benchmark test with 6250 elements. An analysis of the CPU times shows that the two sub-steps (2) and (3a) in Box 3 represents 66.4% and 31.4% of the total computational time in the Box. Different strategies have been applied to both parts in order to efficiently parallelize those two steps.

- The one concerning step (2) in Box 3 is quite trivial as the computation of the elastic stiffness matrices $\mathbf{K}^e$ have no dependence from one element to another one. We apply here a procedure similar to method (3) in the internal forces vector computation.
- Step (3a) in Box 3 is more complicated to efficiently parallelize as in sub-step (3a(iii)) we can notice a writing instruction in the shared vector $\hat{x}_n$. We already know that the use of a `critical` directive for this operation costs a lot of CPU time. Solution adopted in this case is to introduce a private vector $\hat{x}_n^{(i)}$ where superscript (i) represents the thread number and further to collect all vectors $\hat{x}_n^{(i)}$ into a single vector $\hat{x}_n$ using an efficient parallel collecting algorithm.

Fig. 8 shows the Speedup versus number of processors for this implementation. Steps (2) and (3a) present a Super-linear Speedup in the benchmark test used. The so called `collecting vectors` step contains sub-steps (3b–3e) and the added step used to collect all local thread vectors $\hat{x}_n^{(i)}$ into a single vector $\hat{x}_n$. In this step, as the number of processors increases, and therefore the number of local thread vectors to collect, the CPU time decreases slightly so the over-cost induced from the collecting operation is compensated by the gain produced by the parallelization of

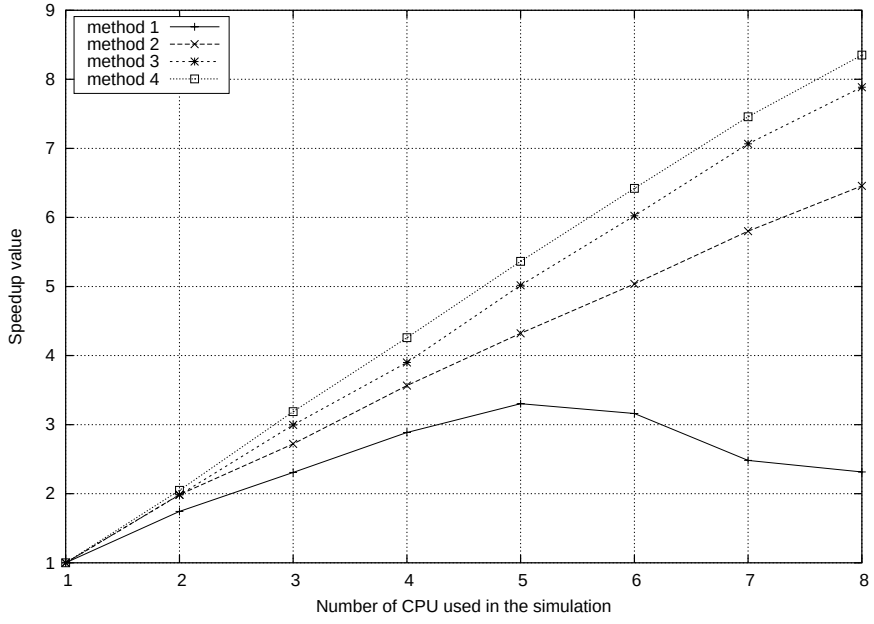


Fig. 7. Speedup of the $\overrightarrow{F^{int}}$ computation for various implementations.

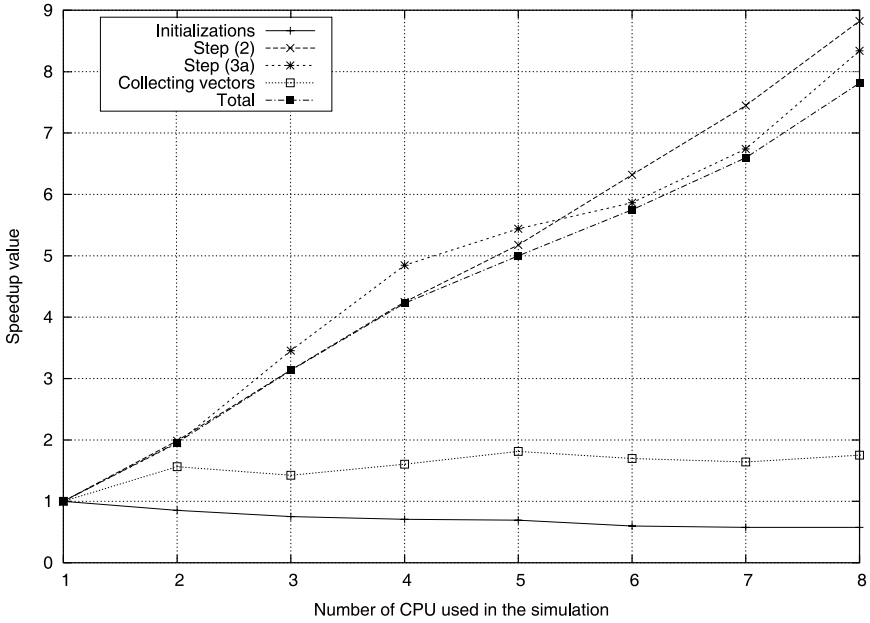


Fig. 8. Speedup results for the time-step computation procedure.

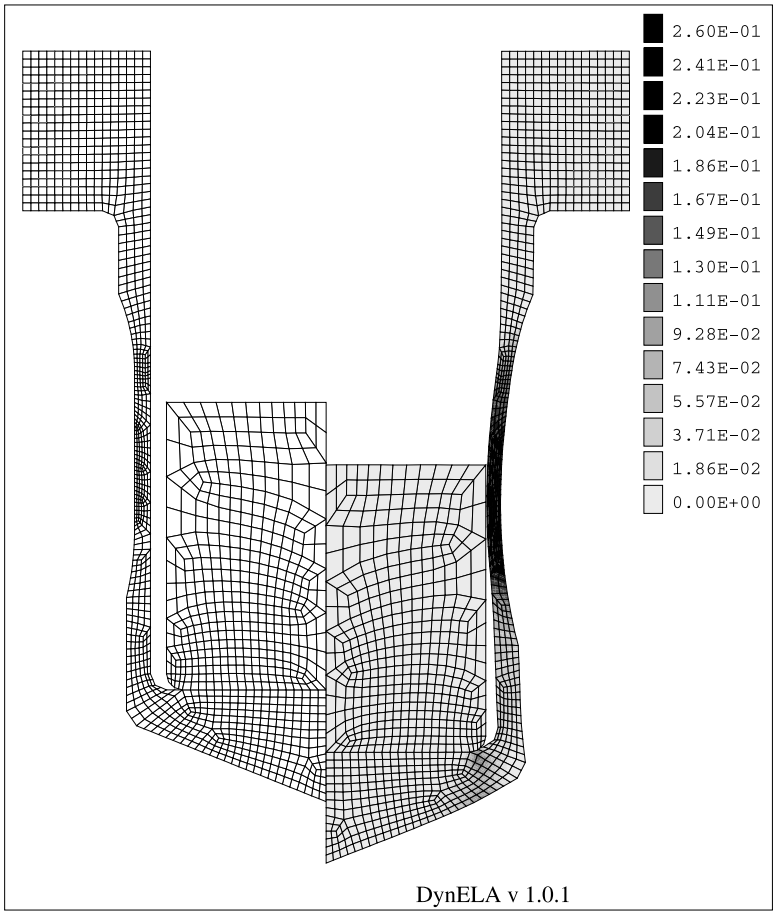


Fig. 9. Dynamic traction: initial mesh and equivalent plastic strain contour-plot.

Table 4
Material properties of the projectile and the target for the dynamic traction test

Nature		Projectile	Target
Young modulus (GPa)	E	193.6	74.2
Poisson ratio	ν	0.3	0.33
Density (kg/m^3)	ρ	7800	2784
Initial flow stress (MPa)	A	873	360
Hardening (MPa)	B	748	316
Coefficient	n	0.23	0.28

Table 5
Comparison of numerical results for the dynamic traction test

FEM code	$\bar{\varepsilon}^{p}_{max}$	Final length (mm)	Inner diameter (mm)	Thickness (mm)
DynELA	0.260	50.84	10.07	0.857
Abaqus	0.259	50.84	10.08	0.856

the sub-steps (3b–3e). The Speedup is around 1.5 for this operation, but we have to notice that this one only represents 2% of the total computational time for the time-step computation procedure. Initializations step presents a Speedup below 1, but in this case, it only represents 0.2% of the computational time. In the presented example, this figure shows a very good `total` Speedup close to the ideal Speedup.

5. Application to an impact simulation

A typical application of the proposed software is presented below showing some results concerning a dynamic traction simulation. This problem simulates the impact of a cylindrical projectile into a closed cylindrical tube. The aim of this test is to identify the constitutive flow law parameters from a set of experiments [23]. We only focus here on the numerical aspect of this test. Only half of the axisymmetric geometry of the structure has been meshed in the model. Initial mesh is reported on the left side in Fig. 9. Numerical model contains 1420 four-nodes quadrilateral elements. Materials of the projectile and the target are different and correspond to a 42CrMo4 steel and an aluminum 2017T3, respectively. Material properties corresponding to an isotropic elasto-plastic constitutive law of the form $\sigma^{v} = A + B\bar{\varepsilon}^{n}$ given in Ref. [23] are reported in Table 4. The projectile weight is $m = 44.1$ gr and the impact speed is $V_c = 80$ m/s. The final configuration is obtained after 110 μs. Right side in Fig. 9 shows the equivalent plastic strain $\bar{\varepsilon}^{p}$ contour-plot at the end of the computation. The model has been exported from DynELA to Abaqus explicit v. 6.4 using the export feature of the DynELA post-processor [19], the meshes are identical in both cases. A comparison of the numerical results is reported in Table 5 and shows a very good level of agreement.

Concerning the parallelization of the code, Fig. 10 shows the general Speedup obtained in this case. The time-step computation procedure presents a good Speedup near the ideal one, while the internal force vector computation shows a falling off after six processors. A fine analysis has shown that this problem seems to be linked to the parallel gather operation at the end of the code in Fig. 6, but we must keep in mind that some extra code has been added in order to measure local CPU time of this subroutine. Therefore, the presence of some extra synchronization directives for CPU measures may interfere with those measures. With the parallelization of only the time-step computation and the internal force vector computation procedures, the total Speedup is 5.61 for eight processors.

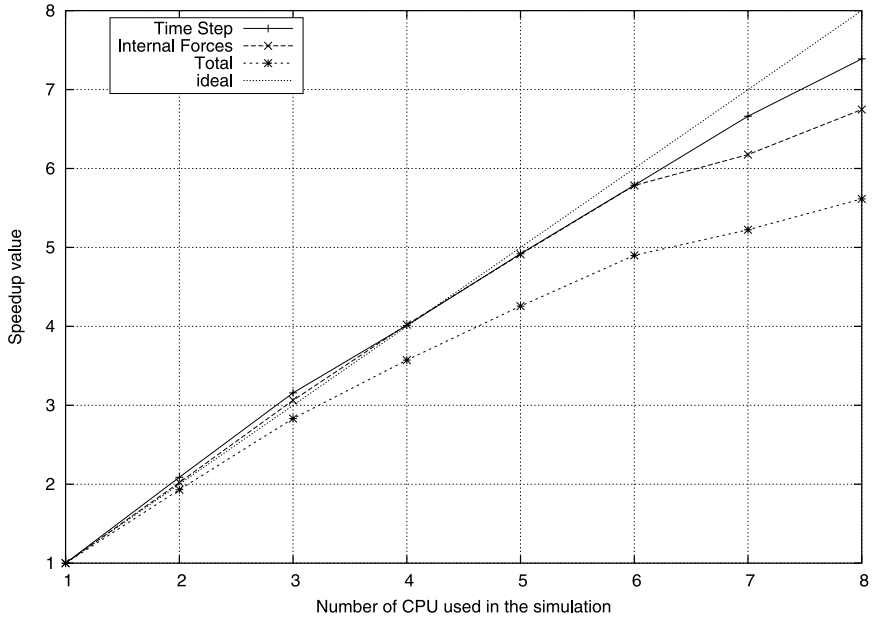
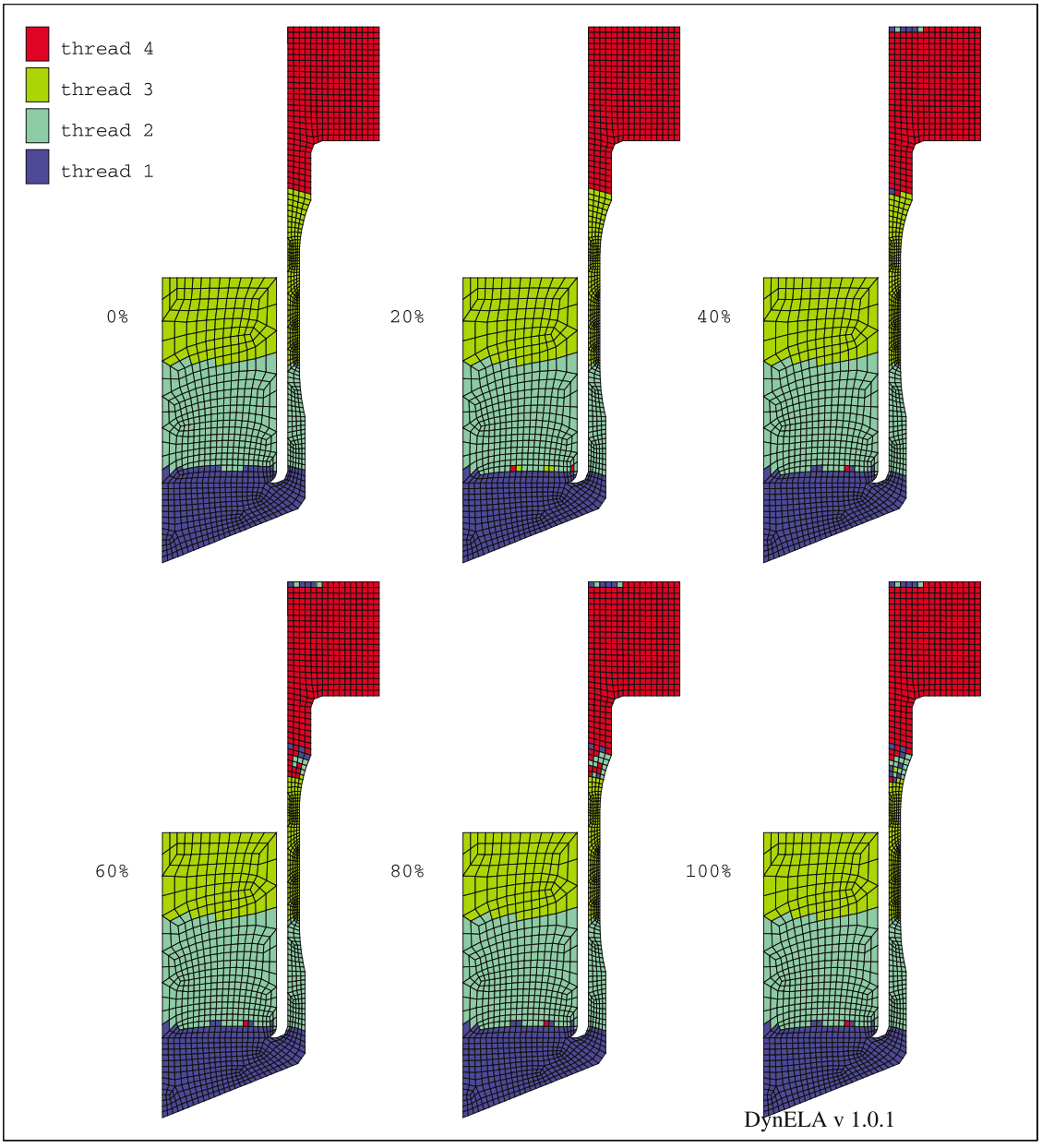


Fig. 10. Speedup for the dynamic traction test.



Distribution of the elements/thread during the computation

Fig. 11. Spatial distribution of the elements during the computation.

Figs. 11 and 12 shows the variation of the number of elements in each thread, for the internal force vector computation, during the run when using four processors. From this later, we can see that the number of elements for each thread vary in the range [329:411] while the average value for 1420 elements is 355.

6. Conclusions

An object-oriented simulator was developed for the analysis of large inelastic deformations and impact processes. The parallel version of this code uses OpenMP directives as SMPs programming tool. The OpenMP version can also be compiled using non-parallel compiler (the pragma directives will be ignored by the compiler). This enforces the portability of the code on different platforms. During this work, it has been found that the use of the OOP facilitates the parallelization of the code.

With the increasing prominence of SMPs computers, the importance of the availability of efficient and portable parallel codes grows. Several benchmark tests have demonstrated the accuracy and efficiency of the developed software. Concerning the parallel performances, the examples presented show a good Speedup with this code.

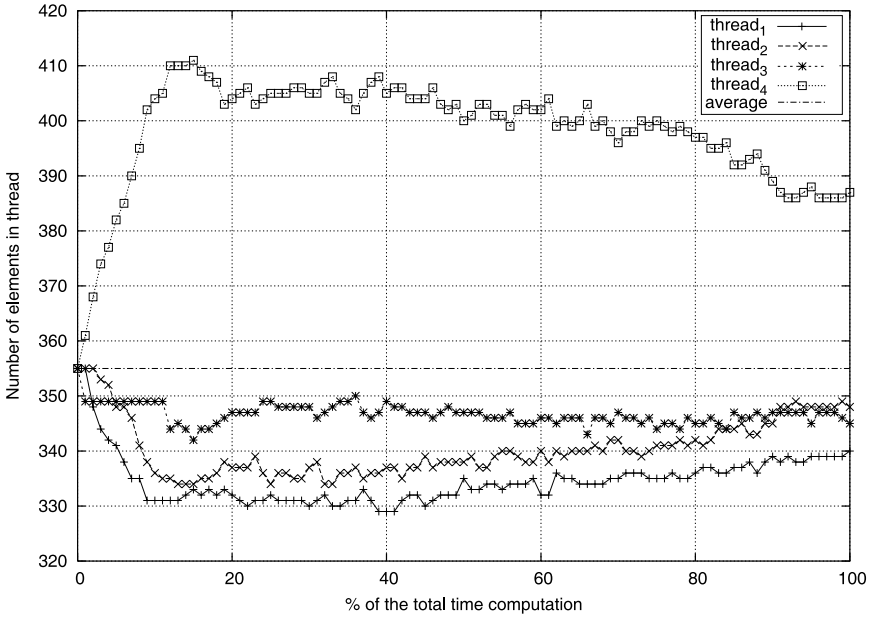


Fig. 12. Distribution of the elements during the computation.

This software is still under development and new features are added continuously. For this moment, the main development concerns more efficient constitutive laws (including visco-plasticity and damage effects) and contact laws. Concerning the parallelization of the code, our efforts are now concentrated on the use of mixed mode MPI/OpenMP parallelization techniques. This will allow us to build a new version of the DynELA code dedicated to clusters of workstations or PC. For this purpose, sub-domain computations must be introduced in the code.

References

[1] Hibbit, Karlsson & Sorensen, inc. www address: http://www.hks.com.
[2] Anderheggen E, Renau-Munoz JF. A parallel explicit solver for simulating impact penetration of a three-dimensional body into a solid substrate. Adv Eng Softw 2000;31:901–11.
[3] Brown K, Attaway S, Plimpton S, Hendrickson B. Parallel strategies for crash and impact simulations. Comput Meth Appl Mech Eng 2000; 184:375–90.
[4] Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R. Parallel programming in OpenMP. New York: Academic Press; 2001.
[5] Hoeflinger J, Alavilli P, Jackson T, Kuhn B. Producing scalable performance with openmp: experimental with two cfd applications. Parallel Comput 2001;27:391–413.
[6] Jia R, Sundén B. Parallelization of a multi-blocked cfd code via three strategies for fluid flow and heat transfer analysis. Comput Fluids 2004;33:57–80.
[7] Rama Mohan Rao A, Appa Rao TVSR, Dattaguru B. A new parallel overlapped domain decomposition method for nonlinear dynamic finite element analysis. Comput Struct 2003;81:2441–54.
[8] Pantalé O. An object-oriented programming of an explicit dynamics code: application to impact simulation. Adv Eng Softw 2002;33(5): 297–306.
[9] Pantalé O, Caperaa S, Rakotomalala R. Development of an object oriented finite element program: application to metal forming and impact simulations. J Comput Appl Math 2004;168(1/2):341–51.
[10] Belytschko T, Liu WK, Moran B. Nonlinear finite element for continua and structures. New York: Wiley; 2000.
[11] Hulbert GM, Chung J. Explicit time integration for structural dynamics with optimal numerical dissipation. Comput Meth Appl Mech Eng 1996;137:175–88.
[12] Ponthot JP. Unified stress update algorithms for the numerical simulation of large deformation elasto-plastic and visco-plastic processes. Int J Plast 2002;18:91–126.
[13] Simo JC, Hughes TJR. Computational inelasticity. Berlin: Springer; 1998.
[14] Benson DJ. Stable time step estimation for multi-material Eulerian hydrocodes. Comput Meth Appl Mech Eng 1998;167:191–205.
[15] Stroustrup B. The C++ programming language. 2nd ed. Reading, MA: Addison Wesley; 1991.
[16] Miller GR. An object oriented approach to structural analysis and design. Comput Struct 1991;40(1):75–82.
[17] Mackie RI. Object oriented programming of the finite element method. Int J Num Meth Eng 1992;35:425–36.
[18] Zabaras N, Srikanth A. Using objects to model finite deformation plasticity. Eng Comput 1999;15:37–60.
[19] Pantalé O. Manuel utilisateur du code de calcul DynELA v. 1.0.0. Laboratoire LGP ENI Tarbes, Av d'Azereix 65016, Tarbes, France; 2003.
[20] Turner EL, Hu H. A parallel cfd rotor code using openmp. Adv Eng Softw 2001;32:665–71.
[21] Rus P, Stok B, Mole N. Parallel computing with load balancing on heterogeneous distributed systems. Adv Eng Softw 2004;34:185–201.
[22] Liu WK, Chang H, Chen JS, Belytschko T. Arbitrary Lagrangian–Eulerian Petrov–Galerkin finite elements for nonlinear continua. Comput Meth Appl Mech Eng 1988;68:259–310.
[23] Pantalé O, Nistor I, Caperaa S. Identification et modélisation du comportement des matériaux metalliques sous sollicitations dynamiques. In: Military Technical Academy, editor. 30th internationally attended scientific conference of the military technical academy, Bucharest; 2003. ISBN 973-640-012-3.