Int. Conf. on Computational Methods for Coupled Problems in Science and Engineering
COUPLED PROBLEMS 2005
M. Papadrakakis, E. Oñate and B. Schrefler (Eds)


# STRATEGIES FOR A PARALLEL 3D FEM CODE: APPLICATION TO IMPACT AND CRASH PROBLEMS IN STRUCTURAL MECHANICS

**Olivier Pantalé*** **and Serge Caperaa***

*Laboratoire Génie de Production - Ecole Nationale d'Ingénieurs de Tarbes
47 av d'Azereix, 65016 Tarbes Cedex, France
e-mail: pantale@enit.fr, web page: http://www.enit.fr/

**Key words:** Finite Element, Parallelization, OpenMP, Explicit integration scheme, Impact simulation, SMP computer

**Abstract.** *Crash and impact numerical simulations are now becoming widely used engineering tools in the scientific community. Nowadays, accurate analysis of large deformation inelastic problems occurring in impact simulations is extremely important due to the high amount of plastic flow. Concerning numerical softwares, number of computational algorithms have been developed in the last years, and their complexity is continuously increasing. 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 for those intensive numerical applications.*

*In this paper, some aspects regarding the parallel implementation of the Object-Oriented explicit Finite Element dynamics code DynELA[9, 11] using the well known OpenMP[4] standard are presented. The Object-Oriented programming (OOP) leads to better-structured codes for the finite element method and facilitates the development, the maintainability and the expandability of such codes.*

*In a first part of this paper, an overview of the Finite Element 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 Shared Memory Processing architecture using the OpenMP standard. A benchmark test is used in this part to compare the performance of the different proposed parallelization methods. Finally, the efficiency and accuracy of the retained implementations are investigated using a numerical example relative to impact simulation.*

## 1 INTRODUCTION

In this presentation, an Object-Oriented implementation of an explicit Finite Element program, dedicated to simulation of impact problems, called DynELA is presented. This code is

developed in the L.G.P. in Tarbes and we refers to Pantalé et al.[9, 11, 10] for more details. In this presentation, we focus on the strategies used to parallelize this FEM code using the OpenMP[4] standard.

## 2 OVERVIEW OF THE FEM CODE

### 2.1 Basic kinematics

In the current version of the DynELA FEM code, the conservative and constitutive laws are formulated using an updated Lagrangian formulation in large deformations. Let $\overrightarrow{X}$ be the reference coordinates of a material point in the reference configuration $\Omega_X \subset \mathbb{R}^3$ at time $t = 0$, and $\overrightarrow{x}$ be the current coordinates of the same material point in the current configuration $\Omega_x \subset \mathbb{R}^3$ at time $t$. Let $\mathbf{F} = \partial\overrightarrow{x}/\partial\overrightarrow{X}$ be the deformation gradient. The spatial discretization based on FEM of the equation of motion leads to the governing equilibrium equation[2]:

$$\mathbf{M}\ \overset{\bullet\bullet}{\overrightarrow{x}} + \overrightarrow{F^{int}}(\overrightarrow{x}, \overset{\bullet}{\overrightarrow{x}}) - \overrightarrow{F^{ext}}(\overrightarrow{x}, \overset{\bullet}{\overrightarrow{x}}) = 0 \tag{1}$$

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

$$\overrightarrow{x}_0 = \overrightarrow{x}(t_0); \quad \overset{\bullet}{\overrightarrow{x}}_0 = \overset{\bullet}{\overrightarrow{x}}\ (t_0) \tag{2}$$

If we use the same form $\varphi$ for the shape and test function, one may obtain the following expressions for the elementary matrices in equation (1):

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

where $\nabla$ is the gradient operator, superscript $T$ is the transpose operator, $\Gamma_x$ is the surface of the domain $\Omega_x$ where traction forces are imposed, $\rho$ is the mass density, $\sigma$ the Cauchy stress tensor, $\overrightarrow{b}$ is the body force vector and $\overrightarrow{t}$ is the surface traction force vector.

### 2.2 Time integration

Solution of the problem expressed by equation (1) is done using an explicit time integration scheme. This is the most advocated scheme for integrating in the case of impact problems. Within an explicit algorithm, the elements of the solution at time $t_{n+1}$ depend only on the solution of the problem at time $t_n$. Stability conditions associated with this integration scheme imposes that the time-step size $\Delta t$ have be lower than a limit as discussed further. In this work, we are using the generalized-$\alpha$ explicit scheme proposed by Chung and Hulbert[6] 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:

$$\overset{\bullet\bullet}{\overrightarrow{x}}_{n+1} = \frac{\mathbf{M}^{-1}\left(\overrightarrow{F^{ext}}_n - \overrightarrow{F^{int}}_n\right) - \alpha_M \overset{\bullet\bullet}{\overrightarrow{x}}_n}{1-\alpha_M} \tag{4}$$

$$\overset{\bullet}{\overrightarrow{x}}_{n+1} = \overset{\bullet}{\overrightarrow{x}}_n + \Delta t\left[(1-\gamma)\overset{\bullet\bullet}{\overrightarrow{x}}_n + \gamma \overset{\bullet\bullet}{\overrightarrow{x}}_{n+1}\right] \tag{5}$$

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

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 $\alpha_M$, $\beta$ and $\gamma$ are linked to the value of the spectral radius $\rho_b$ 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 $\Delta t$ is limited, it depends on the maximal modal frequency $\omega_{max}$ and on the spectral radius $\rho_b$ by the following relation:

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

where $\gamma_s$ is a safety factor that accounts for the destabilizing effects of the non-linearities of the problem and $\Omega_s$ 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-$\alpha$ explicit integration flowchart is given by Algorithm 1. In this flowchart, the two steps 5b and 5f are the most CPU intensive ones. We focus now on some theoretical aspects of those two steps before presenting some parallelizing methods to apply.

### 2.2.1 Internal forces computation

According to the decomposition of the Cauchy stress tensor $\sigma$ into a deviatoric part $\mathbf{s} = \text{dev}[\sigma]$ and an hydrostatic part $p$, the hypo-elastic stress/strain relation can be written as follow:

$$\overset{\triangledown}{\mathbf{s}} = \mathbf{C} : \mathbf{D}; \quad \overset{\bullet}{p} = K\text{tr}[\mathbf{D}] \tag{10}$$

where $\overset{\triangledown}{\mathbf{s}}$ 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

**Algorithm 1** Flowchart for generalized-$\alpha$ explicit integration

1. Internal matrices computation: $\mathbf{N}$, $\mathbf{B}$, $\mathbf{J}$, det[$\mathbf{J}$]
2. Computation of the global mass matrix $\mathbf{M}$
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
   (b) Computation of the vectors $\overrightarrow{F^{int}}$ and $\overrightarrow{F^{ext}}$
   (c) Computation of the corrected quantities at $t_{n+1}$
   (d) If simulation complete, go to 6
   (e) Internal matrices computation: $\mathbf{B}$, $\mathbf{J}$, det[$\mathbf{J}$]
   (f) Computation of the stable time-step of the structure
   (g) Go to 5a
6. Output

velocity gradient $\mathbf{L} = \dot{\mathbf{F}}\ \mathbf{F}^{-1}$. In order to integrate equations (10), we adopt the use of elastic-predictor/plastic-corrector (radial-return mapping) strategy, see for example Refs.[2, 13, 14]. An elastic predictor for the stress tensor is calculated according to the Hooke's law by the following equation:

$$p_{n+1}^{tr} = p_n + K\text{tr}[\Delta e]; \quad \mathbf{s}_{n+1}^{tr} = \mathbf{s}_n + 2G\text{dev}[\Delta e] \tag{11}$$

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 (see Pantalé[10]) between increment $n$ and increment $n+1$. At this point of the computation, we introduce the von Mises criterion defined by the following relation:

$$f = \bar{\sigma} - \sigma^v = \sqrt{\frac{3}{2}\mathbf{s}_{n+1}^{tr} : \mathbf{s}_{n+1}^{tr}} - \sigma^v, \tag{12}$$

where $\sigma^v$ is the current yield stress of the material. If $f \leq 0$ then the predicted solution is physically admissible and the whole increment is assumed to be elastic ($\mathbf{s}_{n+1} = \mathbf{s}_{n+1}^{tr}$). If not, the consistency must be restored using the radial return-mapping algorithm reported in Algorithm 2.

**Algorithm 2** Radial return algorithm for an isotropic hardening flow law

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

### 2.2.2 Stable time-step computation

As presented in the time-integration part, the time-step size must be lower than a critical value as shown in equation (8). In our application, the value of $\omega_{max}$ is evaluated by the power iteration method proposed by Benson[3]. A updated version of the corresponding algorithm is given in Algorithm 3.

## 3 OBJECT-ORIENTED DESIGN

### 3.1 Overview of object-oriented programming

We have made the choice to develop the DynELA FEM code using Object-Oriented Programming (OOP), because this 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[9, 8, 7, 16]. More details concerning the numerical implementation of the code are given in Pantalé et al.[9, 11].

**Algorithm 3** Computation of the maximal model frequency

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

### 3.2 Finite element classes

As it can be found in other papers dealing with the implementation of FEM[8, 7, 16] we developed some specific classes for this application (see figure 1). This approach will simplify the parallelization of the code as we will see later.

## 4 PARALLELIZATION OF THE CODE

With the growing importance of microprocessor-based architectures using shared-memory processing (SMP) or distributed-memory processing (DMP) parallelized code has become more important in the las past years. In SMPs, all processors access the same shared memory as shown in figure 2, while in DMPs each processor has its own private memory.

The parallelization techniques used in FEM codes can be classified into two categories:

- the first one concerns DMPs where MPI (Message Passing Interface) is well established as high-performance parallel programming model.
- the second one concerns SMPs computers with the main use of special compiler directives. The OpenMP[4] standard was designed to provide a standard interface in Fortran

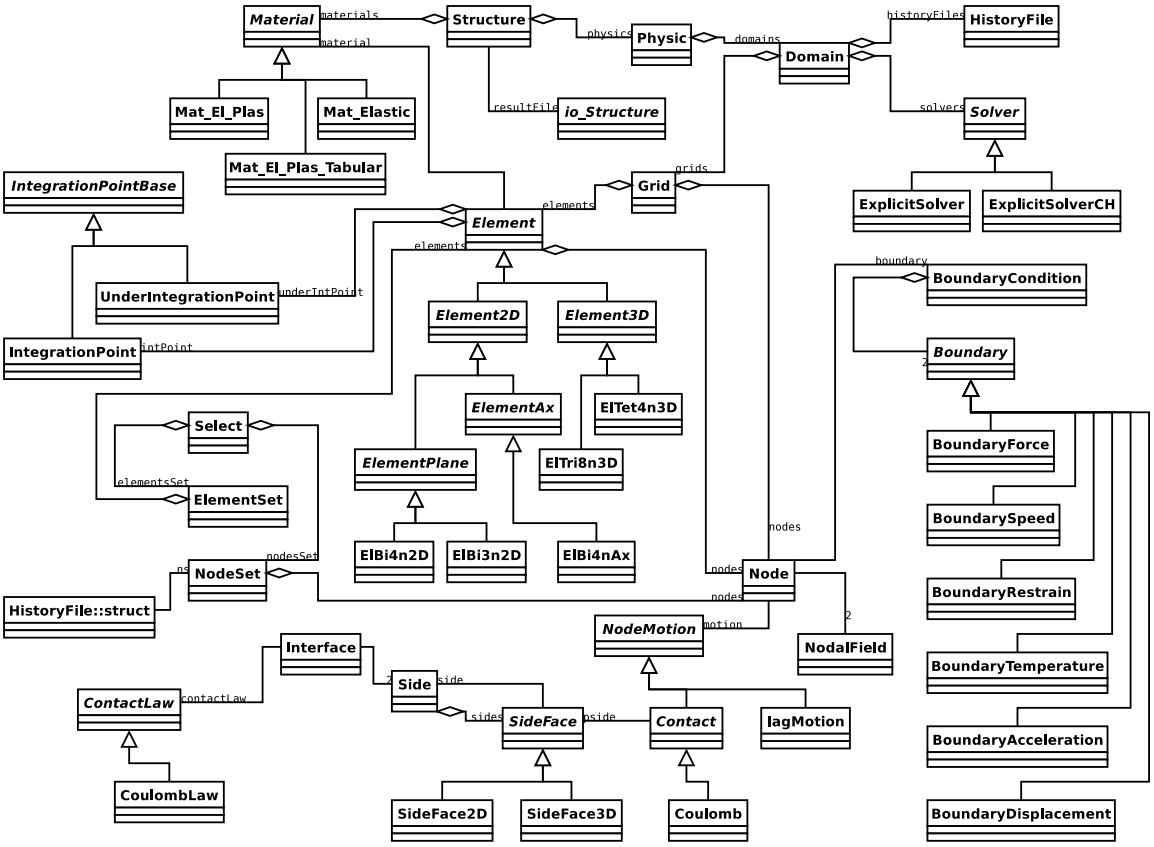


Figure 1: Simplified UML diagram of the Object oriented framework

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.

In this work, we focussed on local parallelization techniques to be applied on some CPU time consuming subroutines inside the explicit integration main loop of the program. Therefore, only the internal force vector and the stable time-step computations are parallelized using some OpenMP parallelization techniques. 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 8 Intel Xeon PIII 550/2Mb processors and 5 Gbytes 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.

As the development platform is a SMPs, the parallelization of DynELA is based on the use of OpenMP[4]. 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

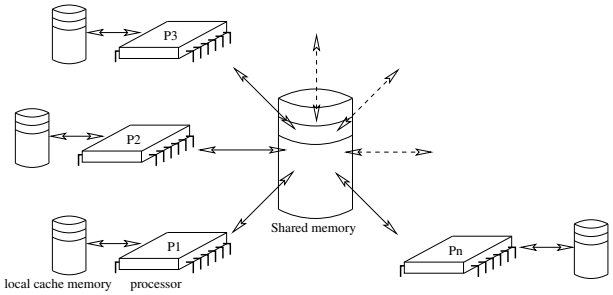


Figure 2: Shared-memory processing (SMP) architecture

figure 3. A thread is an instance of the program running on behalf of some user or process.

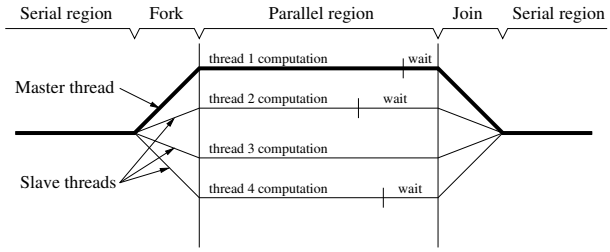


Figure 3: Fork-join parallelism

Parallelization with OpenMP can be done automatically, through compiler flags, or manually. We tested both methods, and as many other authors[15], found that the automatic parallelization of the code leads to very bad Speedup results. Manual parallelizing of the code is achieved by inserting 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 instructs the compiler that the next loop in the program must be forked, and the work must be distributed among available 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 *computeMatrices* method must be *thread-safe*.

The user may also define 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.

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. A *critical* directive is used to restrict access to the enclosed code to only one thread at a time. This is a 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 Chandra et al.[4] for further complements.

### 4.1 Load balancing

In dynamic computations, CPU time/element may vary from one element to an other during the computation of the plastic corrector because plastic flow occurs in restricted regions of the structure. As a consequence, the prediction of the CPU time needed for the computation of the internal force vector $\overrightarrow{F^{int}}$ 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 figure 3 where thread 2 is the faster one and thread 3 the slower one). To avoid such a situation, we developed a dynamic load balance in order to equilibrate the allocated processors work[10].

### 4.2 Benchmark test used for Speedup measures

#### 4.2.1 Impact of a copper rod

The benchmark test used to to compare the efficiency of the various proposed parallelization methods is the impact of a copper rod on a rigid wall. A comparison of numerical results obtained with the DynELA code and other numerical results has already been presented by Pantalé[9]. The initial dimensions of the rod are $r_0 = 3.2mm$ and $l_0 = 32.4mm$. The impact is assumed frictionless and the impact velocity is set to $V_i = 227m/s$. The final configuration is obtained after $80\mu s$. The constitutive law is elasto-plastic with a linear isotropic hardening. Material properties corresponding to an OHFC copper are reported in table 1. Only half of the

| | | |
|---|---|---|
| Young modulus | $E$ | $117.0\,GPa$ |
| Poisson ratio | $\nu$ | $0.35$ |
| density | $\rho$ | $8930\,kg/m^3$ |
| initial flow stress | $\sigma_v^0$ | $400.0\,MPa$ |
| linear hardening | $h$ | $100.0\,MPa$ |

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

axisymmetric geometry of the rod has been meshed in the model. Two different meshes are

used with 1000 ($10 \times 100$) and 6250 ($25 \times 250$) elements respectively. This quite large number of elements has been choosen to increase the computation time.

### 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 $\Delta t = 10ms$. In this application we use the Pentium benchmarking instruction *RDTSC* (Read Time Stamp Counter) 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 $\Delta t = \frac{1}{550E+06} \simeq 1.8ns$.

## 4.3 Internal forces computation parallelization

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 four different ways to parallelize the corresponding block with the influence on the Speedup. 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. 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 follows:

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

We present here after four different techniques from the simplest 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 figure 4 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.

```
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
```

Figure 4: Source code for the method 1 variant

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.

4. This method is similar to the previous one except that we introduce the dynamic load balance operator presented in section 4.1. See figure 5 for the corresponding source code fragment.

Table 2 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 ($n$):

$$s_p = \frac{T_s}{T_m}; \quad e_f = \frac{s_p}{n} \tag{13}$$

Variation in the number of CPU to use is done by specifying this value from the environment variable *OMP_NUM_THREADS*. Table 2 shows that this ratio can be over 100%, this case is usually called Super-linear Speedup. This result comes from the fact that, as a consequence

| | 1 CPU | 4 CPU | | | 8 CPU | | |
|---|---|---|---|---|---|---|---|
| method | time | time | Speedup | efficiency | time | Speedup | efficiency |
| 1 | 167.30 | 57.95 | 2.88 | 72.2% | 57.95 | 2.88 | 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% |

Table 2: Speedup of the $\overrightarrow{F^{int}}$ computation for various implementations

of the dispatching of the work, each processor needs less memory to store the local problem,

```
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();
```

Figure 5: Source code for the method 4 variant

and cache memory can be used in a more efficient way. If we run the same computation test with 6250 elements instead of 1000, we obtain an efficiency value of 90% for 8 processors, and always below 100% for 2 up-to 8 processors. Cache-missing seems to occur in this case. Figure 6 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. Of course this extra time is taken into account in the results presented. Dynamic load balance improves the efficiency by reducing the waiting time.

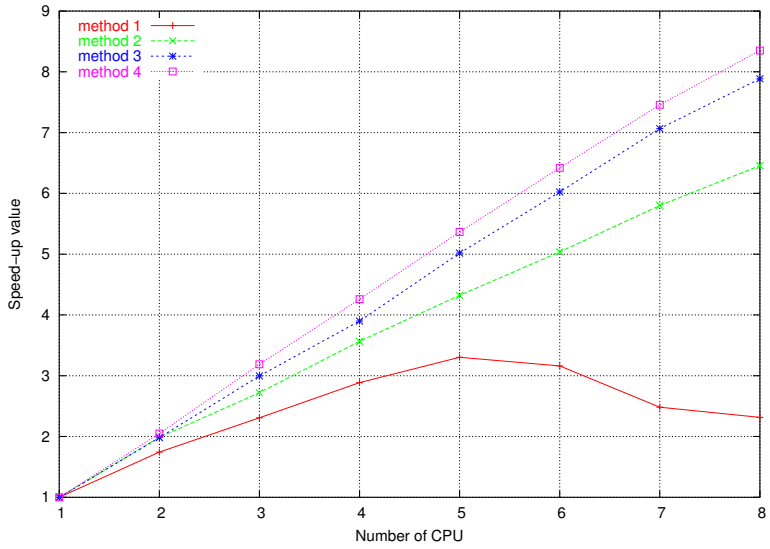


Figure 6: Speedup of the $\overrightarrow{F^{int}}$ computation for various implementations

### 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 an other 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 (3(a)iii) we can notice a writing instruction in the shared vector $\widehat{x}_n$. 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 $\widehat{x}_n^{(i)}$ where superscript $(i)$ represents the thread number and further to collect all vectors $\widehat{x}_n^{(i)}$ into a single vector $\widehat{x}_n$ using an efficient parallel collecting algorithm.

Figure 7 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 $\widehat{x}_n^{(i)}$ into a single vector $\widehat{x}_n$. 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

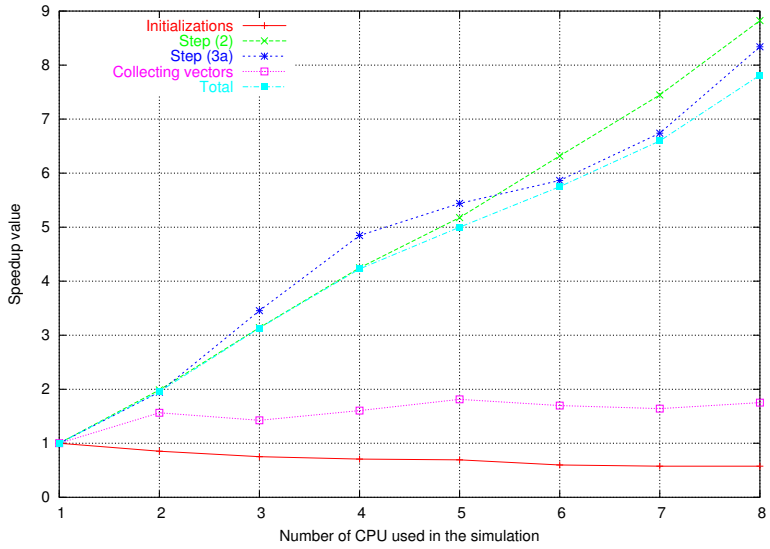


Figure 7: Speedup results for the time-step computation procedure

the collecting operation is compensated by the gain produced by the parallelization of 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

In this part, we simulate the impact of a cylindrical projectile into a closed cylindrical tube[1]. Only half of the axisymmetric geometry of the structure has been meshed in the model. Initial mesh is reported on the left side in figure 8. 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 $\sigma^v = A + B\bar{\varepsilon}^n$ are given by Pantalé et al.[12]. The projectile weight is $m = 44.1gr$ and the impact speed is $V_c = 80m/s$. The final con-

| projectile | | | | target | | | |
|---|---|---|---|---|---|---|---|
| $E$ | $193.6\,GPa$ | $A$ | $873\,MPa$ | $E$ | $74.2\,GPa$ | $A$ | $360\,MPa$ |
| $\nu$ | $0.3$ | $B$ | $748\,MPa$ | $\nu$ | $0.33$ | $B$ | $316\,MPa$ |
| $\rho$ | $7800\,kg/m^3$ | $n$ | $0.23$ | $\rho$ | $2784\,kg/m^3$ | $n$ | $0.28$ |

Table 3: Material properties of the projectile and the target for the dynamic traction test

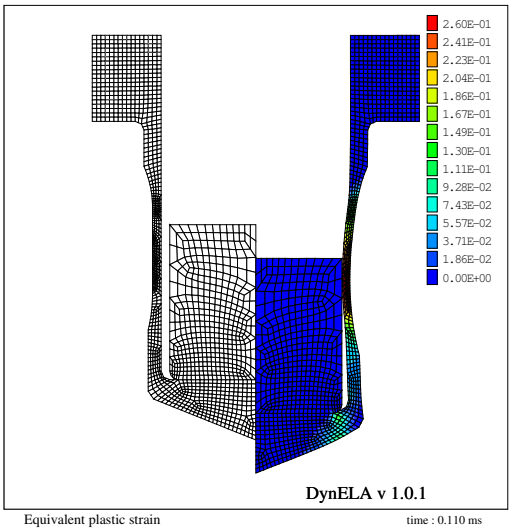


Figure 8: Dynamic traction: initial mesh and equivalent plastic strain contour-plot

figuration is obtained after $110\mu s$. Right side in figure 8 shows the equivalent plastic strain $\overline{\varepsilon}^p$ contour-plot at the end of the computation. A comparison of the numerical results obtained with DynELA and Abaqus/Explicit is reported in table 4 and shows a very good level of agreement.

| FEM code | $\overline{\varepsilon}^p_{max}$ | final length | inner diameter | thickness |
|---|---|---|---|---|
| DynELA | $0.260$ | $50.84\,mm$ | $10.07\,mm$ | $0.857\,mm$ |
| Abaqus | $0.259$ | $50.84\,mm$ | $10.08\,mm$ | $0.856\,mm$ |

Table 4: Comparison of numerical results for the dynamic traction test

Concerning the parallelization of the code, figure 9 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 slight falling off after 6 processors. With the parallelization of only the time-step computation and the internal force vector computation procedures, the total Speedup is $5.61$ for 8 processors. A more detailed analysis is presented in Pantalé[10].

## 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

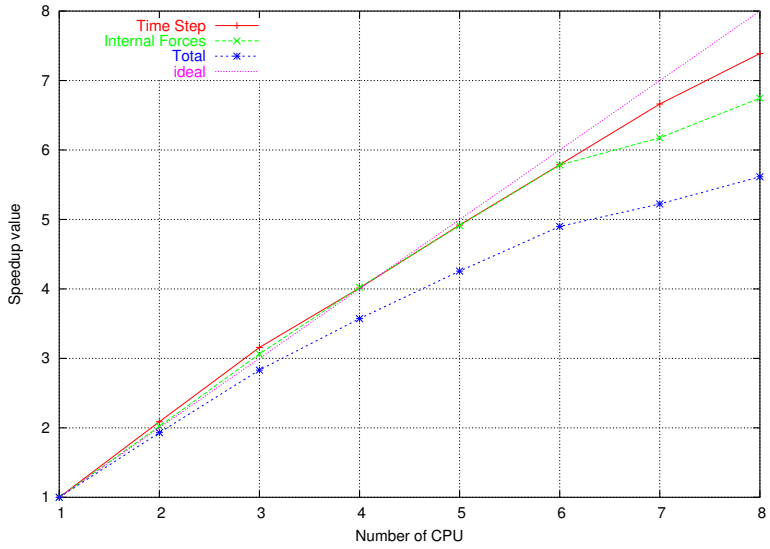


Figure 9: Speedup for the dynamic traction test

pragma directives will be ignored by the compiler). This enforces the portability of the code on different platforms.

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.

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] H. Abichou, O. Pantalé, I. Nistor, O. Dalverny, and S. Caperaa. Identification of metallic material behaviors under high-velocity impact: A new tensile test. In *15th Technical Meeting DYMAT*, Metz, 1-2 June 2004.

[2] T. Belytschko, W. K. Liu, and B. Moran. *Nonlinear Finite Element for Continua and Structures*. Wiley, 2000.

[3] D. J. Benson. Stable time step estimation for multi-material eulerian hydrocodes. *Computer Methods in Applied Mechanics and Engineering*, 167:191–205, 1998.

[4] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Academic press, 2001.

[5] J. Hoeflinger, P. Alavilli, T. Jackson, and B. Kuhn. Producing scalable performance with openmp: experimenta with two cfd applications. *Parallel Computing*, 27:391–413, 2001.

[6] G. M. Hulbert and J. Chung. Explicit time integration for structural dynamics with optimal numerical dissipation. *Computer Methods in Applied Mechanics and Engineering*, 137:175–188, 1996.

[7] R. I. Mackie. Object oriented programming of the finite element method. *International Journal for Numerical Methods in Engineering*, 35:425–436, 1992.

[8] G. R. Miller. An object oriented approach to structural analysis and design. *Computers and Structures*, 40(1):75–82, 1991.

[9] O. Pantalé. An object-oriented programming of an explicit dynamics code: Application to impact simulation. *Advances in Engineering Software*, 33(5):297–306, 5 2002.

[10] O. Pantalé. Parallelization of an object-oriented fem dynamics code: Influence of the strategies on the speedup. *to appear in Advances in Engineering Software*, 2005.

[11] O. Pantalé, S. Caperaa, and R. Rakotomalala. Development of an object oriented finite element program: application to metal forming and impact simulations. *Journal of Computational and Applied Mathematics*, 168(1-2):341–351, 2004.

[12] O. Pantalé, I. Nistor, and S. Caperaa. 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*, ISBN 973-640-012-3, Bucharest, 2003.

[13] J. P. Ponthot. Unified stress update algorithms for the numerical simulation of large deformation elasto-plastic and visco-plastic processes. *International Journal of Plasticity*, 18:91–126, 2002.

[14] J. C. Simo and T. J. R. Hughes. *Computational inelasticity*. Springer, 1998.

[15] E. L. Turner and H. Hu. A parallel cfd rotor code using openmp. *Advances in Engineering Software*, 32:665–671, 2001.

[16] N. Zabaras and A. Srikanth. Using objects to model finite deformation plasticity. *Engineering with Computers*, 15(Special Issue on Object Oriented Computational Mechanics Techniques):37–60, 1999.