Object-Oriented Approach and Distributed Finite Element Simulations

Piotr Breitkopf * — Yves Escaig **

* *Codiciel, UPS CNRS 856*
BP 511, PAT de la Vatine
32, rue Raymond Aron
F-76824 Mont-Saint-Aignan cedex

** *INSA de Rouen, Laboratoire de Mécanique de Rouen*
Place Emile Blondel, BP 8
F-76131 Mont-Saint-Aignan

ABSTRACT. *We present an application of object-oriented approach in the context of distributed computing in the field of structural engineering problems. In this work, conducted within the framework of a general purpose finite element code, we consider two types of distributed algorithms: the cooperation of heterogeneous computing systems and an algorithm for distributing the resolution of the finite element problem. In the first case, the major issue is the transparent distribution of the data base involving data structures and algorithms. In the first part of the present work, we present DDSM (Distributed Data Structures Manager) dealing with this first issue. The second case addressed is that of solution of linear systems by a domain decomposition direct method. Performance results given are those of a Cray T3D system. Communications and process control are implemented using the PVM library.*

RÉSUMÉ. *Une application de l'approche objet dans le contexte de la modélisation distribué des problèmes de mécanique est présentée. Ce travail est effectué dans le cadre d'un logiciel de simulation numérique utilisant la méthode des éléments finis. On considère deux types d'algorithmes distribués : la coopération de systèmes informatiques hétérogènes et la distribution de la résolution de problèmes par éléments finis. Dans le premier cas, la question centrale est celle de la distribution transparente de la base de données. Dans la première partie de l'article, on aborde cette question en présentant le logiciel DDSM (Distributed Data Structures Manager). Le second type d'algorithmes distribués est illustré par une application de la méthode de décomposition de domaines à la résolution de grands systèmes d'équations par une méthode directe. Des mesures de performance obtenues sur Cray T3D sont présentées. Les communications et le contrôle des processus distants sont implantées en utilisant la bibliothèque PVM.*

KEY WORDS: *distributed computing, object-oriented programming, finite element method, domain decomposition methods.*

MOTS-CLÉS : *calcul distribué, programmation orientée objet, méthode des éléments finis, méthodes de décomposition de domaines.*

1. Introduction

The maturity of the field of computational engineering entails the user having to deal with a growing complexity of individual software components. The simulation process involves a number specialized tools such as pre- and post-processors, linear, non-linear and dynamic solvers or auto-adaptive mesh generators. On the hardware side, multiple architecture systems are available: networks of RISC processor systems, vector processor systems, shared memory multiprocessor systems, massively parallel systems, networks of workstations and arrays (or clusters) of shared memory systems.

These two aspects are evidence of the need for new programming paradigms. TAn object-oriented approach permits the partition programs into manageable pieces that closely match the concepts of computational engineering. Objects interact and communicate with each other by exchanging messages. The interesting aspect of the object-oriented approach is that objects do not need to be in the same process or even on the same machine to send and receive messages back and forth to each other. In this sense, objects match well with distributed computing. The adoption of an object-oriented methodology may therefore lead to the distribution of the current centralized numerical simulation's process, limiting at the same time its complexity and increasing its overall performance.

The work presented in this paper has been conducted within the framework of the project SIC (*Système Interactif de Conception*) [BRE 92]. The aim is to conceive a general platform for various kinds of finite element simulations (structural mechanics, heat transfer, fluid mechanics, electromagnetism and any other kind of coupled problems involving concurrent simulations of different aspects of a complex phenomena). This platform allows tight cooperation between different development teams through sharing and reusing common software modules (data manager, solver, pre- and post-processing, finite elements, etc.).

The term "object oriented technology" has no precise definition. A simple use of a programming language does not make a program object-oriented. On the other hand, it is perfectly possible to use an object-oriented approach while programming in a low level language. The issue is to find the right software environment providing a framework for an object-oriented methodology as applied to the domain of application. Two approaches are open to the user:

– using a "true" object-oriented language,
– build a set of tools on top of an existing language.

In the domain of scientific computing, software availability, compatibility with existing code and performance considerations inhibit the former solution. In fact, languages such us Eiffel [MEY 92], Java [ARN 98] do not focus on computationally

intensive applications. The latter approach was used by the authors [STR 91] for the design of the C++ language built on top of the standard C language. The constraints of the project presented in the present work did not permit to directly use C++. For the reasons of compatibility with existing software, we chose the second solution, using Fortran as the base language. Our approach is based on an object-oriented data manager, briefly described in the first section.

The second aspect of our work concerning the distribution of the computations may be achieved in one of the two following ways:

– either by distributing the computational algorithms on a certain number of processing elements (usually identical) to reduce the elapsed execution time of the program,

– or by performing different phases of the simulation process possibly using different software concurrently on different computing systems, in order to optimize the use of available resources.

In the first case, a SPMD (Single Program Multiple Data) programming model is often used, meaning that each processor executes the same program but operates on different data. In the latter case, the main issue is the distribution of the database. In the present paper, we present a project of a simulation software, based on the finite element method, that implements both types of distribution previously mentioned. A key issue in this project is the control of remote process and the communication of data between heterogeneous processing elements.

The rest of the paper is set out in the following way. In the first two sections, we present our object manager along with a run time system for the distribution of objects. The next two sections concern performance aspects and a distributed implementation of the finite element method using a domain decomposition strategy.

2. Design of the Object Manager

While designing the data base for the SIC project, we tried to find a compromise between the following constraints:

– simplicity, reusability, flexibility

– high performance – because of intensive calculations involved in the finite element method, the data has to be accessed in a very efficient way, ideally as fast as a simple memory access

– compatibility with the extensive range of existing libraries and modules written mainly in Fortran

– support for parallel processing

– provide both compiled and interactive interface

The design of GO (*Gestionnaire d'Objets*) [AUN 90], the data manager of the project SIC. GO permits us to implement a number of features commonly found in object-oriented languages such as data abstraction, inheritance, encapsulation. We have to stress that GO is not an object oriented language. Its programming interface is of a relatively low level as it relies on libraries rather than on syntactic constructs. GO may be seen as a research vehicle that permits the experimentation of object-oriented concepts while applied to computationally intensive tasks.

The main characteristics of GO is that it was conceived as a small set of extensions to standard Fortran 77 language in order to provide high level data structures. In this way, existing code could be reused and the training of programmers reduced. The SIC project was originally conceived as an interactive environment providing basic building blocks for the assembly of complex simulations. All data structures, including internal ones manipulated by SIC are "objects". These objects can contain any data types, including fields of variable length as well as references to other objects.

2.1. *Encapsulation*

An important characteristic of GO is that it stores an explicit description of every object in the data base, though allowing generic service routines, like creating, copying, printing, working on any kind of objects

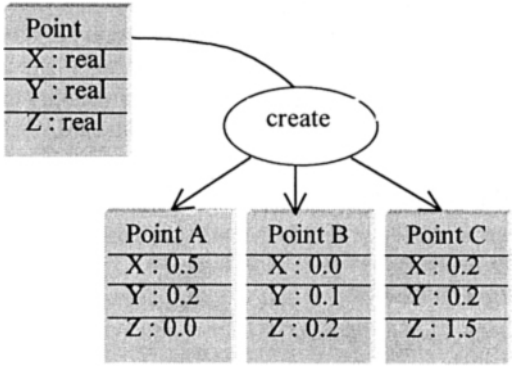

Figure 1. *"Object description" and individual objects*

The object's internal structure: the type and number of its components is defined in an external file. This file is converted into an "object description" object which serves as a pattern for creating actual objects of this type. The term "object" is currently used to refer to both object descriptions and to individual objects. Individual objects are referenced through a unique identifier that is independent from an actual memory location. The current memory location of objects is stored in an array, indexed by object identifiers. This means that referencing an object adds an overhead of one indirection. The independence between identifiers and memory locations allows transparent displacements of objects in the database, for garbage collection for example.

The second basic notion in an operator. An operator is a particular kind of object that references both the data in GO and the Fortran or C function which has to be executed with this data. The operators are accessible to an interactive user via a Matlab-like command-line interpreter. The important feature of the interpreter is that it is tightly coupled to GO in the way that all user input is converted into operator type-objects. In this way, the object manager does not differentiate between data and the executable code.

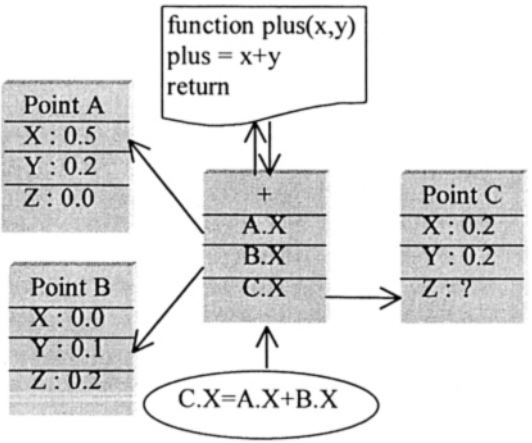

Figure 2. *User input converted into an operator-type object which in turn references other objects and Fortran code*

Figure 2 presents a simple command issued by the user: C.X = A.X + B.X. This command is translated into an operator-type object with pointers to the concerned items. The role of an operator-type object is to prepare data for a function implemented as a native Fortran method. More complex tasks may be accomplished by macro commands. In this case, a list of operators is created and some basic control structures permit the execution of loops, test logical conditions and redirect executions.

An object of type "object description" may be associated with a list of operators allowed to act on objects of a given type. This is achieved by defining links between an object descriptor and operators. A geometric object may for instance reference operators permitting it to be plotted or transformed (Figure 3).

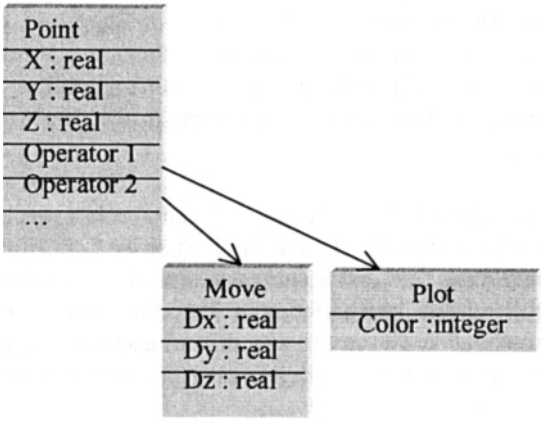

Figure 3. *An object descriptor and its member operators*

In the SIC system, the contents of an object is not protected. There is always a possibility for the user to modify any variable of an object by using a generic modification function. The same remark concerns the member operators. This kind of behavior is due to the problem of the granularity of the operators. The opposite strategy would be to protect internal data of an object by creating access functions. In this case, one would get a high number of low level functions. In our approach, we prefer a small number of generic access functions and medium-sized operators for other operations. The information hiding is therefore achieved rather by respecting the coding rules then by compiler level verification.

2.2. *Inheritance*

Inheritance permits to reuse existing objects for constructing new ones. Subclasses can add variables and methods to the ones they inherited from the superclass. GO provides only a simple inheritance model as opposed to c++ where multiple inheritance is allowed. The Figure 4 presents a simple case when an object "Point" serves as a base class description for an object "Circle". The object "Circle" is obtained from the object "Point" by adding the fields describing the normal to the circle and the radius. The original data representing the "Point" coordinates is reused and is interpreted as the center of the "Circle".

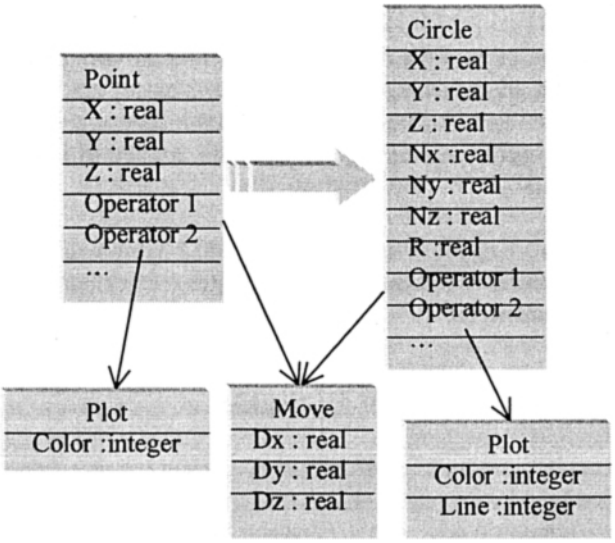

Figure 4. *Simple inheritance model : object "Circle" adds new fields Nx,Ny,Nz,R to an existing object "Point"*

The interest of an inheritance mechanism is that the derived object description reuses not only data of the base class object but also its operators. In our example, we notice that the "Move" function has the same meaning for the Point as for the Circle. In fact, when moving a circle, one has only to modify the coordinates of the center. The remaining attributes remain unchanged, so the original "Move" operator may be shared by both classes. Subclasses can also override inherited methods and provide specialized implementations for those methods. In our case, we have to define a new "Plot" function for a circle.

2.3. *Generic Behavior*

Programmers can implement methods that exhibit "generic" behaviors. Such methods exploit the object descriptors and may be implemented prior to the definition of the objects themselves. Several generic objects are predefined using GO at different levels ranging from container classes for storage of individual objects [AUN 90] through specialized programming patterns for finite element operations [BRE 92] and a complete framework for automatic sensitivity analysis and optimization of structures [MOR 93].

3. Distributing the Data Base

In this section, we present the run time system DDSM (Distributed Data Structure Manager) which allows an easy and efficient displacement of previously described objects between local memories of distributed processing elements. Chaos++ system [SAL 95] proposes an alternative approach applied to objects of the C++ programming language.

There are four major issues in the design of DDSM: remote data access, maintaining pointers across the network, support of heterogeneous computing environments and efficiency considerations.

3.1. *Remote Data Access*

In order to reduce the number of messages exchanged, an access to a remote object is replaced by a local copy of the entire remote object. This principle is equivalent to that of "ghost objects" of the Chaos++ run time system. In our approach however, the objects are accessed directly in the memory rather than through an access function. This compromise, due to the performance considerations, implies that there is no run time system to supervise the remote data access. Therefore, it is up to the application to validate the objects it will use. If the objects are local, the usual consistency validations take place. If the object exists in a remote memory, then the run time system gets a copy of it before resuming execution. Because a single object can have multiple copies, it is necessary to ensure the coherence of these copies. Again, as we do not use a function to access objects, we assume that the coherence is supervised by the application.

3.2. *Distribution of Objects Containing Pointers*

This issue concerns the distribution of objects containing pointers (in our case identifiers) to other objects. Supposing that an object points to another one in a local memory. The run time system must preserve this relation for the remote copies of the two objects. One solution consists in copying the two objects together and in recreating the relation when allocating objects in the remote memory. This is the solution implemented in Chaos++. However, this approach is not always acceptable when dealing with large graphs of objects. Our solution consists in introducing the notion of an "universal identifier" (which contains the local identifier of an object concatenated with a processor number) and in maintaining in each memory the correspondence between local and universal identifiers. When transferring an object to another memory, a generic function scans its content and replaces each local identifier by a universal one. The opposite operation takes place at the reception of

the object, just before putting it into the local object base. If the universal identifier does not yet correspond to a local object, then a new local virtual object appears. If later on, an object with the same universal identifier arrives, then it will take the local identifier of the placeholder object. Therefore, the relation between the two objects is restored.

3.3. *Support of Heterogeneous Computing Environments*

For basic data types, the XDR (eXternal Data Representation) library [KOP 95] ensures the consistency of coding between different binary representations. PVM [GAI 95], for example, uses XDR to support heterogeneous networks. This feature is not sufficient when handling complex data types like C language structures. This is also the case of our objects that can contain fields of different basic data types as well as variable length fields. Therefore, in our system, an XDR encoding takes place at the DDSM level rather than at the PVM level. This is done by a generic function and is transparent to the application.

3.4. *Efficiency*

In actual distributed hardware, the message latency is high as compared to the communication bandwidth. In order to limit this overhead it is necessary to reduce the number of messages. DDSM implements a buffering technique to transfer efficiently large quantities of "small" objects. Messages are sent only when the buffer is full or when the sender requests it.

4. Performance Evaluation

The aim of this section is to quantify the overhead of DDSM as compared to the raw performances of PVM. Performance evaluation tests have been conducted on a network of workstations (SGI Indigo R4000) connected by Ethernet. It can be noticed that all the tests have been run on a non-dedicated network. For this reason, we do not interpret the absolute values of our results. However, the relative values are significant. The following points have been analyzed:

– transfer of a single object with DDSM as compared to the transfer of an array representing the same amount of raw data using PVM, for different object sizes,

– transfer of an increasing number of objects of a given size with DDSM as compared to the transfer of a single object representing the same quantity of data also with DDSM,

– transfer of a large and complex graph of objects with DDSM as compared to the transfer of the same number of disconnected objects also with DDSM, with an increasing number of objects.

4.1. *Transfer of a Single Object*

The aim of these tests is to quantify the overhead of going through an additional software layer when transferring a single large object, made of one dynamic zone of increasing size. This object does not contain any pointer to other objects. Elapsed times are compared with a usual data transfer of an array using directly PVM. Performances obtained are shown in Figure 5.

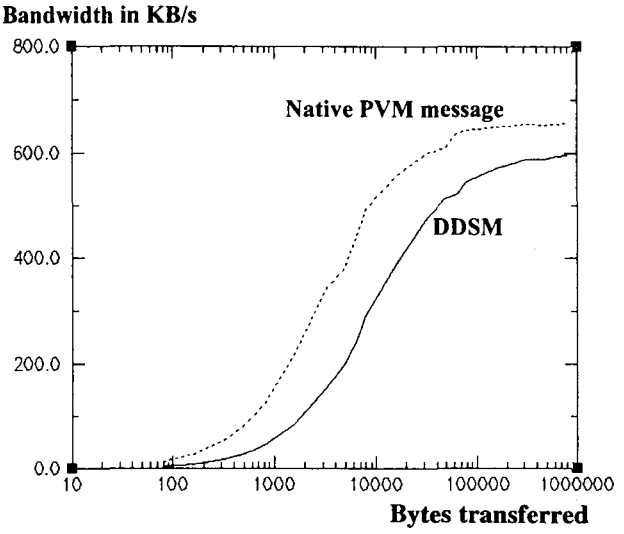

Figure 5. *Transfer rate for 1 object compared to raw PVM*

One can note that for a small amount of data transferred, the latency of DDSM is always higher than that of PVM. The relative importance of this overhead, due to the additional software layer, diminishes as the message size increases. The maximum bandwidth of DDSM is about 10% smaller what is mostly due to an additional copy of the data when creating a new object in the remote data base.

4.2. *Transfer of N Objects*

The aim of these tests is to quantify the additional work needed to transfer a set of N objects as compared to the transfer of a single object of equivalent size. An increasing number of objects of size of 800 bytes each is transferred. Note that this transfer needs only one PVM message due to the buffering technique used. The

The maximum bandwidth is about 25% smaller when transferring N objects. This is due to the overhead associated with the initialization of the transfer of each object and to the time needed to create these objects in the remote database. performances obtained are shown in Figure 6.

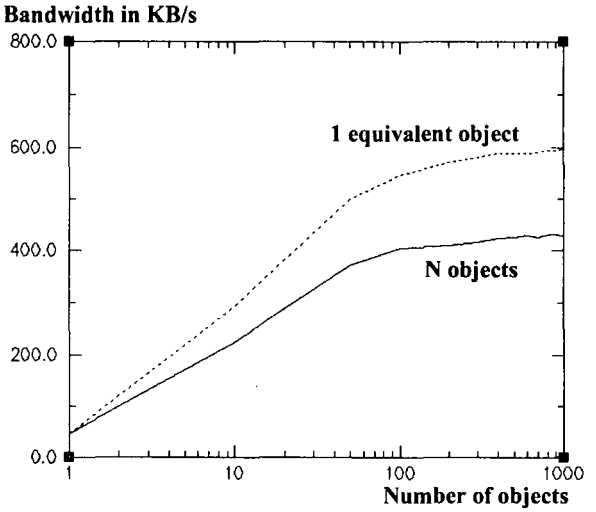

Figure 6. *Transfer rate for N small objects as compared to transfer of one big object*

4.3. *Transfer of a Complex Graph of Objects*

These tests correspond to the transfer of a large, complex graph of objects that corresponds to a finite element mesh (see next section). Tests are run for different mesh sizes, i.e., different numbers of objects. Results are compared to the transfer of the same number of simple objects that do not contain different fields nor pointers. The description of the tests is given in Table 1. Performances obtained are shown in Figure 7.

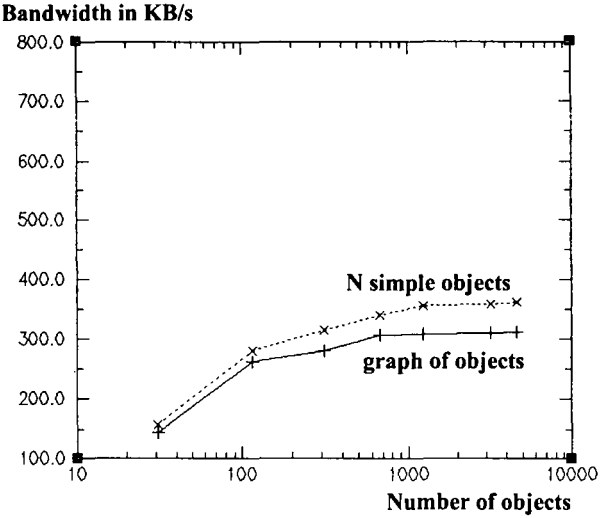

Figure 7. *Transfer rate for complex graph of objects compared to N simple objects*

Mesh	mesh 1	mesh 2	mesh 3	mesh 4	mesh 5	mesh 6	mesh 7	mesh 8
Nb of objects	31	116	313	670	1235	2056	3181	4658
Size in bytes	$5*10^3$	$17*10^3$	$46*10^3$	$96*10^3$	$174*10^3$	$288*10^3$	$443*10^3$	$645*10^3$
Size per object	176	146	146	143	140	140	140	139

Table 1. *Characteristics of the different graphs of objects*

These results show that the overhead necessary to handle links between objects is rather small. The value of the bandwidth for these tests is not very high because of low size of an average object (140 bytes).

All these results show that the performances of DDSM are comparable to those of PVM and confirm the scalability of our approach. Considering the new services added by DDSM, the overall performance of numerical applications enabled by this environment compensates its overhead. However, in particular cases involving a high number of very small objects, the overhead of DDSM can accumulate. Therefore, DDSM should be used with caution in applications where the communications are intensive.

5. Applications

Two classes of distributed finite element applications may be implemented using DDSM. The first class of applications aims to optimally use a set of computing resources: a network involving several processors and a 3D graphic workstation running the user interface. There are many different scenarios for such applications. One of them concerns the analysis of loosely coupled systems. In this case, two models of one problem are executed concurrently and an exchange of data takes place at fixed time intervals. The data transfer must be transparent to the user and very efficient. In this scenario, there is a different process and a different object base on each machine. Communication of objects among these bases is achieved using DDSM.

The second class of applications concerns Single Program Multiple Data (SPMD) programming models. We apply this approach for the distribution of a single finite element model over a network of processing elements. In the following section, we describe a distributed domain decomposition method. All the data of a finite element problem are represented by a graph of objects (see Figure 8 for a simple view of this graph).

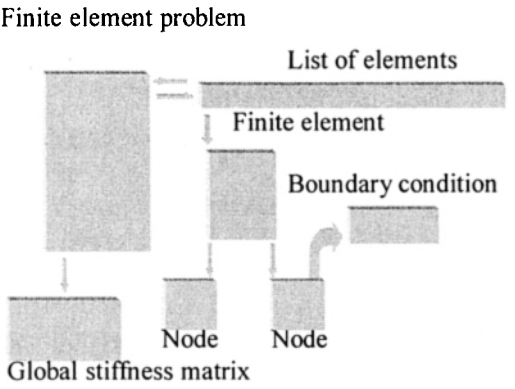

Figure 8. *Partial view of the data graph for a finite element problem*

The graph representing the finite element problem is partitioned into subdomains and distributed to different processing elements.

5.1. *Distributed Domain Decomposition Direct Method*

In this application, we aim to use a certain number of processing elements, usually identical, to reduce the elapsed execution time of the program. On distributed memory systems, the privileged approach for parallelizing the finite element method is the use of domain decomposition methods. The domain decomposition methods solve first the initial problem independently on each subdomain, and then they add constraints to fit the local solutions on the boundaries between the subdomains. For example, in the Schur Complement Method, the constraint is the equality of forces at the interfaces between subdomains. Detailed references can be found in [FAH 94]. For the Schur Complement Method, the decomposition algorithm can be written in the case of two subdomains:

1. Renumber equations so that the internal degrees of freedom (dof) of each subdomain appear first

$$[k]\{u\} = \{f\} \quad \Leftrightarrow \quad \begin{bmatrix} k_{11} & 0 & k_{13} \\ 0 & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \end{Bmatrix} = \begin{Bmatrix} f_1 \\ f_2 \\ f_3 \end{Bmatrix}$$

2. Eliminate all internal dof

$$\begin{bmatrix} k_{11} & 0 & k_{13} \\ 0 & k_{22} & k_{23} \\ 0 & 0 & \bar{k}_{33} \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \end{Bmatrix} = \begin{Bmatrix} f_1 \\ f_2 \\ \bar{f}_3 \end{Bmatrix}$$

where

$$\bar{k}_{33} = k_{33} - k_{31}k_{11}^{-1}k_{13} - k_{32}k_{22}^{-1}k_{23}$$
$$\bar{f}_3 = f_3 - k_{31}k_{11}^{-1}f_1 - k_{32}k_{22}^{-1}f_2$$

3. Solve the interface problem

$$\left[k_{33} - k_{31}k_{11}^{-1}k_{13} - k_{32}k_{22}^{-1}k_{23}\right]\{u_3\} = f_3 - k_{31}k_{11}^{-1}f_1 - k_{32}k_{22}^{-1}f_2$$

4. Back-substitute interface solution on internal degrees of freedom

$$\begin{Bmatrix} u_1 \\ u_2 \\ u_3 \end{Bmatrix} = \begin{Bmatrix} k_{11}^{-1}f_1 + k_{11}^{-1}k_{13}u_3 \\ k_{22}^{-1}f_2 + k_{22}^{-1}k_{23}u_3 \\ u_3 \end{Bmatrix}$$

The conditioning of structural mechanic problems implies the choice of direct rather than iterative methods for eliminating internal dof and for solving the interface problem [ESC 94].

Steps 2 and 3 of the algorithm are the most time consuming ones. Step 2 involves only calculations local to subdomains. Internal unknowns for each subdomain can be eliminated concurrently. For Step 3, the interface problem matrix is distributed over the local memories of the processing elements and it is factorized and solved in parallel. The main issue for this parallelization is the relation between the number of subdomains and the number of processors. The analysis of the sequential execution times shows that:

– total execution time depends on the number of subdomains,

– execution time of Step 2 is dominant for a small number of subdomains (up to 8) and tends to decrease as the number of subdomains increases,

– execution time of Step 3 increases with the number of subdomains, and becomes dominant from about 16 to 32 subdomains.

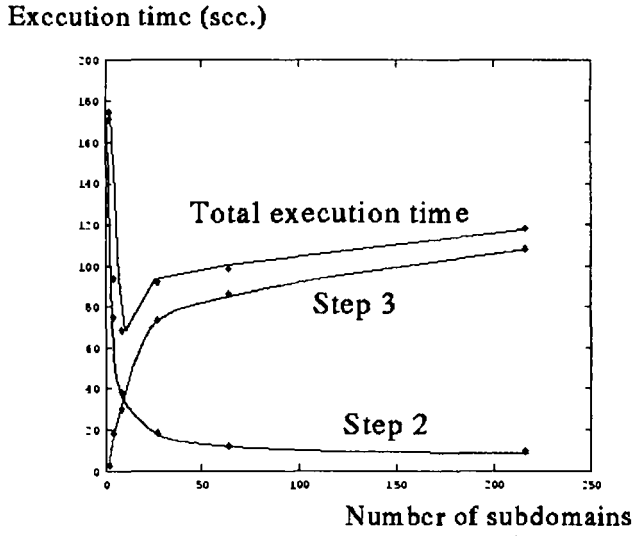

Figure 9. *Sequential performance of a 3D test case with 6591 dof*

Typical execution times are shown in Figure 9. A straightforward choice is to assign one processor per subdomain. For large numbers of processors, most of the execution time is spent in Step 3, which requires large amount of communication and does not lead to high efficiency. Therefore, it could be more interesting to assign several processors per subdomain. On the other hand, load balancing problems occur in the cases when the computing times of Step 2 for each subdomain are not constant. To reduce these problems, it would be necessary to have more tasks and consequently more subdomains than processors. This is in contradiction with the previous remark. In the rest of the paper, the choice of one processor per subdomain will be assumed.

The implementation of the domain decomposition method on a distributed memory system can be described as follows:

– (S1) distribute the data of the finite element problem (nodes, elements, boundary condition, material characteristics, etc.) of the subdomains to the processors using DDSM;
– (S2) for each subdomain concurrently, eliminate all internal dof;
– (S3) for each processor:
  - allocate one part of the interface problem matrix,
  - send the locally condensed matrix (matrices) k_{33} to all other processors,
  - receive the other condensed matrices and assemble the elements which belong to the local part of the interface problem matrix;
– (S4) factorize the interface problem matrix (eventually in parallel);
– (S5) solve the lower and upper triangular systems of the interface problem matrix;

– (S6) for each subdomain concurrently, back substitute the solution of the interface problem on internal dof.

This algorithm is currently being implemented on a Cray T3D system. Performance analysis of the algorithm raises the following issues: there is no possible direct comparison with sequential execution time because the problems solved in parallel do not fit into the memory of a single processor. A comparison with a sequential machine (with a sufficient memory) having a different processor should take into account the relative performances of the processors that are difficult to evaluate (non-linear behavior). For these reasons, the performances presented concentrate on the elapsed (wall clock) execution time for different number of processing elements (PE). These execution times are compared to those obtained sequentially on a Cray J916. As mentioned before, the size of the problem solved depends on the number of PEs.

The test problem considered is a 3D cube partitioned regularly into 4, 8, 16 and 32 subdomains. The execution times obtained are given in Figure 10.

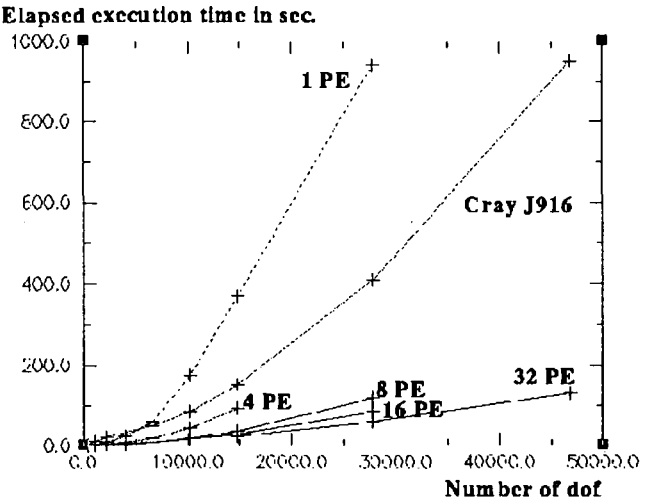

Figure 10. *Elapsed execution times for different numbers of Pes*

The following remarks can be drawn from these results:

– for small numbers of PEs (4 and 8), efficiency approaches 100%,

– for larger numbers of PEs (16 and 32), at fixed problem size the efficiency drops rapidly,

– for the same numbers of PEs (16 and 32), the efficiency increases with the problem size.

The above conclusions mean that the proposed parallel algorithm is not well scalable at fixed problem size, but is scalable if the problem size increases with the number of processors.

6. Conclusion

We have presented different aspects of distributed computing in the context of structural engineering problems. Distributed computing was applied for optimizing the use of heterogeneous computing resources. A runtime system for communicating objects between distributed object bases has been developed. Next, we demonstrated the use of distributed parallel systems to reduce significantly the execution time of finite element simulations. A domain decomposition direct method was applied. The same communication system could be used for these different classes of applications for a wide range of computing systems: networks of heterogeneous workstations, vector supercomputers, massively parallel systems.

The actual work forms a basis for future research aiming at optimization of all the steps of direct domain decomposition methods: communication and assembly of condensed matrices, factorization and resolution of the interface problem, using several processors per subdomain, load balancing problems.

Acknowledgments

The authors gratefully acknowledge the use of the Cray T3D system of the CEA (*Commissariat à l'Energie Atomique*) of Grenoble.

7. References

[ARN 98] K. ARNOLD, J. GOSLING, *The Java™ Programming Language*, second Edition, Addison Wesley, 1998.

[AUN 90] S. AUNAY, Architecture de logiciels de modélisation et traitements distribués, thèse UTC, Université de Technologie de Compiègne, 1990.

[BRE 92] P. BREITKOPF, G. TOUZOT, Architecture des logiciels et langages de modélisation, *Revue Européenne des Eléments Finis*, 1(3) :333-368, 1992.

[ESC 94] Y. ESCAIG, M. VAYSSADE, G. TOUZOT, Une méthode de décomposition de domaines multifrontale multiniveaux, *Revue Européenne des Eléments Finis*, 3(3) :311-337, 1994.

[FAH 94] C. FAHRAT, F.X. ROUX, Implicit parallel processing in structural mechanics, Monograph, *Computational Mechanics Advances*, 1994.

[GEI 95] A. GEIST, J. DONGARRA, W. JIANG, R. MANCHEK, V. SUNDREAM, *PVM : Parallel Virtual Machine ; A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1995.

[KOP 95] C. KOPP, *Introduction to NFS Performance, Part 1*, Open Systems Review, September, 1995.

[MEY 92] BERTRAND MEYER, *Eiffel: The Language*, Prentice Hall, second edition, 1992.

[MOR 93] L. MORANÇAY, Représentation paramétrée et modélisation de sytsèmes physiques pour la conception optimale, thèse UTC, Université de Technologie de Compiègne, 1993.

[SAL 95] J. SAILTZ, R. PONNUSAMMY, S. SHARMA, B. MOON, Y-S HWANG, M. UYSAL, R. DAS, A manual for the CHAOS runtime library, Technical Report CS-TR-3437, University of Maryland, 1995.

[STR 91] B. STRUSTROUP, *The C++ Programming Language*, Second Edition, Addison Wesley, 1991.