Parallel Finite Element Computations on a Network Using NCS

F. H. Bertrand and P. A. Tanguy
Dept. of Chemical Engineering & CERSIM, Université Laval, Québec, Canada, G1K 7P4

Research has been carried out on the use of NCS (Network Computing System) to distribute the processing of a finite element application to multiple computers at once. Coarse-grained parallelism of specific routines on loosely coupled CPUs has been implemented and tested. In a step-by-step fashion, the method of using NCS to convert finite element software POLY2D is explained for its application in parallel computing resources throughout a network. For the modeling of a simple conductive heat transfer problem, the distributed version of POLY2D on a homogeneous token ring network of nine Apollo workstations is used. Some timings given are compared to those obtained from a standard nondistributed simulation.

Introduction

The rapid development of vector computing and the advent of new computing architectures have strongly altered the engineering computing environment over the last decade. The situation has evolved from the classical concept of blind terminals sharing the power of a central computer in the 70s toward the networking of stand-alone workstations sharing common resources ranging from specialized peripherals for mass storage and printing to high-end vector computers. Because the latest workstations offer even more CPU power than many classical mainframes, the user has now access to a most powerful computing environment that tackles complex engineering problems which was considered as outstandingly difficult only a few years ago.

Parallel computing, a concept which has also gained acceptance in recent years, harnesses the power of several CPUs to reduce the computation time for a single application. As a quantum leap for the reduction of today's fastest clock periods ($\sim$ 1 ns) seems most unlikely, even with GaAs chips, parallelism increasingly is considered as the only way to speed-up the process. Parallel computing can be divided into three categories:

1. *Massive parallelism* uses a very large number of microprocessor-based CPUs simultaneously to run one or several applications. The operating system handles the problem of contention (communication bottleneck of processors) and reducing communication time among all the processors, intrinsic to parallelism and most problematic with massively parallel computers. Although still considered by many as rather exotic, commercial computers (cellular automata) based on this concept have already appeared such as Hypercube and Connection Machine. Unfortunately, massively parallel computers still lack efficient software tools (compilers, debuggers,...) for high-level programming languages such as FORTRAN and C, thus slowing down their surge in the engineering community.

2. In *parallelism on multiprocessor computers* (typically up to 32 processors), the CPUs are tightly coupled and generally share a common memory. This approach to parallelism has proven very successful and is now implemented in many commercial computers ranging from workstations to extremely large-scale computers. Synchronization of the processes and access to shared data are handled by the operating system. Again, the problem of developing efficient compilers is still a real one, although more or less efficient parallel FORTRAN compilers have started to appear.

3. In *network parallelism*, also called *parallel distributed computing*, a crowd of loosely coupled processors are used to run a single application. Such a concept is new for scientific computing, but has been widely used in operating systems and related software products such as NFS.

The main objective of this work is to show that network computing is a valuable concept that offers many advantages in engineering computations. In the following, the method of using a software product called NCS to convert finite element software POLY2D will be described for use in parallel computing resources spread out on a network. It should be noted

Correspondence concerning this article should be addressed to P. A. Tanguy.

that the upcoming methodology is by no means restricted to finite elements and could be applied to any engineering problems whose parts are parallelizable.

After recalling the essence of the finite element formulation when applied to a conductive heat transfer equation, software product NCS will be introduced. Next, some basic principles of network computing will be explained, as well as how the time-consuming assembly process can be divided into individual parallel tasks, each of which is associated with an available computer. Finally, some timings will be compared to those obtained from a standard application.

Finite Element Method

The finite element method is one of the best numerical techniques for the resolution of transport phenomena (Finlayson, 1972). This technique consists of breaking up the computational domain into adjacent subdomains called elements, in which the governing equations are discretized using Galerkin test functions and low-order polynomial approximations. The concept of parallelism is inherent in the finite element method. Indeed, elemental discretizations are built independent of one another, so that all the approximations can be built simultaneously provided a sufficient number of processors are available (Figure 1). Of course, other steps in finite element computations are suitable for parallelism like, for instance, the resolution of the matrix problem. The assembly process, however, constitutes one of the most expensive parts of a finite element application and, depending on the problem at hand, may even dominate the resolution of the matrix system.

We consider a conductive heat transfer problem at steady state over a domain Ω as governed by:

$$\text{div } (k \text{ grad } T) = f \text{ on } \Omega \tag{1}$$

where k is the thermal conductivity, T the temperature, and f a heat source. In many problems of industrial interest, this simple equation cannot be solved analytically, and a numerical procedure must be used. In a finite element context, the weak variational formulation of Eq. 1 must be first derived, which yields:

$$(\text{grad } \Phi, k \text{ grad } T) = (\Phi, f) \text{ on } \Omega \tag{2}$$

where Φ is any test function in the space of temperature, and (Φ, f) denotes the standard inner product in the L^2 space. The righthand side of Eq. 2 is written in a general form and includes the contribution of the heat source and the boundary conditions.

The resolution of the weak problem by the finite element method is straightforward. Equation 2 is first expressed as the sum of elemental contributions:

$$\Sigma_{el}\{(\text{grad } \Phi, k \text{ grad } T) - (\Phi, F)\}_{el} = 0 \tag{3}$$

Then, the discretization is carried out, and the element matrices and righthand sides are built element by element, yielding the global matrix:

$$[K]\{T\} = \{F\} \tag{4}$$

where $[K] = \Sigma_{el}[K]_{el}$ and $F = \Sigma_{el}\{F\}_{el}$ (Figure 1). As explained before, this assembly procedure is a natural candidate for distributed processing, since the element computations are completely independent of one another.

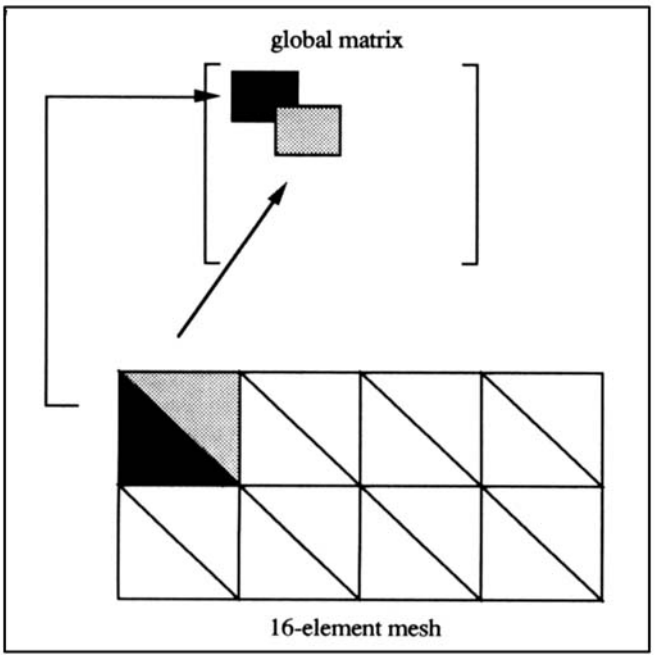

Figure 1. Building of element matrices and assembly.

Network Computing System

Network Computing System (NCS) (Kong, 1990) is a set of portable tools that enables one user to distribute the processing of one application to remote computers. It was designed to support situations where both data and program execution are distributed across a homogeneous or a heterogeneous network. For the sake of portability, it is written in *C* and can therefore be run on many different platforms. NCS implements a client-server communication mechanism (Colouris, 1988), in which client processes send request messages to server processes that in turn send back reply messages (Figure 2). In our case, we can define a client as a process that makes remote procedure calls (RPC), thus requesting that some work be executed on a specific remote computer. On the other hand, we say that a server is a process that implements a set of operations called an interface and provides access to a CPU. Consequently,

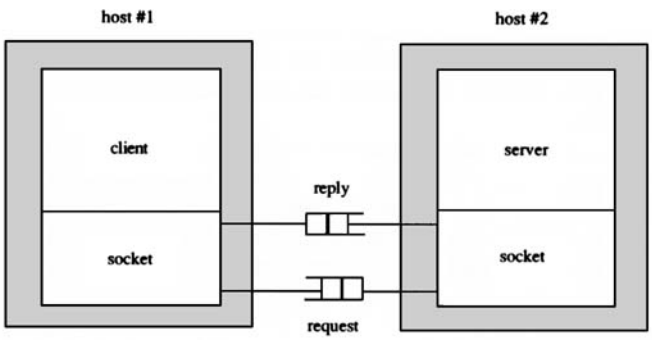

Figure 2. Client-server concept.

whenever a server receives a request from a client, it performs the desired operation and then sends back its response to the calling client.

NCS can be divided into three major components: a RPC runtime library, a Network Interface Definition Language (NIDL) compiler, and a Location Broker.

The RPC runtime library enables a local program to call procedures that are to be executed on a remote computer. RPCs are not written explicitly in the application, but are provided by a set of routines generated automatically by the NIDL compiler. This set of routines makes up what is referred to as the client and the server stubs. These are needed to bridge the gap between the client and the server, so that whenever the former makes a remote procedure call, the execution of the appropriate routine may take place as if it were residing on the local host. In particular, they handle any data transfer inherent to the RPC mechanism and can deal with data conversion in a heterogeneous environment, if necessary. From a practical point of view, the stubs are generated through the NIDL compilation of a user-supplied interface definition like the one to be introduced in the next section.

Finally, the Location Broker can be viewed as a utility program that one can use to query a database that contains information about the available network resources. Its main role is to enable the client to obtain at runtime the address of a required resource like the address of a server exporting a specific interface.

CPS

Concurrent Programming Support (CPS) (Apollo Computer, 1987) is a set of routines that can be used to create and manage multiple and concurrent threads of execution called tasks within a single process. Though it is not strictly part of NCS, it can be used to break down a complex operation into smaller pieces and run them in parallel. Along with NCS, CPS then offers the possibility to parallelize a portion of code in a loosely coupled fashion, whereby the resulting tasks run on different remote CPUs.

Overhead

RPC communications between client and server is not free. RPC overhead time or the time attributable to the RPC setup and breakdown only, without server execution time, can be modeled as (Francisco, 1988):

$$\text{RPC overhead time} = \text{Null RPC time} + \text{Time/byte} * n\text{bytes} \quad (5)$$

where the null RPC time corresponds to the time for an RPC with no data transfer and where $n$bytes stands for the number of bytes to be transferred. In real applications, many factors influence performance achievable through network computing. These are client and server compute powers for processing their respective RPC operations, data conversion in a heterogeneous environment when necessary, communications protocols such as DDS (Domain Distributed Services) or IP (DARPA Internet Protocol), network size, and network activity. On our network, which comprises among all a dozen Apollo workstations, a transfer rate of 150 kbytes/s with a start-up time (time for a null RPC) of 10 ms (elapsed time) has been measured on the average, using NCS version 1.5.1. Consequently, NCS may become a burden that does not pay off if used inappropriately. As seen in Eq. 5, RPC overhead time is linearly proportional to the length of the remote procedure parameter list. For a nonparallel application, prior to making a single RPC, one should make sure that RPC time is less than non-RPC time, where non-RPC time stands for the time needed to execute the procedure in a standard way and where RPC time is best expressed as:

$$\text{RPC time} = \text{server execution time} + \text{RPC overhead time} \quad (6)$$

For a parallel application, the situation is more complex because many threads of execution are run concurrently on remote hosts. Nevertheless, RPC overhead time as given by Eq. 5 is still valid so that, potential developers should remember that the larger the parameter list, the larger the overhead time will be.

Distributed finite element computations

The step-by-step method of using NCS (and CPS) to convert a finite element code is explained here for use in parallel computing resources throughout a network. As in Figure 1, the process of assembling the global matrix (and the global right-hand side also called residual) can be divided into individual pieces. For this purpose, commercial code POLY2D, which is a 2-D finite element code for the resolution of isothermal or nonisothermal fluid flow problems, will be used.

In POLY2D, the assembly process is achieved according to Figure 3 and implemented through the following FORTRAN

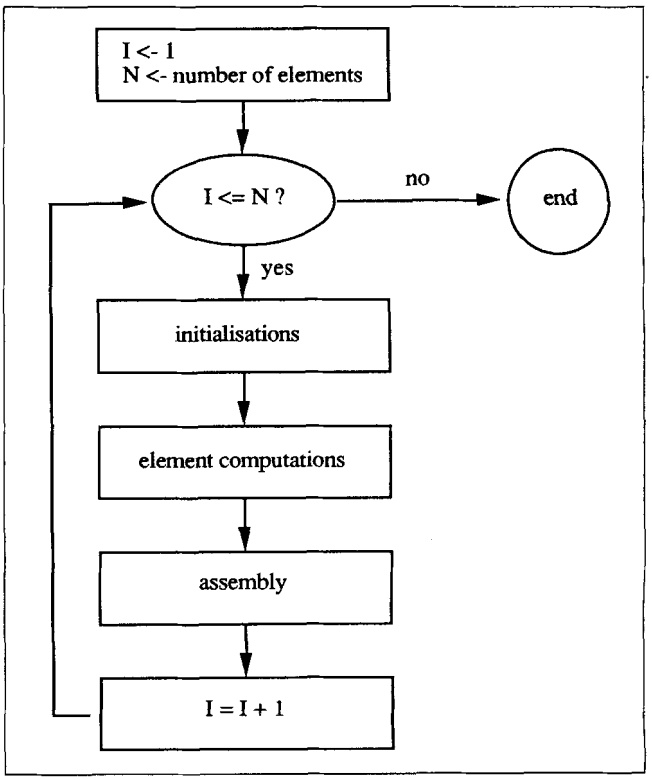

Figure 3. Assembly process.

DO loop:

```
DO   I=1,   number_of_elements
     CALL init (...)
     CALL element (...)
     CALL assembly (...)
END DO
```

In other words, for each element, certain initializations are performed (**init**). Then, one element matrix and one element residual are computed (**element**), and finally these element data structures are assembled into a global matrix and a global residual (**assembly**). In the standard version of POLY2D, these computations are performed element by element in a sequential manner. Under NCS and CPS, these same computations will be performed in parallel. More precisely, because they access shared data, procedures **init** and **element** will be executed on the client host, whereas element matrices and residuals will be computed remotely through RPCs. Figure 4 summarizes this whole idea.

In the context of NCS, a systematic approach leads us now to consider the following topics:

- Definition of an interface in NIDL syntax
- Writing of the server program
- Writing of the client program.

Of course, discussing these topics in full length would be far beyond the scope of this article. Consequently, the reader is referred to the book written by the developers of NCS (Kong, 1990) for thorough details on the matter.

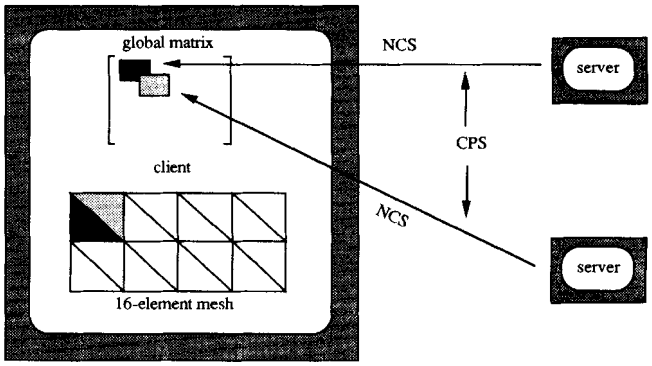

Figure 4. Building of element matrices and assembly: parallel approach.

Interface

The definition of an interface written in NIDL syntax constitutes the first step toward distributing procedure **element** whose goal is to compute one element matrix and one element righthand side. Its purpose is twofold: 1. to determine the calling syntax of remote procedure **element**, that is the list of parameters that must be transferred between client and server; 2. to generate, through a NIDL compilation, stubs that are

```
%c
[
uuid(46718333f000.0d.00.01.e2.95.00.00.00),
version(1)
]

/* bytes transferred : in = 857; out = 336; total 1.17Kb (8 ms) */

interface ELEMENT {

void element$element (
    handle_t [in]       h,
    double   [in]       vfbit[54],
    double   [in]       vpgit[6],
    int      [in]      *nfbit,
    int      [in]      *npgit,
    int      [in]      *ndle,
    int      [in]      *ndlec,
    double   [in]      *diffus,
    double   [in]      *ro,
    double   [in]      *cp,
    double   [in]      *visc,
    int      [in]      *ifac,
    int      [in]       kconnt[12],
    double   [in,out]   vfe[6],
    double   [in]       vcore[12],
    double   [in]       vt[6],
    double   [in]       vu[6],
    double   [in]       vv[6],
    double   [in]       vj[5],
    double   [in]       vnu[6],
    double   [in]       vnru[6],
    double   [in]       vnzu[6],
    double   [in,out]   vke[6][6],
    double   [in]       vnnu[6][6],
    double   [in]       vnnr[6][6],
    double   [in]       vnnz[6][6],
    double   [in]       vcoret[6]
);
}
```

Figure 5. Definition of interface ELEMENT in NIDL syntax.

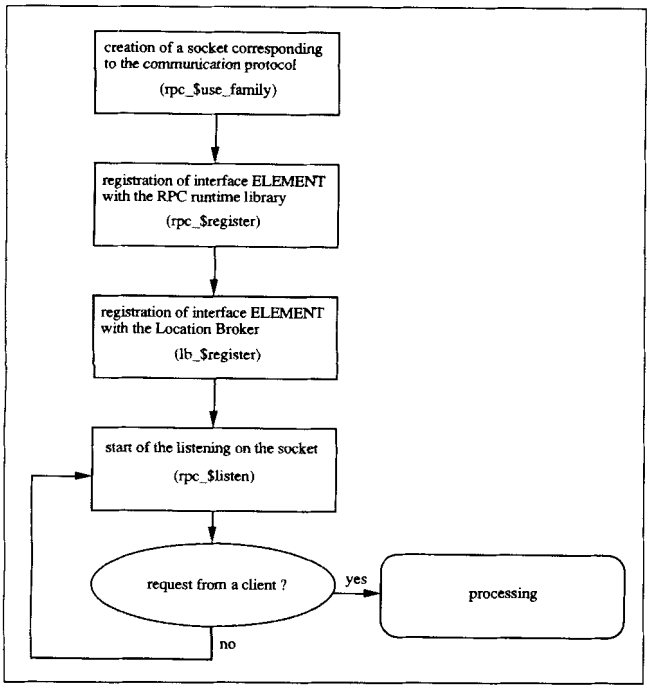

Figure 6. Server program.

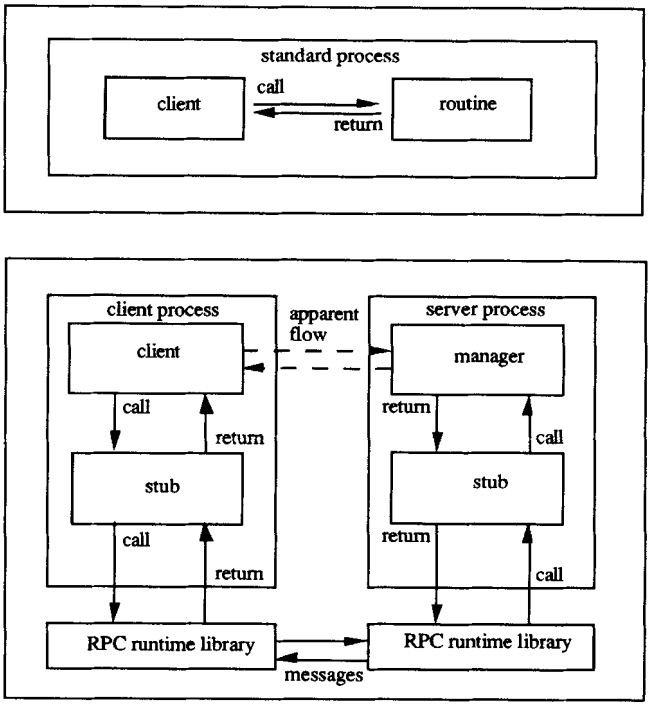

Figure 7. Standard call and remote procedure call via NCS.

needed to link client and server. We recall that these stubs are written in C and make remote procedure calls look like local calls. Interface ELEMENT that exports procedure **element** is given in Figure 5. It contains a unique universal identifier (uuid) useful to the Location Broker, a version number, and a parameter list corresponding explicitly to procedure **element**. One can readily notice integer and real variables as well as arrays. For each such parameter, the type is given along with the status, **in**, **out**, or both. As expected, **in** means that the parameter is to be passed from client to server, and **out**, from server to client. Accordingly, 1.17 kbytes of data are to be transferred for each RPC, which amounts to roughly 8 ms of elapsed time. Furthermore, the asterisk before certain parameters indicates, as in the C language, that the latter are to be passed by reference as opposed to argument passing by value. Finally, data structure **h** is called a handle and transmits information to the RPC runtime library as to the location of the server that is to take over and perform the requested operation. As it will be seen shortly, **h** is given a value in the client application using appropriate NCS system calls.

Server program

The next step toward distributing procedure **element** consists of writing a server program. Typically, the server program written in C, contains four basic operations processed through four NCS system calls, as summarized in Figure 6. First, an endpoint of communication is created for either DDS (Apollo Token Ring) or IP (Ethernet) protocols. This endpoint, which complies with the Berkeley UNIX socket abstraction for interprocess communication (IPC), enables the server to receive messages from clients. Next, interface **ELEMENT** is registered with the RPC runtime library so that any request for it can be dispatched to the server. Third, interface **ELEMENT** along with the server socket address created above are registered with the Location Broker. This information may then be used by clients when they need to locate available resources. At this point, the server may start listening on its socket for any requests from calling clients. When such a request gets through, the server stub performs specific formatting operations on transmitted data and then asks a module called manager to take over and call FORTRAN procedure **element**. One element matrix and one element residual are then computed, gathered into packets by the server stub, and returned to the client (Figure 7).

This form of a server program is quite general and can therefore be viewed as a template that needs only a few modifications to adapt to different distributed applications. On a heterogeneous network, however, provision must often be made for data conversion, as is the case for communication between a 32-bit workstation and a 64-bit supercomputer. Again, the reader is referred to (Colouris, 1980) for more details on this topic.

Finally, server application **ELEMENT_SERVER** can be built as follows:

- Create a server stub through a compilation of interface **ELEMENT** by the NIDL compiler.
- Compile the server stub, the server program, and the manager module.
- Link the resulting binaries.

Client program

The last step consists of writing the client program. It can be viewed as an augmented version of POLY2D whose extra code lines, written in C, will serve to initiate specific NCS and CPS system calls that handle the parallel execution of the assembly loop (Figure 8). First, the address of all servers exporting interface ELEMENT is looked up in the Location Broker database. Next, each available server is bound to the

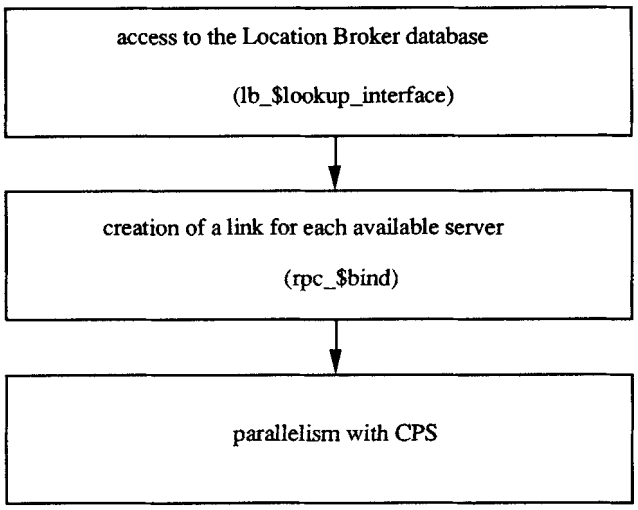

Figure 8. Client program: parallelism of the assembly loop.

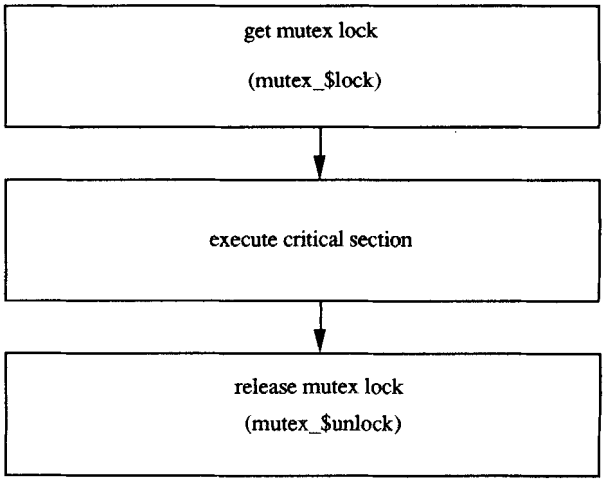

Figure 10. Access to shared data via a mutual exclusion mechanism (CPS).

client so that communication may take place. This is achieved by assigning the address of the corresponding server to a variable called the handle, defined before. Once this is done, parallel computations for the assembly process may begin, as described in Figure 9. A task is associated to each server and initiated on the client host. The first action of the task is to pop one entry from a shared stack of unprocessed element numbers. After preliminary initializations on the client host, it makes a remote procedure call for procedure **element** to its corresponding server. The latter then takes over, computes the related matrix and residual, and returns them to the waiting task on the client host. These are next assembled into shared global matrix and residual. Afterwards, the task starts over with a new element number and the loop continues until the stack gets empty. Then, the task is killed by the main process. When all the running tasks are destroyed, the main process resumes its execution with the factoring of the matrix system and the completion of the simulation.

The whole assembly process is performed by a crowd of tasks running in parallel and accessing shared data. A mutual exclusion (mutex) mechanism (Silbershatz, 1988) provided by CPS is used to control the access to those shared data. Briefly, it uses a semaphore called a mutual exclusion lock that a task must obtain prior to accessing a shared variable and release on termination (Figure 10).

A fault recovery routine is also provided to handle situations where a server fails to respond within a reasonable amount of time. In such a case, the element number that was being processed by this server is pushed back onto the top of the stack so that it can be taken over by another task. The task associated with this server is also killed immediately.

Finally, client application **ELEMENT_CLIENT** can be built as follows:

- Create a client stub through a compilation of interface **ELEMENT** by the NIDL compiler.
- Compile the client stub and the client program.
- Link the resulting binaries.

Figure 11 shows the building of client and server applications.

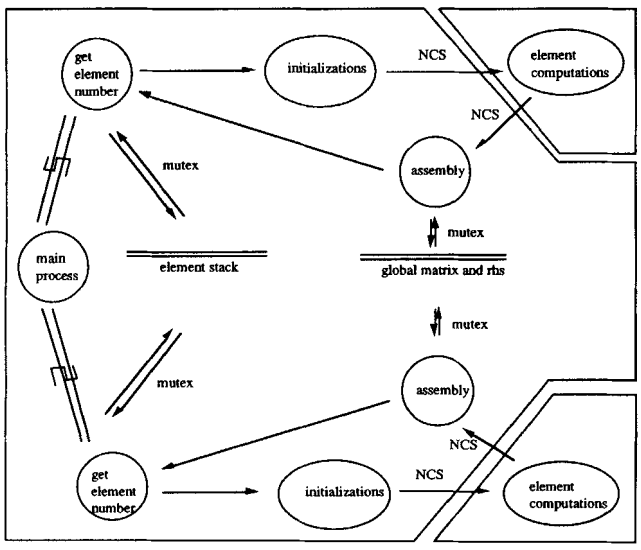

Figure 9. Assembly process using two servers.

Performance

This section aims at assessing the performance of the distributed version of POLY2D which was devised in the previous sections. To achieve this goal, some timings are presented and compared to those obtained with a nondistributed version of POLY2D. The test consists of resolving, using POLY2D, a typical conductive heat transfer problem. The mesh is made of 5,600 quadratic triangular elements resulting in a linear system of approximately 10,000 equations.

The simulations were performed on an Apollo token ring network using from one to nine DN3500 workstations on each of which a server program was initiated in background. Figure 12 shows the elapsed time for the assembly process (including the elemental computations) vs. the number of servers. A simulation using nine servers took 130 seconds as opposed to 650 seconds for a standard simulation, which gives a speed-up factor of 5. In a similar way, the total elapsed time vs. the number of servers is plotted in Figure 13. Again, it can be noted that a simulation with nine servers took 230 seconds as

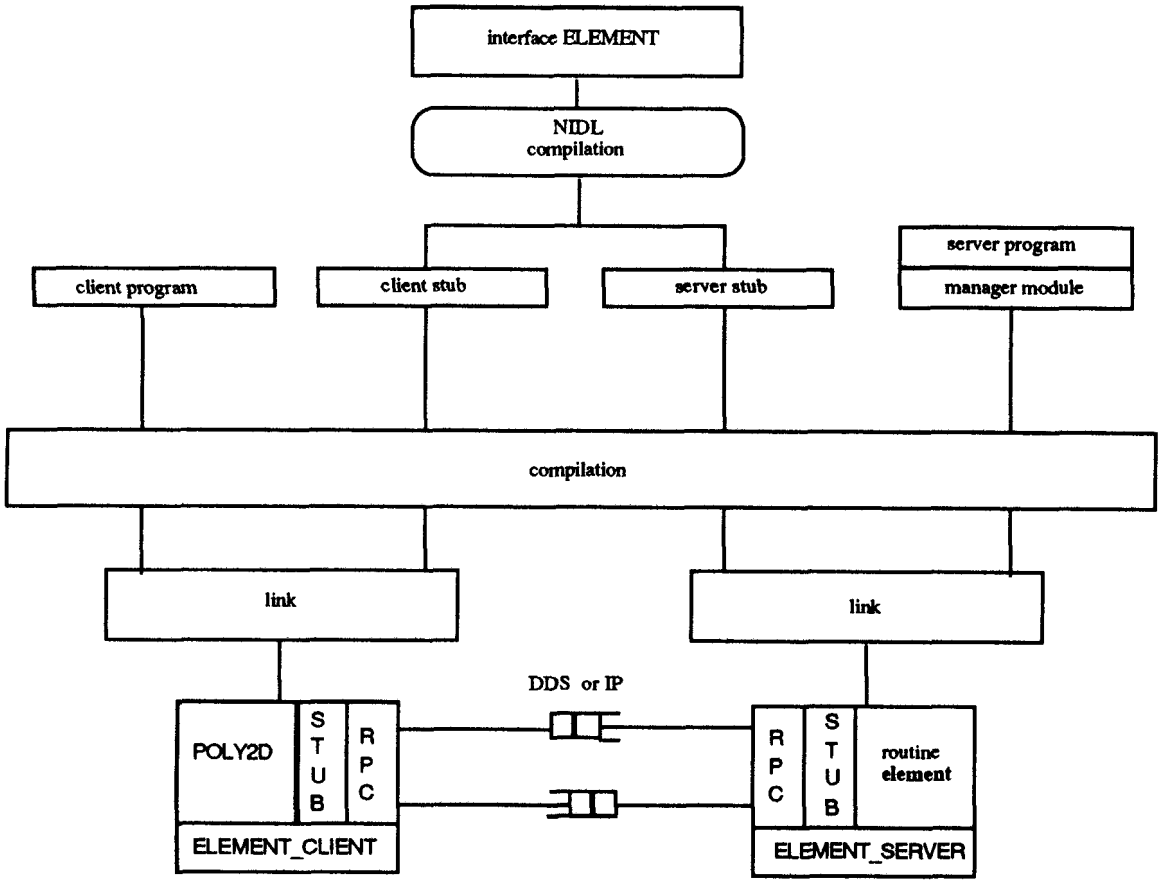

Figure 11. Building of client and server applications.

opposed to roughly 760 seconds for a standard simulation, yielding an overall speed-up of 3.3.

An interesting point in Figures 12 and 13 is that both curves decrease exponentially with the number of servers. Ultimately, simulations using eight and nine servers, respectively, take up approximately the same amount of time. This can be easily explained by the fact that, when information comes back from the servers, it must be assembled into shared data on the client host. As its capacity is limited, there is an optimal number of servers for this application, beyond which a bottleneck occurs. Obviously, one way to increase this optimal number would be to use a faster host.

Parallel computations were carried out with two sets of conditions: when all the workstations are in use (for instance, during the day) and the traffic on the network varies, and when everything is idle. For our application, only minor differences were noted as for the timings obtained. This is likely due to the fact that for each remote procedure call, only a little amount

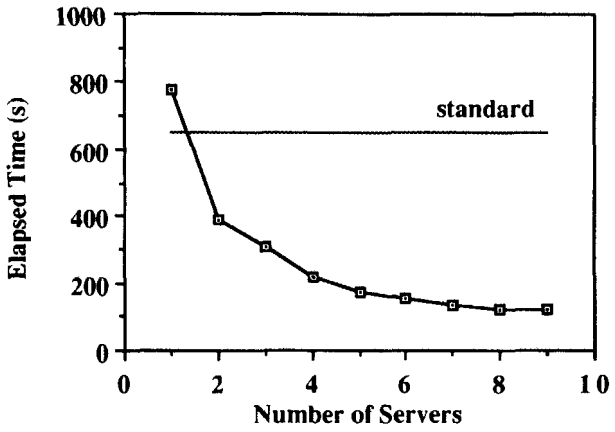

Figure 12. Elapsed time for the assembly process vs. the number of servers.

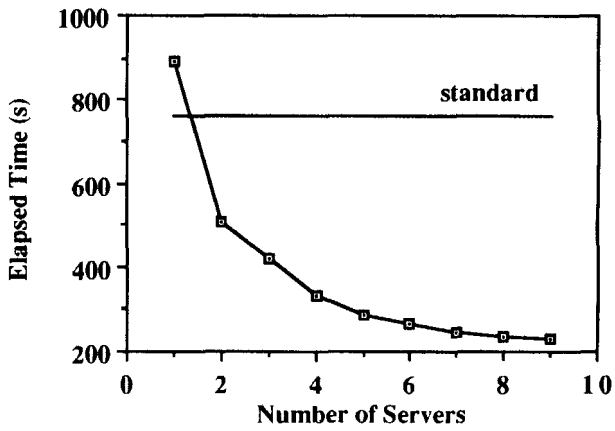

Figure 13. Total elapsed time vs. the number of servers.

of data (1.17 kbytes) needs to be transferred and a few CPU burst are taken up by the server host to perform the appropriate computations.

Parallel computing on a network should obey Amdahl's law (Lakshmivarahan, 1990). If v stands for the speed-up factor for execution in parallel mode and k stands for the percentage of parallelizable code, Amdahl's law predicts that the speed-up factor for the overall simulation, V, should be:

$$V = v/[k + v(1-k)] \tag{7}$$

For our problem, it can be measured that 85% of the elapsed time is taken up by computations of element matrices and residuals, which are performed by the parallel tasks. Consequently, $k = 0.85$. If we set, for instance, $v = 9$, we get using the above formula a speed-up factor of 4.1, as opposed to the value of 3.3 obtained in practice. The difference between these two values is due to the overhead associated with NCS and CPS. In fact, the ratio of actual speed-up to the theoretical speed-up does not seem to be affected by the number of servers and is of the order of $\alpha = 0.8$ in our experiments.

To reflect the overhead in our computations, Amdahl's law can be modified as:

$$V = \{ v/[k + v(1-k)] \} * \alpha \tag{8}$$

Concluding Remarks

The objective of this article is to explain how NCS (and CPS) can be used to distribute the processing of a finite element application to multiple remote computers strung out on a network.

It was shown that at the cost of some initial effort required to become acquainted with these software products, an interesting speed-up factor can be obtained, even though communication between client and server is relatively slow. What appears to be important is that increased throughput was made possible at no extra cost, using existing hardware, and that a few modifications were needed to the standard FORTRAN code.

Acknowledgment

The authors are indebted to the Apollo Computer Division of Hewlett-Packard and the Natural Sciences and Engineering Research Council of Canada for their financial support.

Literature Cited

Apollo Computer, *Concurrent Programming Support (CPS) Reference*, Order No. 010233 (1987).

Colouris, G. F., and J. Dollimore, *Distributed Systems*, Addison-Wesley (1988).

Finlayson, A., *The Method of Weighted Residuals and Variational Principles*, Academic Press (1972).

Francisco, C. R., and D. Labossière, "A Method for Estimating Application Performance in a Network Computing Environment Using the LINPACK Benchmark as an Example," Apollo Computer, Internal Report (1988).

Kong, M., et al., *Network Computing System Reference Manual*, Prentice Hall (1990).

Lakshmivarahan, S., and S. K. Dhall, *Analysis and Design of Parallel Algorithms*, McGraw-Hill (1990).

Silbershatz, A., and J. L. Peterson, *Operating System Concepts*, Addison-Wesley (1988).

Manuscript received Jan. 15, 1991, and revision received Oct. 15, 1991.