August 1992

THE EVOLUTION OF SCIENTIFIC COMPUTING ON PARALLEL COMPUTERS

Gérard Meurant
CEA, Centre d'Etudes de Limeil–Valenton
94195 Villeneuve St Georges cedex

This paper was translated to Portuguese by C. de Moura

Abstract. In this paper, we would like to indicate what could be, from our point of view, the future of large scale scientific computing during the next ten years. We will try to identify what are the major issues to be addressed and how the situation could evolve. First, we define what are the goals of today's and tomorrow's scientific computing showing what is the evolution of the users' needs. Then, we summarize the trends in the architecture of parallel and, particularly, massively parallel computers and we stress the needs for a better programming environment and better software tools to efficiently use the new architectures. We show some examples of the changes in algorithmic development due to the availability of new architectures. Finally, we give some conclusions, showing that even though a lot of progresses have been made, users are still waiting for more performant parallel architectures, software tools and algorithms.

1. The goals of scientific computing.

The main task of scientific computing is computer simulation of physical problems. Usually, one starts with the description of a physical problem and the first thing to do for solving the problem is to describe a model of the physical reality. This phase involves physicists and applied mathematicians. In most of the problems this paper is interested in, this results in a set of coupled partial differential equations. Today, people are interested in simulating phenomenon occurring in three dimensional geometries and most problems are also time dependent. But there can be even more complicated problems where we have to deal with the phase space which results in equations to be solved in six or seven dimensional spaces. An example of this is the simulation of particle transport (neutrons, charged particles, etc...) arising in the nuclear industry and in plasma physics.

It is fair to say that, in most cases the accuracy of the models has been limited by the computers on which the problems are finally solved. There is no point in using sophisticated models involving a lot of variables and equations, if we are not able to numerically compute them. Therefore, more often, scientists derived approximate models from the complete ones. An example of this is the study of turbulence. As we cannot yet solve the full Navier–Stokes equations around a complete aircraft with enough accuracy to describe all the fine details that are needed, approximate models must be used (cf. [Paterson]). But, as the researchers want to get results closer and closer to the field experiments, the models are getting more and more complex and therefore, there is always a need for computers with faster computing capabilities and larger memories.

When one has set up a model of the problem involving a small enough number of variables and equations, then a numerical method must be defined and used to go from the model with an infinite number of degrees of freedom to a finite dimensional problem. For a large class of problems this finally amounts to (repeatedly) solving a system of linear equations. Here also, to get a better accuracy, one needs more and more discretisation points leading to larger finite dimensional problems to be solved.

When the problem is numerically solved, the physicist and the applied mathematician want, of course, to look at the results of the computation hopefully in a condensed graphical form.

To assess the needs of scientific computing, we must distinguish between two main activities: research and design. This distinction can be also done across computer users, most of the research being done in universities and national laboratories and most of the design being done by industrial companies. By 1990 the main users of the fastest computers (usually called supercomputers) were:

• national laboratories working for strategic areas (LANL, LLNL, etc... in the US, CEA, CERN, etc... in Europe)

• Research centers working for industrial applications (like INRIA, ONERA, IFP, METEO, EDF, etc... in France)

• Laboratories from industrial companies (Boeing, Aerospatiale, CGG, PSA, Renault, etc...)

• University research centers (NSF centers, CCVR, etc...).

The industry possesses only around 40 % of the installed supercomputers (by 1990). The situation is much different with smaller computers as, for instance, about 70 % of Convex machines are located in industrial companies.

There is some difference between both activities, research and design.

In leading edge research, most computations are done to get some insights in problems that have not been solved before. Even if we suppose that the researchers have the computing power they need (which is very seldom the case), usually the runs can be quite long (say hundreds of hours) as they do not have to be repeated too often. It has been common these last years to speak of the Grand Challenges of Science, even though they are difficult to define, to refer to problems the solution of which is far beyond today's computer capabilities. One of this challenges is the simulation of the global evolution of the earth climate.

The opposite is true for analysis and design in industrial companies. Until some

recent years, development methodology has been based on an empirical approach including a lot of long and expensive field test experiments. Now, we begin to have the tools to use computer simulation to reduce the design time and more importantly, to include the computations in an optimization loop. As industrial companies have to improve the quality of their products and to lower their design costs, it is very important to reduce the analysis/design time. As computer simulations are available, analysis can be performed on design conditions, more configurations and variations can be tested, the simulations can help defining the field test experiments, the good solutions are found earlier in the development cycle through computations avoiding the need for late and expensive modifications. But, for the computations to be included in the design/optimization cycle, they must be fast enough, that is at most a few hours and hopefully a few minutes, such that the engineers can be able to test a lot of configurations and to try many variations. Finally, this should lead us to automatic optimization and design for many problems. This implies that powerful inexpensive computing engines are available.

Examples of this strategy of product development is aircraft engineering, both for airplanes and engines design (cf. [Karadimas], [Paterson]) and the automobile industry where, for instance, computer crashes simulation is now common within most companies. Also, combustion simulations can reduce dramatically the time design for a car engine. This can lead to reduce the development cycle of a car from four–five years to two years giving an edge to the manufacturers using fast and sophisticated simulation tools.

Of course, the development of numerical analysis and computer technology is sought to permit that today's research methodologies can become the tools of tomorrow's design. Therefore, we need to improve the computers capabilities to reach two goals: to push as far as we can the leading edge of research to develop new methodologies and to solve problems that were beyond our present scope and secondly to greatly improve the models and methods that are used today in design and optimization in

the industry and to reduce the turnaround time for most studies.

Can we assess what are and what will be the needs of the users in the near future? Of course, this is difficult as it is different from one area to another. But, we can give, at least, some examples.

An important area is CFD (computational fluid dynamics) that is involved in the solution of many problems. For instance, direct numerical simulation of turbulence (compressible turbulence in particular), appears to be a very important tool. To study complex physical models, one needs millions of degrees of freedom (called points in the sequel). By the time of writing, the biggest computations that the author has seen were made on Cray and Japanese Supercomputers and involved runs of hundred of hours while using the full capacity of the computers. Two numerical simulations we have recently seen in a CFD meeting show these trends.

The first one is a simulation around a two dimensional cross section of a wing flying at jet liners speed. There were around 4 million points. Although much information can be extracted from such computations, there were still not enough points to predict properly the angle at which the wing would stall (if the aircraft decides to climb at too steep an angle it will eventually fall).

The second one is a computation with 7 million points for the simulation of compressible turbulence in a star. At this resolution one begins to observe interesting statistical behaviors but more is desired.

What computers are needed for CFD computations? Minisuper computers and workstations allow computations with at most a few 100 000 points unless restrictive hypotheses are made about the flow (like periodic boundary conditions for instance). Fast workstations are within a small factor in speed with minisupers at present. So users could do similar computations (100 000 points) if they had to, but these are more likely used for routine computations involving around 10 000 points or to test new algorithms.

[Paterson] reported a turbulence computation (direct numerical simulation of

a flat plate turbulent boundary layer) with 9.4 million points in the computational domain. This required 200 hours of CPU time on a Cray 2. To summarize without deforming too much the truth, we can say that the following computations can be made

- Millions of points on a supercomputer

- Hundred thousands of points on a minisuper

- Ten thousands of points on a workstation.

Of course, as we will see later on, the computing time depends very heavily on the algorithm and on the kind of mesh that are used. Hence, it is difficult to predict what will be going on in the future and the most conservative way is to evaluate the needs using today's algorithms.

For research challenging problems, it is estimated that to solve turbulent flows over plates and through channels for large Reynolds numbers using the Full Navier–Stokes equations, with 1988 algorithms (if this has any meaning) in 200 hours of CPU time, a Tflops (1 Teraflop=10^{12} flops) machine is needed. The memory requirement is about 0.1 Twords (1 Teraword=10^{12} words). Larger parts of aircrafts like airfoils or even complete aircrafts can be simulated using the same computing power and time but with a less accurate model (large eddy simulation).

The upper estimate for solving the full Navier–Stokes equations for a complete aircraft of a transport size airplane in the same amount of CPU time amounts to speeds in excess of Eflops (1 Exaflop=10^{18} flops) and memory size of 1 Pwords (1 Petaword=10^{15} words). Of course, this can seem gigantic but fortunately these levels of computing powers are excessive for engineering applications as not so many fine details are needed. It can be estimated that the power needed for engineering applications is five orders of magnitude less. However, if we want to go further in the study of turbulence, we will need at least a fraction of these huge requirements.

For aircraft analysis and design (at normal altitudes and speeds) several levels of modelling have been defined, ranging from linear inviscid flows, through Reynolds

averaged Navier–Stokes and large eddy simulation, to the full Navier–Stokes equations. For example, runs using the averaged Navier–Stokes equations use about 1 million grid points and 10 hours of CPU time on a Cray 2. The goal for the analysis and design optimization is to reduce this to about 15 minutes and to be able to use more points. This amounts to Tflops and Gwords (1 Gigaword=10^9 words) for averaged Navier–Stokes and Pflops and Twords for large eddy simulation. During the optimization process, thousands of such small 15 minutes runs must be made.

A big challenge is to take into account the structural deformation of the body and its interaction with fluid mechanics.

As we said before, another important area (which is somehow related as fluid mechanics is also involved) is the automobile research and industry. The main computations that are done are crash simulation, structural mechanics, incompressible fluid flow computations. As an example, a 1990 frontal crash simulation (80 msecs) using finite elements with 20 000 shell elements and about 100 000 time steps with an explicit method used 18 hours of Cray X–MP CPU time. To get a good description of the physics and to handle more complicated situations like lateral collisions, many more elements (3 to 5 times more) have to be used. This will lead to more time steps (because of stability restrictions) and the goal will be to do the same computation in less than 15 minutes. To do this much faster computers are needed.

There are many other areas from which examples can be used. In the research category, we can quote:

- Aerospace engineering (hypersonics, spatial structures)
- Plasma physics
- Particle physics
- Weather modelling
- Weapons simulation

In design and analysis for industrial products :

- Oil detection and recovery

- Computational Chemistry
- Biology and medicine
- Visualization

and many more examples.

An area where large scale computing is rapidly growing is the financial companies (like banks and other operators).

As we have seen, all these fields require a huge increase in computing power that can be obtained through hardware and software improvement and also through algorithmic advances.

To satisfy all these users (or potential users), there are a lot of issues to be discussed. Not only are the architectures of computers important but also the context in which the machine is used: through a large computing center with may be several machines but hundreds or thousands of users, on a single user basis for big research users or with longer runs on smaller machines. The computing environment is also very important, for instance, the graphic pre and post processing and the communications. Therefore the problem of which machine to choose cannot be isolated from the other issues and there are many different situations.

2. The evolution of parallel architectures.

The progress in computers is due to two main reasons: the first one is the increase in the commutation speed of the logical gates and in integration resulting in faster and smaller chips, the second one is the improvements in architectural design.

Regarding the speed of integrated circuits, one can look at the clock cycle of past and present machines (and also the number of cycles needed to perform one floating point operation). In Table 1, we list some scientific computers with the (approximate) year of introduction, the clock cycle and the amount of memory available. From this table, one can see that the decrease of the clock cycle is considerably slowing down.

From 1960 to 1980, the cycle time has decreased from 2000 ns to about 10 ns, that is a ratio of 200. During the last ten years, we went only from 10 ns to about 3 ns. The next generation of Cray vector computers have a target of 2 to 1 ns, therefore there won't be too much progress arising from faster clock cycles in the next years on traditional supercomputers.

Table 1

Computer	Year	Cycle time (ns)	Memory
IBM 7090	60's	2000	32K (36 bits)
CDC 6600	1964	100	128K (60 bits)
CDC 7600	1969	27.5	128K + 512K
CRAY 1	1976	12.5	2M (64 bits)
CDC Cyber 205	1979	20	4M
Fujitsu VP 200	1983	7.5 (15 scal)	64M
CRAY 2	1984	4.1	256M
CRAY X–MP/4	1985	8.5	16M
NEC SX–2	1984	6	128M
CRAY Y–MP/8	1988	6	128M
Fujitsu VP2600	1990	3.2	256M
NEC SX–3	1990	2.9	256M
CRAY C90	1992	4	512M

The manufacturers have started to improve the computing speed by using more parallelism. There has always been a lot of parallelism in the fastest computers. For instance, there were already multiple functional units in the CDC 6600, able to operate in parallel. However, this parallelism was at the hardware level, hidden from the user. In the mid 1980s, to improve the performance, Cray choose to switch from mono processor vector computers to multiprocessor shared memory machines whereas the Japanese manufacturers choose to use multiple functional units with only one processor. There are some advantages and drawbacks in both approaches.

The obvious advantage of multiple pipelines is that the user has nothing to do. The hardware is in charge of exploiting this low level parallelism. Unfortunately, to

have a large enough granularity, one needs long vectors and enough floating point operations to be performed for each vector load. So, usually, these machines are not very good at handling short vectors.

On the other side, at least at the beginning of the multiple processor machines, the user was in charge of describing the parallelism in the multiprocessor approach and there is some overhead associated with the use of such systems. As we will see, the situation is improving in that respect.

It seems that now, the two approaches are getting closer to each other, as Cray has introduced more functional units and larger vector registers in the C90 and, for instance, we see Japanese manufacturers like NEC and Hitachi introducing multiprocessor computers, each processor being a vector machine with multiple pipes.

Can the user get the power he needs from these kind of architectures that is a multiprocessor shared memory vector machine with a few processors? The answer is likely to be: no. The Cray C90 delivers a peak performance of 16 Gflops, the peak of the NEC SX–3 is rated at 22 Gflops although it is difficult to get close to that.

The next generation of Cray vector multiprocessor is aimed at delivering about 100 Gflops. Therefore, we are far from the goal we would like to reach for both research and design, that is having a Tflops sustained by the end of the century.

We can notice also that, despite their low cycle time, these machines are still not very good on scalar computations. Unfortunately, even if many algorithms have been vectorized or parallelized, there is still a lot of scalar instructions in most industrial codes.

Another thing to consider is the development cost of these machines and the price to be paid by the user (the Gflops per Dollar). Some ten years ago, the price of a top of the line supercomputer was around 20 Millions of Dollars. Now, the prices are more than 30 Millions of Dollars and it is likely that the next generations will be even more expensive.

Eight years before the end of the century, it can be said that, probably, the

only way to have a Tflops in year 2000 or before at a reasonable price is to design massively parallel computers. As people like simple concepts (even if they don't fully understand what is behind), it is common and fashionable now to speak about MPP (Massively Parallel Processing) and Hypercomputing.

It is difficult to define precisely what "massively" means (it is just like defining what a supercomputer is). From our purpose, we will speak of MPP when the machine has at least a hundred of processors. Of course, we also have to define what a processor means. A processor, in our terminology, will be a computing engine with some kind of control. This is different, for instance, from the 1 bit processing elements of the Connection Machine CM2.

There has been some MPP machines on the market for quite a while now. Some well known machines are SIMD like the Connection machine CM–2 and the MasPar (also known as the DEC MPP). Some others are MIMD like the Intel Hypercube iPSC/860 and the Ncube 2. It is not our purpose here to review the performances of all these machines but just to look at the trends of the market and the future architectures.

The first machines on the market a few years ago were more toys than real practical computers. Think, for instance, of the iPSC/1 and the CM–1. However, the situation has changed. Today, one can solve problems on the CM–2 at about the same speed as on a vector supercomputer. For instance, in our laboratory, we coded a neutron transport Monte–Carlo algorithm on the CM–2 ([Robin]). This code runs at about the same speed on a 16K CM–2 as on a one processor Cray Y–MP although this algorithm is not well suited to the CM architecture. Of course, there are some applications for which much better speed can be reach on the CM–2. There are people having codes running at a few Gflops, that is to say the speed one can reach on an 8 processors Y–MP.

Today most manufacturers have an undergoing project of massively parallel computer. Therefore, there is going to be a lot of competition. Let us review some of

them:

• Thinking Machines Corp (TMC) is now manufacturing the CM–5. The history of this company has begun in the early 80s from research done at MIT to build computers suited for Artificial Intelligence applications. The CM–1 was introduced in 1986 and the CM–2 in 1987. The CM–2 was an SIMD machine with up to 64k 1–bit processing elements (pe). Soon, TMC recognized that this machine can be used for large scale scientific numerical computing and they added floating point chips (1 Weitek chip for each 32 pes). The memory was 256 Kbits per pe. The programming model of the CM–2 is called Data Parallelism. This means that the processors are doing either computation or data communications. This is expressed with a subset of Fortran 90 called CM–Fortran. Compilation and link edit of the codes as well as the scalar parts of the codes are handled on a front end machine which is an off the shelf workstation. The performance of the CM–2 was from almost 0 (when most of the code is run on the front end) to a few Gflops for some special applications.

The CM–5 is a completely different machine. First of all, this is an MIMD machine with distributed memory, each processor having a control unit. There are two kinds of processors: control processors and computing nodes. These nodes are linked by two kinds of networks: the control network and the data network.

The control processor has a SPARC microprocessor, 32 or 64 MB of memory and an interface to the networks (NI). A computing node also has a SPARC microprocessor and a NI, but this is complemented by 4 vector units each with 8 MB of memory. At full speed on triadic operations, each vector unit is able to deliver 128 Mflops. A vector unit has 64 registers of 64 bits, a 16 bits mask register and a 4 bits vector length register. The instructions for the vector units are issued by the SPARC microprocessor.

The control network is used for tigh communications: synchronisations, broadcast of a scalar, etc.... The data network is used for data communication between memories. The topology is a "fat tree" (this is a tree having more links when close to

the root). TMC claims a bandwith of data transfer between NI of 20 Mb/s in a group of 4, 10 Mb/s in a group of 16 and 5 Mb/s otherwise. Two nodes are in a group of 2^k if their networks addresses differ by the last k bits.

The machine can be divided into partitions. Each partition is controlled by a control processor called the partition manager providing the Unix functions. Each partition may be allocated to different kinds of users or different kinds of tasks.

Notice that the maximum speed of a 1024 nodes machine is about 130 Gflops. Therefore to reach a Tflops with this technology, a 10 000 nodes machine is needed. The (1992) cost of a large machine is about $ 40 000 per processor.

The early samples of the machine were released in 1991 without the vector processors. After having been announced for quite a while, the vector processors are now running on TMC prototypes and are said to be deliver to customers by Fall 1992.

• Intel has been working on its Touchstone project for several years. There are several steps in this project. All the machines constructed under this project are MIMD computers based on Intel microprocessors and with distributed memory.

One product already on the market is the iPSC/860 (arising from the Gamma phase of the project) having at most 128 processors. This machine has evolved from the first iPSCs by replacing the Intel i386 by the Intel i860 chips, increasing the memory capacity and improving the router chips transmitting data between memories.

The Delta machine, which is only a prototype and not a commercial product has been delivered in 1991 to a consortium of laboratories and universities (CSC) based in Caltech (USA). This is an MIMD machine, the network topology being a 2D mesh (with periodic connections) connecting 528 i860 processors (16×33) plus some dedicated service processors. The core of the system is the mesh routing chip (MRC) for transmission of messages, to which a computational nodes is linked. Data is sent by fixed length messages. The bandwith between two nodes is 65 MB/s. The processor is the i860 associated with 16 MB of memory. The maximum performance is a few tens of Gflops.

The follow on is the Sigma machine derived from the Delta. This leads to a commercial product to be released soon, the Paragon. The microprocessor is the new i860 XP delivering a maximum speed of 75 Mflops (at 50 Mhz). The memory is 128 MB per node. There is up to 2048 nodes, each node being two i860 XP. One is in charge of the floating point computations and the other one of handling the messages (send and receive) that will be routed by the MRC. The MRC has been improved and the communication bandwith is 200 MB/s. The maximum speed is 150 Gflops. The main programming model, as on the iPSC/860, is by message passing. However, Intel with some other manufacturers is working on extending Fortran to distributed memory architectures. An undergoing project in this area is called HPF (High Performance Fortran).

• Kendall Square Research (KSR) has been working on an MIMD machine ranging from a hundred to a thousand of processors (KSR1). This machine uses a proprietary processor delivering 40 Mflops. The processors are connected with a ring topology with a maximum of 32 processors. Several rings can be tied together to form larger machines. The important point is that this computer is presented as an "all cache" machine. There is a global address space, the memory being divided into pages which are migrated by the hardware to satisfy the data requests from the processors. The hardware is in charge of maintaining the cache coherency as there can be several copies of some data.

• Cray Research is also working on a massively parallel machine, the goal being to reach a Tflops by 1995–97. The project has three distinct phases. The first one (sometimes denoted MPP0) has a goal of a maximum of 150 Gflops in 1993. A peak speed of a Tflops is expected for 1995 and a sustained Tflops for 1997 (which means having a machine with a much larger peak speed). So far, some details are only known for the MPP0.

This is an MIMD machine with a physically distributed memory. Each node uses a DEC Alpha superscalar microprocessor at 150 Mhz. This chip is aimed at delivering

a peak floating point computational speed of 150 Mflops. However, it seems that the attainable speed will be around 100 Mflops for codes with a good data locality. The MPP0 will probably have from 128 to 1024 processors. Each processor is associated with 2 Mwords (64 bits) of memory. The topology of the communication network is a 3D grid (with periodic connections). The data transmission speed is sought to be around 300 MB/s.

The originality of this machine is to have hardware able to support a sort of shared memory programming model. This model (Fortran MPP) is based on Fortran 77 and a few directives. The data can be local (private) or global (shared). When an operation is done on a node some shared data can be located in the memories of some other processors. Directives allow the user to describe the data and work distribution. Cray will also provide a message passing environment based on the syntax of PVM.

So far, this machine is linked to a Y–MP in charge of the compilations and the I/O.

There are many more machines either in the design phases or already on the market. However, our goal is not to describe all the projects but to get the trends of the market. As we have noticed, all of these projects are MIMD machines although some programming models are SIMD–like. Moreover, all these machines have distributed memory, even though in some of them the address space can be viewed as a shared global space by the programmer. This will be done by hardware on some machines and at the software level on some other ones. Almost all of them use off the shelf RISC superscalar microprocessors that have appeared recently and allowed (with appropriate compilers) to efficiently process scalar and vector codes.
Therefore, the global trends towards an MPP machine (with a target of more than a 100 Gflops) are:

◦ a few thousand processors,

◦ a physically distributed but (eventually) logically shared very large memory (a few Gwords),

◦ the use of superscalar microprocessors.

This machine can be either a stand–alone one or to be linked to a more conventional super workstation or mid range supercomputer.

As we said before, a machine is not isolated in a computing center. Hence, many other issues must be addressed, like communications as there must be very fast links between such an hypercomputing engine and mass storage or graphic workstations. If we would like to do real time animation on simple images with a resolution of 1024^2 and 24 bits/pixel, we need a sustained throughput of more than 1 Gb/s. The problem of storage capacities is also becoming crucial as users are able to produce more and more data. Usually, they like to store this data for a while. This implies the need for massive automated storage devices and a short retrieval time for any stored data.

3. The evolution of software.

The software we have to care about for the use of supercomputers or massively parallel computers is mainly the operating system and the compilers, the language of interest for scientific computing being Fortran (Fortran 77 right now and may be Fortran 90 in the near future).

In the scientific computing area, Unix is now a de facto standard for the operating system. Therefore, all the manufacturers (except, until recently, some Japanese companies) have a Unix offer. To be used on massively parallel machines, Unix will have to be distributed somehow. Most implementations use a Mach or Chorus micro kernel. Moreover, the view of the operating system tends to be hidden from the user by the use of workstations using graphical user interfaces like Motif and its competitors.

Regarding compilers and automatic parallelizers, we are, for massively parallel machines, almost in the situation we were at the beginning of the use of vector computers.

We have just to recall that when the first Cray machines were delivered to LANL, they were almost without compilers. At least, the early vectorizing compilers were far from being satisfactory and it took more than ten years to reach a point where

people can almost completely rely on automatic vectorization.

This was done through the progress of research in vectorizing compilers starting in the 70s. By 1985, the research goals were reached and people started looking at more difficult problems like parallelization.

To stress the importance of having efficient compilers, for instance, the performance of the Linpack benchmark has increased from 12 Mflops in 1983 with the CFT 1.12 compiler to 27 Mflops with the CFT77 2.1 compiler using exactly the same hardware. Although the improvement is not always as spectacular as this one, on the Cray X–MP, we noticed an increase of performance of about 30% when we switch from CFT to CFT77. A double saxpy kernel went from 180 Mflops to 220 Mflops. The improvement in speed was particularly noticeable for short vectors.

Regarding parallelization, most of the work done so far was concerned about shared memory multiprocessors. There has been a lot of progresses towards a better dependency analysis. Partitioning of loops has improved allowing loop inversions, chopping of the iteration domain. The task is more difficult when one wants to mix vectorization and parallelization as one must look at nested loops trying to push inside the longer vector loops and to parallelize the outermost loops. Some efforts have been done towards parallelizing loops with non rectangular iteration domains. Also, the situation has been clarified concerning dependency tests.

Many improvements result from the fundamental work of people like D. Kuck and K. Kennedy.

An area that has been developed during the last years is interactive systems that allows the user to see the results of automatic parallelization and to help the compiling system tuning the code.

Recently, there has been an emphasis on generating efficient code for superscalar architectures which is very important to have good performances on massively parallel machines. Experiments have shown that the main problem in most codes that prevents more parallelism to be extracted is control dependencies. Therefore, the

gains that can be made using these architectures heavily depend on the availability of compilers able to efficiently schedule the instructions. A large part of the problem has been switched from hardware (which is simpler) to software (which becomes more sophisticated). This implies to reorder instructions to optimize the use of several parts of the processor working simultaneously. This kind of superscalar processors are particularly important as they will be used as individual components of either traditional vector computers (improving the "scalar" part of the computation) or in massively parallel computers.

An important point that has been also studied and that can have application on distributed memory systems is data allocation for improving cache efficiencies.

We are on the verge of seeing in industrial products inter–procedural dependency analysis that will allow parallelization of loops with subroutine calls.

Unfortunately, the situation is far from being satisfactory for distributed memory architectures. Large efforts have to be done in that direction as we have seen that most future machines have this type of architecture.

There are some active people working in parallelization in France and we would like to mention P. Feautrier in Paris VI university and the team at Ecoles des Mines de Paris whose parallelizer PIPS is an interesting tool.

When the user wants to code on one of todays parallel vector computers like the Cray Y–MP, he can rely on automatic parallelization. For many cases and with a small help from the user, the parallelizer will do an honest job. The important thing to notice is that there is no problem of data assignment on a shared memory machine.

So far, the problem is completely different on a distributed memory architecture where no satisfactory automatic parallelization exist. Therefore it seems that the future is dark. We will have hyper machines without being able to use them up to their peak possibilities without tremendous programming efforts.

However, there is some hope to get something done. One possibility is to use these MIMD hypercomputers with an SIMD–like programming model. Consider, for

instance, the Connection Machine CM–2 and CM–5. They are used with a model of programming which is called data parallelism. Programs are coded using a subset of Fortran 90 and data allocation is done automatically. This type of coding is very easy and the machine is not difficult to program (having good performances might be another story).

Of course, not all the algorithms naturally lead to data parallelism. Therefore, we will probably have to use a kind of SPMD programming model. But the goal is to have something easy to use where the user has just to code in a "natural" programming language like the array features of Fortran 90.

The Cray MPP Fortran also offers a possibility of using an MIMD machine like a shared memory multi processor.

Automatic data allocation will not be easy on all machines, but either hardware or software virtual shared memory will help. The user wants to see a global address space with communications automatically generated either by the compiler or the hardware. We will have such systems in the next years. This will be a mandatory condition for these machines to be used in industrial laboratories.

4. The evolution of algorithms.

There has been a lot of progresses made in numerical algorithms during the last 20 years but not that many directed towards parallelism.

Think, for instance, of the generalization of the finite element method. Very sophisticated algorithms have been devised for different types of problems. In the area we were looking at in the first section, that is computational fluid dynamics, new finite element methods have been introduced to handle Euler and Navier–Stokes equations (cf. [Pironneau]). Particularly, very complex and sophisticated (but efficient) schemes were invented for the gas dynamics system of equations.

As we said before many algorithms boil down finally to solving a large sparse linear system of equations (for elliptic and parabolic equations or systems). During the 70s and the 80s, there was the revival of the Conjugate Gradient (or Conjugate

Gradient like) method. It was shown that to be efficient, this method has to be used with a preconditioner which transforms the system into one that possesses better numerical properties. One can quote also the successes of the multigrid algorithm.

Unfortunately, most of these algorithms were not well tailored to vector and parallel architectures.

At the beginning of vector computers, the results with the finite element method were disappointing because these algorithms generate a lot of indirect addressing and the first machines (like the Cray 1 and the CDC Cyber 205) performed very badly on these operations, even inhibiting vectorization. This need was so obvious that this problem was corrected on the next machines (like the Cray X–MP/4) with a hardware gather–scatter. Today, nobody can imagine marketing successfully a machine that is performing poorly on indirect addressing.

The Conjugate Gradient method is a good one for vector architectures as most operations are directly expressed in terms of vectors (like sums and dot products). However, two steps of the algorithms cause problems. The first one is the matrix–vector product and the second one is the preconditioner solve. Both depend on the problem we are solving, the chosen storage scheme and the design of the preconditioner.

The matrix–vector problem is tougher for finite element on unstructured grids than for finite differences methods on regular grids. However, during the last years, clever storage scheme have been devised (cf. [Erhel]) such that this operation can be done efficiently on vector processors.

The design of the preconditioner depends heavily on the problem to be solved. Efficient solutions have been devised for vector machines for finite difference matrices (cf. [Van Der Vorst], [Meurant]). The situation is less satisfactory for finite element matrices. However, we can consider to have controlled the problems on vector computers.

The same is true for algorithms for CFD problems that can be vectorized effi-

ciently.

But, unfortunately, problems still remain for parallel architectures, especially for distributed memory and massively parallel systems.

If we consider the algorithms for gas dynamics problems, they are not well suited to data parallelism as the different cells of the mesh might require different handlings. This has lead some people to solve this type of problems on massively parallel machines by brute force, using less sophisticated but more parallel algorithms with a much finer mesh size. From the point of view of the computer scientist, this has two advantages: the algorithm is easy to code and, as the problem size has to be larger, the performance in terms of speed is much better. Unfortunately, what we are interested in is to solve problems fast and not necessarily to reach the peak performance of the given computer. It must be noticed that it is always simpler to reach the (almost) peak performance with a stupid algorithm rather than with a sophisticated one (manufacturers know that quite well !).

The picture is not too good either regarding preconditioners. Most of the good preconditioners are not parallel at all. This is understandable as we are trying to solve elliptic or parabolic problems with a strong coupling between all the unknowns. So far, on massively parallel machines only straightforwardly parallel preconditioners like the diagonal preconditioners have been used.

So, are we condemned to use dumb and obvious algorithms on the future generation of massively parallel computers?

This will drive to despair most numerical analysts and drive crazy the rest of them. Fortunately, in this area also, there are some hopes for improvement.

First of all, the situation is much better than it was a few years ago for moderately parallel machines. During the last years there has been a great deal of development, both theoretical and practical, on domain decomposition methods.

Roughly speaking, these algorithms divide the domain of interest (or sometimes directly the equations) in subdomains (either with overlapping or not). Then, sub-

problems are defined on each subdomain that can be solved independently. Finally, the results of the subproblems are pasted together (usually iteratively) to give the solution of the global problem. Most of the time, the algorithm takes the following form:

- parallel subproblems solves,
- a non parallel synthesis phase,
- parallel subproblems solves.

The modern way of using domain decomposition is to use this methodology to derive parallel preconditioners for the Conjugate Gradient method. As an example, we consider a preconditioner called INVDDH (using 8 subdomains), derived in [Meurant] and we compare it to the trivial diagonal preconditioner on the Cray Y–MP using 8 processors in parallel. The results are given in Table 2 in a square domain with a 400×400 mesh solving the Poisson equation.

Table 2

INVDDH

INVDDH	cpu time (s)	wall clock (s)	ratio	Mflops
iter	12.10	1.52	7.96	1472
total	13.69	1.74	7.86	1284

DIAG

DIAG	cpu time (s)	wall clock (s)	ratio	Mflops
iter	17.83	2.25	7.92	1605
total	17.92	2.28	7.86	1597

From the results in Table 2 it may seem that the improvement using the domain decomposition method is not so dramatic. However, the important point to notice is that the computational speed we reach is almost the same as for the straightforwardly parallel diagonal preconditioner. The decrease of the computing time is not large because we solved a too easy problem. If we look at the number of iterations, INVDDH

gives 163 iterations and DIAG gives 1029 iterations, but the domain decomposition iterations are more costly. However, there are problems for which the difference is quite large and for which it pays to use a sophisticated algorithm. A discontinuous coefficient problem like:

$$-\frac{\partial}{\partial x}(\alpha(x,y)\frac{\partial u}{\partial x}) - \frac{\partial}{\partial y}(\beta(x,y)\frac{\partial u}{\partial y}) + \gamma(x,y)u = f, \quad \text{in } \Omega \subset R^2,$$

$$u|_{\partial\Omega} = 0$$

with the diffusion coefficients chosen as

$$\alpha \equiv 1$$

$$\beta = \begin{cases} 1000 & \text{if } 0.24 \leq y \leq 0.76 \\ 1 & \text{otherwise} \end{cases}$$

on a 200×200 mesh, gives 244 iterations for the domain decomposition method and 6050 iterations for the diagonal preconditioner.

Although methods like this one are not well suited for massively parallel computers, one can construct, using the same methodology, other algorithms using a large number of small subdomains more appropriate to a computer with many (hundreds or thousands) of processors.

Another interesting feature of the domain decomposition methods is that they allow coupling different physicals models in different regions. For instance, in CFD, Euler and Navier–Stokes equations have been coupled together by researchers in AMD's team, (cf. [Glowinski–Periaux]). For hypersonics studies, some research has been started to couple Boltzmann and Navier–Stokes equations.

More information can be found about domain decomposition in the Proceedings of the annual Domain Decomposition Conference published by SIAM.

5. Conclusions.

In this short paper, we have seen that there are still a lot of problems that do need computational power that will not be available before the next century. Some of

these problems arise in leading edge research. Others occur when solving industrial design problems trying to include computations in an optimization cycle.

It seems today that the trend in architectures is to have massively parallel machines with thousand of processors, each microprocessor having efficient superscalar and/or vector capabilities. These machines will use a physically distributed memory but it is hoped that the user can see it as a shared global memory.

To use these machines we will need software tools that are in the research stage now. This include distributed operating systems and compiling systems. The automatic parallelizing compilers must be user friendly to help the user cooperating with the system on portions of code that cannot be automatically parallelized. More research has to be done also on automatic data allocation.

Concerning the algorithms, we must continue research in the way of domain decomposition to get algorithms more suited to thousands of processing nodes. It is not satisfactory to have to use dumb algorithms that were rejected because of their poor performances some years ago.

Right now, we are almost in the same situation for MPP as we were for vector computing at the beginning of the 80's. This means that more research has to be pursued at the software level (compilers and algorithms) in order to keep pace with the progresses in hardware and architectures. But, there are some good hopes that the sustained Tflops target can be reach before the end of the century.

References

[J. Erhel], *Sparse matrix multiplication on vector computers*, Int. J. of High Speed Computing, 1990

[R. Glowinski, J. Periaux and G. Terrasson], *On the coupling of viscous and inviscid models for compressible fluid flows via domain decomposition* In Domain decomposition methods for partial differential equations III, T.F. Chan, R. Glowinski, J. Périaux and O. Widlund, eds Siam, 1990 pp 64–97

[G.ÊKaradimas], *Application of computational systems to aircraft engine components development*, SNECMA Report, 1990

[G. Meurant], *Domain Decomposition Preconditioners for the Conjugate Gradient Method*, Calcolo v 25 n 1–2 (1988) pp 103–119

[G. Meurant], *Domain Decomposition methods for solving large sparse linear systems*, in Solving linear systems, the state of the art, E. Spedicato Ed, Nato ASI series, Springer 1991

[V. Paterson], *Computational challenges in aerospace*, Future generation computer systems 5 (1989) pp 243–258

[F. Robin], *Transport de neutrons sur Connection Machine*, Note CEA, 1991

[H.A. Van der Vorst], *A vectorizable variant of some ICCG methods.* SIAM J. Sci. Stat. Comput. v 3 (1982) pp 86–92.