RAL 94-019

Porting Industrial Codes and Developing Sparse Linear Solvers on Parallel Computers [1]

Michel J. Daydé [2] and Iain S. Duff [3]

ABSTRACT
We address the main issues when porting existing codes from serial to parallel computers and when developing portable parallel software on MIMD multiprocessors (shared memory, virtual shared memory, and distributed memory multiprocessors, and networks of computers). We discuss the use of numerical libraries as a way of developing portable and efficient parallel code.

We illustrate this by using examples from our experience in porting industrial codes and in designing parallel numerical libraries. We report in some detail on the parallelization of scientific applications coming from Centre National d'Etudes Spatiales and from Aérospatiale, and we illustrate how it is possible to develop portable and efficient numerical software by considering the parallel solution of sparse linear systems of equations.

Keywords: industrial codes, sparse matrices, multifrontal, BLAS, PVM, P4, MIMD multiprocessors, networks.

AMS(MOS) subject classifications: 65F05, 65F50, 68N99, 68U99. 90C30.

Computing Reviews classification: G.4, G.1.3, D.2.7, D.2.10

Computing Reviews General Terms: Algorithms

Text from invited lectures given at First International Meeting on Vector and Parallel Processing, Porto, Portugal. 29 September - 1 October, 1993.

[1] Part of this work was supported by Aérospatiale, Division Avions, and Centre National d'Etudes Spatiales under contracts 11C05770 and 873/CNES/90/0841/00.

[2] ENSEEIHT-IRIT, 2 rue Camichel, 31071 Toulouse CEDEX, France and CERFACS, 42 av. G. Coriolis, 31057 Toulouse Cedex, France.

[3] Current reports available by anonymous ftp from numerical.cc.rl.ac.uk (internet 130.246.8.23) in the directory "pub/reports".

Central Computing Department
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

February 7, 1994.

Contents

1 **Introduction** **1**

2 **A simplified view of high-performance computing today** **2**
2.1 High-performance CPUs 3
2.2 Use of building blocks 3

3 **Methodology for porting codes and portability** **4**
3.1 Methodology for porting industrial applications to parallel computers . . . 4
3.2 Portability problems 5
3.3 Fortran developments 5

4 **Description of the codes** **6**
4.1 Out-of-core solution of linear least-squares problems 7
4.2 Panel method 8
4.3 Flutter calculation 9
4.4 2D explicit Navier-Stokes solvers 10

5 **Design of mathematical software** **11**
5.1 Solution of full systems 12
5.2 Solution of sparse systems 13

6 **Conclusion** **15**

1 Introduction

One of the common problems for application scientists is the porting of codes from serial to parallel computers. Since most of the existing software has been developed on serial computers, exploiting application parallelism without totally rewriting or redesigning existing codes and algorithms is crucial if parallel computers are to be accepted and used by industry. It is also vital that new mathematical software is designed for efficient execution on parallel computers since the availability of portable and efficient parallel numerical libraries, which can be used as building blocks, is crucial for both simplifying application software development and improving reliability.

One of the primary concerns of the European Centre, CERFACS, that was established in Toulouse in 1987, is the interaction of scientific research with industry. From the start, the Parallel Algorithm Project at CERFACS has had a strong programme in the development of algorithms and the design of software for parallel computers. This has been coupled with heavy involvement in what is now called technology transfer effected by encouraging the use of these techniques by the industrial partners of CERFACS. This has been done both through training courses and by advice and consultation on actual industrial codes of the partners. In this paper, we show that similar building blocks and methodology can be used both in the design of new mathematical software and in the porting of industrial codes. We use as an example the development of codes for parallel computers for the solution of both full and sparse linear equations and the porting of some industrial codes from the CERFACS partners Aérospatiale and CNES.

In this work, we have been involved with different kinds of tasks of increasing complexity :

- Porting existing codes to parallel computers : only local changes are made in the code (use of compiler directives, modifications at loop level, ...), there is no change in the global solution technique, except possibly the substitution of some computational kernels (linear system solution for example). This is the fastest way to obtain some benefits of parallelization but it obviously does not lead to the most efficient parallelization.
- Developing a parallel code : in this case one has to use a solution technique suitable for parallelization that may be very different from that used on a serial computer (use of domain decomposition for PDE problems for example).
- Developing portable and efficient parallel numerical software : obviously portability and efficiency issues are fundamental and complicate the design of the code. For example, the data distribution on a distributed memory machine should not prevent the use of convenient data allocation schemes for the user.

Some of the main factors to be considered before porting industrial applications to parallel computers are the following :

- the suitability of the code for implementation on a parallel computer (the solution technique may be inherently serial),
- the scalability of the code (this dictates its suitability for implementation on massively parallel computers),
- the suitability of the application for functional or data parallelism (important issue for implementation on SIMD computers),

- the amount of effort to be spent in parallelizing the code.

Moreover, there is a large variation in programming environments depending on whether the target machine is a shared memory multiprocessor (where automatic parallelization and loop-level parallelism are available), a distributed memory multiprocessor where message passing or less often virtual shared memory are available, or an SIMD computer.

We work in a close relationship with industrial partners and have been involved in several studies on porting some of their applications to parallel computers. We discuss in particular two studies performed for Centre National d'Etudes Spatiales (CNES), Toulouse, France ([7]), and Aérospatiale, Toulouse, France ([10]). The aim of these contracts was to evaluate the performance of a selection of application codes on a range of parallel computers: ALLIANT FX/80, ALLIANT FX/2800, BBN TC2000, CRAY-2, CONVEX C220, and networks of workstations.

2 A simplified view of high-performance computing today

Todays high-performance computers can be divided into two classes :

- General scientific computers : these are the conventional high performance computers capable of handling efficiently a wide range of applications. Automatic parallelization is generally available, and they support multiprogramming and time-sharing (RISC workstations, shared memory computers with a moderate number of processors including minisupercomputers and vector supercomputers).

- Massively parallel computers : they are characterized by a more limited applicability compared to the current vector supercomputers for example. This typically arises from novel features in the architecture or from limitations in the software (application software, programming tools, ...). They are very competitive in peak performance with the conventional supercomputers and may achieve an interesting price to performance ratio on suitable applications. This class of computers includes both physically distributed memory architectures (both virtual shared memory computers and explicit message passing architectures (multicomputers)) and SIMD computers.

Some of the main architectural factors influencing the performance of a computer are the following ([30]) :

- the memory bandwidth and the latency for accessing data in the memory,

- the relative cost of computing and communication (for remote access on a virtual shared memory computer or for exchanging a message on a multicomputer),

- the synchronization costs,

- the vector start-up time for a vector processor based architecture.

Note that these are only some of the main factors. For example, the I/O requirements of an application are often ignored, while they may be one of the main bottlenecks in performance. Any mismatch in the design of an architecture will penalize the application performance irrespective of the efforts of the algorithm designer.

Some other factors that may affect the performance are application dependent :

- The type of parallelism that can be exploited within the application. It can be either functional or data parallelism.

- Task granularity.
- Scalability which imposes limitations on the number of processors that can be efficiently exploited.
- Load balancing.
- Communication needs which may have a great impact on the performance. Locality of references to data is crucial for efficient implementation on distributed memory architectures.
- Regularity of treatment is important for the vector performance and for efficient implementation on SIMD computers.

Clearly, the solution technique has a great impact on these factors (especially on the global and local communication requirements, particularly crucial on distributed memory architectures).

2.1 High-performance CPUs

Vector processors are commonly used in supercomputers. Recently very fast RISC processors which also can process vectors efficiently have come on to the market ([15]). They are often more efficient than vector processors on scalar applications. We report in Table 1 the performance of some current RISC processors on the double precision 100-by-100 and 1000-by-1000 LINPACK benchmarks ([17]). We also record their peak performance.

Computer	LINPACK	LINPACK	Peak
	100*100	1000*1000	performance
DEC 3000 500X AXP	39.8	133.2	200
IBM RS/6000-580	38.1	105.0	125
HP 9000/755	41.0	107.0	200
SGI Crimson	16.0	32.0	50
SUN SPARC 10/41	7.3	22.4	40

Table 1: Performance in MFlops of RISC workstations on the double precision LINPACK benchmarks

2.2 Use of building blocks

As we have previously discussed, it is very important to use standard building blocks when designing or porting codes. They are extremely useful for simplifying the design of codes while guaranteeing portability and efficiency. Two of the most important such building blocks are Fast Fourier Transforms (FFTs) and Basic Linear Algebra Subprograms (BLAS). We will consider the former when discussing the porting of codes in Section 4. We now discuss the latter in this subsection.

Different levels of BLAS are available. The level terminology arises from the fact that if the vectors and matrices involved are of order N, the Level 1 BLAS provides vector computations of order $O(N)$ ([27], [28]), the Level 2 BLAS provides matrix-vector computations of order $O(N^2)$ ([14]), and the Level 3 BLAS provides matrix-matrix computations with $O(N^3)$ operations ([12], [13]).

The architecture of high-performance computers usually involves a memory hierarchy (main memory, local memory, vector and scalar registers, ...). All the arithmetic computations are performed at the top level of this hierarchy. Therefore the key to efficiency is to keep active data as close as possible to the top of hierarchy. The use of higher Level BLAS provides this capability. The increased granularity of higher Level BLAS also allows more efficient parallelization depending on the synchronization overheads ([9]). We report in Table 2 the performance of some manufacturer-supplied routines from the BLAS using 64-bit arithmetic (DAXPY, DGEMV and DGEMM on ALLIANT and IBM, and SAXPY, SGEMV, SGEMM on the CRAY-2). A tuned manufacturer-supplied version of the BLAS is today available on most high-performance computers and, if not, a standard Fortran implementation is available on the **netlib** electronic server ([11]).

	Level 1 BLAS	Level 2 BLAS	Level 3 BLAS
Computer	DAXPY/SAXPY	DGEMV/SGEMV	DGEMM/SGEMM
ALLIANT FX/80 (1 proc.)	3.3	4.0	13.0
ALLIANT FX/80 (8 proc.)	17.5	31.9	89.4
CRAY-2 (1 proc.)	121.0	316.5	437.5
IBM 3090E/VF (1 proc.)	26.0	60.0	80.0

Table 2: Performance in Mflops of BLAS routines on some computers

3 Methodology for porting codes and portability

3.1 Methodology for porting industrial applications to parallel computers

In most of our studies on porting codes, we have used an incremental approach ([7], [10]). First, the code is parallelized on shared memory multiprocessors (where compilers have automatic vectorization and parallelization capabilities). Then, on the basis of the scalability and data locality properties of the code, we decide whether to port the application to distributed memory computers (whether virtual shared or message passing) or even to networks of computers. This methodology for porting codes can be described as follows :

- **Step 0** : Start with a portable code (standard Fortran or C)
- **Step 1** : Use automatic vectorization and parallelization. Profile the code in order to identify the most-time consuming parts of the code.
- **Step 2** : Clean up the code. Use loop level tuning and tuned computational kernels.
- **Step 3** : Manual parallelization of the code. This may require changing parts or all of the solution technique.
- **Step 4** : Decide whether it is sensible to port the code to massively parallel computers or to networks of workstations.

The gains from cleaning up a code and using appropriate compilation directives may be significant, especially on codes written before the availability of vector computers. Cleaning up a code may involve the following operations :

1. Suppression of some subroutine calls
2. Tuning the vectorization and the parallelization at loop-level :
   - Switch some loops (parallel outer, vectorial inner)
   - Precompute the loop invariants
   - Check and possibly change the dimensions of the arrays
3. Check and possibly modify I/O
4. Identification of building blocks (BLAS, FFT, ...)

3.2 Portability problems

Since standards both for Fortran and C exist, it should be easy to write portable code. However, in practice, problems often arise when porting codes. They may come from poor or non-standard implementations of Fortran or C by the vendor, but often they come from the design of the code (for example use of binary files).

The most frequent portability problems are the following :

- Use of non-standard Fortran or C
- I/O portability problems : use of binary files, non-standard I/O procedures (random I/O,...)
- Non IEEE arithmetic (for example on CRAY and CDC computers)
- Implicit typing of variables

The use of different arithmetics by different vendors may cause noticeable differences in the number of iterations required for convergence of iterative methods. We consider the solution of a linear system used in a 3D transonic aerodynamic code ([10]). The number of iterations required for convergence of a preconditioned conjugate gradient code and the residuals on three computers are given by :

1. CRAY XMP : 15 iterations, residual = 0.21718×10^{-5}
2. CRAY 2 : 17 iterations, residual = 0.61356×10^{-6}
3. CONVEX C220 : 15 iterations, residual = 0.88515×10^{-5}

It is interesting to note the influence of the arithmetic on the convergence rate and on the precision achieved.

3.3 Fortran developments

Some new derivatives of Fortran or extensions to Fortran are available (Fortran 90, Fortran D, High Performance Fortran, and the Parallel Computing Forum extensions) that incorporate features to aid in the exploitation of parallel computers :

- Arrays extensions for data parallelism
- Parallel constructs (such as parallel DO-loop)
- Data distribution directives for distributed memory computers (including SIMD)

Fortran 90 incorporates many new features ([23]). The main one of interest here is array extensions. For example, it is possible to write : $A = B + C$ where A, B, and C are arrays. There also exists new intrinsic functions that are applicable to arrays. There is obviously scope for exploiting data parallelism within these operations. This means that the underlying data parallelism of Fortran 90 could possibly be efficiently supported on architectures such as shared memory architectures and some SIMD computers (MasPar and Thinking Machine computers have provided several Fortran 90 constructs for some time).

Since data distribution is crucial on multiprocessors where the latency for accessing remote data is high, it can be seen that the main drawback of Fortran 90 is that it does not offer ways of specifying an efficient data distribution.

This is the reason why High Performance Fortran, which is a superset of Fortran 90, was initially proposed ([26]). It incorporates data distribution directives similar to those available within Fortran D. HPF Fortran also provides the FORALL statement that can be used to specify identical sequences of operations on elements of arrays permitting simultaneous execution.

The data programming style, already used on SIMD computers, can obviously be supported on MIMD computers. It often scales up to large numbers of processors and can be used on many applications.

Parallelizing applications using message passing is surprisingly more portable since message passing can be emulated on shared and virtual shared memory computers, and since it is the programming paradigm used on multicomputers and networks of computers. Some message passing libraries are available on a wide range of computers such as PVM ([24]), and P4 ([6]). There is currently an international collaboration to define a standard for message passing : the MPI initiative ([16]).

4 Description of the codes

The first part of the studies performed for CNES and Aérospatiale was to evaluate the cost of porting codes to the target computers. The second part concerned the performance obtained by tuning and parallelizing these codes on the target machines. The conclusion of these studies concerned recommendations for a methodology for both porting and developing codes on parallel computers, a performance analysis of the target computers, and comments and recommendations related to the numerical algorithms encountered. Note that the initial parallelization of the applications was always performed on shared memory multiprocessors.

The benchmark of application software from CNES involved the following types of applications ([7]) :

- Two signal processing codes : frequency multiplex demodulation of a signal and telecommunication
- Solution of linear least-squares problems
- Computation of antenna reception patterns
- Teledetection

The benchmark of application software from Aérospatiale involved two aerodynamic codes and one structural engineering code ([10]).

We will examine in more detail in the following subsections the solution of linear least-squares problems and the parallelization of the structural engineering and of the aerodynamic codes from Aérospatiale. In the remainder of this subsection we briefly discuss the porting of the other codes.

The first signal processing code from CNES performs the frequency multiplex demodulation of a signal. This mainly involves calculations such as trigonometric functions and filtering operations. The second one is a library that simulates the distortions of a signal during its transmission through a set of devices. It can be applied to telecommunications, television, radar,.. The main computational task is a 1D FFT. On both codes, the vectorization was improved by rewriting loops and using compiler directives, and for the telecommunication code using a vectorizable 1D FFT. We have been able to achieve significant gains compared with the initial version of the code. These gains mainly arise from improvements in the vectorization. Parallelism is only exploited at loop-level and is not particularly efficient in either case.

The computation of antenna reception patterns is more interesting. The numerical method involves the solution of integral equations using boundary finite elements and the solution of a full complex linear system of equations for computing the surface currents. Most of the improvements were due to the use of a block algorithm from the LAPACK library ([5]) in the linear solution step.

The teledetection application consists of a statistical data analysis to identify the main crop zones from satellite images. The image pixels are transformed into a signal using FFT. Then a statistical data analysis is performed on the signals. Since the treatment on each pixel is independent, the code is embarrassingly parallel. Practically, the parallelization is effected at the row level within blocks of 40 image rows that are loaded from the disk at each step. The I/O required for loading part of the image is the main limiting factor for performance. We achieved speed-ups of approximatively 20 on the computational part and 5 on the whole code on 24 processors of the BBN TC2000.

4.1 Out-of-core solution of linear least-squares problems

We consider the solution of large full linear least-squares problems. These are solved by using a Cholesky factorization of the normal equations. One use of this software is in the computation of the spherical harmonics of the earth potential from measurements of the gradients of this field. The calculation is carried out in two steps :

- Forming the normal equations
- Cholesky factorization

The normal equations system is full and is partitioned into blocks that are stored on disk. The initial version of the software was entirely written in terms of calls to Level 1 BLAS and LINPACK. Rewriting the software in terms of calls to Level 3 BLAS and to the LAPACK Library ([5]) provided a substantial improvement in performance. The parallelization is straightforward because parallel versions of the BLAS are available on some computers. Note that the parallelization in this case is restricted to single blocks. We report in Table 3 the results obtained. The speed-ups achieved are reported in parentheses.

In the initial version of code for forming the normal equations, the computation of $A^t.A$ was performed using dot products that represent 47% of the execution time. The rest of the execution time was spent in I/O. In the Level 3 BLAS version, the matrix-matrix multiplication (GEMM) used instead of the dot products, only requires 0.5% of the execution time. The Cholesky factorization step has the same behaviour. The I/O is therefore the main bottleneck after substitutions of the computational kernels.

Computer	Procs	Forming $A^t.A$		Cholesky factorization	
		Initial	Tuned	Initial	Tuned
ALLIANT FX/80	1	532	110	259	79
	8	181 (2.9)	28 (3.9)	132 (1.9)	35 (2.3)
CONVEX C220	1	159	103	103	94
CRAY-2	1	25	9	17	8

Table 3: Execution time in seconds and speed-up of the initial version (Level 1 BLAS and LINPACK) and of the tuned version (Level 3 BLAS and LAPACK) of the out-of-core linear least-squares solver

4.2 Panel method

We consider the parallelization of a 3D code ([10]) that computes flow around an aircraft in a perfect fluid. The solution technique is a panel method using a mesh surface. The surface of the aircraft is divided into panels. Then, the singularities are computed on each panel. The interactions and the contributions of the panels are assembled into a so-called influence matrix. Then the velocity is computed around the aircraft from the vector of singularities. This requires the solution of a nonlinear system of equations using Gauss-Seidel and **LU** factorization. The computation of the coefficients of the influence matrix is independent. Thus the calculation is easily parallelizable using appropriate data structures. We report in Table 4 the results we obtain on a range of shared memory computers. The speed-ups achieved by the tuned parallel version over the tuned serial version are reported in parentheses. The test problem has 1537 panels for the wings and fuselage and 225 panels for the wake.

Computer	Procs	Initial Version	Tuned Version	
		Serial	Serial	Parallel
ALLIANT FX/80	8	8391	6766	2202 (3.1)
CONVEX C220	2	1691	808	580 (1.4)
CRAY-2	4	140	148	77 (1.9)

Table 4: Execution time in seconds of initial and parallel versions of panel method

The speed-ups are somewhat disappointing. The calculation of the influence matrix is totally parallelized while the rest of the application cannot be efficiently parallelized (the parallelism is restricted to loop-level within blocks of the matrix that are loaded from disk).

Amdahl's law gives an upper bound to the achievable speed-up :

$$S_p = \frac{p}{(f+(1-f)*p)}$$

where p is the number of processors, f the fraction of code parallelized, and S_p is the theoretical speed-up for p processors. Infinite speed-up (theoretical speed-up using an infinite number of processors) is given by $1/(1-f)$.

Using Amdahl's law, we can compare in Table 5 the effective speed-ups obtained and the theoretical speed-ups predicted by Amdahl's law on the ALLIANT FX/80 on the execution of the whole code. We also report the speed-ups on INFLUV, the procedure that performs the calculation of the influence matrix which is entirely parallelized. Note that the effective speed-up exceeds the theoretical one using 4 processors on the ALLIANT FX/80 because the theoretical estimate does not include any allowance for parallelizing other parts of the code than in INFLUV.

	Number of processors		
Speed-ups	2 procs.	4 procs.	8 procs.
Effective speed-up	1.7	2.6	2.7
Theoretical speed-up	1.7	2.5	3.4
Speed-up on INFLUV	2.0	3.7	6.4

Table 5: Effective and theoretical speed-ups of panel method on ALLIANT FX/80

4.3 Flutter calculation

We consider a flutter calculation from Aérospatiale. After cleaning up the code and other optimizations at loop-level (using compiler directives and rewriting some loops), we obtained a tuned version that was much more efficient than the initial one. The code is organized into a set of nested loops :

```
Loop on the mach numbers
     Loop on the structural modes
          Loop on the velocity
               Compute frequency and damping
          End loop
     End loop
End loop
```

The two outer loops are parallelizable since the calculations for each mach number and for each mode are independent. We have only parallelized the loop on the modes since often the application is only for one mach number. Because of the scalability of this application (the degree of parallelism is equal to the number of modes) and the very good data locality (calculation for each mode requires only local data after an initialization step), this application has been successfully parallelized on a range of shared memory and virtual shared memory computers, and on networks of computers (a network of IBM RS/6000-320 connected by Ethernet and a cluster of IBM RS/6000 950, 550, and 530 connected by SOCC) using the PVM message passing package. We report in Table 6 the gains in execution time achieved from tuning and parallelizing the codes. The calculation involves 38 modes. We report in parentheses the speed-ups achieved by the tuned parallel version over the tuned serial version.

The speed-ups are very satisfactory. The effect of implementing an appropriate strategy for handling load balancing efficiently is crucial for the performance. Since the solution times for each mode are unbalanced and unpredictable, a dynamic load balancing scheme was implemented. We have simply taken the self-scheduling strategy used to manage loop-level parallelism on shared memory computers, while on the BBN and on clusters of workstations we have used a master/slave paradigm.

		Initial version	Tuned version	
Computer	Procs	Serial	Serial	Parallel
ALLIANT FX/80	8	5100	1717	326 (5.3)
BBN TC2000	19	-	2994	202 (14.8)
CONVEX C220	2	1919	216	119 (1.8)
CRAY-2	4	624	45	15 (3.0)
IBM RS/6000-320	8	-	622	93 (6.7)
Cluster RS/6000	3	-	235	89 (2.6)

Table 6: Execution time in seconds and speed-up of initial and tuned versions of the flutter calculati on application

4.4 2D explicit Navier-Stokes solvers

We consider the parallelization of a 2D explicit Navier-Stokes solver in turbulent viscous flow. The solution technique uses time averaged equations and a zero equation turbulence model. The discretization uses finite volumes. The time integration uses a 3rd order explicit Runge-Kutta method with local time stepping.

The structure of the computations within the code is fairly simple. Most of the calculations for updating the variables have the following structure :

```
          DO 10 k=1,nvar
             DO 20 j=1,Dim_y
                DO 30 i=1,Dim_x
                   update(X(i,j,k))
30              CONTINUE
20           CONTINUE
10        CONTINUE
```

The parallelization of these calculations is straightforward using loop-level parallelism. We report in Table 7 the execution times obtained on one test case (the grid involves 3718 points) on the ALLIANT FX/80, the CONVEX C220, and the CRAY-2.

Computer	Procs	Serial	Parallel
ALLIANT FX/80	8	5406	1479 (3.6)
CONVEX C220	2	1113	832 (1.3)
CRAY-2	4	236	172 (1.4)
IBM RS/6000-950	1	1020	-

Table 7: Execution time in seconds of the 2D explicit Navier-Stokes solver

The speed-up is not very good. There are several explanations :

- The granularity of the loops parallelized is too fine for efficient parallelization.
- The loops involve strides different from 1 (typically a function of the grid size). This may increase the cache miss ratio dramatically and thus decrease the execution rate. This may be less dramatic on architectures with an interleaved memory such as the

CRAY-YMP or the CONVEX but, even there, some grid dimensions may lead to strides causing memory bank contention.

The poor data locality of that code should not lead to an efficient implementation on distributed memory computers, but it could be a good candidate for implementation on SIMD computers since the parallelism exploited is basically data parallelism. It would be more efficient to exploit coarser grain parallelism by using domain decomposition. This is illustrated by considering the performance of another explicit 2D Navier-Stokes/Euler solver developed at CERFACS by Luc Giraud and Marco Manzini ([25]).

Each processor solves for one subdomain. We report in Table 8 the uniprocessor execution time and the speed-ups achieved on a range of computers. The results are obtained using the BBN Fortran extensions for virtual shared memory, and using the PVM and the P4 message passing packages on the BBN and on a network of IBM RS/6000-320 (connected using Ethernet). The results reported correspond to the first ten iterations of a simulation on a 200×100 grid using 64-bit arithmetic.

	# domains = # procs					
Computer	1	2	4	8	16	20
ALLIANT FX/80	348.1 sec	1.79	3.31	5.61	-	-
BBN Fortran	429.0 sec	1.98	3.91	7.61	13.70	16.14
P4 BBN	429.0 sec	1.99	3.83	6.67	13.98	16.76
PVM RS6000	140.5 sec	1.83	3.39	5.54	-	-
P4 RS6000	140.5 sec	1.92	3.73	7.11	-	-

Table 8: Speed-ups on a 200 × 100 grid of the first ten time marching steps

We report in Table 9 the average computing time per cell. It demonstrates that, with such an implementation, the use of distributed memory architectures and of networks of computers is competitive with the use of conventional minisupercomputers such as the CONVEX C220.

Convex vectorial	677.5 μsec
20 procs BBN Fortran	132.9 μsec
20 procs BBN P4	128.0 μsec
PVM : 8 RS6000	126.9 μsec
P4 : 8 RS6000	98.8 μsec

Table 9: CPU time per cell per timestep on a 200 × 100 grid (μsec)

This approach is also scalable : Giraud and Manzini report impressive speed-ups on the 128 nodes BBN TC2000 configuration at Lawrence Livermore National Laboratory.

5 Design of mathematical software

In this section, we consider the development of mathematical software on parallel computers, using as particular examples codes for the solution of linear equations. As in the case of porting industrial codes, we will make heavy use of building blocks such as the BLAS.

There is a kind of symbiotic relationship between numerical algorithm designers and vendors of high performance computers. On the one hand, the availability of new machines stimulates the development of new (or the refurbishment of old) algorithms. On the other hand, it is necessary to have new algorithms for the efficient use and eventual acceptance of such machines. The basic problem with designing efficient codes is to ensure that the arithmetic pipes do not suffer data starvation and that data is near the top level of the memory hierarchy when it is required, say in the cache or registers in a shared memory machine or in the local memory in a distributed machine. One main way this is achieved is through the use of higher level BLAS. In general, most parallel algorithms adopt some kind of partitioning and/or blocking strategy to enable reuse of data and localization of the data requirement.

We consider the effect of using Level 2 and Level 3 BLAS in the solution of linear equations in the following subsections, first when the coefficient matrix is full, then when it is sparse.

In summary, the basic techniques in designing mathematical software for parallel computers are:

- partitioning and/or blocking (for example, domain decomposition, partitioned **LU**)
- reuse of data (for example, use of explicit copying)
- exploitation of multiple levels of parallelism (see forthcoming example of sparse solution techniques)
- use of building blocks (for example, use of BLAS)

We note that fine tuning may be machine dependent but these general paradigms are not.

5.1 Solution of full systems

We showed the performance of some of the BLAS routines on a range of computers in Section 2.2. We would now like to realize this performance in designing codes for the solution of the linear system

$$Ax = b. \tag{1}$$

There are several ways to do this. All involve partitioning the matrix and performing the elimination operations in batches. One algorithm is to factorize the matrix by blocks of columns, using Level 2 BLAS within the block which has previously been updated by all the previously factorized matrix using the Level 3 BLAS routines TRSM and GEMM.

We do not want to go into the details of the implementation here but refer the reader to Daydé and Duff ([8]) for further information. We show in Table 10 a summary of results obtained by Daydé and Duff ([8]) where, for each machine, the same code, all in Fortran, has been used. This is clearly at least competitive with the manufacturer's Library, normally written in assembler. We also compare in that table two types of parallelization : in the Parallel BLAS version, the parallelism is only exploited within the Level 3 BLAS, while in the Parallel **LU** version, additional freedom is obtained by parallelization over the BLAS. The use of parallel BLAS kernels is more and more common for exploiting parallelism while maintaining portability. Of course, this approach captures only part of the potential parallelism. The Parallel **LU** version is more efficient at the price of a decreased portability.

		Block **LU** factorization			Manufacturer's Library	
Computer	Procs	Serial	Parallel BLAS	Parallel **LU**	Routine	Performance
ALLIANT FX/80	8	12	65	62	PDGEFA	39
CRAY-2	4	379	1027	1072	SGEFA	353
IBM 3090E-400	3	64	132	183	DGEF	72
IBM 3090J-600	6	89	294	418	DGEF	97

Table 10: Performance in MFlops of the **LU** factorization on shared memory multiprocessors using 1000-by-1000 matrices

5.2 Solution of sparse systems

When the coefficient matrix in (1) is sparse, it is not immediately obvious how we can take advantage of the BLAS which are only defined and implemented for full matrices. An obvious way to do this is to observe that if **LU** factorization is performed on a sparse matrix, the reduced partially-factorized matrix becomes increasingly dense due to fill-in and thus, at some point in the sparse factorization we could switch to using full code. This switching technique has been incorporated in MA48, a recent Harwell Subroutine Library code for general sparse unsymmetric systems ([20]), and provides gains of between factors of 3 and 20 over the earlier MA28 code when run on a set of sparse test problems ([19]). Much of this gain comes from the use of tuned versions of the BLAS. However, by a careful design of the sparse code, we can make better use of the BLAS.

We will use a multifrontal approach in designing our algorithm for the solution of sparse sets of equations. The details of such an approach are not necessary to understand this paper but further background can be obtained from the original papers by Duff and Reid ([21], [22]). We now describe the features of multifrontal methods that are needed to follow the subsequent discussion. As is common in sparse elimination, the factorization is split into a symbolic phase, which performs an analysis using only the sparsity pattern of the matrix, and a numerical factorization phase.

In a multifrontal method, the sparse factorization proceeds by a sequence of factorizations on small dense matrices, called frontal matrices. The frontal matrices and a partial ordering for the sequence are determined by a computational tree where each node represents a full matrix factorization and each edge the transfer of data from child to parent node. This tree, which can be obtained from the elimination tree ([18], [29]), is determined solely by the sparsity pattern of the matrix and an initial pivot ordering that can be obtained by standard sparsity preserving orderings such as minimum degree. When using the tree to drive the numerical factorization, eliminations at any node can proceed as soon as those at the child nodes have completed. This gives flexibility for the exploitation of parallelism, which we refer to in the following as tree parallelism. Normally, the complete frontal matrix cannot be factorized but only a few steps of Gaussian elimination are possible, after which the remaining reduced matrix (the Schur complement) needs to be summed (assembled) with other data at the parent node before further factorizations can take place. Thus the frontal matrices can be written

$$\begin{pmatrix} F_{11} & F_{12} \\ F_{12}^T & F_{22} \end{pmatrix}$$

where the matrix F_{11} is factorized and the Schur complement $F_{22} - F_{12}^T F_{11}^{-1} F_{12}$ is assembled with other Schur complements and original data at the parent node. Note that the elimination is done using full linear algebra and direct addressing so that we can use the BLAS. All the indirect addressing is confined to the assembly.

We have developed a parallel multifrontal code for the solution of symmetrically structured unsymmetric equations. The results in this section are from runs with this code, called MUPS, and experimental versions of this code ([1], [2], [4]).

If we only take advantage of tree parallelism, the speed-up is very disappointing. The actual speed-up depends on the problem but is typically only 2 to 3 irrespective of the number of processors. This poor performance is caused by the fact that there is no tree parallelism at the root of the tree and very little close to it. When this is combined with the fact that much of the computation is done in nodes near the root (Amestoy and Duff ([1], [3]) show that typically 75% of the work is performed in the top three levels of the computational tree), the reason for the poor performance is quite evident. To avoid this bottleneck, it is necessary to obtain further parallelism within the large nodes near the root of the tree (so-called node parallelism). This is done simply by using parallel versions of the BLAS in the factorizations within the nodes. When we combine both tree and node parallelism the situation becomes much more encouraging and we show typical speed-ups for a range of computers in Table 11. A medium size sparse matrix, *BCSSTK15* from the Harwell-Boeing set ([19]), is used to illustrate our discussion. This is a structural analysis matrix of order 3948 with 117816 nonzeros. A minimum degree ordering is used in the analysis and the number of floating-point operations for the factorization is 443 million.

		multifrontal factorization			
		(1)		(2)	
Computer	nprocs	Mflops	(speedup)	Mflops	(speedup)
Alliant FX/80	8	15	(1.9)	34	(4.3)
IBM 3090E/3VF	3	83	(1.9)	105	(2.4)
IBM 3090J/6VF	6	126	(2.1)	227	(3.8)
CRAY-2	4	316	(1.8)	404	(2.3)
CRAY Y-MP	6	529	(2.3)	1119	(4.8)

Table 11: Performance summary of the multifrontal factorization on matrix BCSSTK15. In columns (1) we exploit only parallelism from the tree; in columns (2) we combine the two levels of parallelism.

MUPS is designed for shared memory multiprocessors but it was not too difficult to develop a version that could run on the virtual shared memory environment of the BBN TC2000 ([4]). However, because access to memory on this machine is not uniform (remote access is not cached by default and takes about 3.5 times the time of local access for a read, and 3.0 for a write), the performance of the shared memory code was not good. It is necessary to pay more attention to data locality in order to get acceptable performance and we show, in Table 12, two versions of the BBN multifrontal code. Version 1 is the relatively straightforward adaptation of the code while version 4 represents many refinements to this, including explicit copying of data to local memory so that it can be cached and effectively reused during the computation so reducing the amount of remote access. The results in Table 12 show quite clearly that it is still vital to respect data locality when using virtual shared memory. If one does so, then good performance can be obtained. We plan to test this same code on another virtual shared memory machine, the KSR-1.

	Number of processors					
	1	2	4	8	16	26
Version 1	328	227	151	105	103	95
Version 4	216	155	86	55	36	31

Table 12: Times in seconds for multifrontal code on BBN TC2000 on matrix BCSSTK15.

6 Conclusion

There are some industrial codes that are good candidates for easy parallelization. Some other codes require a change of solution techniques or a reorganization of the calculations. Nevertheless, we have seen that the gains achieved do not always come from parallelization but may come from cleaning up codes or from vectorization.

Most of the complications that arise when porting industrial codes to parallel computers come from the lack of maturity in programming environments (languages, debuggers, performance analyzers,..) and algorithm design (since most of the codes have been developed on serial machines).

Virtual shared memory is a very convenient feature that simplifies the porting of existing codes. It also demonstrates how shared memory can scale to hundreds of processors. But data locality is still crucial for efficiency. Therefore, most of the tuning aims at finding a good data distribution which is exactly the same problem as on multicomputers using message passing.

Massively parallel computers are increasingly considered as an alternative to conventional high performance computers. They compare very favourably in peak performance with shared memory computers of a similar cost. However, the application performance depends to a large extent on the ratio of the bandwidth to the computing power which tends to be poor compared with shared memory computers. There are still many research issues concerning the effective use of massively parallel computers as a production facility, some of them much more general (for example ability to handle a large number of users).

The use of heterogeneous networks of computers as a computing resource is quite attractive, because it is cost-effective and because of the availability of programming environments such as PVM that can also be used within a single parallel computer. But, from our experience, it is restricted to applications where the amount of communication is small compared with the computation.

Parallelizing compilers may one day be capable of generating efficient parallel code for multicomputers (and networks of computers) from sequential code but they still have a long way to go.

References

[1] P. R. Amestoy, "Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment", PhD Thesis, Technical Report TH/PA/91/2, CERFACS, Toulouse, France, 1991.

[2] P. R. Amestoy and I. S. Duff, "Vectorization of a multiprocessor multifrontal code", *Int. J. of Supercomputer Applics.* , **3**, 41–59, 1989.

[3] P. R. Amestoy and I. S. Duff, "Memory management issues in sparse multifrontal methods on multiprocessors", *Int. J. of Supercomputer Applics.* , **7**, 64–82, 1993.

[4] P. R. Amestoy, M. J. Daydé, I. S. Duff, and P. Morère, "Linear Algebra Calculations on a virtual shared memory computer", Technical Report TR/PA/92/99, CERFACS, Toulouse, France, 1992.

[5] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, "LAPACK : A portable linear algebra library for high-performance computers", SIAM, Philadelphia, 1992.

[6] R. Butler and E. Lusk, "Users' Guide to the p4 Parallel Programming System", Technical Report ANL-92/17, Argonne National Laboratory, 1992.

[7] J. L. Charles, M. J. Daydé, A. Petitet, L. Prévost, and E. Simonnet, "Evaluation de Calculateurs Multiprocesseurs pour des Logiciels et Bibliothèques Scientifiques du CNES : Rapport Final", Technical Report, CERFACS, Toulouse, France, 1993.

[8] M. J. Daydé, M.J., and I. S. Duff, "Use of Level 3 BLAS in **LU** factorization in a multiprocessing environment on three vector multiprocessors, the ALLIANT FX/80, the CRAY-2, and the IBM 3090/VF", *Int. J. of Supercomputer Applics.* , **5** (3), 92–110, 1990.

[9] M. J. Daydé, I. S. Duff, and A. Petitet, (1992), "A Parallel Block Implementation of Level 3 BLAS Kernels for MIMD Vector Processors", Technical Report TR/PA/92/74, CERFACS, Toulouse, France, 1992. To appear in *ACM Trans. Math. Softw.*

[10] M. J. Daydé, I. S. Duff, J. Y. L'Excellent, and L. Giraud, "Evaluation d'Ordinateurs Vectoriels et Parallèles sur un Jeu de Programmes Représentatifs des Calculs à la Division Avions de l'Aérospatiale : Rapport Final", Technical Report FR/PA/93/19, CERFACS, Toulouse, France, 1993.

[11] J. J. Dongarra and E. Grosse, "Distribution of Mathematical Software Via Electronic Mail", *Comm. ACM*, **30**, 403–407, 1987.

[12] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms", *ACM Trans. Math. Softw.* , **16**, 1–17, 1990.

[13] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, "Algorithm 679. A Set of Level 3 Basic Linear Algebra Subprograms: model implementation and test programs", *ACM Trans. Math. Softw.* , **16**, 18–28, 1990.

[14] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extented set of Fortran Basic Linear Algebra Subprograms", *ACM Trans. Math. Softw.* , **14**, 1–17 and 18–32, 1988.

[15] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, "Solving Linear Systems on Vector and Shared Memory Computers", SIAM, Philadelphia, 1991.

[16] J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker, "A Proposal for a User-Level, Message Passing Interface in a Distributed Memory Environment", Technical Report ORNL/TM-12231, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA, 1993.

[17] J. J. Dongarra, "Performance of Various Computers Using Standard Linear Algebra Software", Technical Report CS-89-85, University of Tennessee, Knoxville, Tennessee, USA, 1992.

[18] I. S. Duff, "Full matrix techniques in sparse Gaussian elimination", Numerical Analysis Proceedings, Dundee 1981, Lecture Notes in Mathematics 912, (edited by G.A. Watson), 71–84, Springer-Verlag, Berlin, 1981.

[19] I. S. Duff, R. G. Grimes, and J. G. Lewis, "Users' Guide for the Harwell-Boeing Sparse Matrix Collection (Release I)", Technical Report 92-086, RAL, England, 1992.

[20] I. S. Duff and J. K. Reid, "MA48, a Fortran code for direct solution of sparse unsymmetric linear systems of equations", Technical Report 93-072, RAL, England, 1993.

[21] I. S. Duff and J. K. Reid, "The Multifrontal Solution of Indefinite Sparse Symmetric Linear Systems", *ACM Trans. Math. Softw.* , **9**, 302–325, 1983.

[22] I. S. Duff and J. K. Reid, "The Multifrontal Solution of Unsymmetric Sets of Linear Systems", *SIAM J. Scient. Statist. Comput.*, **5**, 663–641, 1984.

[23] ISO, "Fortran 90", [ISO/IEC] 1539:1991(E), 1991.

[24] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM 3.0 user's guide and reference manual", Technical Report TM-12187, Oak Ridge National Laboratory, Tennessee, USA, 1993.

[25] L. Giraud and G. Manzini, "A Multi-Domain Euler Solver", Technical Report TR/CFD-PA/93/49, CERFACS, Toulouse, France, 1993.

[26] High Performance Fortran Forum, "High Performance Fortran Language Specification", Technical Report, Rice University, Houston, Texas, 1993.

[27] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage", *ACM Trans. Math. Softw.* , **5**, 308–323, 1979.

[28] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Algorithm 539. Basic linear algebra subprograms for Fortran usage", *ACM Trans. Math. Softw.* , **5**, 324–325, 1979.

[29] J. H. W. Liu, "The Role of Elimination Trees in Sparse Factorization", *SIAM J. Matrix Anal. Appl.*, **11**, 134–172, 1990.

[30] R. Schreiber and H. D. Simon, "Towards the Teraflop in CFD", Technical Report, NASA Ames Research Center, Moffett Field, CA, 1992.