The influence of vector and parallel processors on numerical analysis

I.S. Duff

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 634

THE INFLUENCE OF VECTOR AND PARALLEL PROCESSORS ON NUMERICAL ANALYSIS

Iain S. DUFF

Mars 1987

THE INFLUENCE OF VECTOR AND PARALLEL PROCESSORS ON NUMERICAL ANALYSIS

L'INFLUENCE DES PROCESSEURS VECTORIELS ET PARALLELES SUR L'ANALYSE NUMERIQUE

Iain S. DUFF

Résumé

Voici dix ans que le premier Cray-1 a été livré au Laboratoire de Los Alamos. Depuis, les supercalculateurs vectoriels se sont répandus et sont devenus importants pour la résolution des problèmes posés par de nombreux domaines des sciences et de l'ingénierie qui nécessitent du calcul à grande échelle. Leur influence sur l'analyse numérique a été moins fondamentale, mais nous en précisons la portée.

Dans la dernière année, des "superminis" présentant diverses formes plus générales de parallélisme sont apparus. Nous les décrivons brièvement et donnons les principes de leur programmation. De nombreux exemples tirés de plusieurs domaines de l'analyse numérique dont algèbre linéaire, l'optimisation et la résolution des équations aux dérivées partielles nous permettent de soutenir la thèse que ces derniers auront une bien plus grande influence sur l'analyse numérique que leurs prédécesseurs vectoriels.

Abstract

It is now ten years since the first CRAY-1 was delivered to Los Alamos National Laboratory. Since then, supercomputers with vector processing capability have become widespread and important in the solution of problems in many areas of science and engineering involving large-scale computing. Their influence on numerical analysis has been less dramatic but we indicate the extent of that influence.

In the last year or so, advanced "superminis" that exhibit various more general forms of parallelism have been developed and marketed. We identify these and give some general principles which algorithm designers are using to take advantage of these parallel architectures. We argue that parallel processors are having a much stronger influence on numerical analysis than vector processors and illustrate our claims with examples from several areas of numerical analysis including linear algebra, optimization, and the solution of partial differential equations.

This paper is based on an invited lecture given at the State-of-the-Art Meeting in Birmingham, England, April, 1986.

Keywords : parallel computers, vectorization, parallelism, supercomputers, numerical analysis, linear algebra, optimization, partial differential equations.

Mots clés : ordinateurs parallèles, vectorisation, parallélisme, supercalculateurs, analyse numérique, olgèbre linéaire, optimisation, équations aux dérivées partielles.

CONTENTS

1 Introduction

It is interesting to reflect that, if a paper of this title had been included in the proceedings of the previous State-of-the-Art conference (Jacobs 1977), its content would have been considerably different from the following text. It is not that parallelism was not studied. There were many early papers on the subject (for example, Miranker and Liniger 1967 and Traub 1973) and much analysis principally on abstract machines like the paracomputer (Schwartz 1980) where unlimited parallelism could be allowed without concern for communication or synchronization costs. Nor was it that parallel machines did not exist. Burroughs delivered the ILLIAC IV to Illinois in 1972 (Bouknight *et al.* 1972) and there was much activity in developing numerical algorithms for that machine (for example, Sameh 1977). However, the predominant trend in research ten years ago was that of complexity analysis, and results of the form "the solution of $\mathbf{Ax}=\mathbf{b}$ can be performed in $0(\log_2{}^2 n)$ time on a machine with $2n^{3.31}/\log_2{}^2 n$ processors" (Csanky 1976) would have been considered at great length in a survey paper written at that time.

However, the situation in 1986 is radically different. The emphasis has changed to the construction of algorithms for specific architectures, the design of algorithmic principles for a range of parallel or vector machines, the establishment of a parallel programming methodology, and aids to portability such as language developments, language extensions, and the development of schedulers. A major reason for such a change of emphasis is the availability of machines: both high-powered vector supercomputers and, more recently, several general-purpose superminis with parallel architectures. We discuss such machines in Section 2, distinguishing between vector and parallel architectures but avoiding any detailed description of architectures. We refer the reader to Dongarra and Duff (1985) for a consumer's guide to commercially available hardware. The survey paper by Ortega and Voigt (1985) contains a summary of developments in hardware and algorithms just prior to the arrival in the marketplace of these machines.

It is also worth mentioning that several projects at universities and research laboratories are now stimulating much research and are promising to make real advances in the design of parallel architecture machines. Indeed, some of the current commercial machines have been developed directly from such projects. Examples of projects include the CEDAR project at Illinois, the NYU ULTRA project and the allied IBM RP3 effort, the CMU Warp, LCAP at IBM Kingston, the Connection project at MIT, and the Caltech hypercube.

The influence of vector computers is presented in Section 3 and a discussion of some basic concepts in parallel computing in Section 4. Then, in Section 5, we define several techniques used in algorithms which exploit parallelism; in each case we give concrete examples of such algorithms, with one or two explored in detail. In the following three sections, we briefly consider the influence of both vector and parallel processors in the areas of linear algebra, partial differential equations, and optimization, before concluding with some overall remarks on current and future trends in Section 9.

2 Vector and parallel processors

The most general categorization of parallel architectures is due to Flynn (1966). Two of his main categories are SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data) machines. Broadly speaking, these subdivisions correspond to vector computers and more general parallel computers. We do not, however, discuss any categorization in detail because we do not feel that it is particularly germane to our present concern. Indeed, it is not our intention to give a rigorous classification or to list all machines which are available (see, for example, Dongarra and Duff 1985). We do, however, make a distinction between vector and parallel machines. Although we subdivide the latter between shared and local memory architectures, we make limited use of this distinction in the later sections.

We define a vector processor as one which can process operations on vectors with great efficiency but which does not exhibit a more flexible form of parallelism. For example, when we discuss vector machines per se we assume that it is not possible for them to execute quite different programs concurrently. Examples of vector machines are those of Cray Research (CRAY–1, CRAY X-MP, and CRAY–2), Control Data (CYBER 205), Fujitsu (FACOM VP 50, 100, 200, and 400 ... also marketed by Amdahl and Siemens), Hitachi (S–810 and 820), and NEC (SX–1, SX–2). We concentrate primarily on the influence of these high-powered and expensive supercomputers in Section 3 although we note that there are several other cheaper but less powerful vector machines now available, for example the IBM 3090 VF machine, the Alliant, several FPS machines, and the so-called Crayettes from Convex and Scientific Computer Systems.

Some of these vector machines have models with more than one processor with access to a shared memory (CRAY X-MP, CRAY–2, and Alliant). Other machines with a parallel architecture whose processors share a common memory include the ELXSI 6400, FLEX/32, Sequent Balance 21000, and the ENCORE Multimax. Denelcor, whose HEP computer played a significant role in the development of algorithms and methodology for shared memory machines (for example, Kowalik 1985), has now been liquidated, a testimony to the current competitiveness in this area of scientific computing. Although we have lumped the shared memory machines together in this way, we should stress that many architectural differences exist which can have an important effect on the design and performance of algorithms. For example, access to common memory may be via a global bus or a more costly but faster switch, which itself may have a range of configurations. Furthermore, each processor may have a cache, and cache management may vary greatly from machine to machine (for example, Montry and Benner 1985). However, in the spirit of the rest of this paper we regard such differences as affecting only the implementation details specific for the machine itself and so do not consider them further.

In local memory parallel architectures, each processor has its own memory and communicates with other processors through message passing. The local memory machines can be further subdivided according to the connections between processors. Popular configurations include a linear array, a ring (for example, the CDC Cyberplus), a

two-dimensional lattice with toroidal topology (the Goodyear MPP and the ICL DAP), or a more complex connectivity as in the BBN Butterfly. A cube-connected cycle configuration has been analysed, and algorithms have been designed for it (Preparata and Vuillemin 1981), but we know of no commercially available machine with this architecture. Perhaps the currently most popular local memory topology is that of a hypercube, distinguished by the fact that each processor in a hypercube with 2^n processors is connected to n processors which can be identified by flipping a single bit of a binary string of length n which labels the processor. As well as the logarithmic growth in connection complexity, the hypercube is also easily scaled to reasonably many processors (> 1000) and has the attractive property that a message can be passed from one processor to any other by communication paths whose length is logarithmic in the number of processors. An attractive feature is the parallel communication feature whereby messages can be sent from all the processors to all other processors concurrently in $2 \log_2 n$ steps. Examples of computers exhibiting this topology are the Ametek System 14, INTEL iPSC, NCUBE, the Thinking Machines Connection Machine, and the FPS T-Series machines.

Two comments are worth making when comparing vector supercomputers with parallel machines. The first is that, although the latter are usually much slower than the former, they are far cheaper, typically costing between \$500K and \$1.5M as opposed to the more than \$10M price tag on most supercomputers. Their manufacturers claim a better price-performance characteristic compared with the supercomputers. A second difference is that many of the superminis are built from widely available chips and indeed owe their genesis to research projects of universities or non-commercial laboratories. Thus most of the companies are small, new, and presently financed by venture capital rather than sales or a major company. The viability of these companies will therefore depend on establishing a niche in the marketplace. This again justifies our determination to stand back from discussing detailed implementations, since by the time of publication any architecture being so addressed could well be obsolete. Our main reason for the foregoing catalogue of available computers is to emphasize the prevalence of vector and parallel architectures. As we said in the introduction, this availability is one of the most important factors in the influence of vector and parallel computers on numerical analysis.

3 The influence of vector processors

Without doubt the advent of vector machines has had a profound influence on computation in science and engineering. Indeed, one can argue that the field of scientific computation owes its birth to the existence of such machines. Certainly, since the first CRAY was delivered to Los Alamos in 1976, increasingly complex calculations have been undertaken using vector supercomputers whose computational rates now approach one Gigaflop (a thousand Megaflops or one thousand million floating point operations per second). In spite of this impact, we would argue that the influence on numerical analysis due to such machines is relatively small. Certainly there are few numerical analysis research papers in which the use of such architectures is the primary concern. The reason for the lack of influence lies both in the very nature of such supercomputers and in the lack of in-house access by most numerical analysts.

By definition, a supercomputer is a powerful computing engine which means that, even in non-vector mode, its capacity for computation exceeds that of other computers. Thus, without paying much attention to optimizing or designing algorithms to exploit the architecture, the use of a supercomputer should give considerable computational gains. Much of the influence of vector supercomputers in computational science has been due to this fact alone. Indeed vectorization is sometimes considered as merely an added bonus. In this context, it is not surprising that the influence on numerical analysis has not been dramatic. In particular applications, however, significant attempts have been made to exploit vectorization, and the computational science literature abounds with papers on this subject. Note that our definition of supercomputers is not machine specific, and indeed machines presently considered supercomputers will not be so considered in a few years time.

Since many vector machines require the use of fairly long vectors (well over 1000 in some instances) to exploit their hardware fully, much effort has been spent in reorganizing the data so that inner-loop operations are performed on long vectors (for example, see Rizzi 1985). This is usually a top-down procedure and often results in a significant redesign of the entire method. In this sense, the influence of vector machines on computational science is very great, but few numerical analysts would regard this as numerical analysis, although some would argue that such considerations should be of more concern. The other major way in which vector processing has influenced computational science has been in the choice of solution algorithms employed, for example the choice of method for the solution of linear equations may be very dependent on the architecture being used.

However, the concept of a vector is a rather simple one in a mathematical sense and so has not caught the interest of most numerical analysts. When dealing with vectors, the natural tendency has been to think in terms of a bottom-up approach, and a primary concern has been to develop techniques to assist in the portability of codes while maintaining efficiency over a range of architectures. Much use has been made of the BLAS (Lawson *et al.* 1979) and most manufacturers have implemented efficient versions for their machines, usually using assembler level coding.

As an example of the use of the BLAS, we consider the product of a matrix $\mathbf{A}$ with a vector $\mathbf{c}$. One can view this product as a linear combination of the columns of $\mathbf{A}$, so that

$$\mathbf{Ac} = \sum c_j \mathbf{A}_{\bullet j} \tag{3.1}$$

where $\mathbf{A}_{\bullet j}$ is the $j-th$ column of $\mathbf{A}$, or one can obtain the $i-th$ component of the product as a scalar product between $\mathbf{c}$ and the $i-th$ row of $\mathbf{A}$. That is,

$$(\mathbf{Ac})_i = \mathbf{A}_{i\bullet}^T \mathbf{c} \tag{3.2}$$

where $\mathbf{A}_{i\bullet}^T$ is the $i-th$ row of $\mathbf{A}$. In the former case (3.1), use can be made of the SAXPY routine from the BLAS, which adds a multiple of one vector to another. In case (3.2), the scalar product routine, SDOT, is appropriate. On some vector machines, the use of SAXPY is much to be preferred. For example, on the CYBER 205, the SAXPY routine is intrinsically more efficient than the SDOT code (Louter–Nool 1985) and accesses the columns, which are stored contiguously, rather than the rows which are not. On the other hand, a hardware peculiarity in

the CRAY–1 allows the SDOT routine to perform significantly faster than SAXPY on that machine (Duff and Reid 1982). Thus, although efficient kernels can be used to effect matrix-vector multiplication on vector machines, the choice of kernel can be very machine dependent.

This fact, coupled with a desire for portability and efficiency over a wide range of parallel and vector machines, has led many people to use larger computational modules when designing code for vector machines. An obvious candidate is matrix-vector multiplication itself. Dongarra *et al.* (1984) have formalised the higher-level modules in a proposal for an extended set of BLAS, sometimes called $O(n^2)$ BLAS since the arithmetic involved is $O(n^2)$ for vectors of length n rather than the $O(n)$ of the original BLAS. The proposal includes BLAS for matrix-vector multiplication (where the matrix can be general, symmetric, triangular, or banded), rank-one and rank-two updates to matrices, and the solution to triangular systems. It is possible to formulate many of the routines from LINPACK and EISPACK using these kernels (for example, Dongarra *et al.* 1986a), and it is hoped that the establishment of an agreed set of extended BLAS will persuade manufacturers to provide efficient code for them in a similar way to the current situation for the original BLAS. The omens are good since already many manufacturers supply efficient code for matrix-vector multiplication.

It is worth reflecting that much debate has gone into the definition of what should be included in a set of extended BLAS. For example, Gustavson (private communication 1985) has found that, in order to obtain high performance in the solution of equations on the IBM 3090 VF, it is necessary to code the kernel as a matrix-matrix product, while the frontal code, MA32, at Harwell (Duff 1983) uses a double Gaussian elimination step (rank-two change to an unsymmetric matrix) to obtain nearly maximum performance on both a CRAY–1 and a CRAY–2 (Dave and Duff 1986). It is inevitable that any proposal will not cover all needs or present usage, but the general feeling and that of the authors of the proposal is that the important thing is to establish a standard which manufacturers will recognize.

A form of parallelism more general than simple vectorization consists of the independent execution of different instantiations of the same computational sequence. It is possible to exploit this very common form of general parallelism on vector computers by vectorizing across the independent streams. We can illustrate this with the ADI method as used in the solution of elliptic or parabolic equations. At each half-step of the ADI method it is required to solve a set of k tridiagonal systems, each of order n,

$$\mathbf{T}_i\mathbf{x}_i = \mathbf{b}_i , \qquad i = 1,2,\ldots,k, \tag{3.3}$$

which are independent of each other. Now, the inherently recursive nature of the solution of a tridiagonal system has made it somewhat of a "cause célèbre" in the world of vectorization and, although algorithms exist which are superior to straightforward Gaussian elimination (see Section 6), highly vectorized codes do not exist. However, if one chooses as vectors the corresponding components from each system, for example $[\mathbf{x}_1]_1[\mathbf{x}_2]_1[\mathbf{x}_3]_1\cdots[\mathbf{x}_k]_1$, then the systems (3.3) can be solved as a single block tridiagonal system with n blocks of order k and vectors of length k.

It is difficult to interpret more general forms of parallelism in this way but, as we shall see in Section 5, many of these forms are of a similar kind to that described above.

4 Basic issues in parallelism

Since it is recognized that much of the intended audience of this paper may have little prior experience of concepts and issues in parallelism, some of the basic terms are discussed in this section.

Returning to vector computers for a moment, a common approximation for the time to perform a calculation on a vector of length n is

$$t_n = (n+n_{1/2})r_\infty^{-1} \text{ microseconds} , \tag{4.1}$$

where r_∞ is the maximum asymptotic rate of computation, and $n_{1/2}$ is the vector length at which half the asymptotic rate is attained (Hockney and Jesshope 1981). We use the notation $n_{1/2}$ since it is now standard in the literature. We would like to stress, however, that the quantity $n_{1/2}$ is independent of n. The formula (4.1) gives rise to a performance curve of the form shown in Figure 4.1 whose shape gives quantitative justification to our desire for long vectors in Section 3. Typical values for r_∞ and $n_{1/2}$ on a 2–pipe CYBER 205 utilizing linked triads in 64–bit arithmetic are 200 and 200 respectively. The size of $n_{1/2}$ is important when designing and comparing algorithms on vector machines.

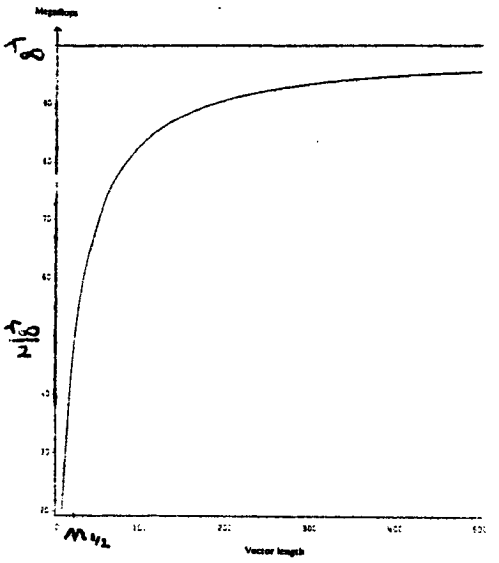

Figure 4.1. Typical performance curve for vector machines.

Another issue pertinent to both vector and parallel machines is the effect of vectorization (or parallelism) on the run time of a complete program or subroutine. Clearly the higher the percentage of calculations performed in vector or parallel mode, the greater the gains from vectorization or parallelism. This can be formulated in many ways and we choose one of the earliest and simplest as applied to vectorization, viz.

$$R = \frac{1}{fv^{-1} + (1-f)s^{-1}} , \tag{4.2}$$

where R is the overall computational rate, v the vector rate, s the scalar rate, and f the fraction of the number of operations performed at the vector rate. This is often termed Amdahl's Law

(Amdahl 1967) and, although it is a gross over-simplification, it is adequate for our present purpose. We have plotted equation (4.2) in Figure 4.2 where we have chosen values of v and s appropriate for the CRAY-1.

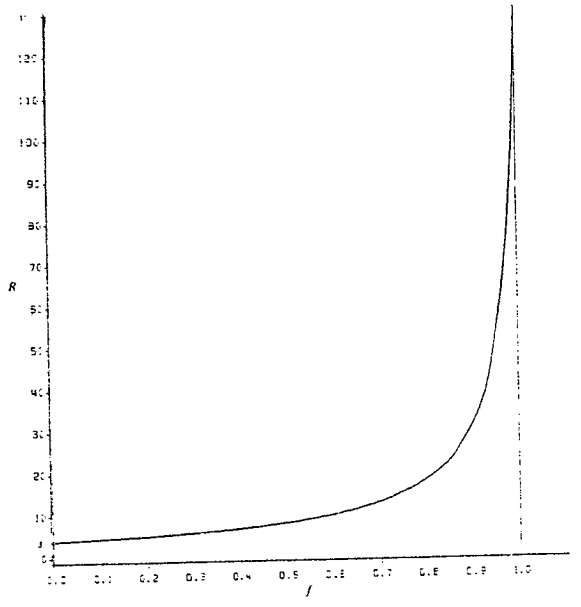

Figure 4.2. Graph of Amdahl's Law for machine with v=130 and s=4.

On looking at Figure 4.2 it is immediately apparent that a very high percentage of vectorization is necessary if overall computational rates comparable to the vector rate are to be attained. For example, even if 90 percent of the calculation were at the vector rate and this part could be done in zero time, then the speed-up would be limited by the 10 percent scalar part to a factor of 10. Fortunately, for some important calculations, for example the solution of linear equations, a very high percentage of the calculation vectorizes.

The counterpart to Amdahl's Law for general parallelism is Ware's Law (Ware 1973) given by

$$S = \frac{1}{fp^{-1} + (1-f)} \quad , \tag{4.3}$$

where f is the fraction of calculation in parallel mode, p the number of processors (amount of parallelism), and S the speed-up. Performance measures for parallel computation (including a definition of speed-up) are currently a matter for some debate. The S in equation (4.3) is the ratio of time for a code on one processor to that of the identical code on p processors. Some people prefer to define speed-up as the ratio that compares the best parallel algorithm with the best sequential one, which may of course be different. Indeed some people define a similar measure using the increase in problem size, for the same time of solution, as the number of processors increases. Of course equation (4.3) is quite inadequate in the parallel case (see, for example, Buzbee 1984) since communication and synchronization delays are absent, but we use it to illustrate the risk of making over-optimistic extrapolation. For example, a speed-up of

	30 × 30 grid	10 × 100 grid
No parallelism within nodes	3.7	7.5
Parallelism within nodes	30	47

Table 4.1. Illustration of effect of change in granularity. Values given are, in each case, the maximum speed-up.

where load balancing is of concern are multisectioning (Section 5.2) and global optimization (Section 8).

The perimeter effect relates to the ratio of the amount of computation within a processor to the amount of data that must be transferred between the processor and other processors. In some cases, for example when solving grid-based problems by subdividing the grid between processors, the amount of data transferred (the *boundary*) grows sublinearly with the computation. The implication is that the overheads of a message passing environment become less significant as problem size (and granularity) increases. Currently, this effect is rarely encountered because of the sizes of problems being run on message passing architectures. Indeed there is some debate concerning this phenomenon, because of the efficiency of passing long messages, and because the implications of local memory size can prevent an arbitrary increase in problem size. McBryan and Van de Velde (1986) have, however, observed the perimeter effect when solving large systems of hyperbolic partial differential equations on the Caltech hypercube, and Lichnewsky (1982) has observed the phenomenon when solving linear systems arising in finite-element calculations.

Our aim in this section has been to introduce some of the basic issues in parallel processing of concern to the numerical analyst, and we will feel free to use the terms without further comment in later sections. We have not, however, been completely exhaustive and recommend the book by Hockney and Jesshope (1981) for further background reading. The books by Kronsjö (1985) and Schendel (1984) discuss the design of parallel algorithms in some detail, although they are more theoretical than our present approach.

5 Techniques for the development of parallel algorithms

In this section, we classify several commonly used techniques for exploiting parallelism. In doing so we are extending earlier work of Voigt (1985), van Leeuwen (1985), and te Riele (1985). Like them, we preface our categorization with the caveat that any particular algorithm may exhibit traits of more than one category. Our main intent in following this line is to give numerical analysts a feeling for the issues which influence parallel algorithm development. By giving concrete examples of each issue, we illustrate both the particular technique and its influence on numerical analysis. Because of the detail with which we explore these techniques, we have chosen to identify each in a separate subsection. In all cases we have kept references to specific parallel architectures to a minimum so that the generality of each paradigm is emphasized.

5.1 Vectorization

We discussed vectorization in Section 3. It is clearly a very powerful tool and, since some computers combine aspects of both vector and parallel machines (for example, CRAY X-MP, CRAY-2, and ALLIANT), the advent of more general forms of parallelism will not remove the need for vectorization. Indeed all today's supercomputers use vectorization as their principal means of achieving high-speed computation. Additionally, as we illustrated with the example of ADI in Section 3, it is often possible to reformulate parallelism as vectorization in a rather straightforward manner. A technique which uses the same philosophy as the hardware of vector processors is pipelining. We discuss a pipelined technique for **QR** factorizations in Section 6.

5.2 Divide and conquer

Without doubt, divide and conquer is the most widespread technique used in the development of parallel algorithms. The idea is the first that would occur to anyone contemplating parallel algorithm design. One first partitions the problem into several subproblems and then solves the subproblems separately. Unless the subproblems are independent, data must be communicated between them. For the success of a divide and conquer technique, it is necessary that the gains in solving the subproblems in parallel are not outweighed by the work required to construct the solution to the whole problem from that of the subproblems.

There are many areas in numerical analysis where divide and conquer methods are employed. In linear algebra, they are used in the solution of banded and tridiagonal systems (Wang 1981) and in obtaining eigenvalues of symmetric tridiagonal matrices (Cuppen 1981). Bisection or more general subdivision techniques include using Sturm sequences to isolate eigenvalues (Barlow *et al.* 1983), root-finding techniques (Kronsjö 1985), and line-search methods (Schnabel 1984). Problems defined over two or three space dimensions often are amenable to a subdivision process where the only information that needs to be communicated is data on the boundaries of the subdivisions. In the solution of partial differential equations, techniques of this kind are termed domain decomposition (see for example, Glowinski *et al.* 1982 and Chan and Resasco 1985). In structures problems, the term substructuring is used (Noor *et al.* 1977), and in the solution of linear systems we use the term dissection (George 1973). The original motivation for these techniques was not to design algorithms for parallelism but rather to solve very large problems by splitting them into tractable subproblems. As in "sectioning methods", which can be viewed as a one-dimensional form of substructuring, it is the task of the numerical analyst to control and combine the solutions of the subproblems in order to solve the main calculation. The added bonus of easy parallelism has spurred much work in this area, particularly in domain decomposition for the solution of partial differential equations.

Similar decomposition techniques are used in other areas, including approximation theory (see Cox 1987, for example), global optimization (Schnabel 1984), and combinatorial optimization (Kindervater and Lenstra 1985, Roucairol 1986), in addition to many non-numerical and semi-numerical areas, such as sorting (Tseng and Lee 1984) and

convex-hull problems (Evans and Mai 1985), which are outside the scope of this present discussion.

We will now indicate the use of divide and conquer in more detail by looking further at algorithms in two of the above areas, namely tridiagonal eigensystems and parallel multisectioning.

The divide-and-conquer algorithm for symmetric tridiagonal eigensystems was originally proposed by Cuppen (1981) and has been developed for shared memory multiprocessors by Dongarra and Sorensen (1986) and for local memory machines by Ipsen and Jessup (1986). The idea is simply to observe that any $n \times n$ tridiagonal matrix, $\mathbf{T}$, can be partitioned as

$$\mathbf{T} = \begin{pmatrix} \mathbf{T}_1 & \beta \mathbf{e}_k \mathbf{e}_1^T \\ \beta \mathbf{e}_1 \mathbf{e}_k^T & \mathbf{T}_2 \end{pmatrix} = \begin{pmatrix} \hat{\mathbf{T}}_1 & 0 \\ 0 & \hat{\mathbf{T}}_2 \end{pmatrix} + \beta \begin{pmatrix} \mathbf{e}_k \\ \mathbf{e}_1 \end{pmatrix} \begin{pmatrix} \mathbf{e}_k^T & \mathbf{e}_1^T \end{pmatrix}$$

where $\mathbf{T}_1$, $\mathbf{T}_2$, $\hat{\mathbf{T}}_1$, and $\hat{\mathbf{T}}_2$ are tridiagonal matrices, β is the (k,k+1)th entry of $\mathbf{T}$, and $\mathbf{e}_1$ and $\mathbf{e}_k$ are the first and k-th columns of the $(n-k) \times (n-k)$ and $k \times k$ identity matrices respectively. If the eigendecompositions of $\hat{\mathbf{T}}_1$ and $\hat{\mathbf{T}}_2$ are given by

$$\hat{\mathbf{T}}_1 = \mathbf{Q}_1 \mathbf{D}_1 \mathbf{Q}_1^T \qquad \text{and} \qquad \hat{\mathbf{T}}_2 = \mathbf{Q}_2 \mathbf{D}_2 \mathbf{Q}_2^T$$

then $\mathbf{T}$ can be written as

$$\mathbf{T} = \begin{pmatrix} \mathbf{Q}_1 & 0 \\ 0 & \mathbf{Q}_2 \end{pmatrix} \left\{ \begin{pmatrix} \mathbf{D}_1 & 0 \\ 0 & \mathbf{D}_2 \end{pmatrix} + \beta \begin{pmatrix} \mathbf{q}_1 \\ \mathbf{q}_2 \end{pmatrix} \begin{pmatrix} \mathbf{q}_1^T & \mathbf{q}_2^T \end{pmatrix} \right\} \begin{pmatrix} \mathbf{Q}_1^T & 0 \\ 0 & \mathbf{Q}_2^T \end{pmatrix} ,$$

where $\mathbf{q}_1$ is the last row of $\mathbf{Q}_1$ and $\mathbf{q}_2$ is the first row of $\mathbf{Q}_2$. Thus, the eigendecomposition of $\mathbf{T}$ has been reduced to the eigenproblem for a rank-one change to a diagonal matrix. This has been considered by Bunch *et al.* (1978), among others. Naturally, to obtain more parallelism the algorithm should be applied recursively by subdividing $\hat{\mathbf{T}}_1$ and $\hat{\mathbf{T}}_2$, and so on. The results of Dongarra and Sorensen (1986) show good speed-up on both the Denelcor HEP and the Alliant FX/8. An interesting feature of this work is that the divide-and-conquer algorithm can save work by deflation in the rank-one update problems to the extent that, even on a sequential machine, it can outperform the **QR** implementation in the `TQL1` routine of `EISPACK`. For a general symmetric matrix, one must first use orthogonal similarity transformations to reduce it to symmetric tridiagonal form. Parallel methods for this calculation are discussed by Dongarra and Sorensen (1986) and Moler (private communication 1986).

Multisectioning methods require a good adaptive load-balancing strategy since it is not known in advance where the work will be. When using the method to obtain the eigenvalues of a symmetric matrix using Sturm sequences, each processor can work independently on a different subinterval and the main problem lies in ensuring that a similar amount of work is required in each subinterval. Ipsen (private communication, 1986) does this by first running *all* processors on one large interval that is known to contain all the eigenvalues (obtained, for example, from Gerschgorin bounds) in order to obtain an approximate eigenvalue distribution that is used to determine the apportionment of subintervals to processors during the main phase. Bernstein and Goldstein (1986) have also developed a parallel Sturm sequence algorithm.

5.3 Reordering

The second main technique used in designing parallel algorithms is that of reordering the problem or data to enhance the underlying parallelism. Perhaps the area in which this technique has been most used is in the solution of linear equations from finite-difference discretizations of partial differential equations. For example, when using the SOR method to solve for a five-point discretization of the Laplacian on an $m \times n$ grid, the natural pagewise ordering of the points forbids parallelism since a value at a point is computed from its immediate predecessor in the same row. This is easily overcome by reordering the points by diagonals so that all points on each diagonal can be computed in parallel. When using successive line over-relaxation (SLOR), parallelism is most easily obtained by updating first the odd rows and then the even ones (called a zebra ordering). A red-black, or chequerboard, ordering is very beneficial for vectorization since vector lengths can be increased from n to $mn/2$. Much work has been done to evaluate the effect of different orderings on the convergence of SOR (see, for example, Adams and Jordan 1984, O'Leary 1984, and Adams 1985) which seems remarkably resilient to such assaults. The use of SOR or incomplete **LU** factorization (ILU) preconditionings in conjunction with conjugate gradients or similar methods can also be similarly implemented in parallel by the use of multicolour orderings on the original grid (Adams 1983). Lichnewsky (1984) combines dissection with multicolouring to develop preconditioners appropriate for multiple-processor vector machines. Erhel *et al.* (1985) investigate the effect of reorderings on preconditioned conjugate gradients in the context of vectorization.

In the one-dimensional case, if a red-black ordering is performed on a tridiagonal system being solved by a direct method, cyclic reduction results. This reordering can also be obtained from a nested-dissection technique. Thus we see one instance of the relationship between divide and conquer and reordering, illustrating the point we made in Section 5.1 concerning the overlap between categories for parallel algorithms.

Although nested dissection and minimum-degree orderings are very similar in behaviour for the amount of arithmetic and storage when used as a pivoting strategy for the solution of grid-based problems using sparse Gaussian elimination, they give fairly different levels of parallelism when used to construct an elimination tree for the parallel implementation of multifrontal methods (Duff 1986). We illustrate this point with some results from Duff and Johnsson (1986) in Table 5.3.1.

Ordering	Minimum degree	Nested dissection
Number of levels in tree	52	15
Number of pivots on longest path	232	61
Maximum speed-up	9	47

Table 5.3.1. Comparison of two orderings for generating an elimination tree for multifrontal solution. The problem is generated by a 5–point discretization of a 10×100 grid.

Another reordering of a computation, particularly suitable for local memory machines, is the wavefront version of Choleski's $\mathbf{L}\mathbf{L}^T$ factorization (see, for example, Kung *et al.* 1981, and O'Leary and Stewart 1985). We indicate the order of computation of the entries of $\mathbf{L}^T$ in Figure 5.3.1, where all calculations of the same number are performed in parallel and data are transferred from a calculation at stage i to stage i+1 only. An important thing about the wavefront method is that, as soon as the operations with the first pivot have been performed on entry (2,2) (at step 3), this entry can then immediately be used as the second pivot and calculations using the second pivot row and column can sweep down the matrix immediately following the calculations using the first pivot. Subsequent pivots can follow in like fashion so that the whole decomposition is effected in 3n–2 parallel steps. A similar strategy can be adopted in solving time-dependent problems where different waves can simultaneously be computing at different time steps. This windowing technique is discussed by Saltz and Naik (1985).

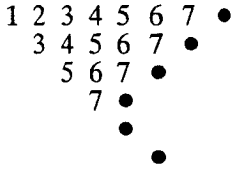

Figure 5.3.1. Order of computation in the wavefront method.

The use of ADI techniques on hypercubes poses a rather interesting ordering problem since the data must be transposed between half sweeps in order to maintain parallelism in both directions. On some machines (for example the CYBER 205) it is best not to transpose the data, but rather to solve the systems in the second direction as a very large concatenated single tridiagonal system. However, Saad and Schultz (1985c) have found it sometimes preferable to do the transposition on the hypercube and keep the parallelism over the tridiagonal systems (see Section 3) in both directions. McBryan and van de Velde (1986) discuss in some detail algorithms for matrix transposition on hypercubes.

In the **QR** method, reordering to obtain parallelism is obtained by using the suggestion of pairwise pivoting of Gentleman (1976). Clearly this freedom allows several Givens rotations to proceed at once, the overall efficiency being determined by the ordering chosen. Sameh and Kuck (1978) and Modi and Clarke (1984) have described possible orderings, and the latter scheme can be shown to require only about $\log_2 m + (n-1)\log_2(\log_2 m)$ steps on a system with m rows and n columns, where $m \gg n \gg 1$. Pairwise elimination can also be applied to Gaussian elimination in a stable fashion (Sameh 1985, Sorensen 1983).

5.4 Recursive doubling

Recursive doubling techniques were originally suggested for vectorization but, as we shall shortly indicate, they give rise to inconsistent methods and so are asymptotically inferior. They have, however, been suggested as being more viable on parallel architectures where extra work may not be penalized. As an example of recursive doubling suppose that we wish to compute

$$\prod_{i=1}^{k} r_i \ , \qquad k=1,2\ldots,n,$$

for given r_i, $i=1,2,\ldots,n$.

A straightforward computation yields the n products in $n-1$ multiplications. If, however, we place $r_1, r_2, \ldots, r_n$ in a vector, $\mathbf{r}_0$ say, and denote by $\text{shift}_k(\mathbf{r})$ the operator that shifts the vector $\mathbf{r}$ down by 2^k positions and places 1's in the first 2^k positions, then the sequence of operations

$$\mathbf{r}_{k+1} = \text{shift}_k(\mathbf{r}_k) * \mathbf{r}_k \qquad\qquad k=0,1,\ldots \ ,$$

will produce a vector $\mathbf{r}_k$, after $\lceil \log_2 n \rceil$ steps, that holds the desired products. That is, the successive $\mathbf{r}_k$, $k=0,1,\ldots$, are given by

r_1	r_1	r_1		r_1
r_2	r_1r_2	r_1r_2		r_1r_2
r_3	r_2r_3	$r_1r_2r_3$		$r_1r_2r_3$
r_4	r_3r_4	$r_1r_2r_3r_4$		●
r_5	r_4r_5	$r_2r_3r_4r_5$		●
r_6	●	●	.....	●
r_7	●	●		●
●	●	●		●
●	●	●		●
●	●	●		●
r_n	$r_{n-1}r_n$	$r_{n-3}r_{n-2}r_{n-1}r_n$		$r_1r_2\ldots r_n$

Unfortunately, the complexity of each vector multiply is $O(n)$, giving $O(n \log_2 n)$ multiplications in total as opposed to $O(n)$ for the straightforward algorithm. Van Leeuwen (1985) gives a list of functions amenable to recursive doubling.

An algorithm for solving tridiagonal systems based on recursive doubling was proposed by Stone (1973). If one defines

$$q_0 = 1,\ q_1 = a_1, \quad q_i = a_i q_{i-1} - c_i b_{i-1} q_{i-2} \ , \ i=2,\ldots,n \ ,$$

then the **LU** factorization of the matrix

$$\begin{array}{cccc} a_1 & b_1 & & \\ c_2 & a_2 & b_2 & \\ & c_3 & a_3 & \bullet \\ & & \bullet & \bullet \ \bullet \\ & & & \bullet \ \bullet \\ & & & \bullet \end{array}$$

includes the entries

$$u_i = \frac{q_i}{q_{i-1}} \ ,$$

where the u_i are the diagonal entries of **U**. We see this by comparing the expression for generating the q_i above with the normal factorization equation

$$u_i = a_i - \frac{c_i b_{i-1}}{u_{i-1}} \ .$$

Stone observed that the recurrence for q_i can be written as

$$\mathbf{Q}_i \equiv \begin{pmatrix} q_i \\ q_{i-1} \end{pmatrix} = \begin{pmatrix} a_i & -c_i b_{i-1} \\ 1 & 0 \end{pmatrix} \begin{pmatrix} q_{i-1} \\ q_{i-2} \end{pmatrix} \equiv \mathbf{G}_i \mathbf{Q}_{i-1} = \prod_{j=2}^{i} \mathbf{G}_j \mathbf{Q}_1$$

from which the use of recursive doubling immediately follows. Because of its instability and inconsistency this method fell from favour but is now being revived, together with an investigation of methods for stabilization, in a more parallel environment. Doubling algorithms for Toeplitz systems are discussed by Brent *et al.* (1980) and Morf (1980).

5.5 Asynchronous techniques

As discussed in Section 4, the need for synchronizing parallel processes can be a major problem in achieving effective levels of parallelism. Therefore techniques have been developed where processes can execute asynchronously without having to suspend computation while awaiting the results from another process.

It is fairly easy to devise an asynchronous version of almost any iterative process. For example, Kronsjö (1985) discusses an asynchronous version of the Newton–Raphson iteration. However, it is usually more difficult to analyse the convergence or stability of such a technique.

An algorithm of this type, whose initial development predated the current interest in parallelism, is chaotic relaxation (Chazan and Miranker 1969, Miellou 1975, and Baudet 1978) where, for example, new values of the variables on a five-point discretized mesh are calculated from the four neighbouring values without regard to when these values were calculated. Experimentation has been done on parallel architectures (for example, at Los Alamos, Hiromoto, private communication 1985) that confirms the earlier analysis of Chazan and Miranker (1969) and indicates the feasibility of such techniques.

We discuss in more detail a suggested asynchronous algorithm for checking the convergence of an iterative process (Saltz *et al.* 1986). Here we assume that a region has been divided into subdomains, each assigned to a processor. Iterations continue on each subdomain with neighbouring subdomains communicating with each other, as mentioned in Section 5.2. There is therefore a local synchronization between neighbouring subdomains but this is not very costly if there is reasonable load balancing. Checking for convergence does, however, present more of a problem, and it is this part of the algorithm for which an asynchronous method is used. Each processor can conduct a convergence check on data in its subdomain leading to either a non-convergent result or a tentatively converged result (tentative because later data coming in from the boundaries may cause future non-convergence). Each processor keeps track of its number of iterations and a "header" is defined as the iteration count at the beginning of a tentatively converged sequence. A separate processor is designated as a "host". The algorithm then proceeds as follows, with action (2) being performed by the host and actions (1) and (3) by the other processors, all in an asynchronous fashion.

(1) As soon as a subdomain (assigned to processor k, say) reaches tentative convergence, processor k sends its header, i_k, to the host.

(2) Host waits until all processors have reported and then calculates

$$i_{\max} = \max\{i_k\} \quad .$$

It compares $i_{\max}$ with its previously calculated value (if any) and, if equal, the host broadcasts a stop to all processors and stops itself. Otherwise, the host broadcasts the new value to all processors and repeats action (2).

(3) When the processor receives prompt from host, it sends back its current header if tentatively convergent, or the next one if it is not.

Saltz *et al.* (1986) have programmed this method on a hypercube and find that the number of unnecessary calculations because of processors continuing iterations after convergence is very low (3–5%).

5.6 Explicit methods

Clearly explicit methods are very suitable for vectorization or parallelism. Thus, if

$$x_i^{n+1} = f_i(x^n, x^{n-1}, \ldots) \quad i=1,2\ldots$$

then each new x_i can be calculated in a parallel or vector operation.

An example of an explicit method for solving one-dimensional time-dependent partial differential equations is Euler's method, but its stability is only assured by restricting the time step drastically. The Crank–Nicolson method is unconditionally stable but requires the solution of a tridiagonal system at each time step. Evans and Abdullah (1983) suggest combining two steps of a semi-explicit method of Saul'yev (1964) to get an unconditionally stable method which is only implicit because a set of 2×2 systems must be solved at each stage. These systems are trivially invertible to yield an explicit method, termed the group explicit method by Evans and Abdullah. Indeed any method that is implicit because of the solution of small blocks may also exploit parallelism by solving its implicit systems in parallel. Gelenbe *et al.* (1982) experiment with both implicit and explicit methods for the one-dimensional heat equation using two processors and develop a probabilistic model to assist in their analysis.

A simple example of converting an implicit algorithm to an explicit one is the use of a Neumann series expansion

$$(\mathbf{I}-\mathbf{A})^{-1} = \mathbf{I} + \mathbf{A} + \mathbf{A}^2 + \ldots$$

to approximate the inverse of a matrix. Thus, the solution of a set of equations is sometimes possible by matrix-vector multiplication only – an explicit process. An example is given by van der Vorst (1982), who uses a truncated Neumann series approximation to $(\mathbf{I}-\mathbf{L})^{-1}$, $\mathbf{L}$ a lower triangular matrix, as a preconditioner for conjugate gradients.

5.7 "New" algorithms

It is often claimed that no new algorithms have been developed because of the advent of parallel processors. To some extent this is correct, although it is largely because such machines have only just recently become readily available, and much of the current effort is in trying to implement known techniques and modifications of them on the new architectures. Certainly many methods with obvious implications for parallelism have been proposed over the last few years. We have seen some examples in earlier sections and would maintain that much of the

present interest in block techniques, including block preconditioning methods (Concus *et al.* 1985), has been generated because of their application to parallel and vector machines. Indeed claims have been made for new algorithms which are just block factorizations. For example, the **WZ** algorithm of Evans and Hatzopoulos (1985) is identical to block 2×2 pivoting and can also be used as a splitting for SOR like methods or as a preconditioner (Evans and Sojoodi–Haghighi 1982).

New interest in old algorithms has been sparked by the advent of parallel architectures. For example, Gauss–Jordan elimination avoids the need for the recursive back-substitution step and is preferable to Gaussian elimination on some architectures in spite of its larger number of arithmetic operations (Bowgen *et al.* 1984). Indeed, the use of explicit inverses and a resurrection of Cramer's rule (Swarztrauber 1979) have also been suggested on parallel machines.

One novel idea is the use of multi-splittings (O'Leary and White 1983), where several iterations with different splittings of the same matrix are executed in parallel. Some analysis of this idea has been done, but no results have been reported of practical experience on parallel architectures.

Recently there has been development of a class of methods for solving partial differential equations called cellular automata or micro-random-walk methods. The technique is based on monitoring the motion of particles in cells and the principle calculations involved are non-numerical in the sense that most of the work involves manipulation of bit vectors. Because of their possible importance in the solution of numerical problems, it is clearly appropriate to mention them here. Much work in this area has been done by Wolfram on the Connection machine, and cellular automata have also been studied by Frisch *et al.* (1986a, 1986b), inter alios. Indeed it may well be that the existence of parallel architectures gives rise to the further development of this whole class of methods.

6 Influence on linear algebra

In the next three sections, we survey the influence of vector and parallel architectures on three major areas in numerical analysis. As with many new developments in the field, the area in which most effect has been felt is linear algebra. This is both because of the very well-defined nature of many linear algebra algorithms and their central role in other numerical areas. Indeed, some people have argued that the advent of vector and parallel machines has revitalized numerical linear algebra in a similar way to the development of sparse matrix techniques in the 1970s. We discuss the influence on linear algebra in this section by indicating some of the areas in which parallel or vector algorithms have been developed. From bibliographies on vector and parallel computing, for example that of the Bochum Computing Centre (Bernutat–Buchmann *et al.* 1983), it is evident that the other areas most influenced are the solution of partial differential equations and optimization. Indeed, out of around 1000 numerical references in the Bochum bibliography, 192 are on linear algebra, 116 on partial differential equations, and 57 on optimization (te Riele 1985). We discuss the influence on partial differential equations in Section 7 and on optimization in Section 8. In our conclusions in Section 9, we touch briefly on other areas of numerical analysis.

Since we are discussing the influence on partial differential equations in the next section, we will leave a discussion of linear algebra algorithms particularly geared to discretizations of partial differential equations until then. Here we do not explicitly consider the source of the linear algebra problem. Earlier reviews in this area have been given by Gentleman (1978), Heller (1978), Sameh (1983), and Ipsen and Saad (1985).

Tridiagonal systems play a critical role in numerical analysis so it is not surprising that they have received much attention. They have particularly been of interest in the study of parallelism because of the inherently recursive and sequential nature of such systems. We have already discussed several algorithms for tridiagonal systems which can exploit vectorization or parallelism, including Stone's method (Stone 1973) and cyclic or odd-even reduction (Buneman 1969). Another variant arises when we effectively perform an odd-even reduction on all equations instead of on alternate ones. This is called odd-even elimination, and one stage of this algorithm on the matrix

$$\begin{bmatrix} a_1 & b_1 & & & & & & \\ c_2 & a_2 & b_2 & & & & & \\ & c_3 & a_3 & b_3 & & & & \\ & & \bullet & \bullet & \bullet & & & \\ & & & \bullet & \bullet & & & \\ & & & & & \bullet & & \\ & & & & & \bullet & \bullet & \\ & & & & & \bullet & \bullet & \end{bmatrix}$$

eliminates the entries adjacent to the diagonal in the way that produces the form

$$\begin{bmatrix} a_1^1 & 0 & b_1^1 & & & & & \\ 0 & a_2^1 & 0 & b_2^1 & & & & \\ c_3^1 & 0 & a_3^1 & 0 & b_3^1 & & & \\ & c_4^1 & 0 & a_4^1 & 0 & & & \\ & & \bullet & 0 & \bullet & & & \\ & & & \bullet & 0 & 0 & \bullet & \\ & & & & \bullet & \bullet & 0 & \bullet \\ & & & & & 0 & \bullet & 0 \\ & & & & & \bullet & 0 & \bullet \end{bmatrix}$$

Next we eliminate similarly the new nonzero off-diagonal entries, and after $\log_2 n$ such reduction steps, the resulting system is diagonal. Like Stone's method, odd-even elimination, termed PARACR by Hockney and Jesshope (1981), is not consistent and is unstable, but further work on this algorithm continues. A divide-and-conquer strategy applied to tridiagonal matrices gives a partitioning scheme where most of the work can be performed on separate tridiagonal systems, the communication between them being kept low. An example of a partitioning scheme is that of Wang (1981), discussed in detail by Duff, Erisman, and Reid (1986).

Partitioning schemes can be extended to more general banded systems (Johnsson 1985 and

Dongarra and Johnsson 1986) and are suitable for implementation on both shared and local memory machines. Another possibility is to consider the banded system as a block tridiagonal system and use block cyclic reduction (Hockney and Jesshope 1981) in an analogous fashion to the point case discussed earlier. Wavefront methods akin to those discussed in Section 5.3 have also been proposed and developed for banded matrices, particularly those whose band is large (Saad and Schultz 1985b).

For full systems of equations, the use of the extended BLAS can have a marked effect on vector machines (Dongarra and Eisenstat 1984), although we do not know of good applications of this approach to local memory parallel architectures. On message passing architectures, several implementations of Choleski's method (Geist and Heath 1985) and Gaussian elimination (Geist 1985) have been proposed. These divide the columns in an a priori way across the processors (the cyclic reuse of processors is recommended on the hypercube by, for example, Geist and Heath 1985) and perform a column oriented reduction, passing information from a reduced column to its successors. On systolic or data-flow architectures, wavefront algorithms as discussed in Section 5.3 have been proposed (Kung *et al.* 1981 and O'Leary and Stewart 1985). There is, of course, abundant small granularity parallelism in Gaussian elimination, and one way of exploiting this is to perform pairwise pivoting, where, for example, at the first minor step rows 1,3,5,... are used to produce zeros in the first column of rows 2,4,... respectively. Pairwise pivoting has been discussed for ring architectures by Sameh (1985) and its stability analysed by Sorensen (1983). Block elimination methods (for example, the **WZ** algorithm of Evans and Abdullah 1983) have also been proposed and studied on local memory machines like the ICL DAP and the loosely-coupled Loughborough NEPTUNE (4 fully connected VAX 750s). Gauss–Jordan elimination has been proposed for the DAP, and a hybrid algorithm that uses block Gaussian elimination with Gauss–Jordan on the individual blocks has been developed (Bowgen *et al.* 1984). The often discredited techniques of Cramer's rule (Swarztrauber 1979) and explicit inverses have also been considered.

The use of a **QR** decomposition for solving full matrix equations, or more commonly for use in the least-squares solution of overdetermined systems, has received much attention. Pairwise ordering of the kind mentioned for Gaussian elimination in the previous paragraph can be used, and indeed many orderings have been proposed to exploit the independence of orthogonal rotations between different pairs of rows. Ordering schemes have been suggested by several people including Sameh and Kuck (1978) and Modi and Clarke (1984), and experiments have been performed on a range of parallel architectures. In pipelined **QR**, the matrix **R** is developed by reducing each row in turn using Givens rotations and is stored in a linear array (pipe) divided into segments. Each row of the matrix in turn is passed through the pipe with the only synchronization being that a row cannot enter a segment until it has been cleared by the previous row in the sequence (Dongarra *et al.* 1986b). Another technique which has been programmed for the hypercube by Chamberlain and Powell (1986) is to partition the matrix into block rows and perform most of the reduction within each block in an asynchronous fashion, communicating information between the blocks when required. Dongarra *et al.* (1986b) discuss and compare three methods for **QR** decomposition, namely the

use of the extended BLAS in LINPACK routines, the windowed Householder method which is similar in philosophy to the wavefront methods discussed in Section 5.3, and the pipelined Givens method. Parlett and Schreiber (1986) describe a block **QR** method suitable for implementation on systolic arrays.

For sparse systems, variants of the distributed Choleski algorithm (George *et al.* 1986b) and the pipelined Givens method (Heath and Sorensen 1986) can be developed as trivial extensions of the algorithms for full systems. Multifrontal techniques are also proving popular, either with automatic subdivision yielding small granularity (Duff 1986, Liu 1985), or using larger granularity by generating a simple subdivision of an underlying region followed by use of the frontal method on each region (Benner 1986, Geist private communication 1985, Berger *et al.* 1985). In general, partitioning methods can be used to split a matrix into subproblems (for example, Duff *et al.* 1986) that can then be handled in parallel. A good splitting or tearing is one which produces several approximately equal subproblems with only a few variables in the tear set. Several algorithms have been proposed but we know of no extensive comparisons or implementations on parallel architectures. Also there are unresolved stability problems when this approach is used on general systems. There are several approaches available for sparse systems arising from discretized partial differential equations but we defer discussion of these until the next section.

We discussed a divide-and-conquer technique for the symmetric tridiagonal eigenproblem in Section 5.2. A Sturm sequence approach using multisectioning has also been proposed (Barlow *et al.* 1983). Jacobi methods have made a comeback because of the ease with which [n/2] sets of transformations can be performed in parallel (for example, Sameh 1971, Modi and Pryce 1985, and Karp and Greenstadt 1986). Methods which only use the matrix in forming matrix-vector products, such as simultaneous iteration or Lanczos methods, can exploit any parallelism both in the matrix-vector product and in the solution of any small auxiliary eigensystem. Although such methods are particularly useful for large sparse systems (see, for example, Parlett 1980 and Cullum and Willoughby 1985), they have recently been recommended for solving large dense eigenproblems when only a few eigenpairs are required (Grimes *et al.* 1985).

One research area in which many algorithms and much theory have been developed is the implementation of numerical linear algebra algorithms on systolic architectures (see, for example, Robert and Tchuente 1982, and Bentley and Kung 1983). Much work has been done on algorithms for eigenvalues and the singular value decomposition, in addition to linear equations (see, for example, Heller and Ipsen 1983, Kung 1984, Brent and Luk 1985, and Brent *et al.* 1985). We have not, however, discussed this work here, since we are unaware of any commercially available systolic architecture computer and a main theme of our presentation is the influence of readily available machines. The wavefront method mentioned in Section 5.3 is, however, suitable for systolic architectures.

7 Influence on partial differential equations

An obvious way of exploiting parallelism in the solution of partial differential equations is to develop methods for the parallel solution of the resulting discretized systems. Most iterative methods, for example conjugate gradients, require one or more matrix-vector multiplications, some scalar products, and some vector additions at each iteration. Although it is not trivial (see, for example, the discussion of a parallel version of conjugate gradients by Meurant 1985 or Saad and Schultz 1985a), it is usually possible to obtain high levels of vectorization or parallelism. For fast convergence, some form of preconditioning is normally required and it is usually this part of the algorithm where parallel implementation is difficult. We discussed the use of a truncated Neumann series by Van der Vorst (1982, 1986) in Section 5.6, and more recently Meurant (1985) and Seager (1986) have discussed more general parallel preconditionings.

We considered block and point SOR techniques in Section 5.3. Here parallelism can be obtained through multicolourings of the underlying grid (Adams and Jordan 1984, Adams 1985, Schreiber and Tang 1982). Other splittings for exploitation of parallelism are possible, for example the QIF method of Evans and Sojoodi–Haghighi (1982) or the multisplitting techniques of O'Leary and White (1983). Chaotic relaxation (Chazan and Miranker 1969) has also been tried on parallel architectures (Hiromoto, private communication 1985). Spectral methods can make ready use of any efficient implementation of FFTs, and efficient use of both vector and parallel architectures is possible (Ortega and Voigt 1985). Indeed the central role of the FFT algorithm is clear from the fact that 111 references in the 1983 Bochum bibliography were to methods for implementing vectorized or parallel version of FFTs (Swarztrauber 1986). FACR techniques (Hockney and Jesshope 1981) also benefit from the parallel implementation of FFTs.

Multigrid techniques have also been examined with a view to designing and implementing algorithms on parallel architectures. Many approaches have been suggested including those of Gannon and van Rosendale (1982) and Greenbaum (1985).

ADI methods have a quite natural form of parallelism when used in the solution of either time-dependent problems or elliptic problems. A technique for time-dependent problems called windowing, akin to the wavefront method of Section 5.3, has been proposed for parallel implementation (Saltz and Naik 1985). Here simultaneous computation is performed on different time steps.

Clearly finite-element calculations admit ready parallelism both in independent computation within each element (for example, Adams and Voigt 1984) and in the assembly process (Berger *et al.* 1985). Domain decomposition techniques are becoming increasingly more refined and give immediate parallelism for either finite-difference or finite-element discretizations (see, for example, Glowinski *et al.* 1982 and Keyes and Gropp 1985).

8 Influence on optimization

A central problem in numerical optimization is often the solution of a set of linear equations. We discussed it in Section 6 and so will here consider other ways in which optimization techniques can exploit parallelism.

An obvious area where divide-and-conquer strategies can be used to good effect is that of global optimization (Schnabel, private communication 1985). Other allied problems are obtaining feasible points, and nonlinear minimax problems (Schnabel 1984 and Lootsma 1984).

Two ways in which many optimization techniques can benefit from parallelism are within the function evaluations themselves (for example, these might involve the solution of partial differential equations), and in the parallel evaluation of difference approximations to gradients, where, for example, the necessary function evaluations could be computed on separate processors and gains could be achieved if these evaluations were expensive (thus keeping the granularity large). Such parallelism could also be exploited in the sparse case although there many less evaluations would be required.

Recently, methods based on holding and updating a set of conjugate directions (Powell 1964) have been proposed in a parallel context (Han 1986) where independent line searches along each direction can be conducted in parallel and can be used effectively to compute a new estimate of the solution. Indeed, a line search itself can be implemented using parallel multisection techniques although there are few instances when sufficient accuracy is required to merit such a thorough search.

A natural candidate for parallel exploitation is that of partial separability (Griewank and Toint 1984). Obviously, full separability is embarrassingly parallel and the more separable a problem is, the more amenable it is to parallel techniques.

Dixon (1985) discusses the use of optimization techniques in the solution of partial differential equations. He proposes exploiting the parallel nature of finite elements so that the parallelism then lies with the differential equation rather than the optimization technique. It is an example of parallel function evaluation.

Often the minimum values of several functions

$$F(\mathbf{x}, p)$$

are required for a range of the parameter p. Sometimes individual problems in this parametric family can be solved independently; at other times the dependence may not be total. In both cases, parallelism can be exploited.

Finally, there are many problems in combinatorial optimization which are amenable to parallelism, many of them using branch-and-bound techniques, for example the travelling-salesman problem, and the knapsack problem. A good review of this area is given by Kindervater and Lenstra (1985) so we will not discuss it further here.

9 Conclusions

We hope that we have illustrated the wide and strong influence that parallel computers are having on numerical analysis. We have chosen some particular algorithmic paradigms and particular areas in numerical analysis but make no claims that these are exhaustive. Indeed parallel algorithms have been proposed for quadrature, the solution of ordinary differential equations, and approximation problems, although their influence has been much less strong than in the areas discussed in Sections 6 to 8.

When polling several colleagues for suggestions on the theme of this paper, more than one suggested that by far the most important influence of parallelism on numerical analysis is that it is difficult to get funding without mentioning it in a proposal. The effect of this is somewhat two-edged. It does indeed mean that there is an increased amount of research on parallel algorithms and some of it is good, but it also means that much inferior work is supported and many poor reports distributed. Such is the danger of the bandwagon.

It seems appropriate, when discussing the current state of the art, to stick one's neck out and make some comments on likely future trends. A major problem with the flurry of new and perhaps poorly-tested ideas is that it soon becomes very difficult to see the wood for the trees. Few comparisons have been conducted between competing methods. For example, Karp and Greenstadt (1986) have suggested that even well implemented versions of parallel Jacobi methods will be inferior to an optimized **QR** algorithm on almost all architectures, even for near-diagonal matrices. This somewhat contentious claim warrants further investigation. Indeed I suspect that there will be much more work in the future aimed at weeding out the less successful suggestions for parallel algorithms (on any architecture) in a similar way to the demise of recursive doubling on vector machines. Further consolidation should result in the equivalence or near equivalence of suggested approaches, such as the **WZ** algorithm and 2×2 block elimination. Another exciting trend could well arise from the development of methods like that of cellular automata for solving partial differential equations that we mentioned in Section 5.7. Here essentially non-numerical algorithms can be used to solve numerical problems. Additionally these methods require high degrees of parallelism for efficient implementation.

Finally, perhaps the real crunch question will be answered in the next ten years. That is, can high levels of architectural parallelism be used in the solution of real problems. We live in hope.

Acknowledgements

I would like to thank the people who responded to a request of mine for information on parallelism in other areas than linear algebra and to thank my colleagues and visitors to Harwell, John Reid, Nick Gould, and Ilse Ipsen for their comments on an early draft. The first draft of this paper was written while the author was visiting INRIA at Rocquencourt near Paris, and I am grateful to Alain Lichnewsky and Francois Thomasset and the CAPRAN project for their support and comments. I am also grateful to Mike Powell, the editor of the State-of-the-Art Proceedings, for his many detailed comments.

References

Adams, L. (1983). An M–step preconditioned conjugate gradient method for parallel computation. Proceedings International Conference on Parallel Processing, Bellaire, Michigan, August 1983. *IEEE Computer Society Press,* 36–43.

Adams, L. (1985). Reordering computations for parallel execution. Report 85–35, ICASE, NASA Langley, Hampton, Virginia.

Adams, L. M. and Jordan, H. F. (1984). Is SOR color-blind? *SIAM J. Sci. Stat. Comput.* **7**, 490–506.

Adams, L. and Voigt, R. G. (1984). A methodology for exploiting parallelism in the finite element process. In *High-speed Computation,* J.S. Kowalik (ed.), Springer–Verlag, 373–392.

Amdahl, G. (1967). The validity of the single processor approach to achieving large scale computing capabilities. AFIPS Conf. Proc. **30**, 483–485.

Barlow, R. H., Evans, D. J., and Shanechi, J. (1983). Parallel multisection applied to the eigenvalue problem. *Computer J.* **26**, 6–9.

Baudet, G. M. (1978). The design and analysis of algorithms for asynchronous multiprocessors. Report CMU-CS–78–116, PhD Thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Benner, R. E. (1986). Shared memory, cache, and frontwidth considerations in multifrontal algorithm development. Report SAND85–2752, Fluid and Thermal Sciences Department, Sandia National Laboratories, Albuquerque, New Mexico.

Bentley, J. L. and Kung, H. T. (1983). An introduction to systolic algorithms and architectures. *Naval Research Reviews* **35** (2), 3–16.

Berger, P., Dayde, M., and Fraboul, C. (1985). Experience in parallelizing numerical algorithms for MIMD architectures use of asynchronous methods. *La Recherche Aerospatiale* **5**, 325–340.

Bernstein, H. J. and Goldstein, M. (1986). Parallel implementation of bisection for the calculation of eigenvalues of tridiagonal matrices. *Computing* **37**, 85–91.

Bernutat–Buchmann, U., Rudolph, D., and Schlosser, K.–H. (1983). Parallel Computing I, Eine Bibliographie, Bochumer Schriften zur Parallelen Datenverarbeitung 1, Second Edition. Computing Centre, Bochum.

Bouknight, W., Denenberg, S., McIntyre, D., Randall, J., Sameh, A., and Slotnick, D. (1972). The ILLIAC IV System. *Proc. IEEE* **60**, 369–379.

Bowgen, G. S., Hunt, D. J., and Liddell, H. M. (1984). The solution of n linear equations on a p processor parallel computer. Report 2.30, DAP Support Unit, Queen Mary College, London.

Brent, R. P. and Luk, F. (1985). The solution of singular value and symmetric eigenproblems on multiprocessor arrays. *SIAM J. Sci. Stat. Comput.* **6**, 69–84.

Brent, R. P., Gustavson, F. G., and Yun, D. Y. Y. (1980). Fast solution of Toeplitz systems of equations and computation of Padé approximants. *J. Algorithms* **1**, 259–295.

Brent, R., Luk, F., and Van Loan, C. (1985). Computation of the singular value decomposition using mesh connected processors. *J. VLSI and Computer Systems* **1**, 242–270.

Bunch, J. R., Nielsen, C. P., and Sorensen, D. C. (1978). Rank-one modification of the symmetric eigenproblem. *Numerische Math.* **31**, 31–48.

Buneman, O. (1969). A compact non-iterative Poisson solver. Report 294, Institute for Plasma Research, Stanford University, California.

Butel, R. (1985). Conflicts between 2 vector transfers in CRAY-XMP computers. Report 418, Institut National de Recherche en Informatique et en Automatique, Rocquencourt, France.

Buzbee, B. L. (1984). Plasma simulation and fusion calculation. In *High-speed Computation,* J.S. Kowalik (ed.), Springer–Verlag, 417–423.

Chamberlain, R. M. and Powell M. J. D. (1986). **QR** factorization for linear least squares problems on the hypercube. Report CCS 86/10, Department of Science and Technology, Chr. Michelsen Institute, Fantoft–Bergen, Norway.

Chan, T. F. and Resasco, D. C. (1985). A domain-decomposed fast Poisson solver on a rectangle. Report YALEU/DCS/RR–409, Department of Computer Science, Yale University, Connecticut.

Chazan, D. and Miranker, W. (1969). Chaotic relaxation. *Linear Alg. and its Applics.* **2**, 199–222.

Concus, P., Golub, G. H., and Meurant, G. (1985). Block preconditioning for the conjugate gradient method. *SIAM J. Sci. Stat. Comput.* **6**, 220–252.

Cox, M. G. (1987). Data approximation by splines in one and two independent variables. In *The state of the art in numerical analysis,* A. Iserles and M.J.D. Powell (eds.), Oxford University Press, London.

Csanky, L. (1976). Fast parallel matrix inversion algorithms. *SIAM J. Computing* **5**, 618–623.

Cullum, J. K. and Willoughby, R. A. (1985). *Lanczos Algorithms for Large Symmetric Eigenvalue Computations. Vol. 1. Theory. Vol. 2. Programs.* Birkhäuser Verlag, Basel and Stuttgart.

Cuppen, J. J. M. (1981). A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerische Math.* **36**, 318–340.

Dave, A. K. and Duff, I. S. (1986). Sparse matrix calculations on the CRAY–2. Report CSS 197, Computer Science and Systems Division, AERE Harwell. In Proceedings International Conference on Vector and Parallel Computing, Loen, Norway, June 2–6, 1986. *Parallel Computing* (To appear).

Dixon, L. C. W. (1985). An introduction to parallel computers and parallel computing. Report 166, Numerical Optimisation Centre, The Hatfield Polytechnic, England.

Dongarra, J. J. and Duff, I. S. (1985). Advanced computer architectures. Report TM 57, Mathematics and Computer Science Division, Argonne National Laboratory.

Dongarra, J. J. and Eisenstat, S. C. (1984). Squeezing the most out of an algorithm in CRAY Fortran. *ACM Trans. Math. Softw.* **10**, 221–230.

Dongarra, J. J. and Johnsson, S. L. (1986). Solving banded matrices in parallel. In Proceedings International Conference on Vector and Parallel Computing, Loen, Norway, June 2–6, 1986. *Parallel Computing* (To appear).

Dongarra, J. J. and Sorensen, D. C. (1986). A fully parallel algorithm for the symmetric eigenvalue problem. Report TM 62, Mathematics and Computer Science Division, Argonne National Laboratory.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. (1984). A proposal for an extended set of Fortran Basic Linear Algebra Subprograms. Report TM 41, Mathematics and Computer Science Division, Argonne National Laboratory.

Dongarra, J. J., Kaufman, L., and Hammarling, S. (1986a). Squeezing the most out of eigenvalue solvers on high-performance computers. *Linear Alg. and its Applics.* **77**, 113–136.

Dongarra, J. J., Sameh, A. H., and Sorensen, D. C. (1986b). Implementation of some concurrent algorithms for matrix factorization. *Parallel Computing* **3**, 25–34.

Duff, I. S. (1983). Enhancements to the MA32 package for solving sparse unsymmetric equations. AERE R11009, HMSO, London.

Duff, I. S. (1986). Parallel implementation of multifrontal schemes. *Parallel Computing* **3**, 193–204.

Duff, I. S. and Johnsson, S. L. (1986). Node orderings and concurrency in sparse problems: an experimental investigation. Proceedings International Conference on Vector and Parallel Computing, Loen, Norway, June 2–6, 1986. Harwell Report. (To appear).

Duff, I. S. and Reid, J. K. (1982). Experience of sparse matrix codes on the CRAY–1. *Computer Physics Communications* **26**, 293–302.

Duff, I. S., Erisman, A. M., and Reid, J. K. (1986). *Direct Methods for Sparse Matrices.* Oxford University Press, London.

Erhel, J., Lichnewsky, A., and Thomasset, F. (1985). Vectorizing finite element methods. Report 383, Institut National de Recherche en Informatique et en Automatique, Rocquencourt, France.

Evans, D. J. and Abdullah, A. R. B. (1983). A new explicit method for the diffusion equation. In *Numerical Methods in Thermal Problems. Volume III,* R.W. Lewis, J.A. Johnson, and W.R. Smith (eds.), Pineridge Press Limited, Swansea, 330–347.

Evans, D. J. and Hatzopoulos, M. (1985). A parallel linear system solver. *Int. J. Comp. and Maths.* **7**, 227–238.

Evans, D. J. and Mai, S.–W. (1985). Two parallel algorithms for the convex hull problem in a two dimensional space. *Parallel Computing* **2**, 313–326.

Evans, D. J. and Sojoodi–Haghighi, R. (1982). Parallel iterative methods for solving linear equations. *Int. J. Comp. and Maths.* **11**, 247–284.

Flynn, M. J. (1966). Very high speed computing systems. *Proc. IEEE* **14**, 1901–1909.

Frisch, U., Hasslacher, B., and Pomeau, Y. (1986a). Lattice-gas automata for the Navier Stokes equation. *Phys. Review Letters* **56**, N14.

Frisch, U., d'Humieres, D., and Lallemand, P. (1986b). Lattice-gas models for 3–D hydrodynamics. *Europhysics Letter* (To appear).

Gannon, D. and van Rosendale, J. (1982). Highly parallel multigrid solvers for elliptic PDEs : an experimental analysis. ICASE, NASA Langley, Hampton, Virginia.

Geist, G. A. (1985). Efficient parallel **LU** factorization with pivoting on a hypercube processor. Report ORNL–6211, Engineering Physics and Mathematics Division, Oak Ridge National Laboratory, Tennessee.

Geist, G. A. and Heath, M. T. (1985). Parallel Cholesky factorization on a hypercube multiprocessor. Report ORNL–6190, Engineering Physics and Mathematics Division, Oak Ridge National Laboratory, Tennessee.

Gelenbe, E., Lichnewsky, A., and Staphylopatis, A. (1982). Experience with the parallel solution of PDEs on a distributed system. *IEEE Trans. Comput.* C–**31**, 1157–1164.

Gentleman, W. M. (1976). Row elimination for solving sparse linear systems and least squares problems. In *Numerical Analysis, Dundee 1975,* G.A. Watson (ed.), Springer–Verlag; 122–133.

Gentleman, W. M. (1978). Some complexity results for matrix computation on parallel processors. *J. ACM* **25**, 112–115.

George, A. (1973). Nested dissection of a regular finite-element mesh. *SIAM J. Numer. Anal.* **10**, 345–363.

George, A., Heath, M. T., and Liu, J. (1986a). Parallel Cholesky factorization on a shared-memory multiprocessor. *Linear Alg. and its Applics.* **77**, 165–187.

George, A., Heath, M., Liu, J. and Ng, E. (1986b). Sparse Cholesky factorization on a local-memory multiprocessor. Report CS–86–01. Department of Computer Science, York University, Ontario, Canada.

Glowinski, R., Periaux, J., and Dinh, Q. V. (1982). Domain decomposition methods for nonlinear problems in fluid dynamics. Report 147, Institut National de Recherche en Informatique et en Automatique, Rocquencourt, France.

Greenbaum, A. (1985). A multigrid method for multiprocessors. Report UCRL–92211, Lawrence Livermore National Laboratory, Livermore, California.

Griewank, A. and Toint, Ph. L. (1984). Numerical experiments with partially separable optimization problems. In *Numerical Analysis. Proceedings, Dundee 1983,* D.F. Griffiths (ed.), Springer–Verlag, 203–220.

Grimes, R., Krakauer, H., Lewis, J., Simon, H., and Wei, S. H. (1985). The solution of large dense generalized eigenvalue problems on the CRAY X-MP/24 with SSD. Report ETA-TR–32, ETA Division, Boeing Computer Services, Seattle, Washington.

Han, S-P. (1986). Optimization by updated conjugate subspaces. In *Numerical Analysis 1985,* D.F. Griffiths and G.A. Watson (eds.), Pitman, London.

Heath, M. T. and Sorensen, D. C. (1986). A pipelined Givens method for computing the **QR** factorization of a sparse matrix. *Linear Alg. and its Applics.* **77**, 189–203.

Heller, D. (1978). A survey of parallel algorithms in numerical linear algebra. *SIAM Review* **20**, 740–777.

Heller, D. and Ipsen, I. C. F. (1983). Systolic networks for orthogonal decompositions. *SIAM J. Sci. Stat. Comput.* **4**, 261–269.

Hockney, R. W. and Jesshope, C. R. (1981). *Parallel Computers.* Adam Hilger Ltd., Bristol.

Ipsen, I. C. F. and Jessup, E. (1986). Solving the symmetric tridiagonal eigenvalue problem on the hypercube. Department of Computer Science, Yale University, Connecticut. (To appear).

Ipsen, I. C. F. and Saad, Y. (1985). The impact of parallel architectures on the solution of the eigenvalue problem. Report YALEU/DCS/RR–444, Department of Computer Science, Yale University, Connecticut.

Iqbal, M. A., Saltz, J. H., and Bokhari, S. H. (1986). Performance tradeoffs in static and dynamic load balancing strategies. Report 86–13, ICASE, NASA Langley, Hampton, Virginia.

Jacobs, D. A. H. (Ed.) (1977). *The State of the Art in Numerical Analysis.* Academic Press, New York and London.

Jalby, W. and Meier, U. (1986). Optimizing matrix operations on a parallel multiprocessor with a memory hierarchy. Report, Center for Supercomputing Research and Development, University of Illinois at Urbana–Champaign.

Jalby, W., Frailong, J-M, and Lenfant, J. (1984). Diamond schemes: an organization of parallel memories for efficient array processing. Report 342, Institut National de Recherche en Informatique et en Automatique, Rocquencourt, France.

Johnsson, S. L. (1985). Solving narrow banded systems on ensemble architectures. Report YALEU/DCS/RR–418, Department of Computer Science, Yale University, Connecticut.

Johnsson, S. L., Saad, Y., and Schultz, M. H. (1985). Alternating direction methods on multiprocessors. Report YALEU/DCS/RR–382, Department of Computer Science, Yale University, Connecticut.

Karp, A. H. (1986). A parallel processing challenge. *IMANA Newsletter,* Institute of Mathematics and its Applications, Southend-on–Sea. **10** (2), 25–26.

Karp, A. H., and Greenstadt, J. (1986). An improved parallel Jacobi method for diagonalizing a symmetric matrix. Report G320–3484, IBM Scientific Center, Palo Alto.

Keyes, D. E. and Gropp, W. D. (1985). A comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation. Report YALEU/DCS/RR–448, Department of Computer Science, Yale University, Connecticut.

Kindervater, G. A. P. and Lenstra, J. K. (1985). An introduction to parallelism in combinatorial optimization. In *Parallel Computers and Computations,* J. van Leeuwen and J.K. Lenstra (eds.), CWI Syllabus 9, Department of Numerical Mathematics, Centre for Mathematics and Computer Science, Amsterdam, 163–184.

Kowalik, J. S. (Ed.) (1985). *Parallel MIMD Computation: HEP Supercomputer and its Applications.* MIT Press, Cambridge, Mass.

Kronsjö, L. (1985). *Computational Complexity of Sequential and Parallel Algorithms.* John Wiley & Sons, Ltd., Chichester, New York, Brisbane, and Toronto.

Kuck, D. (1978). *The Structure of Computers and Computation.* John Wiley & Sons, Ltd., Chichester, New York, Brisbane, and Toronto.

Kung, H. T. (1980). The structure of parallel algorithms. In *Advances in Computers. Volume 19,* M. Yovitts (ed.), Academic Press, 65–112.

Kung, S.-Y., Arun, K., Bhuskerio, D., and Ho, Y. (1981). A matrix data flow language/architecture for parallel matrix operations based on computational wave concept. In *VLSI Systems and Computations,* H.T. Kung, R. Sproull, and G. Steele (eds.), Computer Science Press, Rockville, Maryland.

Kung, S.-Y. (1984). On supercomputing with systolic/wavefront processors. *Proc. IEEE* **72**, 867–884.

Lambiotte, J. J. and Voigt, R. G. (1975). The solution of tridiagonal linear systems on the CDC STAR–100 computer. *ACM Trans. Math. Softw.* **1**, 308–329.

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

Lichnewsky, A. (1982). Sur la résolution de systèmes linéaires issus de la méthode des éléments finis par une machine multiprocesseurs. Report 119, Institut National de Recherche en Informatique et en Automatique, Rocquencourt, France.

Lichnewsky, A. (1984). Some vector and parallel implementations for preconditioned conjugate gradient algorithms. In *High-speed Computation,* J.S. Kowalik (ed.), Springer–Verlag, 343–359.

Liu, J. W. H. (1985). Computational models and task scheduling for parallel sparse Cholesky factorization. Report CS–85–01. Department of Computer Science, York University, Ontario, Canada.

Lootsma, F. A. (1984). Parallel unconstrained optimization methods. Report 84–30, Department of Mathematics and Informatics, Delft University of Technology, Delft, The Netherlands.

Louter–Nool, M. (1985). BLAS on the CYBER 205. Report NM-R8524, Department of Numerical Mathematics, Centre for Mathematics and Computer Science, Amsterdam.

McBryan, O. A. and Van de Velde, E. F. (1986). Hypercube algorithms and implementations. Report DOE/ER/03077–271, Courant Mathematics and Computing Laboratory, NYU, New York.

Miellou, J. C. (1975). Algorithmes de relaxation chaotique à retards. *Revue d'Automatique Informatique et Recherche Operationnelle* **9**, 55–82.

Meurant, G. (1985). Multitasking the conjugate gradient on the CRAY X-MP/48. Report NA–85–33, Department of Computer Science, Stanford University, Stanford, California.

Miranker, W. L. and Liniger, W. M. (1967). Parallel methods for the numerical integration of ODEs. *Math. Comp.* **21**, 303–320.

Modi, J. J. and Clarke, M. R. B. (1984). An alternative Given's ordering. *Numerische Math.* **43**, 83–90.

Modi, J. J. and Pryce J. D. (1985). Efficient implementation of Jacobi's diagonalisation method on the DAP. *Numerische Math.* **46**, 443–454.

Montry, G. R. and Benner, R. E. (1985). The effects of cacheing on multitasking efficiency and programming strategy on an ELXSI 6400. Report SAND85–2728, Fluid and Thermal Sciences Department, Sandia National Laboratories, Albuquerque, New Mexico.

Morf, M. (1980). Doubling algorithms for Toeplitz and related equations. International Conference on Acoustics, Speech, and Signal Processing, Denver, April 1980. *IEEE Press.* 954–959.

Noor, A. K., Kamel, H. A., and Fulton, R. E. (1977). Substructuring techniques – status and projections. *Computers and Structures* **8**, 621–632.

O'Leary, D. P. (1984). Ordering schemes for parallel processing of certain mesh problems. *SIAM J. Sci. Stat. Comput.* **5**, 620–632.

O'Leary, D. P., and Stewart, G. W. (1985). Data-flow algorithms for parallel matrix computations. *Communications ACM* **28**, 620–632.

O'Leary, D. P. and White, R. E. (1983). Multi-splittings of matrices and parallel solution of linear systems. Report 1362, Computer Science Center, University of Maryland, Maryland.

Ortega, J. M. and Voigt, R. G. (1985). Solution of partial differential equations on vector and parallel computers. *SIAM Review* **27**, 149–240.

Parlett, B. N. (1980). *The Symmetric Eigenvalue Problem.* Prentice–Hall, New Jersey.

Parlett, B. N. and Schreiber, R. (1986). Block **QR** for systolic architectures. Technical Report. SAXPY Computer Corporation, Sunnyvale, California.

Powell, M. J. D. (1964). An efficient method for finding the minimum of a function of several variables without calculating derivatives. *Computer J.* **7**, 155–162.

Preparata, F. P. and Vuillemin, J. (1981). The cube-connected cycles: a versatile network for parallel computation. *Communications ACM* **24**, 300–309.

Rizzi, A. (1985). Vector coding the finite-volume procedure for the CYBER 205. *Parallel Computing* **2**, 295–312.

Robert,Y. and Tchuente, M. (1982). Calcul en parallele sur des reseaux systoliques. In *Actes Colloque AFCET-GAMNI-ISINA,* A. Bossavit (ed.), Serie C, n^0 1, Bulletin de la Direction des Etudes et Recherches, EDF, France, 125–128.

Roucairol, C. (1986). A study of parallel branch and bound algorithms for the travelling salesman problem. 8th European Conference on Operational Research, September 1986, Lisbon. (To appear).

Saad, Y. and Schultz, M. H. (1985a). Parallel implementations of preconditioned conjugate gradient methods. Report YALEU/DCS/RR–425, Department of Computer Science, Yale University, Connecticut.

Saad, Y. and Schultz, M. H. (1985b). Parallel direct methods for solving banded linear systems. Report YALEU/DCS/RR–387, Department of Computer Science, Yale University, Connecticut.

Saad, Y. and Schultz, M. H. (1985c). Data communication in hypercubes. Report YALEU/DCS/RR–428, Department of Computer Science, Yale University, Connecticut.

Saltz, J. H. and Naik, V. K. (1985). Towards developing robust algorithms for solving partial differential equations on MIMD machines. Report 85–39, ICASE, NASA Langley, Hampton, Virginia.

Saltz, J. H., Naik, V. K., and Nicol, D. M. (1986). Reduction of the effects of the communication delays in scientific algorithms on message passing MIMD architectures. Report 86–4, ICASE, NASA Langley, Hampton, Virginia.

Sameh, A. H. (1971). On Jacobi and Jacobi-like algorithms for a parallel machine. *Math. Comp.* **25**, 579–590.

Sameh, A. H. (1977). Numerical parallel algorithms – a survey. In *High Speed Computer and Algorithm Organization,* D.J. Kuck, D. Lawrie, and A.H. Sameh (eds.), Academic Press, 207–228.

Sameh, A. H. (1983). An overview of parallel algorithms in numerical linear algebra. Bulletin de la Direction des Etudes et Recherches, EDF, France. Serie C., 129–134.

Sameh, A. H. (1985). On some parallel algorithms on a ring of processors. *Computer Physics Communications* **37**, 159–166.

Sameh, A. H. and Kuck, D. J. (1978). On stable parallel linear system solvers. *J. ACM* **25**, 81–91.

Saul'yev, V. K. (1964). *Integration of equations of parabolic type by the method of nets.* Macmillan, New York.

Schendel, U. (1984). *Introduction to Numerical Methods for Parallel Computers.* Ellis Horwood Ltd., Chichester.

Schnabel, R. B. (1984). Parallel computing in optimization. Report CU-CS–282–84, Department of Computer Science, University of Colorado, Boulder, Colorado.

Schreiber, R. and Tang, W-P. (1982). Vectorizing the conjugate gradient method. Unpublished document. Department of Computer Science, Stanford University, Stanford, California.

Schwartz, J. (1980). Ultracomputers. *ACM Trans. Program. Lang. Syst.* **2**, 484–521.

Seager, M. K. (1986). Parallelizing conjugate gradient for the CRAY X-MP. *Parallel Computing* **3**, 35–47.

Sorensen, D. C. (1983). Analysis of pairwise pivoting in Gaussian elimination. *IEEE Trans. Comput.* C–**34**, 274–278.

Stone, H. S. (1973). An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *J. ACM* **20**, 27–38.

Swarztrauber, P. N. (1979). A parallel algorithm for solving general tridiagonal equations. *Math. Comp.* **33**, 185–199.

Swarztrauber, P. N. (1986). Vector-concurrent FFT's. Invited talk at International Conference on Vector and Parallel Computing, Loen, Norway. *Parallel Computing* (To appear).

te Riele, H. J. J. (1985). Applications of supercomputers in mathematics. Report NM-N8502, Department of Numerical Mathematics, Centre for Mathematics and Computer Science, Amsterdam.

Traub, J. F. (Ed.) (1973). *Complexity of Sequential and Parallel Numerical Algorithms.* Academic Press, New York and London.

Tseng, S. S. and Lee, R. C. T. (1984). A new parallel sorting algorithm based upon min-mid-max operations. *BIT* **24**, 187–195.

Van der Vorst, H. A. (1982). A vectorizable variant of some ICCG methods. *SIAM J. Sci. Stat. Comput.* **3**, 350–356.

Van der Vorst, H. A. (1986). The performance of FORTRAN implementations for preconditioned conjugate gradients on vector computers. *Parallel Computing* **3**, 49–58.

Van Leeuwen, J. (1985). Parallel computers and algorithms. In *Parallel Computers and Computations,* J. van Leeuwen and J.K. Lenstra (eds.), CWI Syllabus 9, Department of Numerical Mathematics, Centre for Mathematics and Computer Science, Amsterdam, 1–32.

Voigt, R. G. (1985). Where are the parallel algorithms? Report 85–2, ICASE, NASA Langley, Hampton, Virginia. Proceedings of the 1985 National Computer Conference. **54**, 329–334. AFIPS Press.

Wang, H. H. (1981). A parallel method for tridiagonal equations. *ACM Trans. Math. Softw.* 7, 170–183.

Ware, W. (1973). The ultimate computer. *IEEE Spectrum* March, 89–91.

Benner, R. E. (1986). Shared memory, cache, and frontwidth considerations in multifrontal algorithm development. Report SAND85–2752, Fluid and Thermal Sciences Department, Sandia National Laboratories, Albuquerque, New Mexico.

Berger, P., Dayde, M., and Fraboul, C. (1985). Experience in parallelizing numerical algorithms for MIMD architectures use of asynchronous methods. *La Recherche Aerospatiale* **5**, 325–340.

Duff, I. S. and Johnsson, S. L. (1986). Node orderings and concurrency in sparse problems: an experimental investigation. Proceedings International Conference on Vector and Parallel Computing, Loen, Norway, June 2–6, 1986. Harwell Report. (To appear).

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique