Abstract: In this paper we present a parallel implementation of Arnoldi's subspace method on the Connection Machine. With a 16K-processor CM2, we obtained performances of a few hundred Megaflops for a matrix size of several thousands when computing a small number of eigenvalues and eigenvectors. The extrapolated performance on a 64K-processor CM2 indicates that the asymptotic speed will be greater than 1 Gigaflop for very large matrices. We show that it is possible to use the subspace method with a good throughput or speed-up on massively-parallel architectures like the CM2. We remark that other classical methods for linear algebra problems such as, for example, back substitution and the QR method, cannot exploit all the potential power of massively-parallel machines.

Next, we propose using the subspace method as a programming methodology for massively-parallel machines in order to obtain a good performance when solving some large linear algebra problems, especially eigenproblems. This method is the most frequently one used for very large eigenproblems and it is also well-adapted to massively-parallel architectures. The choice of the subspace size is very important both for numerical stability and speed-up. We conclude with a discussion on the effects of the orders chosen for the subspaces on performance.

Keywords: *Linear algebra, non-Hermitian eigenproblems, matrix computation, iterative subspace methods, massively-parallel algorithms.*

Yale University
Department of Computer Science

Parallel Subspace Method for non-Hermitian Eigenproblems on the Connection Machine (CM2)

Serge G. Petiton

YALEU/DCS/RR-859

August 1991

To appear in the Applied Numerical Mathematics Journal, North Holland

This work has been supported in part by U.S. Office Naval of Research contract N00014-89-J-1906 and in part by an INRIA postdoctoral fellowship.

Parallel Subspace Method for non-Hermitian Eigenproblems on the Connection Machine (CM2)

Serge G. Petiton*

1 Introduction. In the past few years many parallel architectures have been introduced. ¿From MIMD architectures, with loosely or tightly coupled memories, to VLIW machines, the current range of supercomputers is very large. Computers based on the hypercube or grid connection of thousands of elementary processors are now reality. Studies in massively-parallel algorithms are numerous, especially since the introduction of the Connection Machine 2 (CM2) by Thinking Machines Corporation.

During these years the research in Numerical Analysis has been oriented, in part, towards the explicit consideration of parallel architecture parameters. The memory size and the power of the supercomputers now proposed or announced make the solution of large and very large problems possible. Many of these, after discretisation, become Linear Algebra problems. The matrices are often Hermitian or positive definite. Their sizes are sometimes very large. An increasing fraction of involved problems concerns non-Hermitian matrices. Many scientific research fields now generate very large non-Hermitian eigenproblems. Quantum chemistry provides good examples of such applications, where one needs to find one or a few eigenvectors of non-Hermitian matrices with dimensions often larger than one thousand. Semi-conductor or composite material analyses are also important applications that generate these particular problems. See [21] for an interesting example of such applications.

In this article we consider the computation, on massively-parallel architectures, of r eigenvalues of a large non-Hermitian matrix A of size n, where n is very large compared to r. The iterative subspace methods are well-adapted to this kind of problem. They are based on the convergence of the Krylov subspace of order m towards the dominant invariant subspace of order m, where the subspace order m is a few times larger than r. Though many techniques, such as reorthogonalization, can increase the numerical stability of these algorithms, the question of the stability of many of them is an open problem. If we want to compute the eigenvalues having the largest real part, the iterative method of Arnoldi is the most common and the best adapted subspace method.

This work has been supported in part by U.S. Office Naval of Research contract N00014-89-J-1906 and in part by an INRIA postdoctoral fellowship.

*Department of Computer Science, Yale University, P.O. Box 2158, Yale Station, New Haven, CT 06520, USA and Site Expérimental en Hyperparallélisme, Etablissement Technique Central de l'Armement (ETCA), 16 bis Avenue du Prieur de la Cote d'Or, 94114 Arcueil Cedex, France

In this article we present an adaptation of this method to massively-parallel architectures. We also provide an overview of the parallel QR algorithm used in this method. We consider only massively-parallel SIMD architectures and we do not study in this article the SIMD-vector-like use of this kind of supercomputers. We also assume that we do not have any front-end floating-point computations after data have been distributed among local memories and before the end of the massively-parallel computation. Experiments were principally performed on a 4K [1] and a 16K [2] Connection Machine 2 (CM2), each of them with 32 Kbytes of local memory per processor. We present results obtained using 32-bit floating-point pipeline accelerators included in the CM2. Programs are developed using the C/PARIS language [24].

2 A Subspace Method to Solve Large non-Hermitian Eigenproblems

2.1 A brief survey of the subspace method principle. We are often faced in numerical analysis with the study of general problems P on a finite n-dimensional vector space E. The goal is to compute a vector $u \in E$, $u \neq 0$, with $Fu = 0$. Thus, let this n-problem be

$$P_E \quad : \quad \mathit{Find}\ u \in E,\ u \neq 0,\ \mathit{with}\ Fu = 0.$$

The solutions computed are approximations of the exact values. Let K be a vector subspace of E with $K \subset E$. Thus for solving P_E, we can use the Rayleigh-Ritz-Galerkin approximation [2] :

$$P_K \quad : \quad \mathit{Find}\ \tilde{u} \in K,\ \tilde{u} \neq 0,\ \mathit{with}\ F\tilde{u} \in K^{\perp}.$$

When we want to compute the solution of a large eigenproblem we have $F = A - \lambda I$. Thus, we can use the generalised Lanczos process whose principle is as follows [2, 10, 20] : We orthogonally project a *problem* of order n (let E^n be the original vector space) to a Krylov subspace E^m of order m. Given an initial vector v_0, this space is spanned by $v_0, Av_0, ..., A^{m-1}v_0$: $E^m = span(v_0, Av_0, A^2v_0, ..., A^{m-1}v_0)$. We solve the projected problem in this subspace in terms of a suitable basis and then transform the solution to the bassis of the full space. The eigenvalues and eigenvectors in E^m are called the Ritz eigenvalues and the Ritz eigenvectors of A respectively [19]. Several projection algorithms and global methods can be proposed, with respect to the expected properties of the projected problems and the numerical stability criteria. Iterative versions of these methods are common : after each step, we re-start the process starting with a *new initial* vector equal to a linear combination of a subset of the approximated eigenvectors computed previously. The choice of the coefficients is almost empirically. If the approximated solution is not good enough, we iterate until we obtain satisfactory precision.

2.2 The Iterative Method of Arnoldi to Solve non-Hermitian Eigenproblems. As an example of this kind of method we study the method of Arnoldi [6, 19] on massively-parallel

[1] part of the 8K CM2 at Department of Computer Science, Yale University

[2] at Site Expérimental en Hyperparallélisme, Etablissement Technique Central de l'Armement (ETCA)

architectures. This method uses an orthogonal projection process and the projected problem has properties of interest for solving the Ritz eigenproblem.

The method of Arnoldi consists of building an orthogonal basis of the Krylov subspace spanned as described in the above section. In this basis, the restriction of A to E^m is represented by an upper Hessenberg matrix of dimension m. Then, this Krylov space converges towards the dominant spectral subspace. With m fixed, the Arnoldi projection is the iterative calculation of the vector basis $v_j, j = 1, m-1$, starting with a given initial normalised vector $v_0 \in E$, as follows:

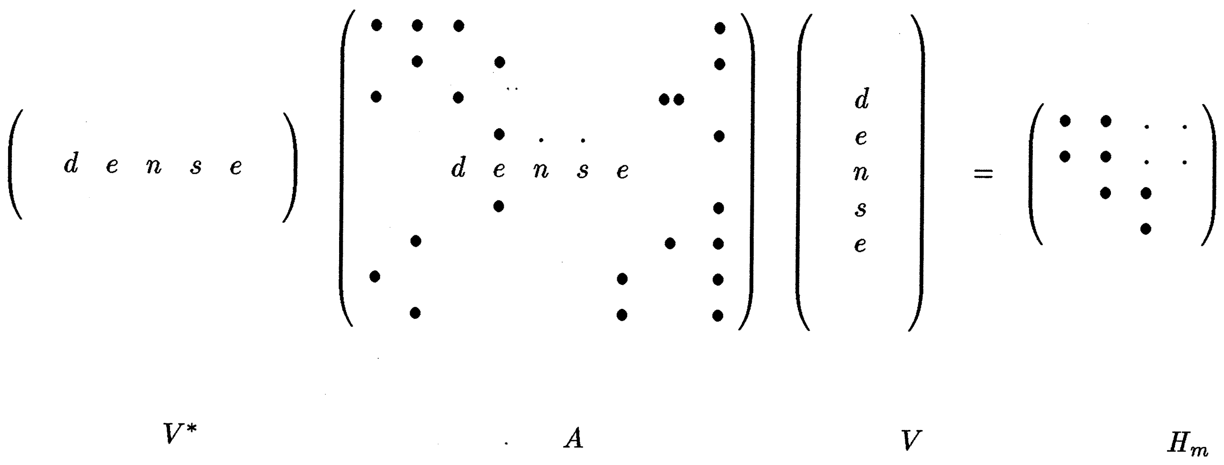
$$v_j = Av_{j-1} - \sum_{i=0}^{j-1} h_{i,j-1}v_i, \tag{1}$$

$$\text{with } h_{i,j-1} = (Av_{j-1}, v_i); i = 0, j-1, \tag{2}$$

$$h_{j,j-1} = ||v_j||, \tag{3}$$

$$v_j = v_j/||v_j||. \tag{4}$$

Hence, the orthogonal projection consists of the steps (1), (2),(3) and (4). Therefore, we need to implement the matrix-vector product and the inner product. We note that the matrix H_m is an upper Hessenberg matrix: this is an important property for the solution of the Ritz eigenproblems. The matrix-algorithm view of this projection is, with $V = (v_0, v_1, ..., v_{m-1})$:



Each iteration of the iterative method of Arnoldi can be decomposed into three parts. We study here only one iteration :

α : Projection from the n-space to an m-subspace; m is often taken equal to τr, where τ is chosen equal to 3 in several examples [19]. The projected matrix in the subspace is a Hessenberg matrix, denoted by H_m.

β : Computation in the subspace E^m. We solve the Ritz eigenproblem using the implicit QR method with Wilkinson shift and deflation. As H_m is a Hessenberg matrix, the QR method is very well-adapted to this operation. We compute the Ritz eigenvectors using the inverse iteration method, starting with the matrix H_m. In this case the initial matrix for the inverse iteration method is a Hessenberg matrix.

γ : If convergence has not occurred yet, we compute a new *initial* vector for the next iteration. Otherwise we compute the expression of the eigenvectors in the full-space E^n.

We remark that the large initial matrix A is loaded once and never modified or reloaded, if the local memories are sufficiently large. We assume that in this article.

The vectors in the sequence $v_0, v_1, ..., v_{m-1}$ introduced in Arnoldi's algorithm are orthogonal, i.e. $(v_i, v_j) = \delta_{i,j}$, with i and j between 0 to $m-1$. We do not discuss reorthogonalisation and convergence testing in this paper.

The QR method is a general method used by itself on sequential or vector machines to compute eigenproblems of moderate size (often with matrix size less than one thousand), especially when one wants to compute all the eigenvalues and eigenvectors. In this case, the original matrix is transformed to Hessenberg form in order to optimize floating-point computations [25]. In practice, the QR method is always used for a matrix in Hessenberg form.

In the next section, we first present a massively-parallel implementation of the QR method. Second, we describe a massively-parallel version of Arnoldi's projection.

3 Ritz Eigenvalue and Eigenvector Computation on Massively-Parallel Architectures

3.1 The QR Method to Compute Ritz Eigenvalues. In the subspace we implemented the QR method to find the eigenvalues of the Hessenberg matrices obtained by the projection described above. We noticed that, when H_m is the Arnoldi projected matrix, it is already in upper Hessenberg form. Then, we can implement the classical QR method, which conserves this matrix form and hence optimises the number of operations. To accelerate the convergence of the method we use a shift as described by Wilkinson [25]. Let λ_k denote the Wilkinson shift at the k^{th} iteration. Then the QR iteration k corresponds to the direct computation of :

$$H_m^k = (Q^k)^*(H_m^{k-1} - \lambda_k I)Q^k + \lambda_k I. \tag{5}$$

The optimised sequential algorithm, based on Householder orthogonal transformations, needs approximately 24 $m^2 + O(m)$ floating-point operations per iteration. The QR method gives all the eigenvalues of the complex matrix A.

3.2 A Massively-Parallel Algorithm. In previous studies the author implemented a vector version of this algorithm [15] and one solution was to use the CM2 as a SIMD-vector machine. But, to obtain maximum performance with this approach, we must map each element

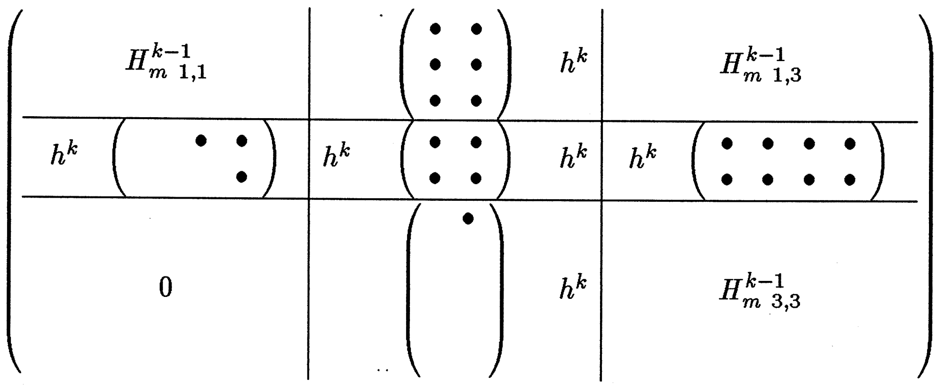


Figure 1: Elements read and updated at the k^{th} step of one QR iteration ; example : $m > 10$, $k = 4$. The sub-block matrices (1,1) and (3,3) are upper Hessenberg matrices

of one row (or column) onto one processor and we must load one column (or row) into each local memory of each processor. This is only profitable if we have very large matrices H_m, at least of order 4K (or 16K) for the 4K-processor (or 16K-processor) CM2 machine. Such a large projected matrix size means that the size of the initial problem, i.e. n, is enormous (because, by hypothesis, $n >> m$). In this situation, two problems arise. First, we may have enough local memory to store the Hessenberg matrix but we will not have enough local memory to store the matrix A. Second, if we map one row (or column) onto all the processors, we no longer have massively-parallel access to the columns (or rows) of A. But, in the QR method, we need to have fast parallel access to both the rows and the columns of the matrix A.

We conclude that a SIMD-vector programming approach is not well-adapted to this problem. If we map one element of H_m onto each virtual processor, we need to ensure only that $m^2 \geq$ 4K (or $m^2 \geq$ 16K) on a 4K (or 16K) CM2, in order to have sufficient data for all the processors. Then we can compute in parallel along both the rows and the columns of A. Let us assume, therefore, that the i^{th} element of the j^{th} column of the matrix H_m is mapped on the i^{th} processor of the j^{th} column of a $m \times m$ grid of processors.

We have described in [16] a massively-parallel implementation of this QR method starting with a Hessenberg matrix. As described in Figure 1, only two rows and two columns of the matrix are read and modified at each step of an implicit QR iteration. Let h^k be a Householder matrix of order 2 computed starting with the vector composed by the elements (k,k) and $(k+1,k)$ of H^{k-1}_m. Then on massively-parallel machines each step of one QR iteration in the subspace is composed of :

1. Wilkinson shift and scalar computation, computed in a redundant way to minimise communication overhead.
2. Preparation for the massively-parallel computation, that is, distribution of data to be used as pivots for the computation.
3. Massively-parallel computation corresponding to the innermost loops of the sequential

algorithm.

On massively-parallel machines with m^2 processors, the innermost loops of each iteration of the sequential QR algorithm are computed with only 16 massively-parallel operations. Added to the redundant scalar computations, one step of the algorithm is completed in 40 massively-parallel operations. So one QR iteration is completed in approximately $40m$ floating-point massively-parallel operations. We note therefore that the complexity of the QR is $O(m^2)$, because it takes a few iterations to compute each eigenvalue; compare this to the $O(m^3)$ complexity of this algorithm on sequential machines.

To compute the eigenvectors we implemented a SIMD massively-parallel inverse iteration method. Then we need to solve a linear system where the matrix is of Hessenberg form. The massively-parallel computation of the QR factorisation of this matrix H_m is similar to the row updates of the QR iteration implementation discussed above. The other parts of the inverse iteration, like the back substitution, are easy to implement on a massively-parallel architecture. The complexity of this method is of the same order as that for the QR method, and the execution time is approximately one-half.

We conclude that the degree of parallelism of these algorithms to compute the Ritz eigenvalues and eigenvectors (i.e., the ratio of the sequential algorithm complexity and the massively-parallel complexity) is of order $O(m)$, with m^2 physical processors. But is throughput efficient?

3.3 Experiments on the Connection Machine 2. Let *vpr* denote the virtual processor ratio: i.e., the ratio of the number of virtual processors and the number of physical processors. We presented above the mapping that we need to use and showed that only a few virtual processors are involved at each step of an iteration. In fact, to implement the QR method on the CM2 we always use fewer than $2(m-k+3)/\sqrt{vpr}$ physical processors in parallel at the fixed step k of one QR iteration, but each processor is active during at least two steps.

One step of the algorithm is completed in 40 CM floating-point operations and one QR iteration in approximately $40m$ CM floating-point operations. We present in Table 1 the performance of this Parallel QR implementation. The communication time represents 75 percent of the computation. The majority of communications are one-dimensional NEWS [3] send or spread operations. We also present performance obtained for the inverse iteration method on the CM2, with the distinction between the QR factorisation and the back substitution. When $m = 512$, there is too little local memory on a 4K processor CM2 for the algorithm to execute, and this value of m is the largest one that can be used on a 16K CM2. In this case, we have a *vpr* equal to 64 and just 128 32-bits words per virtual processor. The significant amount of redundant scalar computation in this massively-parallel implementation (and the associated temporary variables) requires more space. Furthermore, the operating system uses part of these local memories.

We observe that for $m \leq 64$ (or $m \leq 128$) on a 4K CM2 (or 16K CM2) the time is approximately

[3] North East West South fast neighbour communications on the Connection Machine 2.

	4K Processors				16 K processors			
m	Method QR 1 iteration	$H = QR$	$Rx = b$	Inverse iteration	Method QR 1 iteration	$H = QR$	$Rx = b$	Inverse iteration
4	0.05	0.014	0.010	0.035	0.034	0.011	0.09	0.029
8	0.11	0.042	0.028	0.075	0.089	0.038	0.024	0.06
16	0.24	0.096	0.049	0.16	0.18	0.081	0.041	0.12
32	0.47	0.20	0.11	0.32	0.37	0.16	0.09	0.25
64	0.97	0.42	0.21	0.64	0.70	0.35	0.18	0.51
128	3.5	1.8	0.81	2.7	1.49	0.69	0.35	1.0
256	20.7	10	4.0	14.6	6.47	3.11	1.51	6.2
512	NA	NA	NA	NA	34.8	15.4	7.3	30.7

Table 1: **QR method and inverse iteration method performance on the CM2, in seconds** (including one factorisation $H_m = QR$, H_m being an upper Hessenberg matrix, and one back substitution $Rx = b$).

doubled when m is multiplied by two. In this case we have *vpr* always equal to one, because the number of mapped elements is less than or equal to the number of physical processors. We have twice the number of steps for each iteration and each of them takes the same time. Only the number of allocated physical processors is different. For m larger, the number of elements mapped is larger than the number of physical processors. Then, when m is multiplied by two, the number of steps, for each iteration, is again doubled. But, the time consumed by each step is approximately twice as long. On each physical processor, we have four times more virtual processors, but only two times more one-dimensional communications between physical processors when we spread data along each dimension. This is because communications between virtual processors along one dimension are just internal moves when they are mapped onto the same physical processor (we always do only $\sqrt{vpr}$ communications between physical processors in this case). Since these communications are the dominant factor in the time for each step, we may expect a ratio a little larger than two in the time needed to complete each step. Then, when *vpr* is larger than one, each iteration is more than four times longer when we double the order of the matrix A, as we show in Table 1.

We also remark that when *vpr* is equal to one, $m \leq 64$, the execution time is comparable for the 4K and the 16K machines. When *vpr* is larger, with m fixed, the time is approximately shorter by one half on a 16K than on a 4K machine. Thus, we do not have a ratio of four, as expected and often claimed, when we have four times more physical processors. The total time is again driven by the one-dimensional spread communications between physical processors, which is only approximately doubled when we have four times more elements mapped.

The throughput of these methods is less than one megaflop. The degree of parallelism is of order $O(m)$ but there exists no well-adapted mapping that would use all the potential of the large number of processors. We can make the general remark that on massively-parallel architectures, some linear algebra problems, like back substitution or the QR method, do not use all the potential power of these machines. These algorithms are vectorial but we cannot map the data on a large number of processors to sufficiently exploit the potential parallelism of these methods. Notice that in practice we always use the QR method starting with Hessenberg matrices, even on sequential machines and with matrices of small size. If δ is the size of the 2D array-grid of a massively-parallel architecture like the CM2, the speed-up is often $O(\delta)$ instead of $O(\delta^2)$ when using processors with the desired 2D geometry.

We also remark that we cannot choose a privileged axis because we access elements in an equivalent way along both rows and columns. Thus we cannot define a virtual geometry over the physical processors to optimise communications along a specific axis.

We conclude that the QR method is not well-adapted to massively-parallel architectures like the CM2. Therefore, even if the other portions of Arnoldi's method are efficient on these architectures, the QR method is potentially a strong bottleneck.

4 Krylov Space's Basis Computation on Massively-Parallel Architectures

4.1 A Massively-Parallel Algorithm.

To compute the orthogonal Krylov space's basis as described in (1), (2), (3), and (4) we need to compute in sequence, $j = 0, m-1$:

α : $w = Av_{j-1}$,
β : The scalar products $(w, v_i) = h_{i,j-1}$ for $i = 0, j-1$,
γ : $s = \sum_{i=0}^{j-1} h_{i,j-1} v_i$,
δ : $u = w - s$,
ϵ : $h_{j,j-1} = \|u\|$,
ζ : $v_j = u/h_{j,j-1}$.

The sequential orthogonal projection described above needs $8n^2m + 8nm^2 + O(n)$ floating-point operations when the matrix is dense and complex.

The computation is composed primarily of matrix-vector multiplications, scalar products and vector additions. Each of these operations is well-adapted to massively-parallel machines like the CM2 when the matrices are dense. The problem is just to find a compatible mapping of the matrix A and of the vectors v_j, $j = 0, m-1$, with respect to these basic linear algebra operations. We have seen from the massively-parallel QR analysis that a large degree of parallelism is not always sufficient to obtain good performance. We also need to have well-adapted communication patterns and we have distribute the computation well, on average, over the larger number of physical processors.

A classical mapping of a square matrix A, in order to compute the matrix-vector product $w = Av_0$, is to store each element of the matrix in a 2D $n \times n$ array of virtual processors. Let $a(i,j)$ be the i^{th} element of the j^{th} column of A and $p(i,j)$ be the i^{th} processor of the j^{th} column of the 2D virtual processor grid. We map $a(i,j)$ on the virtual processor $p(i,j)$. The vector v_0 is mapped for each row of a 2D grid of processors, i.e., the element $v_0(j)$ is mapped on each virtual processor of the j^{th} column of the grid. Thus, on each processor $p(i,j)$ we have mapped $a(i,j)$ and $v_0(j)$. Then, in one massively-parallel step we calculate all the multiplications of the matrix-vector product. Let

$z(i,j) = a(i,j) * v_0(j)$ denote the partial results, then we have to compute the sum $\sum_{j=0}^{n-1}$ z(i,j), $\forall$ $j \in [0, n-1]$. For each row of the matrix, we have to add the partial results. With the mapping described above we just have to carry out a reduction-and-spread operation on each row of the 2D virtual processor grid, in parallel. Then, for each row i, $\forall i \in [0, n-1]$, on each virtual processor we calculate the i^{th} element of w. The time of this massively-parallel operation is $O(log_2(n))$. Computing w with a unique result on one fixed processor of each row takes approximately the same time as obtaining these results on each virtual processor of the rows. Then, we obtain the vector $w = Av_0$ on each column after this massively-parallel computation. Hence, the complexity of this matrix-vector product is $O(log_2(n))$.

Let us notice that w is mapped in each column, compared to the row mapping of the original v_0. Hence, to compute the scalar product (w, v_0) the two vectors concerned are not well-mapped. In virtual processor $p(i,j)$ we have $v_0(j)$ and $w(i)$ and we have to change the mapping of one of them. Hence, we have to permute the vector w on the grid of processors; i.e. we have to send data from virtual processors $p(i,j)$ to virtual processor $p(j,i)$. Then we have $w(j)$ on processor $p(i,j), \forall i \in [0, n-1]$. After this permutation, in order to obtain the scalar product we can apply just a massively-parallel multiplication and a reduction with addition. Hence, all the other computations of the first iteration (steps γ, δ, ϵ and ζ of the algorithm presented above) are massively-parallel computations between elements stored in the same local memory (except for the normalisation of v_1, where a scalar product is required).

We use the results v_1 (or v_j) to compute Av_1 (or Av_j). Thus, the mapping of A and v_1 (or v_j) is coherent only if we have chosen to permute the vector w in the previous iteration, i.e., if v_0 and v_1 have a row mapping (or, if the v_j are row mapped $\forall j \in [0, m-1]$). Note that this permutation is important and necessary on these massively-parallel machines. We have tried such optimisations as a double iteration method with storage of the transpose of A on the virtual processors, but this was not more efficient. This transposition of data on the two-dimensional grid generates some general communications, i.e., communications not between neighbours using NEWS but over longer routes, requiring a general-purpose router. This is a very expensive part of the implementation of the Arnoldi projection on a massively-parallel machine.

After the computation of $w = Av_1$ (resp. $w = Av_j$), we permute w and we have to compute $h_{0,1} = (w, v_0)$ and $h_{1,1} = (w, v_1)$ (and, $h_{i,j-1}$ $\forall i \in [0, j-1]$). Let us assume that at the end of each iteration j, v_j is saved on the j^{th} row of a 2D grid of virtual processors. Thus, for j fixed, let $Pvar$ be a massively-parallel variable. We have $Pvar$ equal to $v_k(l)$ on the virtual processor $p(k,l)$ for $k = 0, j-1$ and $l = 0, n-1$. We also have $w(l)$ on each processor $p(k,l)$. Then, we can compute the element $h_{i,j-1}$ for $i = 0, j-1$ of the Hessenberg matrix in parallel. On each row k of virtual processors, with $k < j$, we compute the scalar product $h_{i,j-1} = (w, v_k)$. Next, in $1 + log_2(n)$ massively-parallel operations we compute all the j^{th} columns of H_m. As the vector v_j is row mapped, $h_{i,j-1}$ is available on each virtual processor $p(i,k), \forall k$. All the products $h_{i,j-1} * v_i$, $i = 0, j-1$ can be done, therefore, simultaneously in one massively-parallel operation. We remark that we use jn virtual processors to compute these parts. We may conclude that we really use the massively-parallel power of this 2D grid of processors to compute the Hessenberg matrix, and not only for the matrix-vector product, since all the elements of each column of the upper Hessenberg matrix are computed in parallel. Each non-zero element of the k^{th} column of H_m is computed by the row k of the two-dimensional grid of virtual processors.

Using n^2 processors, the number of massively-parallel operations of each step of the Arnoldi projection is $O(log_2(n))$. Hence, the number of massively-parallel operations in the projection is $O(mlog_2(n))$, compared to $O(mn^2)$ for the sequential algorithm.

4.2 Complete vs Incomplete Orthogonal Projection on Massively-Parallel Architectures. The result of the Arnoldi projection is often already useful if the basis V is only incompletely orthogonalised, according to Saad [19]. This comes from an empiric but generally accepted observation. Let C be the number of diagonal bands of the projected Hessenberg matrix. Then we have $\delta_{i,j} = (v_i, v_j)$ only for $|i - j| < C + 1$. These techniques permit us to optimise the number of floating-point operations. The incomplete orthogonal basis of the Krylov space is generated as follows, for j fixed :

$$v_j = Av_{j-1} - \sum_{i=max(1,j-C-1)}^{j-1} h_{i,j-1} v_i, \tag{6}$$

$$\text{with } h_{i,j-1} = (Av_{j-1}, v_i);\ i = max(1, j - C - 1), j - 1, \tag{7}$$

$$h_{j,j-1} = \|v_j\|, \tag{8}$$

$$v_j = v_j / \|v_j\|. \tag{9}$$

We have seen that for the complete orthogonalisation, each element of one column of H_m is computed in parallel and that the multiplication of these elements with the already available vector v_j takes just one massively-parallel operation using jn virtual processors. If we use incomplete orthogonalization, only $min(j, C)$ elements of the j^{th} column of H_m need to be computed and only the same number of elements from the already-computed vectors are affected. Thus, we would always use fewer than $min(j, C) \times n$ virtual processors. The time to perform one massively-parallel operation on a SIMD massively-parallel machine as the CM2 is always $vpr \times C \times \phi$, where ϕ is the time on one physical processor for the operation involved, even if only one virtual processor is allocated. Thus, elapsed time will be the same as when using complete orthogonalization as studied above. Only the number of physical processors running is different. This implies that throughput will be smaller in the case of one incomplete orthogonalisation, so that complete orthogonalisation is better in our case. This is in contrast with the situation for sequential machines, where incomplete orthogonalisation is recommended for efficiency reasons.

We may try to optimise the massively-parallel incomplete orthogonalisation variant of the Arnoldi projection algorithm. For example, we can map the C band of the upper Hessenberg matrix on a

C by n grid of processors. But this mapping of the elements of H_m would be incompatible with the mapping of the vector v_j. To accommodate it, we would have to change the other parts of the massively-parallel algorithm and would no longer have any regular neighbour communications in these parts. Thus overall communications would be slower. When we study a complete numerical method, we need to find a well-adapted mapping for the main parts of the implementation. We are often faced with the problem of not optimising some part sufficiently in depth to retain a large-enough average degree of parallelism and well-adapted communications.

Since the complete orthogonalization is better from a numerical point of view, and is not more expensive than an incomplete one on these architectures, all the performance figures presented in this paper are based on using a complete orthogonalization.

5 Arnoldi's Method on Massively-Parallel Architectures

5.1 A Massively-Parallel Algorithm. Let q_1 denote the number of QR method iterations that we need to calculate one eigenvalue, q_2 the number of inverse iteration method iterations to compute each eigenvector, and let $q = \frac{m}{r}$. During the QR iterations we deflate the matrix after each eigenvalue has been calculated. We take m equal to a power of two (we suppose that r is also equal to a lower power of two). We take $q = 4$, the smallest power of two larger than three (three being the value commonly proposed on sequential machines). We take the average values of q_1 and q_2 as suggested in [25] and [10], i.e., $q_1 = 2$ and $q_2 = 1$. Then, we can estimate the time required to compute the Ritz elements. Let t_{Ritz} be this time. Let t_{proj} be the time for the massively-parallel Arnoldi projection described in the previous section. Let $t_{V\tilde{v}}$ be the time of the part γ, see section 2.2. This part represents a massively-parallel multiplication of an $n \times m$ matrix by a vector of size m when we re-start the method, and a massively-parallel multiplication of the $n \times m$ matrix V by the $m \times r$ matrix composed of the r Ritz eigenvectors when convergence occurs. Hence, the average time t to compute one Arnoldi's method iteration with the parameters above is :

$$t = t_{Proj} + t_{Ritz} + t_{V\tilde{v}}.$$

We will focus in this paper on a general iteration step, i.e., when convergence has not yet occurred. Thus, the massively-parallel complexity of the computation of $V\tilde{v}$ is $O(log_2(m))$. We can compare this with the complexity of the two other parts of one iteration of Arnoldi's method, i.e., respectively $O(m^2)$ and $O(mlog_2(n))$. The exact number of massively-parallel operations to compute $V\tilde{v}$ is $1 + log(m)$. Then, we can remark that the time $t_{V\tilde{v}}$ is very small compared to t_{Ritz} and t_{Proj}. Hence, the global time of Arnoldi's method on a massively-parallel machine is dominated by t_{Proj} and t_{Ritz}.

The problem when we use several massively-parallel algorithms in combination is to find a common efficient mapping for all the parts. Is the n by n mapping proposed for the Arnoldi's method well adapted for the Ritz computation? Is this mapping coherent with the massively-parallel $n \times m$ matrix by m vector multiplication? The answer depends strongly on the target machine used.

Nevertheless, in general the n by n mapping is very efficient for the Arnoldi projection part. We have seen that this is the logical mapping for the matrix-vector product, but is it also the best for the other computation of this part?

When we compute the j^{th} vector of the Krylov's basis, after the matrix-vector product, we use only $j-1$ ($j \leq m$) rows of virtual processors out of n (with $n >> m$ by assumption). Might an m by n mapping be better? With it, we have vpr equal to $\frac{n*m}{p}$. In this case the vpr is $l = \frac{n}{m}$ times less but the average proportion of physical processors running is larger. This is because, during the matrix-vector multiplication, all the processor are allocated and during the computation of v_j, at j fixed, we have $(j-1)n$ processors running under mn virtual processors instead of $(j-1)n$ under n^2, as with the first mapping. Since the matrix-vector part of this Arnoldi's projection can be decomposed on l parts, we have to arrange, for this part of each iteration, some array of order l on each virtual processor. Then, with a $m \times n$ mapping of the data onto the physical processors, the average efficiency of the projection part of Arnoldi's method might be better than with a $n \times n$ mapping (which, in turn, is better for the matrix-vector product). But, as we have seen in section 3, the Ritz computation only needs fewer than $2m$ virtual processor per fixed step. So, in this part and with this mapping we utilize fewer than 2 in each group of n virtual processors (n large by assumption).

And how about an m by m mapping, that would be very well-adapted to the Ritz computation? In this case, the projected matrix is mapped with one element per virtual processor. Hence, we use a block version of the matrix-vector multiplication and of the computation of the Krylov's basis vector. Then, on each virtual processor we have an $l \times l$ matrix, a vector of length l and one element of H_m. We also have some redundant scalars. Of course if m^2 is smaller than the number of physical processors p, we select a 2D grid of order $\sqrt{p}$ as a virtual processor geometry. We note this is a compromise between the largest well-adapted massive-parallelism and the mapping with the minimum vpr.

We argued in section 3 that a mapping of 1D geometry is impractical because of massively-parallel properties of the QR method. So we cannot continue to limit the number of virtual processors in order to optimise the average efficiency.

In general, there are so many parameters depending on a given target machine and software environment that only experimentation can indicate the best mapping. The theoretical best average efficiency for Arnoldi's method on a massively-parallel machine is achieved using a $m \times m$ geometry to map the data onto p physical processors.

Now let us evaluate the global throughput of the method of Arnoldi for solving eigenproblems. Let $CX()$ be the complexity of a part of the algorithm at the mathematical level (i.e., it is not the massively-parallel complexity). For example, $CX(n,m)$ means that we need $CX(n,m)$ floating-point operations and that this complexity depends on the parameters n and m. Then, the throughput is as follows :

$$d_{Arnoldi} \simeq \frac{CX_{Proj}(m,n) \ + \ CX_{Ritz}(m) \ + \ CX_{V\tilde{v}}(m,n)}{t_{proj} \ + \ t_{Ritz} \ + \ t_{V\tilde{v}}}. \tag{10}$$

Here, as shown before, $CX_{Proj} = O(mn^2)$, $CX_{Ritz} = O(m^3)$ and $CX_{V\tilde{v}} = O(mn)$). But the massively-parallel complexity of these parts is respectively $O(mlog_2(n))$, $O(m^2)$ and $O(log_2(m))$. An in-depth study permits us to conclude that, if $l = \frac{n}{m}$ is not very large, then $d_{Arnoldi} \simeq \frac{CX_{Proj}}{t_{Ritz}}$; i.e., the elapsed time is dominated by the time for the Ritz computation, and the majority of operations is performed during the projection part. The Ritz computation is thus really a significant bottleneck. But if the ratio l increases, we may have, for n fixed, $d_{Arnoldi} \simeq \frac{CX_{Proj}}{t_{Proj}}$. Then the throughput of the method becomes very close to that of the projection part. In this case, the Ritz computation is not longer dominant. The exact value of l that leads to this situation depends on the target machine. But, as performance always increases with l, for n fixed, the criteria for choosing m are not only performance-based, as we will see in the next section.

We may conclude that the massively-parallel complexity of the Arnoldi method is $O(mlog_2(n))$ + $O(m^2)$.

5.2 Experiments on the Connection Machine 2. We now present some experiments on the CM2 concerning a few Arnoldi iterations without any reorthogonalisation. We start with dense matrices.

We have implemented both an n by n and an m by m mapping algorithm. For example, with n equal to $1K$, on the CM2 the throughput is from two to three times better with an m by m mapping than with an n by n mapping, for m from 512 to 128. We will present obtained performances using a $m \times m$ geometry.

In choosing between two mappings on the CM2, it is not customary to select the one with the smaller *vpr*. Conventional CM2 programming methodologies, like those described in [18], always call for the largest *vpr* possible (because the machine uses pipelined floating-point accelerators that have a large start up time). This is a valid viewpoint when all the processors are executing. However, for a complicated program we often cannot find a mapping with a large *vpr* and a large average number of active processors. On other massively-parallel machines, such as the MP-1 from Maspar Computer Corporation, the time is linear with the *vpr*. Then, the effects of the *vpr* are consistent with our choice and perhaps more in the philosophy of massively-parallel programming.

We present in Table 2 the performances obtained on a 16K CM2. We first note that when n or m is doubled, then the time of the Arnoldi projection is multiplied by approximately two. The throughput variation of the projection part of the method is not significant for n fixed. Each computation of one Krylov space basis vector and of one column of the projected matrix takes an almost constant time. Though the number of processors allocated (and hence the number of floating-point operations executed) changes, the number of SIMD massively-parallel operations does not change. Especially on machines like the CM2, where the assignment of virtual processors to physical processors is dynamic and cannot be managed by the user, the time variation and the variation in number of floating point operations are both linear when m increases. Thus the throughput variation is quasi-constant with respect to m, at n fixed.

Throughput variation is completely different if we consider Arnoldi's method in its totality. From this standpoint, the variation in time for the method stems principally from the variation in times of its projection and Ritz phases. We have seen above that the first of these is doubled, and the second is four times larger, when m is multiplied by two. Hence the variation in total throughput may be very difficult to analyse. It can depend on many parameters, some dependent on the application (like l), others on the architecture, language or compiler used.

As expected, Arnoldi's method gives the best performance for l large. For l small, we have confirmed that, on the CM2, the Ritz computation is the real bottleneck. But, for n fixed, as m decreases we observe that this part of the algorithm becomes less and less significant. Hence, for n fixed, performance improves as m decreases. Since we often use the iterative subspace method to find only a few eigenelements of very large matrices, we often have a large n and a small r. In these cases, we have a small m and a large l. But, if we want to compute a larger number of eigenelements, the massively-parallel method of Arnoldi may be very slow on a CM2.

Let d^p be the throughput obtained with p physical processors. We have also seen from our experiments that:

$$d^{16K}(n) \simeq 3d^{4K}(n) \text{ with } vpr^{16K} = \frac{vpr^{4K}}{4}.$$

Hence, if we assume that performance with respect to the number of physical processors continues to scale up in the same way for numbers of processors greater than 16K, we may expect, for example, more than one Gigaflop for $n = 28K$ and $m = 16$ on a 64K CM-2.

6 Related Works.

The analysis of non-Hermitian or Hermitian eigenproblems is discussed in depth in [2, 10, 25]. The implementation on parallel architectures of subspace methods for symmetric matrices is proposed in [1] (MIMD machine with shared memories) and [6, 17] (vector machines). Parallelisation on the CM2 of Jacobi's method for symmetric eigenproblems is presented in [7]. The parallel solution of symmetric tridiagonal eigenproblem on a hypercube is presented by Jessup [14]. An implementation of the double-QR algorithm is also proposed for an array-processor in the symmetric case [23]. Parallel implementations of eigenproblems are surveyed by Ipsen and Saad in [12]. The use of the subspace method to solve dense small symmetric matrix eigenproblems is discussed in [11]. There, the authors proposed to use these to solve dense and medium matrix eigenproblems for Hermitian matrices (Lanczos) and not only for very large matrices. We have seen that on massively-parallel machines the assumption $n >> m$ must be satisfied in order to achieve good performance. If not, the Ritz computation is a strong bottleneck on SIMD massively-parallel machines like the CM2.

The parallelisation of the non-Hermitian case is not commonly studied. However, an array processor implementation of the QR method is proposed in [9], not starting with the Hessenberg

						Projection		Arnoldi's method	
n	$log_2(n)$	m	l	r	vpr	time (sec.)	Throughput (Mflops)	time (sec.)	Throughput (Mflops)
7168	< 13	256	28	64	4	225	**483**	2184	50
		128	56	32	1	125	432	350	151
		64	112	16	1	63.8	420	117	227
		32	224	8	1	31.8	422	45.8	293
		16	448	4	1	15.6	419	19	342
		8	896	2	1	7.7	422	8.54	**380**
4096	12	512	8	128	16	218	354	20935	3.7
		256	16	64	4	94	385	2093	17
		128	32	32	1	53	336	278	51
		64	64	16	1	28.3	311	82.3	107
		32	128	8	1	13.6	316	27.6	158
		16	256	4	1	6.6	327	10	215
		8	512	2	1	3.1	337	4.04	268
2048	11	512	4	128	16	99	217	20864	1.13
		256	8	64	4	37	256	1995	5
		128	16	32	1	21	215	246	18
		64	32	16	1	10	208	64	34
		32	64	8	1	5.2	206	19	57
		16	128	4	1	2.6	208	6.0	90
		8	256	2	1	1.3	202	2.1	122
1024	10	512	2	128	16	48	132	20813	0.5
		256	4	64	4	17	155	1975	1.5
		128	8	32	1	9.3	130	234	5.3
		64	16	16	1	4.6	129	58	9.7
		32	32	8	1	2.3	120	16	17
		16	64	4	1	1.1	119	4.6	30
		8	128	2	1	0.6	120	1.4	48
512	9	512	1	128	16	28	76	20793	0.26
		256	2	64	4	8.7	91	1967	0.36
		128	4	32	1	4.5	74	229	1.45
		64	8	16	1	2.2	69	56	2.8
		32	16	8	1	1.1	64	14	4.8
		16	32	4	1	0.5	69	3.9	8.5
		8	64	2	1	0.25	68	1.05	16
256	8	256	1	64	4	5.1	52	1963	0.273
		128	2	32	1	2.4	41	227	0.59
		64	4	16	1	1.2	35	54	0.85
		32	8	8	1	0.6	30	14.8	1.33
		16	16	4	1	0.3	29	3.8	2.4
		8	32	2	1	0.15	28	0.95	4.7
128	7	128	1	32	1	1.4	23	226	0.3
		64	2	16	1	0.7	18	46	0.37
		32	4	8	1	0.35	15	14.15	0.4
		16	8	4	1	0.14	16	3.6	0.68
		8	16	2	1	0.7	16	0.87	1.3

Table 2: Complete Orthogonal projection and Arnoldi's method performance on a 16K-CM2; dense case; with an $m \times m$ geometry for the virtual processors and $q = 4$.

matrix and using deferred shifts. Shroff presents in [22] a Jacobi-like implementation on the CM2 to solve the non-Hermitian problem. He proposes a $O(nlog_2^2(n))$ massively-parallel method. Compare this to the $O(n^2)$ method proposed in [16] (and summarised in part in this paper) and to the $O(mlog_2(n)) + O(m^2)$ method of the Arnoldi implementation described in detail in section 4 and 5. For n small the timings presented in [22] show that the QR variant of our massively-parallel Arnoldi method is more time-efficient. When the size of the matrix is large, Schroff's Jacobi-like implementation is faster than the QR method presented in [9], but slower than the Arnoldi method described in this paper. We note that the numerical stability of these algorithms may be different.

An MIMD parallel implementation of the Arnoldi method is proposed in [15] for both tightly as well as and loosely coupled memory machines with vector elementary processors and large granularity. This study has already shown that QR method is the most significant bottleneck on these MIMD architectures. We have to use redundant QR computations on elementary processors for efficient implementation ot the Arnoldi method on architectures with require large granularities.

7 Conclusions.

With a 16K-processor CM2, we obtain performances of a few hundred Megaflops for $n > 1024$. The expected performance for a 64K-processor CM2 shows that the asymptotic speed may be larger than one Gigaflop for very large matrices. The projection onto the subspace may be carried out at almost 500 Megaflops with $n = 7K$ using a 16K CM2. If we extrapolate, using coefficients derived as described in this paper, we may expect 1.35 Gigaflops for the Arnoldi projection on a 64K CM2. The performance figures presented were obtained using 32-bit floating-point pipeline accelerators, but we also experimented with our programs on a 4K CM2a[4] having 64-bit floating-point chips. On this machine, the throughput of our programs was approximately from three quarters to half as fast as with single precision floating-point operations.

Our performances were obtained in C/PARIS without too many target-level optimizations. It is possible to use a communication compiler [5] to optimize the permutation of vectors on the 2D grid of virtual processors, because the pattern of these general communications is known at compile time. Recent results for *all-to-all* communications on the CM2 [13, 4] can also be applied to permute these data. Hence, the performances presented in this paper can be improved, but we can safely conclude based only on the figures presented here that the Arnoldi method is well-adapted for massively-parallel machines.

We have seen that we need to have $n >> m$ to obtain high throughput. Otherwise, computations on the projected problem form a bottleneck. However, experimentation yielded a reasonable value for the ratio $\frac{n}{m}$, which was also consistent with the assumptions made for the Arnoldi method. We also have seen how important it is to choose a good geometry for the mapping of virtual processors onto physical processors. When we implement a combination method of different numerical basic methods, it may be difficult or impossible to find one general mapping which permits a good average utilisation of physical processors. But, given such a mapping, and with a good choice of

[4]at Site Expérimental en Hyperparallélisme, ETCA

$\frac{n}{m}$, the massively-parallel complexity of Arnoldi's method is $O(mlog_2(n)) + O(m^2)$, as opposed to $O(mn^2)$ on a sequential machine. A few other related iterative subspace methods, such as GMRES, for example, will have the same kind of properties on a massively-parallel machine.

The speed of convergence for such methods usually increases when the subspace size m is choosen larger. The number of floating point operations, and therefore the time required by the algorithm, rapidly increases with m. Futhermore, we have seen that we really need to take m as small as possible if we want to avoid QR to become a bottleneck. The choice between increasing parallelism and reducing the number of iterations is important here. One of the most significant future debates in scientific computing, in our opinion, will be about the interaction between conserving numerical stability (convergence speed in this case) and successfully utilising potential parallelism.

Acknowledgments

The author would like to thank Martin Schultz, Youcef Saad and Joel Saltz for their helpful discussions. We also wish to thank Harry Berryman and Kay Crowley for their programming suggestions and other information on the CM2. We also gratefully acknowledge the comments and suggestions of an anonymous referee of the Applied Numerical Mathematics Journal that helped me to improve both content and presentation.

References

[1] S. Bostic and R. Fulton, *A Lanczos Eigenvalue Method on a Parallel Computer*, Tech. Rep. 89097, NASA, 1987.

[2] F. Chatelin, *Spectral Approximation of Linear Operators*, Academic Press, 1983.

[3] ——, *Eigenvalues of Matrices*, Masson, 1989 (in French).

[4] D. Delasalle, D. Trystram, and D. wenzek, *Optimal Total Exchange on a SIMD Distributed-Memory Hypercube*, in Proceeding of the Sixth Distributed Memory Computing Conference, 1991.

[5] D. Dahl, *Mapping and Compiled communication on the Connection Machine System*, in Proceeding of the Fith Distributed Memory Computing Conference , 1990.

[6] N. Emad, *Contribution to the Resolution of Very LargeEigenproblems*, PhD thesis, Université P. and M. Curie, Paris VI, January 1989 (in French).

[7] L. Ewerbring, F. Luk, and A. Ruttenberg, *Matrix Computation on the Connection Machine*, in Twenty-first Conference on Signals, Systems and Computers, MAPPLE PRESS, San-Jose CA, November 1988.

[8] L. Ewerbring, and F. Luk, *Computing the Singular Value Decomposition on the Connection Machine*, IEEE Transactions on Computers, Vol. 39, 1990.

[9] R. van de Geijn, *Implementing the QR-Algorithm on an Array of Processors*, Tech. Rep. 1897, Dept. Computer Science, University of Maryland at College Park, 1987.

[10] G. Golub and C. Van Loan, *Matrix Computation*, North Oxford Academic Oxford, second ed., 1989.

[11] R. Grimes, H. Krakaeur, J. Lewis, H. Simon, and S. Wei, *The Solution of Large*

Dense Generalized Eigenvalue Problems on the Cray X-MP/24 with SSD, in Second SIAM Conf. Par. Proc. and Sci. Comp., 1985.

[12] I. Ipsen and Y. Saad, *The Impact of Parallel Architectures on the Solution of Eigenvalue Problems*, Tech. Rep. 444, Dept. Computer Science, Yale University, 1985.

[13] S. Johnson, and C. Ho, *Matrix Transposition on Boolean n-cube Configured Ensemble Architectures*, SIAM J. Matrix Anal. Appl., Vol. 9, 1988.

[14] E. Jessup, *Parallel Solution of the Symmetric Tridiagonal Eigenproblem*, Tech. Rep. 728, Dept. Computer Science, Yale University, 1989.

[15] S. Petiton, *On the Development of Numerical Software in Parallel Environments*, PhD thesis, University P. and M. Curie, Paris VI, December 1988 (in French).

[16] ——, *Parallel QR Algorithm for Iterative Subspace Methods on the Connection Machine (CM2)*, in Parallel Processing for Scientific Computing, Dongarra et al, ed., SIAM, 1990.

[17] B. Philippe and Y. Saad, *Solving Large Sparse Eigenvalue Problems on Supercomputers*, in In Parallel and Distributed Algorithms, Cosnard et al, ed., North-Holland, 1989.

[18] R. Pozo and A. MacDonald, *Performance Characteristics of Scientific Computation on the Connection Machine*, Tech. Rep. 440, Dept. Computer Science, Boulder University, 1989.

[19] Y. Saad, *Variations on Arnoldi's method for computing eigenelements of large unsymmetric matrices*, Linear Algebra and its Applications, Vol. 34, 1980.

[20] ——, *Partial Eigensolutions of Large Nonsymmetric Matrices*, Tech. Rep. 214, Dept. Computer Science, Yale University, 1985.

[21] M. Said, M. Kanehisa and Y. Saad, *Higher Exited States of Acceptors in Cubic Semiconductors*, Physical Review B, Vol. 35, 1987.

[22] G. Shroff, *Parallel Jacobi-like Algorithms for the Algebraic Eigenvalue Problems*, PhD thesis, Rensselear Polytechnic Institut, Troy, May 1990.

[23] G. Stewart, *A Parallel Implementation of the QR Algorithm*, Parallel Computing, vol. 5, North Holland, 1987.

[24] Thinking Machines Corporation, *Introduction to programming in C/PARIS*, 1989.

[25] J. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford University Press, 1965.