Working Paper

A parallel implementation of a deflation algorithm for systems of linear equations

Author(s):
Williams, A.; Burrage, Kevin

Publication Date:
1994

Permanent Link:

https://doi.org/10.3929/ethz-a-004284232 →

A parallel implementation of a deflation algorithm for systems of linear equations

A. Williams[1] and K. Burrage[1]

Research Report No. 94-12
September 1994

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

[1]Department of Mathematics, University of Queensland, Brisbane 4072, Australia

A parallel implementation of a deflation algorithm for systems of linear equations

A. Williams[1] and K. Burrage[1]

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

Research Report No. 94-12 September 1994

Abstract

A general deflation technique for solving arbitrary systems of linear equations was described in Burrage et al. (1994a). This technique can be used with any iterative scheme. As the iterations proceed information is obtained about the eigenvalues of the iteration matrix which either cause slow convergence or divergence. These eigenvalues (and associated eigenvectors) are then deflated into a stiff subspace. This then leads to a coupled iteration process between the underlying iteration on the nonstiff space and a Newton iteration on the stiff system. This process can rapidly accelerate the convergence of even very ill-conditioned systems. In this paper a parallel implementation of the algorithm is presented for a distributed memory MIMD environment. A number of numerical results are given showing the efficacy of this approach.

[1]Department of Mathematics, University of Queensland, Brisbane 4072, Australia

A parallel implementation of a deflation algorithm for systems of linear equations.

Alan Williams and Kevin Burrage

September 28, 1994

1 Introduction

Effective scientific algorithms to exploit parallel computers are still in a considerable state of flux. This is due to the wide variety of parallel machines each with widely varying performance characteristics due to processor performance, network topology, communication characteristics and memory management. Currently, there is little portable applications software for parallel computers, but this state of affairs is now slowly changing.

Perhaps one of the most important aspects of scientific computing involves the solution of systems of linear equations. There are essentially two approaches, based on direct methods such as LU factorisation and iterative methods. Direct methods are in general more robust but such algorithms are hard to implement especially on distributed memory MIMD machines. Most iterative methods are relatively simple to implement in parallel because they are rich in matrix-vector and matrix-matrix products. However, convergence can be very slow and this necessitates the use of effective preconditioners. However, it is often the case that the most effective preconditioners (such as those based on incomplete LU factorisation) are not readily parallelisable.

In this paper we report on an initial attempt at the parallelisation of the deflation techniques (introduced in Burrage et al. (1994a, 1994b)) for solving systems of linear equations. These techniques were originally applied to the implementation of a Generalised Cross Validation technique used for fitting surfaces to noisy rainfall data gathered from up to 16,000 meteorological stations scattered irregularly across Australia. The ensuing systems are extremely ill-conditioned. An interactive visualisation environment ADVISE (Lau et al. (1994)) has been developed which allows a number of environmental factors associated with drought conditions in Australia to be viewed in an interactive manner. At the moment the surfaces are fitted by running the GCV code and then this data is incorporated into a number of simulation programs which are run in real time. The hope is that these surfaces can be calculated in real time and then incorporated into ADVISE. However, in order for this to happen a number of criteria must be satisfied:

- In order to fit rainfall surfaces to 16,000 stations across Australia, approximately 2.6 GBytes of main memory is required. This is not usually available on most vector machines. However, the Intel Paragon sited at ETH Zürich has approximately 3 GBytes of main memory distributed across 96 processing nodes.

- In order to do the complete simulations in realtime, one surface would have to be fitted in 1 to 5 seconds. For most computers this is currently unrealistic for 16,000 data points, but is a realistic proposition for 2000 to 3000 data points (corresponding to one state within Australia, such as Queensland).

- ADVISE should be ported to a parallel computer. This has been addressed in Rezny (1994).

The aim of this preliminary paper is to investigate whether some of these aims are realistic for the Intel Paragon using the deflation techniques of Burrage et al. (1994a, 1994b).

Thus the outline of the paper is as follows. In section 2 we describe the deflation algorithm from a computational viewpoint. Section 3 discusses a number of issues affecting the performance of parallelized codes in general, including communication overheads, use of BLAS, load balancing. In section 4 we describe one way of parallelizing the deflation algorithm. Section 5 gives some numerical results of a preliminary parallelization running on the Paragon while section 6 gives conclusions and future work.

2 Deflation techniques

The deflation technique is fully described in Burrage et al (1994a, 1994b) and will only be briefly described here, from a computational perspective. Consider the system of linear equations given by

$$Ax = b, \quad A \in R^{n\times n}, \quad x, b \in R^n. \tag{1}$$

Then a splitting of A by $A = M - N$ leads to a general stationary iterative scheme

$$Mx^{(k+1)} = Nx^{(k)} + b. \tag{2}$$

Here M is some easily invertible matrix such as the diagonal of A in the case of the Jacobi method or ωI in the case of Richardson iteration.

The convergence of these fixed point iteration schemes is directly affected by the distribution of the eigenvalues of the iteration matrix $J = M^{-1}N$. Convergence is fastest when the eigenvalues of this matrix are closely clustered around the origin. In particular, convergence will only take place if $\rho(J) < 1$.

This deflation technique uses subspace projections to extract or project the 'problem' eigenvalues into one space (hopefully of small dimension), and keep the rest of the eigenvalues in the complement space. Then a coupled iteration is performed where a direct solve (such as that based on LU factorisation) is applied to the small 'stiff' system, in conjunction with

the fixed point iteration on the large non-stiff system. As will be seen later, this dramatically accelerates the convergence of the underlying scheme.

The deflation routine takes as input two parameters $freq$ and eig which determine how often eigenvalues are extracted, and how many are extracted each time, respectively. Typically, extracting 2 or 3 eigenvalues every 4 to 8 iterations gives good results, but this is of course problem dependent.

Thus initially, $freq$ Jacobi iterations are carried out and a matrix δ (of size $n \times eig$) is kept where, at each iteration, the first column of δ is updated by

$$\delta(:,1) = x^{(k)} - x^{(k-1)} \tag{3}$$

and the differences from previous iterations are shifted across through the other columns of the matrix δ. This matrix is then orthogonalised using a Gram-Schmidt scheme. The matrix $w = \delta^T J \delta$ is then formed, and a Schur decomposition of w produces the matrices Z and T which satisfy

$$w = TZT^T, \quad T^T T = I. \tag{4}$$

A matrix (of size $n \times eig$) containing eig eigenvector estimates is given by $V = \delta Z$, and a matrix (of size $eig \times eig$) containing eigenvalue estimates on its diagonal is given by $E = V^T J V$. This technique is essentially the Power method so the dominant eigenvalues and associated eigenvectors are obtained.

Denoting the two projection subspaces by Q and P where

$$Q = I - VV^T, \quad P = VV^T, \quad V^T V = I, \quad QP = 0, \tag{5}$$

and writing

$$x = (P + Q)x = Px + q = Vy + q, \tag{6}$$

(with y a vector of dimension eig), and

$$\begin{aligned} (I - E)y &= V^T M^{-1} b + V^T J q \\ q &= Q(M^{-1} b + Jq + JVy) \end{aligned} \tag{7}$$

leads to the coupled Reverse Gauss-Seidel iteration scheme given by

$$\begin{aligned} q^{(k+1)} &= Q(M^{-1} b + Jq^{(k)} + JVy^{(k)}) \\ y^{(k+1)} &= (I - E)^{-1}(V^T M^{-1} b + V^T J q^{(k+1)}). \end{aligned} \tag{8}$$

From a computational perspective this leads to the scheme

$$\begin{aligned} x &= J(Vy + q) + M^{-1} b \\ q &= x - V(V^T x) = qx \\ y &= (I - E)^{-1} V^T (M^{-1} b + Jq) \\ x_{new} &= Vy + q. \end{aligned} \tag{9}$$

Here y is calculated by a direct solve using the previously stored LU factors of $(I - E)$. It should be noted here that by changing the super-scripts of y and q on the right hand side of (8) other iteration schemes can be obtained which are the equivalent of Jacobi iteration, or Gauss-Seidel iteration. (See Burrage et al. (1994a))

Thus every $freq$ iterations the eigenvalue extraction process is repeated in which a further eig columns are appended to V, E is updated by the double matrix product $E = V^T JV$, and the LU factors of $I - E$ are recalculated.

3 Issues affecting the performance of parallelised code

As a preliminary task to implementing the deflation algorithm in parallel, we first implemented the Jacobi method for solving linear systems of equations. The Jacobi method is very good for illustrating the level of performance that can be achieved by the iterative schemes that we are concerned with here.

The Intel Paragon is a distributed memory MIMD machine, with processors located on a 2-D grid and communcating with each other by message passing which must be explicitly controlled by the programmer. Peak performance per processor is 75 MFLOPS for 64 bit floating point computation, and the machine at ETH Zürich has 96 compute nodes, so peak performance is approximately 7.2 GFLOPS.

The compute nodes on the Paragon are i860 RISC (Reduced Instruction Set Computer) processors. The attainable performance is heavily dependent on how effectively the cache can be utilised. Careful attention to array access stride lengths is important, for example. Performance is also dependent on instruction scheduling, with peak performance figures assuming that certain floating point operations can be executed concurrently, which is done quite effectively in assembly coded library routines, such as the BLAS (Basic Linear Algebra Subprograms).

The work in Jacobi iterations consists almost entirely of calculating matrix-vector products, so we are able to make extensive use of the BLAS library provided on the Paragon.

Each node of the Paragon has 32 Mbytes of memory, of which about 26 are available to user programs. Some of this is used for message buffering etc. It has been found that it is best to keep the size of the executable program under about 23 Mbytes, depending on how much communication is to be done, how many processors are to be used, and thus how big message queues are likely to get. If too big an executable is run on a large number of processors then the message buffering requirements can result in paging which severely degrades performance.

3.1 Communication overheads

In the case of the iterative linear system solvers being discussed here, there is some communication which needs to take place between processors at each iteration.

As can be seen in Figure 1, we have produced timings for 3 different problem sizes. The MFLOP rate achieved on a single node of the Paragon by the Jacobi iteration program is

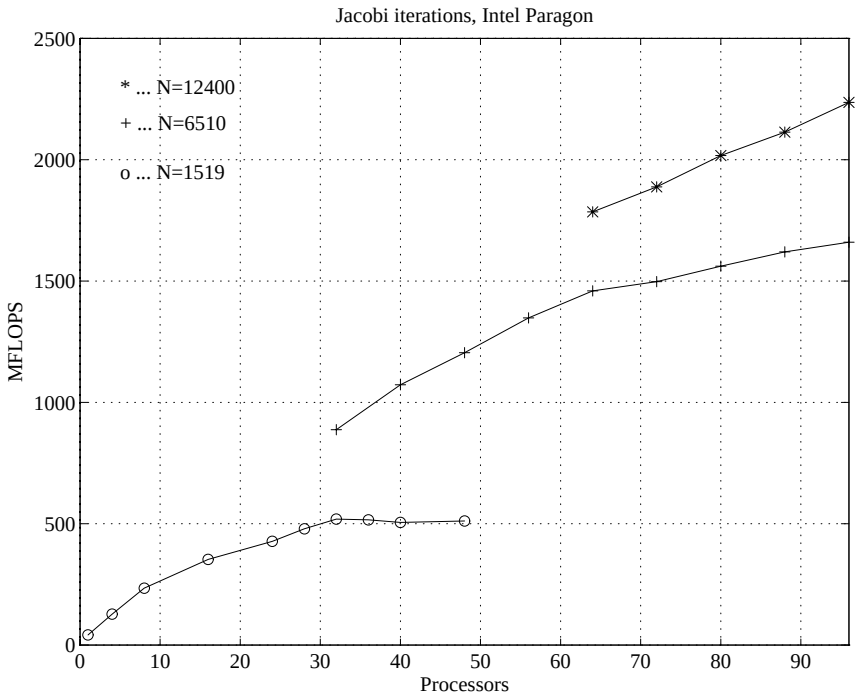

Figure 1: Observed MFLOPS on the Paragon

about 41 MFLOPS, which is nearly 55% of claimed peak performance. It should be noted that this algorithm uses only level 2 BLAS routines, not level 3. Higher MFLOP rates can typically be achieved with level 3 BLAS due to the better cache reuse and sub-blocking strategies which are possible with these routines. When the number of processing nodes is increased, the problem size remains unchanged, and so the total amount of work done by the program is constant. Clearly adding more processors increases communication overheads, and reduces the per-processor MFLOP rates. The work load is distributed by sharing the iteration matrix J in blocks of rows across the processors, and then at each iteration, each node forms a part of the updated iterate by $x^{(k+1)} = Jx^{(k)} + M^{-1}b$. Thus each node uses the entire previous iterate to form a section of the next iterate, necessitating a vector collection and broadcast at each iteration. Timings on a test problem of size $n = 1519$ are produced for up to 48 nodes, where it is clear that communication time is beginning to become significant and performance is no longer scaling up.

If the assumption is made at compile time that the program will be run on no less than p nodes, then a problem size p times larger can be used, and then the time spent in doing calculations dominates that spent in communication for a larger number of processors. The two larger problem sizes $n = 6510$ and $n = 12400$ illustrated in the graph show this. This demonstrates the effectiveness of parallel machines on very large problems.

3.2 Cache efficiency

Cache efficiency is an important factor affecting the performance of RISC processors. Because of the large difference between the time required to access main memory compared with accessing the cache, a very significant performance increase can be observed by accessing array elements column-wise (varying the first index in the inner loop) rather than row-wise. This is because if the array is accessed row-wise, then elements of memory are addressed with a stride length of n rather than 1, requiring the cache to be reloaded much more often.

In the case of a matrix that is shared row-wise across processors, it is clear that significantly more stride n memory accesses than stride 1 accesses in the course of doing the matrix-splitting, for instance will occur. A noticeable performance increase can be obtained by storing the rows as columns and then performing operations on the transpose of the matrix.

3.3 Proportion of serial work

Almost all algorithms have at least a small amount of work which cannot be done in parallel. The proportion of total execution time that must be spent in serial calculations has an obvious but surprisingly large effect on the potential speedup that can be achieved by increasing the number of processors. As a trivial example, an algorithm that spends just half of one percent of execution time doing work that cannot be parallelised, would achieve a speedup of slightly less than 167 if it were run on 1000 processors. That figure is an upper bound on the speedup possible, and assumes that there are no communication overheads due to the parallelisation.

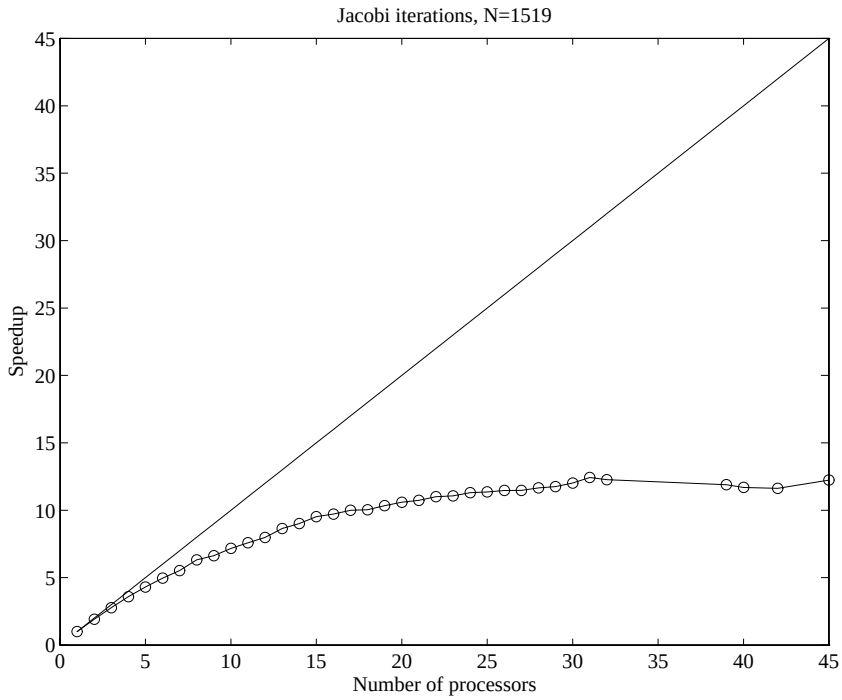

Figure 2: Jacobi speedup for a small problem size

4 Parallelisation of the deflation technique

The biggest component of execution time for this deflation technique is spent in the matrix-vector and matrix-matrix products, so these tasks are the main candidates for parallel execution. There are also other components, such as the initial splitting of the matrix, numerous vector adds and updates, etc.

The size of the problem to be solved, and the number of processors that will be used, both affect the way data will be distributed and the way work will be shared. As was illustrated with the Jacobi iterations, there are two significant classes of problem size, namely one that is small enough to reside in the memory of one processor, and one that is bigger than that. If the problem can be contained on one processor, then the program can be written in a general way so that at runtime the user can specify any number of processors. In this case, since the memory on the Paragon is distributed and not shared, the coefficient matrix will be of size $n \times n$ on each processor, but not necessarily fully initialised on all processors, and each processor would only access a certain portion of the matrix. If, however, assumptions are made at compile time as to how many processors will be used, then just n/p rows (or columns, depending on how the work is to be organised) of it need to be declared on each node, with p being the minimum number of processors to be used. Then of course matrices p times larger can be stored, enabling us to make better use of the very large amounts of memory available on MPP machines. However, a different indexing strategy is required for accessing the matrix.

4.1 Issues affecting the parallelisation

When a program is adapted for parallel execution, a useful measure of how well it performs can be obtained by plotting a speedup graph, which shows the execution time of the serial program divided by the execution time of the parallel one. If no overheads are introduced by parallel execution, and there are no significant regions of necessarily serial execution, then speedup should be linear. In practice this is of course rarely the case.

The matrix-vector and matrix-matrix products executed in the deflation routine consist of two broad types, as far as work-sharing is concerned:

- products involving the $n \times n$ matrix J,
  (both matrix-vector and matrix-matrix products)

- and products involving only matrices of smaller dimension
  (V $(n \times r)$ and E $(r \times r)$, r the number of eigenvalues extracted so far) which increase in size as more eigenvalues are extracted.

In addition, the matrix-vector products involving V have different implications to those involving V^T since, for example, $d = Vy$ is an $n-$vector while $c = V^T x$ is only of dimension r. Clearly it isn't practical to parallelise the calculation of a result vector which has dimension smaller than the number of processors we would typically expect to be using on an MPP machine.

One of the most significant operations performed in the routine is a matrix double product ($w = V^T JV$, w of size $r \times r$). This is coded as two matrix-matrix products, where the first call involves J and results in a temporary matrix which is then used in the second call to produce the result w ($tmp = JV, \quad w = V^T tmp$). Clearly it is not practical to parallelise the calculation of the second matrix-matrix product (which results in w), since the value of r is typically at most 10% of the dimension of the problem. It is worthwhile, however, to parallelise the first product. But then the node which does the second one must collect the 'pieces' of the intermediate result from the slave nodes. Depending on whether we distribute the large matrix J column-wise or row-wise across the processors, the triple product can be done in a couple of different ways:

$$\begin{array}{rcl} tmp = JV & , & w = V^T tmp \\ tmp = V^T J & , & w = tmpV \end{array} \tag{10}$$

We can of course use the fact that $J^T V = (V^T J)^T$ to change the order in which the matrix arguments appear, and also to cater for row-wise or column-wise distribution of J but in any case the intermediate result must be collected onto a single node before the second product is formed. Also, the data distribution should be arranged so that the matrix-matrix product can be done by each node simply calling a BLAS routine for its block of the matrix, which implies that if $C = AB$ is to be calculated, for instance, then A should be shared row-wise or B should be shared column-wise across the nodes.

As Figures 3 and 4 illustrate, it is possible to to arrange it so that each node forms either rows or columns of a result matrix.

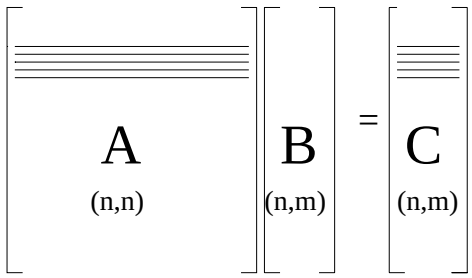

Figure 3: Matrix A shared row-wise

Since Fortran arrays are stored in column-major format, such that the first index varies fastest, then the collection of the tmp matrix onto one node is simplified if each slave node has columns. This way the starting address of the first column to be sent can be specified, and a single message of an appropriate length can be sent containing the p columns. If rows of the matrix are to be sent, they must first be packed into a vector one after each other, and then unpacked into the receiving matrix on the target node. However, this consideration is countered by the fact that another significant operation performed at each iteration is the matrix-vector product Jq. For this to be shared among a pool of slave processors, J

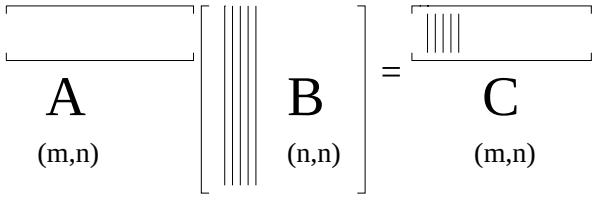

Figure 4: Matrix B shared column-wise

really needs to be distributed row-wise. For this reason it has been decided to distribute J row-wise.

The strategy that is used for this implementation, is to share all operations involving J, and non-transpose products involving V, among a pool of slave nodes. Products involving V^T, and the LU factorisations and solutions of the small system, will be done by a single 'master' node. Thus it is intended that the master node can perform certain serial tasks concurrently with the large products that the slave nodes are doing, and also act as a collector and distributor of distributed vectors and stopping test norms. So the coupled reverse Gauss-Seidel iteration (9) could be parallelised as follows.

Parallel iteration

node 0		slave nodes
$tmp = V^T x$	$\rightarrow$	$tmp2 = V * tmp$
	$\leftarrow$	
$q = x - tmp2$	$\rightarrow$	$Jq + M^{-1}b$
$y = (I - E)^{-1}V^T(Jq + M^{-1}b)$	$\rightarrow$	
$(JVy) = JV * y$		$x = Vy + q$

The arrows in the above figure indicate the communication that must happen between the master and slave nodes. It is clear that for this tightly coupled iteration there are considerable serial regions where the slave nodes must wait for results to come from the master, and conversely for the master node. So the speedup is limited to the reduced time required for the products calculated by the slaves, together with the concurrency provided by smaller tasks such as the shuffling of the δ matrix and some vector updates being able to be done in the background. These effects have been confirmed by using the software package known as Paragraph, which can be used to analyse the performance of a parallel program.

It is important to consider though, that the operations involving the matrix V (containing the eigenvector estimates) will take longer as more eigenvalues are extracted. For example, if a problem is solved in 150 iterations with 3 eigenvalues being extracted every 5 iterations, then the time taken to do operations on the matrix V and on E increases by at least an order of magnitude during the execution of the program. This means that load balancing and parallelism considerations change during program execution if these operations (done by a single node) are done concurrently with a product involving the matrix J (size $n \times n$) which is shared among the pool of slave nodes.

5 Results

In this section we present results comparing the convergence properties of the deflation technique with those of the Jacobi iteration, and also timing results showing the performance of the deflation technique. The program was written in Fortran77 with Intel NX native message passing routines.

5.1 Comparison of deflation with Jacobi iterations

A simple symmetric test matrix was generated with the off-diagonal elements filled by *sine* values and an arbitrary larger value put on the diagonal. In this way the condition number can be varied easily. The matrix used for the following comparison is of size $n = 1116$, and has a condition number of 83.78. For this problem Jacobi iterations give approximately linear convergence to an error tolerance of 10^{-6} in the residual (difference between successive iterates) in 1012 iterations.

The deflation algorithm gives convergence to the same accuracy in just 16 iterations. The points at which the eigenspace approximation was updated can be clearly seen in Figure 5 where the convergence accelerates. This matrix provides an almost ideal case for the deflation algorithm, having only a few widely scattered eigenvalues. This also effectively illustrates the fact that the deflation routine is useful for solving a class of problem that cannot be solved by the Jacobi method.

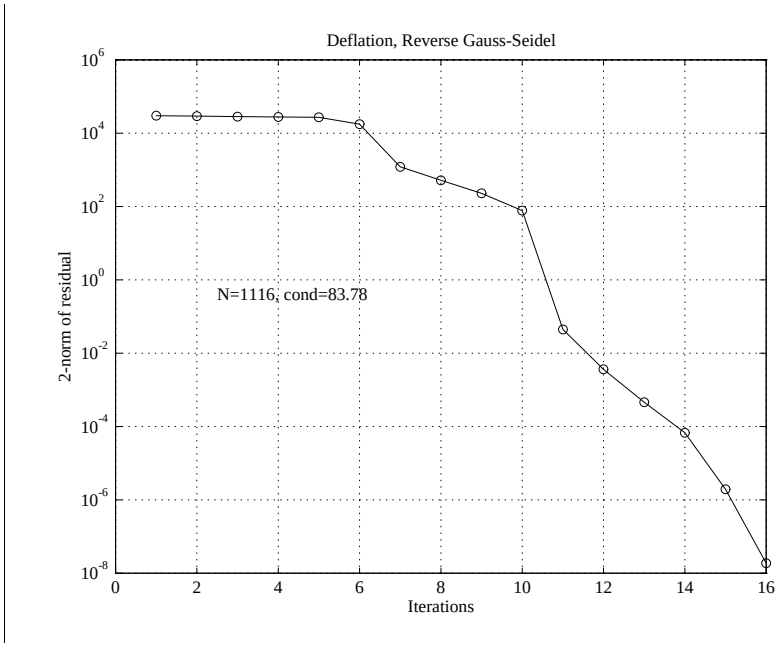

Figure 5: Convergence of the deflation algorithm

5.2 Deflation performance

In this section we present preliminary performance results for the deflation algorithm. Com-

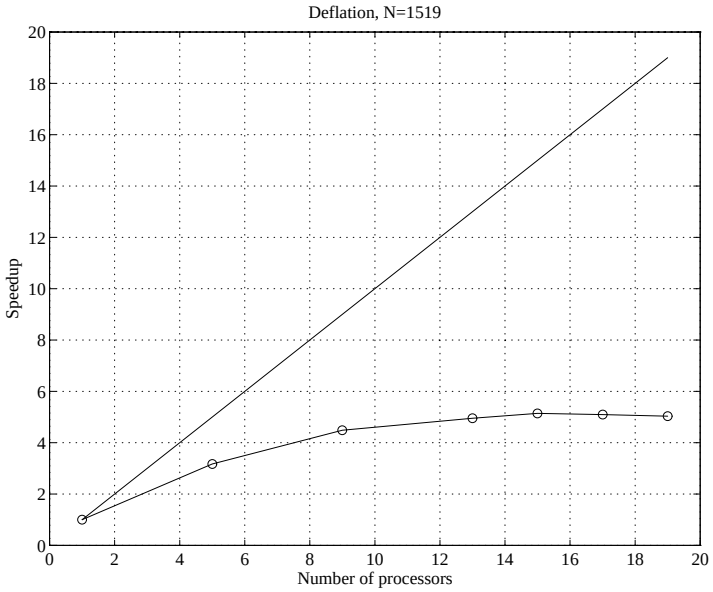

Figure 6: Speedup on a small problem size

paring the two Figures 6 and 7 shows that the deflation algorithm exhibits the same behaviour as the Jacobi algorithm described in Figure 2. In particular, as the problem size is increased, greater speedups are obtained for a larger number of processors. Thus in the case of a problem of dimension 1519 performance does not seem to improve with more than 15 processors. However, for a problem of size 6200, performance continues to improve for up to 45 processors. This phenomenon would be expected to continue as the dimension of the problem increases further. Since we are ultimately concerned with solving very large problems (dimension 16000 and more) we would expect to be able to use all 96 processors in an effective manner.

It should be noted that there are some difficulties when attempting to evaluate the performance of parallel algorithms on very large problems, in that performance comparisons cannot be made with a single processor (or indeed a small number of processors). This is because there is insufficient memory to store such problems.

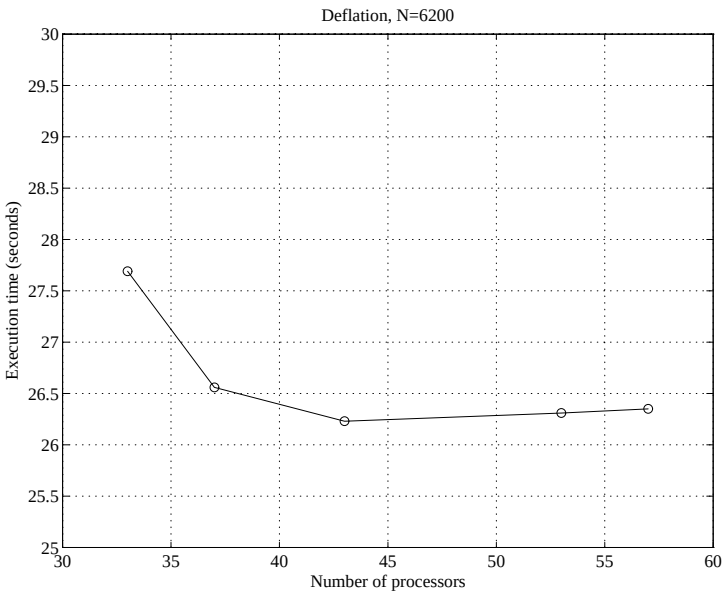

Figure 7: Deflation algorithm

6 Conclusions and future work

It has been demonstrated that the aims stated in the introduction are realistic, and in particular that very large matrix systems can be efficiently solved on the Paragon. However, in order to obtain 'real-time' behaviour within ADVISE greater performance is needed from the deflation implementation. A number of further approaches would lead to improvements.

- Different iteration schemes:
  As has been mentioned, the reverse Gauss-Seidel iteration scheme which underlies this implementation of the deflation method results in significant amounts of communication and large regions of serial execution. A less tightly coupled iteration would offer better parallelism and less communication (by using fewer but larger messages to send more than one vector at a time). The convergence properties would, however, be changed somewhat. It may be possible to use some preconditioners at this stage.

- Communication: Collection of a distributed vector onto a single node.
  There are a number of considerations to be made when deciding how to arrange the communication involved in collecting the pieces of a vector. Depending on the workload of the node that will do the collecting, there are at least three possible ways of doing it 'optimally', the goal of course being to minimise the time that any node spends idle or waiting to receive a message.
  As the slave nodes finish calculating their sections of the vector, the collecting node can be in one of three states:

  - standing idle waiting at the receive statement. In this case the best strategy is simply to have each slave node send its piece of the result vector to the master node, which places all of the pieces into the appropriate places.
  - not yet waiting at the receive statement. The best strategy here would be to collect the result vector on one of the slave nodes and then send it to the master in a single message.
  - arriving at the receive statement essentially 'on time'. If all nodes arrive at essentially the same time, then the most effective way of minimising the collection time is to form a tree structure where each node first passes its piece of the vector to a neighbouring node and so on, so that the master node receives in the order of $log_2(P)$ messages.

  As has already been mentioned, in this algorithm the workload of the master node varies as execution progresses. So all of the above can be expected to occur at some time in the execution of the routine. Ideally the structure of the communication would be varied adaptively as the load balancing changes.

- Asynchronous iteration:
  One of the things hampering the speedup of the deflation algorithm is the large serial

region which occurs where the eigenspace approximation is done. While the master node is doing the LAPACK routine calls to form the schur decomposition and LU factorisation etc., the slave nodes are idle. If they would continue iterating with the 'old' eigenspace approximations while they wait for the updated one to be calculated then some extra convergence could be achieved while greatly reducing processor idle time. It is known that asynchronous iteration can, in general, improve parallel performance by removing sychronization points.

- Adaptive eigenspace approximation:
  Currently the frequency with which the eigenspace approximation is updated is set at the beginning of the program. Performance gains could be obtained by monitoring convergence and updating the eigenspace at different intervals as necessary. Also, the number of eigenvalues to be extracted could be varied.

It is hoped to further continue this work on the parallel implementation of deflation algorithms. We expect considerable improvements if some of the above factors are taken into account.

7 Acknowledgement

The authors would like to thank Professor Dr. Rolf Jeltsch of the Seminar for Applied Mathematics, ETH Zürich, for all his help in establishing this project, and to Intel Corporation for financial support. Thanks also to Dr. Bert Pohl at SAM, ETH and Michael Vollmer at Intel for much appreciated help with various aspects of using the Paragon.

References

Burrage, K., Erhel, J. and Pohl, B. (1994a). A deflation technique for linear systems of equations. *Res. Rep. 94-02, Seminar für Angewandte Mathematik, ETH Zürich.*

Burrage, K., Williams, A., Erhel, J. and Pohl, B. (1994b). The implementation of a Generalised Cross Validation technique using deflation techniques for linear systems. *Res. Rep. 94-05, Seminar für Angewandte Mathematik, ETH Zürich.*

Lau, L., Rezny, M., Belward, J., Burrage, K., Pohl, B. (1994) ADVISE - Agricultural Developmental Visualisation Interactive Software Environment *Res. Rep. 94-03, Seminar für Angewandte Mathematik, ETH Zürich.*

Rezny, M. (Parallel Implementation of ADVISE on the Intel Paragon) *Res. Rep. 94-10 Seminar für Angewandte Mathematik, ETH Zürich.*