Parallel Algorithms for Linear Algebra on a Shared Memory Multiprocessor

K. Wright and D. Kaya
Computing Science Department,
University of Newcastle-upon-Tyne,
Newcastle-upon-Tyne,
NE1 7RU, UK.

Abstract

This paper describes a variety of parallel algorithms for linear algebra problems developed using shared memory Encore Multimax Multiprocessors. Algorithms using dynamic task allocation are compared with ones which do not.

Problems considered include QR and LU decomposition, orthogonal reduction of General Matrices to upper Hessenberg form and symmetric matrices to tridiagonal form.

The experimental results to be presented show that dynamic task allocation can be very effective on this machine, and that very high efficiency is obtainable with careful construction of the parallel algorithms even for relatively small matrices.

Keywords: Parallel Algorithms, Linear Algebra, Shared Memory.

1 Introduction

This paper considers parallel algorithms for a variety of linear algebra problems which have been developed using shared memory Encore Multimax Multiprocessors. The aim has been both to develop efficient algorithms and to investigate more generally what techniques lead to efficient implementations using this architecture.

The problems considered are QR and LU decomposition of a square matrix, orthogonal reduction of a general matrix to upper Hessenberg form and orthogonal reduction of a symmetric matrix to tridiagonal form. Considerable work has been done on parallel implementations of algorithms for these problems though this has been concentrated on other architectures such as distributed memory machines or shared memory machines with multiple vector processors. The

multiprocessor systems used to perform the experiments were two bus-connected shared memory Encore Multimax computers running the UMAX operating system. One of the machines (Turing) has 8 NS32332 processors and 32Kb cache memory per processor. The other machine (Newton) has 14 faster NS32532 processors with 256Kb processor cache memory. This second machine was used for most of the work.

Three languages and associated parallel facilities were used. Firstly, Pascal with the Encore multi-tasking library. Secondly, Encore Parallel Fortran (EPF) which is an extension to FORTRAN77 including parallel constructs. Thirdly, C++ with the Encore Parallel Threads package [4]. All theses systems provide facilities for setting up parallel control and various synchronization facilities. Except where stated otherwise the results described were obtained using C++. The C++ language was particularly convenient as we used a simple Matrix class to represent the matrices and this could be altered internally to give storage of the matrix by rows or by columns and to include array bound checking or not.

Using any language the program does not have direct control of the allocation of either processors or storage. The transfer of data between shared and cache memory is controlled by the hardware. Variables can be declared locally or globally, but in either case will be stored in shared memory with possible copies in cache. The programmer can ask for a number of parallel tasks or threads, but again the actual allocation is controlled by the operating system, and this may depend on the current usage of the machine as it is time-shared.

This last point suggests that dynamic allocation of work may be particularly appropriate. However, it is very difficult to measure the effectiveness of this under heavily loaded conditions, and all the results in the sequel are the based on the best times observed over a number of runs when the machine was lightly loaded. Dynamic allocation is likely to improve processor utilization but does require inter-processor communication for control. Predetermined (or static) allocation is less flexible but avoids the need for this communication.

In the construction of the parallel implementations we aimed to get good efficiency by attempting to avoid waiting, by avoiding the need for synchronization and avoiding contention in accessing the shared memory, as far as possible.

The graphs given plot efficiency against number of processors p where effi-

ciency E_p is defined as

$$E_p = T_s/(pT_p)$$

where T_p is the observed parallel time and T_s the corresponding time for the appropriate standard sequential algorithm for that problem, assuming the same environment is being used. Normally the efficiency is expected to be less than one, however, in a few cases efficiencies greater than one are observed indicating the sequential form of the new algorithm is an improvement on the standard one.

The results show that careful implementation taking these points into account produces significantly better times than those for simple parallel versions for all these algorithms, and in most cases very good use of the parallel facilities is possible.

2 QR and LU Decomposition

The first problem tackled was the decomposition of a matrix into a product of an orthogonal and an upper triangular matrix or QR decomposition, and its use for the solution of sets of linear algebraic equations. This work is described in detail in [12], so only the main points will be summarized here. Reduction using Givens transformations was significantly poorer than using Householder transformations. For the Householder versions the most significant factor was using storage which represented the matrix by columns rather than by rows. This meant using the transpose of the matrix in PASCAL.

The usual sequential algorithm processes pivotal columns and then uses the pivotal columns to update the later columns with column heads in the same row. This leads immediately to a simple parallel implementation where pivotal columns are treated sequentially followed by updates of the later columns in parallel as these are all independent.

Noting that all these updates do not need to be performed before the next pivotal column can be treated, various alternative ways of allocating column updates were investigated where pivotal and normal columns were treated simultaneously. The most effective of these was an algorithm which carried out as many column updates to a column as possible by the same processor, and when no further updates were possible due to the relevant pivotal column not

having been treated, then starting on a different column. In addition, copying the column of the matrix being updated to a vector and only copying back at the end of a set of updates made a significant improvement to the PASCAL and C++ versions of this algorithm. In EPF this effect was obtained by passing a column of the matrix as a parameter. So, it seems that this improvement is partly due to reduction of subscript calculations and partly due to requiring less writing to the shared memory. Following the work in [12] a further version was investigated using the 'EVENT' facility in EPF where if a pivotal column is not yet available the process just waits until it is. This version was found to be only marginally slower than the previous version.

Precisely the same strategy can be used for LU decomposition if no row interchanges are carried out, and the single processor version is then equivalent to the GAXPY Gaussian elimination described in [7] p99. For LU decomposition with interchanges the GAXPY version described [7] p113 has updates to columns which are not independent because the whole row is interchanged together. Also the interchanges are applied to the pivotal information stored in the lower triangle of the matrix. However, the interchange strategy may be modified so that firstly, the multipliers in L are not interchanged and secondly, interchanges in U are delayed until just before a column update the column independence is recovered. If this is done the same parallel strategies used for the QR decomposition can be applied. This has the minor disadvantage that element updates cannot be carried out using scalar products (as in the Crout algorithm) as the modifications to an element are interleaved with interchanges. This possibility is mentioned by Gallivan et. al. [5], but does not seem to have been investigated further.

Versions of this algorithm for both unit upper (Crout like) and unit lower (Doolittle like) triangular matrices were implemented. Three strategies were used. The first chose columns dynamically, with a new column chosen when a required pivotal column was not available. The second allocated columns statically with waiting when necessary, where waiting was carried out using a simulation of 'EVENT' synchronization using a loop and with a test protected by a lock. The third used dynamic column allocation with waiting as in the second version. Additional variations of these strategies copied the column that was being updated to a vector at the start of a set of updates and copied it back

at the end of the set.

As for the QR decomposition, using column storage was significantly better than using row storage and copying of the columns to vectors made a substantial saving in time, while the other differences were relatively minor. The version using unit upper triangle with column changing came out marginally the best. All were significantly better than a simple parallel implementation of the Crout algorithm, where no overlapping of the pivotal stages with other updates was used. Graphs of efficiency for various of these implementations are given in Fig. 1, where average efficiencies for matrices of sizes 100(100)500 are plotted against the number of processors. More details of the LU decomposition work is given in [9].

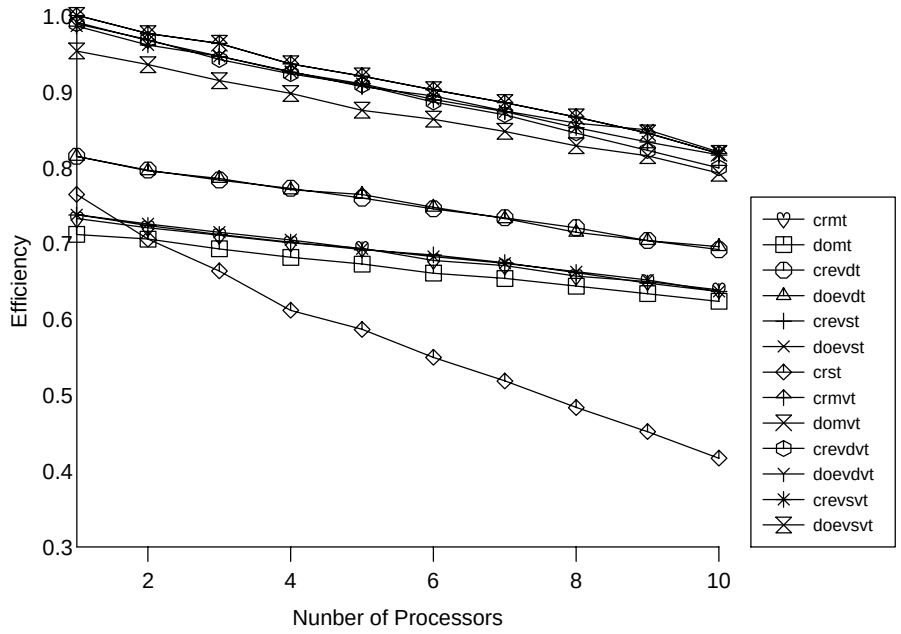

Figure 1: Efficiency graphs: LU decomposition

3 Reduction to upper Hessenberg form

The usual sequential algorithm for the orthogonal reduction of a general matrix to upper Hessenberg form, is described, for example, in Hager[8]. In this algorithm a sequence of Householder matrices $H_1, H_2, ...H_{n-2}$ are chosen such that

$$A_{n-1} = H_{n-2}H_{n-3}...H_1AH_1...H_{n-3}H_{n-2}$$

is upper Hessenberg. Each stage in the reduction of A to an upper Hessenberg matrix can be written as

$$A^* = HAH$$

where H is a Householder matrix and A^* indicates the new matrix. The multiplication is taken in the form $(HA)H$, unlike the algorithm described by Dongarra et. al. [3] which combines the two multiplications. This combination has benefits in reducing data transfer but involves more arithmetic. As good speed up with the simple algorithm is obtained, the alternative algorithm was not considered further. Other investigations of parallel algorithms are given in Dongarra et. al. [1] and Geist and Davis[6].

This sequential algorithm chooses an H using the pivotal column, headed by $a_{k+1,k}$ for $k = 1, 2, \ldots, n-2$, so that the elements below the head of the new pivotal column are zero. The matrix H is then used to update the later columns $k+1, k+2, \ldots, n$ while the columns $k = 1, 2, \ldots, k-1$ are not altered. When the column updates are completed, corresponding to the first multiplication, row updates for all the rows are carried out, corresponding to the second multiplication.

Following this the next pivotal column is treated and the whole process repeated for the next stage.

The diagram in Fig. 2 illustrates the data dependencies in this algorithm, considering updates to pivotal columns, standard columns and rows rather than individual elements. This is a simplification of the situation but sufficient for the present paper. In the diagram the k^{th} pivotal update is denoted by Pk, the update to the column headed by a_{jk} which uses Pj-1 is denoted by Cjk and the update to the row starting with element a_{jk} using Pk-1 is denoted by Rjk.

Considering this diagram it is clear that the row updates at any particular stage can be divided into two groups: the first group which will be denoted by

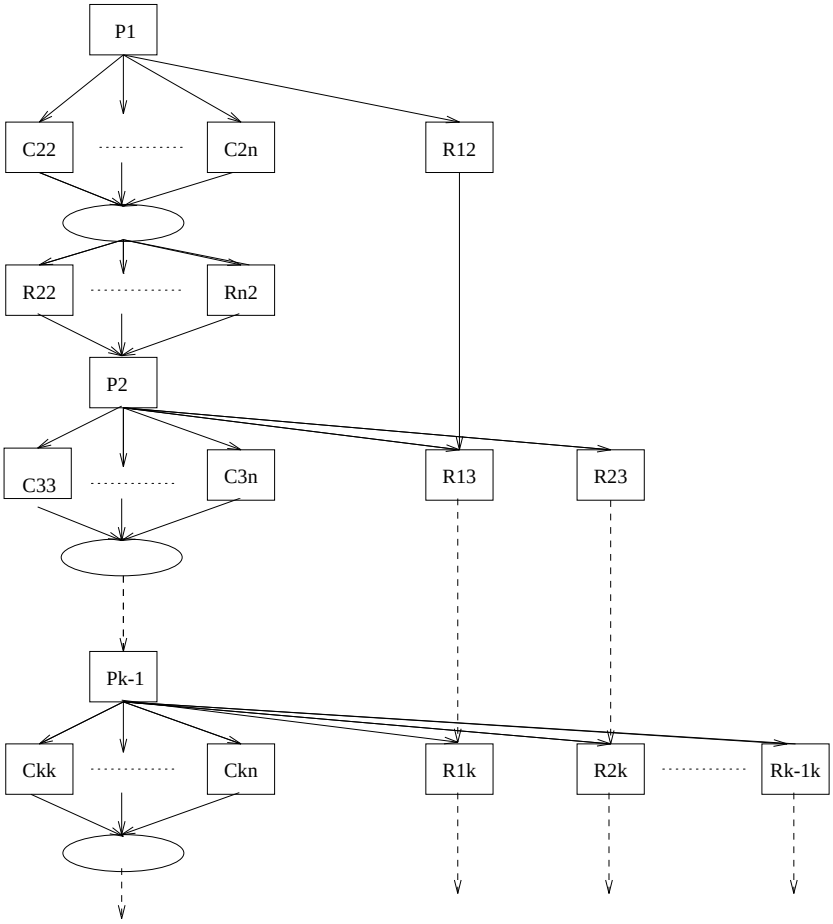

Figure 2: Dataflow Diagram: Hessenberg Reduction

rowa consists of the rows with row number less than or equal to the the latest pivotal stage number, the other rows are denoted by *rowb*. The *rowb* rows must be completed before the next pivotal update can be made, while this is not true for the *rowa* rows. This observation will be made use of in algorithms 4 and 5 described below.

Five different parallel versions were implemented. The first one is a simple implementation carrying out the pivotal column updates sequentially, followed by column updates done in parallel, and when these are all completed followed by row updates in parallel. The next two algorithms are based on the observa-

tion that at any stage the updates to *rowa* rows can be carried out at the same time as the column updates while the *rowb* row updates can not. The pivotal updates are again carried out sequentially. The *thread* chooses columns for updating until there are no more columns to allocate. If all columns are allocated, then a row is chosen from rows 1 to n taken in order. Updates are carried out immediately for *rowa* rows, but for *rowb* rows the thread must wait until all the columns have been completed. The waiting is carried out by using a loop checking a count of the number of columns which have been completed to simulate an 'EVENT'. In algorithm two the columns and rows are allocated to threads dynamically, while in algorithm 3 they are allocated in a static (scattered) way.

The fourth algorithm avoids the sequential updating of the pivotal column by allowing a start on the next pivotal column while *rowa* are still being processed. The calculations are still carried out in stages corresponding to the pivotal updates. A stage starts after a pivotal update and the previous *rowa* rows have been completed. The next set of columns is allocated for updating in parallel, with *rowa* rows allocated when no more columns remain to be allocated. However, as soon as all the set of column updates is completed *rowb* rows are then allocated for updating even though all *rowa* rows may not have been allocated. When the updates of the *rowb* rows are completed, the next pivotal column is started, whether all *rowa* rows are completed or not. The stage is finished by completing any *rowa* updates which have not been treated previously.

The fifth algorithm was designed to avoid splitting the algorithm into stages by making use of the observation made clear in the dataflow diagram that the pivotal, column and *rowb* row updates are not dependent on the *rowa* row updates being completed. In fact when this algorithm is run on a single processor all the *rowa* row updates are left until all the other updates have been completed. Work is allocated to *threads* in the order indicated in the left hand column of the dataflow diagram, that is, a pivotal update is followed by a set of column updates which is in turn followed by a set of *rowb* updates, leading to the next pivotal update. However, when any thread finds none of this work available for allocation, any *rowa* rows which are ready for updating are allocated instead. When *rowa* rows are processed all updates currently possible are carried out, that is all updates dependent on pivotal columns which have

been already treated excluding any that have been previously carried out. This should allow transfer between cache and shared memory to be reduced.

The columns and rows are allocated dynamically to the threads, and a count for the columns and a count for the *rowb* rows are used. As we indicated above, the *rowa* rows are updated independently so we used a count for each of the *rowa* rows in addition to an indicator for the next *rowa* row. This should ensure that the pivotal columns are given high priority in the early part of the calculation when they are more critical, but gradually less priority as the calculation proceeds, as in the latter stages there will be plenty of *rowa* rows ready for updating. Two implementations of this algorithms were developed for which results are illustrated, they only differ in the way the reduction is terminated. In the version $Hella$ a thread terminates when all the *rowa* rows are completed, in the $Helle$ version the thread terminates when all the *rowa* rows are allocated and the current thread has no further work. In practice little difference was found between them. These implementations are expected to be more efficient than the previous versions.

Results illustrating the performance of these algorithms were obtained. The results use the notation He for our implementations of parallel upper Hessenberg reduction. The numerical results were obtained for the five versions outlined in Section 3. The first method, the simple implementation is indicated by ss, the second implementation using dynamic allocation by ld the third implementation using static allocation by ls, the fourth implementation using pivotal processing done in parallel by mml, the fifth implementations using delayed *rowa* allocation by lla and lle. The representation of the matrix by columns is indicated by final character t, otherwise the representation is by rows. We tested the algorithms using from one up to ten processors (1,2,4,6,8,10) with matrices of sizes 100(100)500. Graphs of efficiencies are displayed in Figs. 3 and 4. More details of this work are given in [10]

There is a clear conclusion that the fifth implementations ($Hella$) and ($Helle$) are better than the other implementations, with the second version marginally better than the first. As the number of processors increases the graphs for this algorithm are very nearly horizontal indicating little loss, and remarkably high efficiencies, with both row and column representation of the matrix.

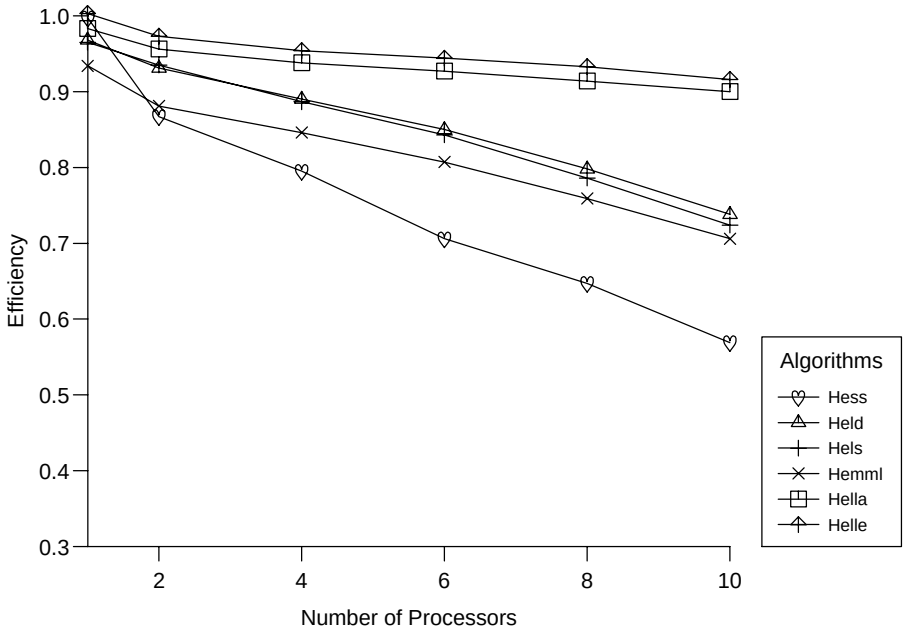

Figure 3: Efficiencies: Hessenberg Reduction (row representation)

4 Reduction of a Symmetric Matrix to Tridiagonal Form

The algorithm described for reduction to Hessenberg form can also be used for the reduction of a symmetric matrix to tridiagonal form, and there are some rather obvious savings from the symmetry. The algorithm described in Watkins [11] makes further savings by combining the pre and post multiplication phases. However, this algorithm has much greater data dependencies so limiting the possibilities for parallelization.

Experiments were carried out on parallelizing this algorithm, and some minor changes did give some improvements, but all the efficiencies obtained were much poorer than for the other algorithms described here. Due to lack of space further details will be described in a later paper.

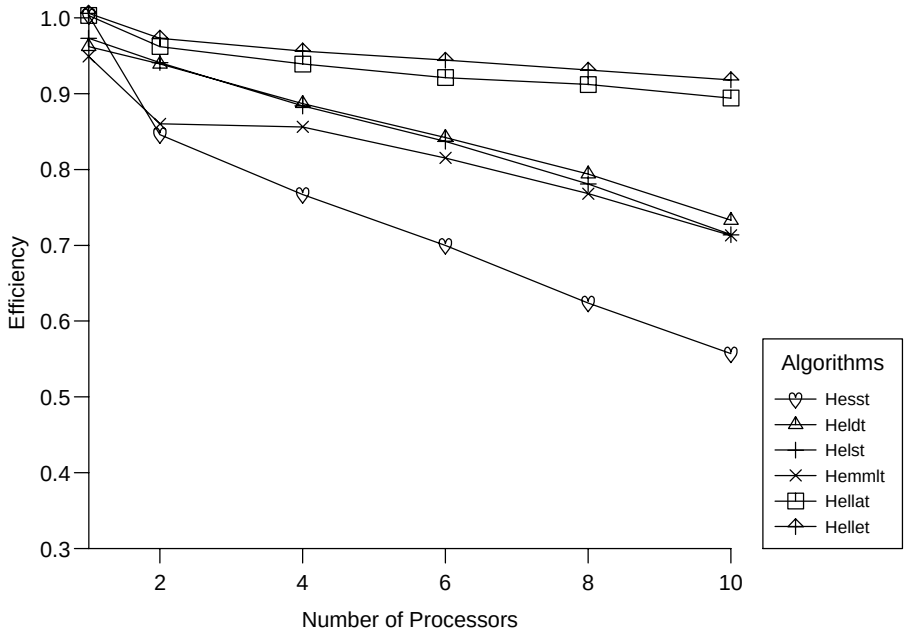

Figure 4: Efficiencies: Hessenberg Reduction (column representation)

5 Conclusions

The results described here indicate that for QR and LU decomposition and Hessenberg reduction very high parallel efficiencies are obtainable on the shared memory machine used, as long as care is taken in the implementation of the algorithms. For reduction os a symmetric matrix to tridiagonal form the results are not so good, but here a combination with finding the eigenvalues as suggested by Dongarra et. al. [2] might compensate for this problem and we hope to investigate this in detail in due course.

Acknowledgement

The second author wishes to acknowledge reciept of a grant from Firay University, Turkey to support this work.

References

[1] J.J. Dongarra and R.A. van de Geijn, Reduction to Condensed form for the Eigenvalue Problem on Distributed Memory Architectures, Parallel Comput. 18(1992) 973-982.

[2] J.J. Dongarra and D.C. Sorensen, A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem, SIAM J. Sci. Stat. Comput. 8(1987) s139-s154.

[3] J.J. Dongarra, D.C. Sorensen and S.J. Hammarling, Block reduction of matrices to condensed forms for eigenvalue computations, J. Computat. Applied Maths 27(1989) 215-227.

[4] Encore Computer Corporation,1988. Encore Parallel Threads Manual. No.724-06210 Rev. A.

[5] K.A. Gallivan, R.J. Plemmons and A.H. Sameh, Parallel Algorithms for Dense Linear Algebra Computations. SIAM Review 32, (1990) pp.54-135.

[6] G.A. Geist and G.J. Davis, Finding Eigenvalues and Eigenvectors of Unsymmetric Matrices Using a Distributed-Memory Multiprocessor, Parallel Comput. 13(1990) 199-209.

[7] G.H. Golub and C.F. Van Loan, *Matrix Computations*, 2nd Ed. Johns Hopkins University Press, Baltimore and London, (1989).

[8] W.W. Hager, *Applied Numerical Linear Algebra*, Prentice-Hall International Editions, (1988).

[9] D. Kaya and K. Wright, Parallel Algorithms for LU Decomposition on a Shared Memory Multiprocessor. University of Newcastle-upon-Tyne, Tecnical Report No.450 (1993).

[10] D. Kaya and K. Wright, Parallel Algorithms for Reduction of a General Matrix to upper Hessenberg form on a Shared Memory Multiprocessor. University of Newcastle-upon-Tyne, Tecnical Report No.490 (1994).

[11] D.S. Watkins, *Fundamentals of Matrix Computations*, John Wiley and Sons, Inc., (1991).

[12] Wright, K., Parallel Algorithms for QR Decomposition on a Shared Memory Multiprocessor. Parallel Computing 17, (1991) pp. 779-790.