The LINPACK Benchmark in Co-Array Fortran

J. K. Reid
Atlas Centre, Rutherford Appleton Laboratory,
Chilton, Didcot, Oxon OX11 0QX, UK

J. M. Rasmussen and P. C. Hansen
Department of Mathematical Modelling,
Technical University of Denmark, DK-2800 Lyngby, Denmark

Abstract

Co-array Fortran, abbreviated to CAF, is an extension of Fortran 90/95 for parallel programming that has been designed to be easy both for the compiler writer to implement and for the programmer to write and understand. It offers the prospect of clear and efficient parallel programming on homogeneous parallel systems.

A subset of Co-Array Fortran is available on the Cray T3E and the aim of this talk is to explain how the LINPACK benchmark can be written in this language and compare its performance with that of ScaLAPACK. We pay particular attention to the solution of a single set of equations since we have not found a clear description in the literature of an algorithm that is asymptotically fully parallel.

1 Introduction

Co-array Fortran (Numrich and Reid 1998), abbreviated to CAF, is an extension of Fortran 90/95 for parallel programming that has been designed to be easy both for the compiler writer to implement and for the programmer to write and understand. It offers the prospect of clear and efficient parallel programming on homogeneous parallel systems.

Each processor has an identical copy of the program and has its own data objects.

Each co-array is evenly spread over all the processors with each processor having a part of exactly the same shape. The language is carefully designed so that implementations will usually use the same address on each processor for the processor's part of the co-array. Subscripts in round brackets are used in the usual way to address the local part and subscripts in square brackets are used to address parts on other processors.

References without square brackets are to local data, so code that can run independently is uncluttered. Only where there is are square brackets, or a procedure call to code that involves square brackets, is there communication between processors.

There are intrinsic procedures to synchronize processors, return the number of processors, and return the index of the current processor.

A subset of Co-Array Fortran is available on the T3E and the aim of this talk is to explain how the LINPACK benchmark involving the solution of a dense set of 1000 linear equations can be written in this language and compare its performance with that of ScaLAPACK (Blackford et al. 1997).

2 LU Factorization of a Full Matrix

We have written a CAF code for LU factorization with partial pivoting by rows of a large dense unsymmetric matrix of order N. To give good load balance, we have followed ScaLAPACK and used a block cyclic distribution by rows and columns. The matrix is treated as a block matrix with all blocks of size $r \times r$ except those in the last block column, which may have fewer columns, and the last block row, which may have fewer rows. The processors are arranged in a $P \times Q$ rectangular array and the blocks are stored cyclically in both directions, that is, block i, j is stored on processor $[1 + \text{mod}(i-1, P), 1 + \text{mod}(j-1, Q)]$.

We use a co-array of local shape $(\lceil N/P \rceil, \lceil N/Q) \rceil$ and co-shape $[P, Q]$. This permits the local part of each block Schur-complement update to be performed by a single call of the level-3 BLAS routine GEMM, with very good cache usage. We obtained the following comparisons with ScaLAPACK on the Linkoping T3E-600 and the Manchester T3E-1200, using block sizes of 32 and 48.

LU factorization			Co-shape			
times in ms	1×1	2×1	1×2	2×2	3×3	4×4
T3E-600, block size 32						
CAF	4.13	2.49	2.43	1.35	0.70	0.59
ScaLAPACK	3.79	2.21	2.36	1.38	0.95	0.80
T3E-600, block size 48						
CAF	3.73	2.47	2.18	1.28	0.71	0.54
ScaLAPACK	3.19	2.19	2.03	1.42	1.03	0.84
T3E-1200, block size 32						
CAF	2.69	1.80	1.54	1.00	0.52	0.47
ScaLAPACK	2.36	1.51	1.43	0.94	0.69	0.59
T3E-1200, block size 48						
CAF	2.37	1.72	1.32	0.86	0.49	0.41
ScaLAPACK	1.97	1.47	1.20	0.95	0.74	0.63

It may be seen that the Co-Array Fortran code is slower for small numbers of processors, but scales better and is faster for larger numbers of processors. We do not see it as sensible to run a problem of this size on more than 16 processors. The differing block sizes can alter the speed by some 10%, but do not alter the relative performance of CAF and ScaLAPACK much.

3 Using the Factorization to Solve a Set of Equations

Given the LU factorization and a record of the permutation used, we can solve a system of linear equations by applying the permutation to the right-hand side vector, performing the forward substitution

$$L_{ii}y_i = b_i - \sum_{j=1}^{i-1} L_{ij}y_j,\ i = 1, 2, \ldots, m$$

followed by the back-substitution

$$U_{ii}x_i = y_i - \sum_{j=i+1}^{m} U_{ij}x_j,\ i = m, m-1, \ldots, 1$$

Here, we assume that each b_i, x_i or y_i is a block of order r (except perhaps for the last ones) and $m = \lceil N/r \rceil$. We assume that the matrix blocks are as left after the LU factorization, that is, block i, j is stored on processor $[1 + \text{mod}(i-1, P), 1 + \text{mod}(j-1, Q)]$. We also assume that the vectors are similarly blocked, with each block on the same processor as the corresponding block on the diagonal of the matrix.

There is a danger that applying the permutations may be as time consuming as a forward or backward substitution. Our original code applied each interchange in turn. The speed was greatly improved by holding the permutation explicitly and making each processor involved responsible for calculating its part of the permuted vector. However, for the numbers of processors that we have used here, our best speed was obtained by collecting the data onto one processor, performing the permutation there, and distributing the permuted vector.

How to best solve a single triangular system of linear equations in parallel is not obvious and we have not found a clear description in the literature of an algorithm that is asymptotically fully parallel. We developed our algorithm independently, but believe that it is a generalization of that of Bisseling and van de Vorst (1991). The main differences are that we work with blocks so that Level-2 BLAS (Dongarra, Du Croz, Hammarling, and Hanson, 1988) can be employed locally and we use a rectangular array of processors instead of a square array.

We will describe the solution of a lower triangular system. It is straightforward to adapt the algorithm to an upper triangular system.

For an off-diagonal block, the main task of its processor is the multiplication

$$L_{ij} y_j$$

For a diagonal block, the main task of its processor is to perform a forward substitution. These operations are performed by Level-2 BLAS.

Once a processor holding a diagonal block has computed its part of the solution, this must be passed to the other processors involved in the block column. We do this by passing it from neighbour to neighbour down the block column until all processors have received it.

The products computed by the off-diagonal blocks in a block row must be accumulated and passed to the processor holding the diagonal block. This accumulation is initially performed locally; for blocks whose distance to the diagonal is less than Q, the partially accumulated sums are passed from neighbour to neighbour along the block row, each further accumulating the sum.

Each processor performs actions associated with its first diagonal block, then the rest of that block column, then its second diagonal block, then the rest of that block column, etc. For each block whose distance to the diagonal is less than Q, the processor must wait for data from its neighbour to the left; and for each off-diagonal block whose distance to the diagonal is less than P, the processor must wait for data from its neighbour above it. No other synchronizations are needed. In particular, no other synchronizations are needed for blocks whose distance to the diagonal is at least $max(P, Q)$.

To see that the algorithm is fully parallel asymptotically, consider the processing of the first Q block columns. All processors are involved, but each works on a single block column. Processor [1,1] performs the first forward substitution, places the solution on processor [2,1], and synchronizes with it. Processor [2,1] performs its multiplication, places the result on processor [2,2], and synchronizes with it. It also places the vector it received from [1,1] on [3,1] and synchronizes with it. This continues until the processors are working on all Q block columns and no further synchronizations are needed. The critical path for this 'start-up phase' involves the blocks [1,1], [2,1], [2,2], [3,2], [3,3], ... involving a total of about $r^2(3Q/2-1)$ sequential operations and a small amount of communication. The rest of the calculation is fully pararallel and involves about $r^2 N/(Pr) = rN/P$ operations on each processor. Applying similar arguments to the other block columns, we get a critical path total of about $3Nr/2$ operations and about $N^2/(PQ)$ operations on each processor. The ratio of these counts is $N/(3PQr/2)$, which tends to infinity with N for fixed P, Q and r. Note that each processor that has critical path tasks in a block column attends to these before others for the block column. An outline of the CAF program is as follows:

```
  me = this_image()
  m = ceiling(N/r) ! Number of blocks in a row or column
  do j = 1, m
     do i = j, m
        prow = mod(i-1,P) + 1 ! Row co-index
        pcol = mod(j-1,Q) + 1 ! Column co-index
  ! Perform work only if this block is owned by this processor
        if (me /= procgrid(prow,pcol) ) exit
        right = procgrid(prow,mod(pcol,Q)+1)
        left = procgrid(prow,mod(pcol-2,Q)+1)
        above = procgrid(mod(prow-2,P)+1,pcol)
        below = procgrid(mod(prow,P)+1,pcol)
        s    = min(r,N-r*(i-1)) ! Size of this block
        i1 = (i-1)/P*r+1 ! First local index
        i2  = i1+s-1     ! Last local index

        if (i-j >= max(P,Q) then
           call sgemv(..) ! Multiply the block by xj and add into asum
           EXIT  ! No synchronization needed
        end if

        if (i > j) then
           if (i-j < P) then
              call sync_images( (/ me, above /) )
              xj = temp_xj ! receive part of x from image above
           end if
           call sgemv(..) ! Multiply the block by xj and add into asum
        end if

        if (j > 1 .AND. i-j < Q-1) then
           ! accumulate sum with data from image to the left in rvec
           call sync_images( (/ me, left /) )
           asum(i1:i2) = asum(i1:i2) + rvec(1:s)
        end if

        if (i == j) then
           ! perform local forward-substitution
           b(i1:i2) = b(i1:i2) - asum(i1:i2)
```

```
            call strsv(..) ! Solution in b(i1:i2)
            xj(1:s) = b(i1:i2)

         else if (i-j < Q) then
            ! send accumulated sum to image to the right
            rvec(1:s)[prow,mod(pcol,Q)+1] = asum(i1:i2)
            call sync_images( (/ me, right /) )
         end if

         ! send part of x to image below
         if (i < m .AND. i-j < P-1) then
            temp_xj(:,:)[mod(prow,P)+1,pcol] = xj(:,:)
            call sync_images( (/ me, below /) )
         end if
      end do
   end do
```

Using our CAF code, we obtained the following comparisons with ScaLAPACK on the LINPACK test involving a single right-hand side for a problem of order $N = 1000$. For the CAF code, we include the time for applying the permutations as well as performing the forward and back substitution. For the ScaLAPACK code, we found that the permutation time was significant, so we also show the time excluding the permutation time, that is, the time for the forward and backward substitution.

Solution times in ms	Co-shape 1×1	2×1	1×2	2×2	3×3	4×4
T3E-600, block size 32						
CAF	67	41	61	25	15	13
ScaLAPACK	72	47	47	39	44	44
ScaLAPACK, no perm.	59	21	35	15	15	15
T3E-600, block size 48						
CAF	70	41	63	25	15	15
ScaLAPACK	49	61	38	45	47	47
ScaLAPACK, no perm.	36	22	26	18	17	18
T3E-1200, block size 32						
CAF	33	23	30	13	10	10
ScaLAPACK	30	29	22	26	31	32
ScaLAPACK, no perm.	23	13	16	10	11	11
T3E-1200, block size 48						
CAF	40	24	35	14	10	10
ScaLAPACK	25	42	21	29	32	33
ScaLAPACK, no perm.	18	12	14	11	11	12

We note that, again, the Co-Array Fortran code is slower for small numbers of processors, but scales better and is faster for larger numbers of processors.

4 Conclusions

We have demonstrated that Co-Array Fortran can be used to write codes for the LINPACK benchmark that are clear and perform well. In particular, they scale better with increasing numbers of processors than the ScaLAPACK codes. We have also provided a straightforward description of a fully parallel algorithm for solving a triangular set of equations.

Acknowledgements

We would like to express our thanks to the National Supercomputer Centre at Linkoping University for making the Linkoping T3E available to us for the project 'Investigation of the effectiveness of Co-array Fortran' and to Bo Einarsson for the substantial amount of help that he has given us.

We would also like thank EPSRC for making the University of Manchester Computer Services for Academic Research (CSAR) T3E available to John Reid under the project GR/M7850Z.

References

[1] Bisseling, R. H. and van de Vorst, J. G. G. (1991). *Parallel triangular system solving on a mesh network of transputers*, SIAM J. Sci. Stat. Comput., 12, 787-799.

[2] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. and Whaley, R. C. (1997). *ScaLAPACK users' guide*. SIAM, Philadelphia.

[3] Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. (1988). *An extended set of Fortran Basic Linear Algebra Subprograms*. ACM Trans Math. Software, 14, 1-17 and 18-32.

[4] Numrich, R. W. and Reid, J. K. (1998), *Co-Array Fortran for parallel programming*, ACM Fortran Forum 17, 2, 1-31.