A distributed parallel solver for sparse linear systems obtained from THCM using METIS, MRILU and MPI

A. Meijster (a.meijster@rc.rug.nl)
Center for High Performance Computing and Visualisation
University of Groningen
P.O. Box 11044
9700 CA Groningen

F.W. Wubs (wubs@math.rug.nl)
Institute for Mathematics and Computing Science
University of Groningen
P.O. Box 800
9700 AV Groningen

This research has been supported by the Stichting Nationale Computerfaciliteiten (National Computing Facilities Foundation, NCF)

Abstract

In this paper a parallel solver for sparse matrices obtained from THCM (see [2]) using MRILU (library for solving sparse linear systems) is discussed. MRILU is a multi-level ILU factorization. Both the ordering and dropping are determined by the magnitude of the elements in the matrix. At each step of the elimination process a nearly independent set of unknowns is sought and eliminated. Currently, the library is sequential. In this paper we aim at parallelization for the class of distributed memory systems using a coarse grain approach. A nearly optimal distribution of the matrix is computed using a library for graph partitioning called METIS [3] . For communication between processes the MPI message passing library is used.

1 Introduction

In this paper a parallel solver for sparse matrices obtained from THCM (see [2]) using MRILU (library for solving sparse linear systems) [1] is discussed. MRILU is a sequential library for solving sparse linear systems by means of constructing Incomplete LU-factorizations (ILU factorizations). MRILU is an

abbreviation of Matrix Renumbering Incomplete LU-factorization. It is a multi-level ILU factorization. Both the ordering and dropping are determined by the magnitude of the elements in the matrix. At each step of the elimination process a nearly independent set of unknowns is sought and eliminated.

From a previous attempt to parallelize the library (see [4]) for shared memory systems using fine-grain concurrency we concluded that this yields only a modest speedup. We tried to implement basic parallel building blocks, like matrix-vector and matrix-matrix multiplication for sparse matrices, and build a version of MRILU on top of them. This approach has several drawbacks. The implementation relied on the use of a large shared memory such that each processor could access any coefficient of the matrix. This led to complicated algorithms that heavily relied on synchronization primitives like mutexes and semaphores. Standard problems like memory contention, cache coherency, and cost of thread scheduling was the cause of only a modest speedup.

This time we aim at the class of distributed memory systems using a coarse grain approach. For communication between processes the MPI message passing library is used. Clearly, such an implementation allows to execute the program on physically distributed architectures (like clusters) as well as shared memory systems with an MPI interface.

The goal is to build a variant of nested dissection. Nested dissection leads to effective orderings for direct methods. It is based on a recursive subdivision of the domain using separators, leaving only isolated domains of elementary size (one element in scalar equations). The ordering is now such that these domains are eliminated first followed by the last added separators continuing with the second-last added separators and so on until finally the separator dividing the domain in two parts is eliminated. It is known that the work in this process is dominated by the elimination of the last separators.

In the present method the domain is partitioned in subdomains using separators,where each subdomain is treated by a separate processor. As with nested dissection we eliminate the internal subdomains (using MRILU) creating connections between all separators surrounding a domain. In a direct method one would go on with the elimination of separators, however this will be quite costly. Therefore we make an incomplete factorization (based on a drop tolerance) of the associated matrix.

In the last step small elements in the matrix will be dropped. Obvious it is a waste of effort to create such small elements and, fortunately, it is possible to avoid that. It is clear that there are more small elements if the subdomains are large, because often the strength of the connection between two unknowns belonging to the separators surrounding a domain depends on their distance. So small elements can be avoided by keeping the subdomains of moderate size.

Thus the original sequential code of MRILU is left as it is, and we have build a parallel shell around it. The input matrix is a sparse matrix. Therefore a compressed storage data structure, called CSR format, is used to avoid storing unnecessary zeroes. The *CSR format* (Compressed Sparse Row) stores matrices row-wise. The CSR data structure consists of a 4-tuple (nr, cf, col, beg), where nr is an integer that represents the number of rows. The elements cf, col, and

$$A = \begin{pmatrix} a & 0 & b & 0 & 0 \\ c & d & e & f & 0 \\ g & 0 & h & 0 & 0 \\ i & 0 & 0 & j & k \\ l & 0 & 0 & m & n \end{pmatrix}$$

nr	5													
beg	1	3	7	9	12	15								
cf	a	b	c	d	e	f	g	h	i	j	k	l	m	n
col	1	3	1	2	3	4	1	3	1	4	5	1	4	5

Figure 1: A sparse matrix (top) and its CSR representation (bottom). The entries of A are deliberately chosen to be symbolic, instead of actual numbers, in order to avoid confusion between entries and indices.

beg are arrays. The array cf contains the non-zero entries of the matrix, stored row-wise. These values are floating-point numbers (doubles). The array col is of the same length as cf, and beg has length $nr + 1$. Both arrays are integer valued. For entry $cf(i)$, its corresponding column number is found in $col(i)$. Since entries of the same row are stored consecutively in cf, we only need to know the index of the first entry of this row, and the index of the last entry. The index of the first element of row i is stored in $beg(i)$, while the index of its last element is $beg(i + 1) - 1$. The array beg has length $nr + 1$, since we need to know the index of the last element of row nr. An example of the format is given in Fig. 1.

As written above, we have decided to split the matrix in as many parts as there are processors available. We chose to not use a static distribution, but prefer to determine it during run-time instead. This has the advantage that a nearly optimal distribution can be computed based on the content of the matrix. To determine this distribution a graph-partitioning library, called METIS (see [3]), has been used. Given some undirected graph $G = (V, E)$, METIS determines a partition of the vertex set V into sets V_i ($1 \leq i \leq n$, where n is the number of requested subsets). Let $\#A$ denote the number of elements of the (countable) set A. METIS tries to find a nearly optimal solution satisfying the following criteria :

- $|\#V_i - \#V_j|$ is minimal, for all $i \neq j$.
- $\#(E \cap (\bigcup_{i \neq j} V_i \times V_j))$ is minimal.

In words, METIS tries to partition V in equally sized subsets V_i, such that the number of edges which have adjacent vertices in different subsets is minimal. Finding the optimal solution is a NP-hard problem, and therefore cannot be done in reasonable computation time. Therefore, METIS computes a (good) approximation of the optimal solution. The solution is returned as an integer

array dom of size $\#V$, where $dom[i] = j$ denotes that unknown i is contained in subset V_j. This array dom is usually called a coloring of V.

To compute a distribution using METIS, we first have to construct a graph from the CSR representation of the matrix. This graph is constructed as follows. Start with empty sets E and V. For each row i insert i in V, and for each non-zero coefficient of this row with column j (with $i \neq j$) insert the edges (i, j) and (j, i) in E. METIS only handles symmetric graphs. Now, if we have n ($n > 2$) processors at out disposal, we request METIS to split this graph in $n - 1$ subdomains.

The reason for creating $n - 1$ subdomains, is that we are interested in a number of special unknowns, called the *interface unknowns*. An unknown is an interface unknown if it is adjacent to at least two subdomains. A separator is an interface unknown $x \in V_i$, of which all its adjacent subdomains V_j satisfy $i < j$ ($j \neq i$).

This way we split the unknowns in 'normal' unknowns and separators. We chose to implement a distribution strategy, based on standard Master-Slave computations, where the separators remain at the master, while all other unknowns are distributed to the slave processors.

In the next sections the actual implementation is described in some detail.

2 Structure of the program

The global structure of the program is as follows:

1. Construct an undirected graph from the matrix, and partition it using METIS.
2. Determine the separators from the partitioning.
3. Reorder the unknowns and the right-hand side domain by domain and put the separators at the end. Perform a corresponding symmetric reordering of the matrix.
4. Distribute the matrix, unknowns and right-hand side over the processors; the part corresponding to the separators resides at the master.
5. Make an ILU factorization of the matrices on the slaves using MRILU. Use this to compute the slave's contribution to the Schur complement matrix.
6. Subtract these contributions from the part of the matrix residing at the master to form the actual Schur complement and make an ILU factorization of that matrix using ILUT (thresholded ILU).
7. Enter a distributed FGMRES process.
   (a) Compute the inner products of global vectors domain by domain, add them together, and send them to all processors.
   (b) Compute matrix vector products by using the distributed matrix.

(c) Apply the preconditioner; this entails a distributed application of the solve part of MRILU.

From this global structure it is clear that much can be done in parallel, however the work on the master in steps 6 and 7c cannot be performed in parallel with that on the slaves. In the next subsections, more details on some of the less straightforward steps are given.

2.1 Partitioning, separators

In this subsection we focus explicitly on the matrices obtained from THCM. These matrices stem from a 3D ocean circulation flow problem. In order to minimize the number of separators we make use of properties of the problem. We know that we deal with a 3D problem on a Cartesian grid with only a limited number of unknowns in the vertical direction. Therefore, the basic idea is to base the coloring on the surface layer only. From this the coloring of an unknown at another layer is simply that of corresponding unknown on top of it at the surface.

Therefore the partitioning is performed as follows.

1. Select the two submatrices corresponding to the two top layers. Consider the unknowns that are on top of each other as the same unknown, so the two submatrices correspond to two graphs for the same unknowns. The union of these graphs is wanted which amounts simply by adding the two matrices.

2. Eliminate all Dirichlet unknowns from the equations.

3. Reduce the resulting matrix by only considering the part corresponding to the horizontal velocities u and v and the temperature T. The coloring of the other unknowns can be derived from these ones. In doing this we have to account for the coupling of the velocities in the continuity equations. Again the union can be taken with the corresponding momentum equation. It turns out that this extra coupling can be found by simply multiplying the matrix corresponding to the continuity equation with its transpose (which is present in the matrix as the pressure gradient)

4. Symmetrize the resulting matrix and make an undirected graph from it in order to be able to feed it to METIS.

5. Produce a coloring by METIS for u, v and T.

6. Deduce the u, v and T separator from the coloring and give the unknowns of the separator just a new color.

7. Give the salinity S the same color as T (they reside at the same grid point in physical space).

8. Derive the coloring of the vertical velocity w and the pressure p from that of u and v. We know that w and p are always internal to a domain and never can occur as a separator we just give them the color of a neighbor that is not part of the separator. Since w and p are on top of each other physically they get the same color.

9. Color simply the unknowns at the other layers the same as those at the top layer.

10. Since the pressure in each domain is determined up to a constant, one pressure unknown is picked from each domain and given the color of the separators.

11. The S unknown for which the equation is replaced by a salt conservation law, resulting in a full row in the matrix, is also given the color of the separators.

12. Built a permutation matrix from the thus found coloring.

The described process is performed on the master and has to be performed only once per geometry.

2.2 ILU factorization

For simplicity we illustrate the factorization process using only 3 processors for the following reordered matrix

$$A = \begin{bmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \tag{1}$$

Slave processors Processor i (i=1,2) obtains from the master A_{ii}, A_{i3} and A_{3i}. Next an incomplete factorization is made from A_{ii} by MRILU. In order to compute $A_{3i}A_{ii}^{-1}A_{i3}$ for every nonzero vector a in A_{i3} the system $A_{ii}\hat{a} = a$ is solved (using the solve part of MRILU) and multiplied by A_{3i}. In fact the number of nonzero columns in A_{i3} and the number of nonzero columns in A_{3i} is precisely the number of separators surrounding domain i. Hence, the nonzero data of $A_{3i}A_{ii}^{-1}A_{i3}$ can be stored in a full matrix the order of which is equal to the number of surrounding separators. The processor sends this full matrix including information on the original row and colum position to the master. After the contributions are sent they can be deleted on the slaves.

In the factorization part of MRILU we can control the accuracy of the factorization and by stopping criteria from the solve part one can control the accuracy of the computed $A_{3i}A_{ii}^{-1}A_{i3}$.

The contributions to the Schur complement are collected and added via a binary tree. Here processor 2 sends its result to processor 1 where it is added and sent to the master. (For 5 processors, processors 2 and 4 send their result to processors 1 and 3 respectively, where it is added to the part residing at that

processor and next processor 3 sends the result to processor 1 where it is added to the result there and sent to the master (processor 5).)

Master processor The contributions from the slaves are subtracted from A_{33} resulting in the Schur complement. Next an incomplete factorization of the Schur complement is made using ILUT.

2.3 Matrix-vector product

Suppose one wants to compute $y = Ax$ and suppose the vectors x and y are partitioned according to (1), hence $x = (x_1, x_2, x_3)$ and the vectors x_1 and x_2 reside at processors 1 and 2 and x_3 on the master; similarly for y. Hence, to compute y_i, for $i = 1, 2$, x_3 should be received from the master processor.

Since A_{3i} resides at the slaves we also compute $A_{3i}x_i$ there. The results thereof are added via a binary tree and on the master added to $A_{33}x_3$ to give y_3. In this example, processor 2 sends its result to processor 1 where it is added and sent to the master. (For 5 processors, processors 2 and 4 send their result to processors 1 and 3 respectively, where it is added to the part residing at that processor and next processor 3 sends the result to processor 1 where it is added to the result there and sent to the master (processor 5).)

2.4 Application of the preconditioner

On the slave processor i we solve $A_{ii}y_i = b_i$ by the solve part of MRILU and premultiply y_i by A_{3i}. The results of the respective processors are again added in a tree and subtracted from b_3. Once the new b_3 is computed, the system with the Schur complement matrix is solved using the ILUT factorization giving x_3. This result is sent to all slaves where it is premultiplied by A_{i3} and subtracted from b_i to give a new b_i. Once again MRILU is used to solve $A_{ii}x_i = b_i$ giving the result on processor i.

3 Numerical Experiments

In Tables 1 and 2 results of experiments of matrices of order 12228 and 49152 obtained from THCM are presented. From the latter also a plot is made in Figure 2. The experiments were performed on the SGI ONYX-3400 machine at RuG consisting of 16 processors (500 MHz MIPS R14000). This machine is just a smaller variant of the TERAS, with exactly the same processors and operating system.

In the first column the number of used processors is given. The number of separators found by METIS are given in column SepSurf. According to the discussion in Section 2.1 the number of separators for the full problem is approximately found by multiplying the SepSurf value by 4/3 times the number of planes in the vertical direction, the latter being 8 for both matrices, so the number of separators of the full system is an order of magnitude more as those given

np	SepSurf	Part (msec)	BrTim (s)	SCreat (s)	SchFac (s)	FacTim (s)	SolTim (s)	It
4	73	18	0.08	137	3.7	141	2.7	5
6	111	20	0.11	25	8.3	34	0.72	8
8	149	21	0.14	8.4	11.3	20	0.82	11
10	181	35	0.21	6.5	15.5	22	0.78	12
12	202	23	0.22	3.9	17.8	22	0.99	14
14	224	26	0.27	3.0	18.1	21	0.92	14
16	235	26	0.28	2.4	19.5	22	0.97	15

Table 1: Timing results for matrix of order 12228

np	SepSurf	Part (msec)	BrTim (s)	SCreat (s)	SchFac (s)	FacTim (s)	SolTim (s)	It
8	310	92	1.0	1246	113	1359	46	16
10	382	94	1.1	470	148	617	23	18
12	430	96	1.3	194	162	357	11	19
14	487	97	1.4	114	173	288	12	26
15	485	108	1.5	76	159	235	11	26
16	512	97	1.5	63	185	248	9.6	26

Table 2: Timing results for matrix of order 49152

in this column. The partition time (column Part), consisting of the partition by METIS and the extension of that to the whole problem is negligible in all cases. Columns BrTim, SCreat, SchFac, Factim and SolTim contain the timings for respectively the sends of the matrix parts to the slaves, the construction of the Schur complement (which includes the factorization on the domains), the factorization of that matrix, the total factorization time (the sum of SCreat and SchFac) and the time needed to reduce the residual of the problem by 8 decimal digits. Finally column It gives the number of FGMRES iterations to get the desired reduction of the residual.

We have to deal with two effects in the construction phase. For a small number of domains the matrix per domain is large, resulting in a large factorization time for that matrix and, due to an increase of the perimeter, more systems to be solved for the contribution of this domain to the Schur complement matrix. One observes that the time for this part, given in column SCreat decreases rapidly if the number of domains (=np-1) increases. However if the number of domains increases, the number of separators increases, making the last Schur complement larger (also sparser) resulting in a longer factorization time (see SchFac). In future improvements we have to deal with bringing down this factorization time, by a further parallelization. For the time being we have to balance the times of SCreat and SchFac, which limits the number of processors.

As written in the introduction an attractive method based on accurate factor-

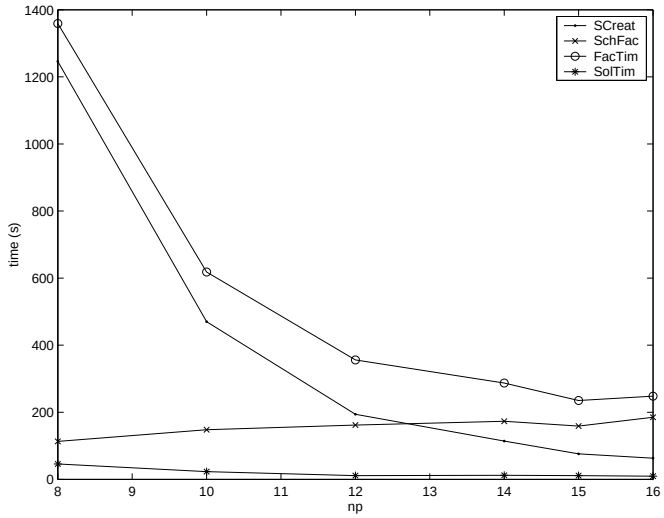

Figure 2: Performance on matrix of order 49152

izations like the present one can be achieved only for a high number of domains of not too large size since then much less elements are created in the Schur complement that will be dropped during its incomplete factorization.

Comparing the two problems, the number of unknowns in the problem of order 49152 is made 4 times more than that of order 12228 by doubling the unknowns in the horizontal directions. This means that problems of about equal size are solved on the subdomains if (np-1) on the finer grid is 4 times larger than that of the coarser one. So for np=4 on the coarse grid we find 137 seconds for SCreat and we see that this value is between that of np=12 and 14 on the fine grid. Hence this part of the algorithm scales.

On the TERAS (using 1 processor) the standard MRILU takes about 300 seconds to factor the 49152 matrix and it takes about 30 seconds to gain 5 digits using about 30 iterations. We see that the factorization time is of comparable order and the solution time is significant longer, which means that the quality of the standard MRILU preconditioner is slightly less than that of the parallelized version.

The timing of the solution process (SolTim) contains both the times on the slaves and that on the master. There is no doubt about it that also here for the larger number of processors the solution time on the master dominates.

During the experiments we encountered a few memory complications. First, the workspace buffer in MRILU is statically defined in COMMON (MRILU is written in FORTRAN 77). This means that every processor has the same workspace buffer. So if one processor needs a large workspace buffer all others have the same large workspace buffer too. This could be avoided using dynamic allocation which is available in Fortran 90 or C++. In fact the builders of MPI

assumed that one uses dynamic allocation.

A second problem with our workspace buffer is that space is claimed at both ends of the underlying array in order to minimize the number of separate free parts in the array. Now, if the array becomes too large for one processor then the last end is stored on another processor which causes long waiting times for data stored in the last end. For this reason we have observed that the performance decreased with increasing size of the workspace buffer.

4 Conclusions

From this research one can draw the following conclusions.

1. It is hard to parallelize sparse linear equation solvers for sparse matrices, specifically for matrices arising from THCM. The builders of PETSC for example were not able to solve a system with the matrix of order 49152. It is not simply adding some MPI directives but the whole process has to be designed carefully.

2. The partitioning based on part of the equations at the surface appears to be working fine, thereby reducing the size of the last Schur complement.

3. The parallelization is effective for the creation of the Schur complement.

4. Various parts of the algorithm call for an improvement, however, the most challenging part is in creating a parallelization of the Schur complement.

5 Where to download

The code can be downloaded from the site: `http://www.rug.nl/rc/hpcv/projects`
There is also a document available which describes how to use the library.

References

[1] Botta, E., and Wubs, F. MRILU: An effective algebraic multi-level ilu-preconditioner for sparse matrices. *SIAM J. Matrix Anal. Appl. 4* (1999), 520–528.

[2] H.A. Dijkstra, H. Öksüzŏglu, F. W., and Botta, E. A fully implicit model of the three-dimensional thermohaline ocean circulation. *J. Comput. Phys. 176* (2001), 685–715.

[3] Kirk Schloegel, G. K., and Kumar, V. Graph partitioning for high performance scientific simulations. Tech. rep., University of Minnesota, Minneapolis, 2000.

[4] Meijster, A., and Wubs, F. Towards an implementation of a multilevel ILU preconditioner on shared-memory computers. In *HPCN Europe* (2000), pp. 109–118.