NASA Contractor Report 189563
ICASE Report No. 91–80

ICASE

EFFECTS OF PARTITIONING AND SCHEDULING SPARSE MATRIX FACTORIZATION ON COMMUNICATION AND LOAD BALANCE

Sesh Venugopal
Vijay K. Naik

Contract No. NAS1–18605
October 1991

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665–5225

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Effects of Partitioning and Scheduling Sparse Matrix Factorization on Communication and Load Balance*

Sesh Venugopal
Dept. of Computer Science
Rutgers University
New Brunswick, NJ 08903

Vijay K. Naik
IBM
T. J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

We present a block-based, automatic partitioning and scheduling methodology for sparse matrix factorization on distributed memory systems. Using experimental results, we analyze this technique for communication and load imbalance overhead. To study the performance effects, we compare these overheads with those obtained from a straightforward "wrap-mapped" column assignment scheme. All experimental results were obtained using test sparse matrices from the Harwell-Boeing data set. The results show that there is a communication and load balance trade-off. The block-based method results in lower communication cost whereas the wrap-mapped scheme gives better load balance.

*This research was partially supported by the National Aeronautics and Space Administration under NASA contract NAS1-18605 while the first author was in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665.

1 Introduction

Partitioning and scheduling the parallel execution of large scientific applications on distributed memory systems is a difficult and time consuming task. If the dependencies involved are unstructured, as in the case of sparse linear systems, then the task becomes even more complex. Use of naive techniques to extract parallelism often results in large communication overhead and/or in large load imbalance. To reduce communication overhead, locality of data must be exploited and to balance the load, the computations must be evenly distributed at all times. When the data dependencies are non-uniform and unstructured, achieving these two goals simultaneously is difficult. As a result, in such cases, the overall performance may turn out to be poor, even if an application has a high degree of extractable parallelism. One possible way to minimize the overhead is to make use of the structure of the sparse system which can usually be determined prior to performing the numerical computations. When direct methods are used to solve the sparse systems, this information in the form of the structure of the factored matrix is routinely used to reduce computation and/or storage costs. Recently, this information has also been applied in extracting parallelism while maintaining low communication and load imbalance costs [5], [6], [14]. However, in most cases, parallelism has been extracted manually, which tends to be extremely tedious, error prone, and inflexible. Thus, automation is the key to successful parallelization of such applications. To summarize, there are two important issues in the efficient parallelization of sparse matrix based computations:

- Developing technology for the automatic parallelization of the computations.
- Developing a methodology for the extraction of the available parallelism with minimum communication and load imbalance costs.

To address these issues, we have developed an automatic, block-based scheme for partitioning and scheduling the computations in factoring a sparse matrix. The scheme makes use of the structure of the factor and is targeted towards distributed memory systems. To reduce communication, it takes advantage of locality. However, to maintain proper load balance and a high degree of parallelism, the scheme makes use of an adaptive technique in distributing the computational work.

To demonstrate the usefulness of such a partitioning scheme and to bring out the performance limitations that are inherent in sparse matrix computations, we compare the communication overhead and the degree of load balance in the automated block-based approach with that obtained from a straightforward and widely used column-based approach. In the latter scheme, computations associated with an entire column or row are assigned to a processor and the assignment of these columns or rows is usually done in a "wrap-around" fashion. We refer to this scheme as the *wrap-mapping*

or *wrap* scheme. For comparing the performance on practical applications, we present results for some of the Harwell-Boeing test matrices.

In the following discussion, it is assumed that the reader is familiar with the standard terminology used in the context of sparse matrix computations. For an explanation, see [7],[3].

The organization of the rest of the paper is as follows. In the next section, the Cholesky factorization is briefly described and some of the terminology used in the paper is introduced. The partitioning and scheduling strategies that are used for automation are presented in Section 3. Performance results are described in Section 4 and Section 5 concludes the paper.

2 Cholesky factorization

The partitioning and scheduling methodology is described in this paper assuming Cholesky factorization as the model numerical method of computation. The Cholesky algorithm is simple, well understood, and is widely used. Note, however, that the techniques presented here are applicable to other factoring methods as well. In the following, we highlight only those aspects of this algorithm that are essential for describing the partitioner. For details on the Cholesky factorization scheme, see [9].

For the sake of completeness, we first briefly describe the four steps involved in the direct solution of $A\mathbf{x} = \mathbf{b}$. (For details see, for example, [8].) It is assumed that A is symmetric, positive definite and that Cholesky factorization is used in computing the factor L, where $A = LL^T$.

1. *Ordering*: Find a good ordering of the unknowns for elimination. The ordering is given by a permutation matrix P. Most often, a "good" ordering implies one which would lead to a sparse factor and fewer arithmetic operations in the numerical factorization step.

2. *Symbolic Factorization*: Determine the sparsity structure of the factor L.

3. *Numerical Factorization*: Compute L.

4. *Triangular Solutions*: Using the computed L, solve the triangular systems $L\mathbf{u} = P\mathbf{b}$, $L^T\mathbf{v} = \mathbf{u}$ and set $\mathbf{x} = P^T\mathbf{v}$.

The basic element-level data dependencies in the factorization process are shown in Figure 1.

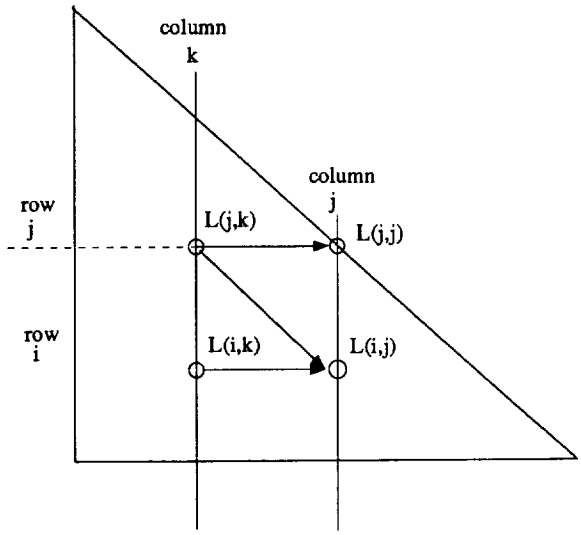

Figure 1: Inter-element dependencies in Cholesky factorization

In that figure, only the lower triangular part of the matrix to be factored is shown. $L_{i,j}$ denotes the element in row i and column j. The direction of the arrows indicates the data flow. Thus, elements $L_{j,k}$ and $L_{i,k}$ from column k of the factor L are required in computing element $L_{i,j}$. $L_{i,j} = L_{i,j} - L_{i,k} * L_{j,k}$ is the corresponding operation in the Cholesky factorization. (Initially $L_{i,j}$ is set to $A_{i,j}$.) We refer to this operation as a single *update* operation. Note that in computing the final value of $L_{i,j}$, it must be updated by all pairs of non-zero elements $L_{j,\bar{k}}$ and $L_{i,\bar{k}}$, $1 \leq \bar{k} < j$. Finally, after all the updates are performed, the element is *scaled* by the square root of the diagonal element in that column.

3 Partitioning and scheduling

The partitioning scheme presented here is static in the sense that all the computations are partitioned before any of the computations are scheduled for execution. For this, the partitioner takes as an input the structure of the factor for the sparse matrix. However, the scheme is general and does not have knowledge of any matrix structure embedded in it.

As stated in the introductory section, the aim of the partitioner and the scheduler is to reduce communication and at the same time maintain a balanced work load among processors at all times. To achieve this, wherever possible, data locality is exploited. This leads to some variation of block-based partitioning; such partitioning approaches have been proposed in several linear algebra related problems [2], [12].

With blocking, it is possible to achieve a high ratio of computation to communication per block. In [11], it is shown that for an important class of problems, the block-based partitioning schemes result in an optimal utilization of the data accessed (and thus reduce data traffic). Blocking, however, could lead to load imbalance because the increase in the size of schedulable units results in a loss of flexibility in distributing work among processors. To avoid this, the partitioner described here partitions the factored matrix into blocks of varying sizes that can be assigned in an equitable manner to the processors. It makes use of a heuristic where the block sizes are subject to adaptive manipulation. In the following we describe the functioning of the partitioner in some detail.

The partitioning starts with the zero-nonzero structure of the filled sparse matrix obtained *after* the symbolic factorization phase has been completed. Blocks of non-zero areas are identified in the filled matrix. We refer to these as *dense* blocks. On occasions, blocks are formed by including small regions that correspond to zeros in the factored matrix in order to obtain larger blocks. Inclusion of such areas with zero elements is kept to a minimum. The work in these dense blocks is partitioned into sub-blocks which are the basic schedulable units. These *unit blocks* have a regular shape - each unit block is either a column, a rectangle or a triangle. After all the unit blocks are identified, the dependencies between these blocks are determined. Finally the unit blocks are assigned and scheduled on processors.

Thus, the steps involved in the automatic partitioning and scheduling are:

- Identify dense blocks in the symbolic factor.
- Partition each dense block into schedulable unit blocks.
- Generate and store dependency information for the unit blocks.
- Schedule these units on the processors of a message passing system.
- Consolidate the non-local memory access information for each processor so as to minimize communication overhead.

In the remainder of this section, we will describe the first four steps.

3.1 Identification of dense blocks

To identify the dense blocks, first clusters of columns are determined in the sparse triangular factor. A *cluster* is a either a column or a strip of consecutive columns. If it is a strip, it contains a dense triangular block at the top and (possibly) a set of off-diagonal dense rectangular blocks. This is illustrated using an example shown in

Figure 2. In that figure, non-zero elements in the filled 41×41 matrix are indicated by the dark areas. The matrix corresponds to a 5-point finite element 5×5 grid and is ordered using Liu's multiple minimum degree algorithm [10]. It was generated using the Sparse Matrix Manipulation System developed at the University of Wisconsin [1].

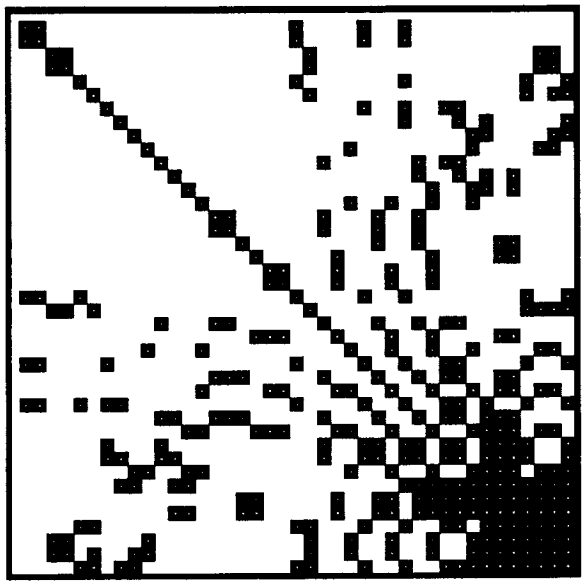

Figure 2: A 41×41 filled matrix.

In Figure 2, note the following in the lower triangular part. Cluster 1 spans columns 1 and 2 and cluster 2 spans columns 3 and 4. Both clusters 1 and 2 have a three-element dense triangular block at the diagonal. Cluster 1 has three dense rectangles below the triangle, each of which is 1×2, while cluster 2 has two dense rectangles, the upper one being 1×2 and the lower one being 2×2. Clusters 3 through 12 are single columns starting with cluster 3 at column 5. The last cluster consists of columns 35 through 41. This cluster has one dense triangle and no rectangles below it. Note that in this illustration we do not consider column 34 as part of the last cluster because of the zero in row 38 of this column. But this can be over-ridden by allowing some zeros to be a part of a triangle.

Once the clusters and the triangular and rectangular blocks within each cluster are identified, the algorithm processes the clusters left to right in the matrix. When a cluster is processed, each block in the cluster is partitioned into sub-blocks which are schedulable units. Next, for each unit, the dependencies are determined and stored. These steps are explained below.

3.2 Partitioning of a block

A cluster with a single column is considered to be a schedulable unit and is not subject to further partitioning. In a multi-column cluster, the triangular block is partitioned first. In general, the number of partitions of a triangle are determined by (a) the number of processors that are assigned to the blocks on which the triangle depends, (b) a certain minimum work requirement per unit sub-block. The first parameter restricts communication to the group of processors that work on the triangle and its predecessors. The second parameter is used to ensure a satisfactory ratio of computation to communication for each unit block and is an architecture dependent parameter. This parameter may be used to vary block sizes from one cluster to the next. For the results presented here we use a fixed size - one for all the triangular block and another for the rectangular blocks. This is referred to as the *grain size* and is the minimum number of matrix elements required in each unit block. The grain size dictates a maximum number of partitions, say P_d. A block is partitioned into at most P_d equal sized units; *at most* because it may not always be possible to break up a block into exactly P_d equal sized units.

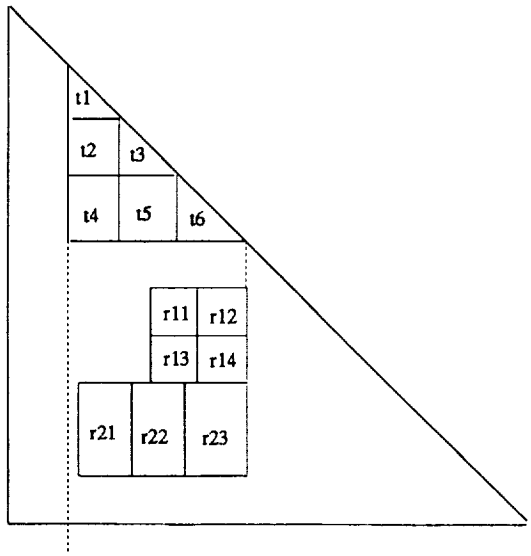

Figure 3: Partitions

Figure 3 illustrates this partitioning. The triangle is partitioned into six parts. One of the rectangles is partitioned into four parts and the other is partitioned into three parts.

3.3 Identification of dependencies

The dependencies in a single update operation at the element level of Cholesky factorization are shown in Figure 1. However, for allocation and scheduling of the units, it is necessary to identify the dependencies at the block level. In this step, for each unit block, the dependencies are determined and the information on the actual data needed in the update operation is stored. This step also identifies columns or block units that are independent, i.e., those that are not updated by any other units. To automate this process, it is necessary to classify the dependencies at the inter-block level. We have classified these dependencies into ten categories which are enumerated next. Using this classification and the interval tree structure, the partitioner computes the dependencies efficiently. The implementation details are given elsewhere.

In the following discussion, a column is represented by its column number in the matrix, a rectangle is represented by its column extent (c_i, c_j), $c_i \leq c_j$ and row extent (r_p, r_q), $r_p \leq r_q$, and a triangle is represented by its row extent (or column extent, which is the same as the row extent) (r_i, r_j), $r_i \leq r_j$.

1. A column updates a column

   This forms the base case for the dependencies. A column k updates a column j if $L_{j,k}$ is non-zero. (see Figure 1).

2. A column updates a triangle

   Let triangle T's row extent be (r_1, r_2). A column k, $k < r_1$, updates the triangle if $L_{i,k}$ is non-zero, $r_1 \leq i \leq r_2$. In Figure 4(a), the non-zero elements of column k that are involved in the update are in rows i_1, i_2 and i_3. The points of intersection of the dotted lines with each other and of the dotted lines with the diagonal are the points of triangle T that are updated by column k.

3. A column updates a rectangle

   Let rectangle R's column extent be (c_1, c_2) and row extent be (r_1, r_2). A column k updates this rectangle if it has pairs of non-zero elements $L_{i,k}$ and $L_{j,k}$, where $c_1 \leq i \leq c_2$ and $r_1 \leq j \leq r_2$. In Figure 4(b), the non-zero elements in rows i_1 and i_2 combine with the non-zero elements in rows j_1, j_2 and j_3 to update a portion of R. This updated portion is the set of points given by the intersection of the dotted lines in R's interior.

4. A triangle updates a rectangle

   Let the column extent of rectangle R be (c_1, c_2) and the column extent of triangle T be (c_3, c_4). The triangle updates the rectangle if there is an intersection in their column extents. In Figure 4(c), the shaded portion of T updates the shaded portion of R.

5. A triangle and a rectangle update a rectangle

   Let rectangle R_1 have column extent (c_1, c_2) and row extent (r_1, r_2) and let rectangle R_2 have column extent (c_3, c_4) and row extent (r_3, r_4). Let $c_2 < c_3$. Let the column extent of triangle T be (c_5, c_6). T combines with R_1 to update R_2 if (c_1, c_2) intersects (c_5, c_6), (c_3, c_4) intersects (c_5, c_6) and (r_1, r_2) intersects (r_3, r_4). In Figure 4(d), the shaded rectangular portion of T combines with the entire shaded rectangle R_1 to update the entire shaded rectangle R_2.

6. A rectangle updates a column

   Let the row extent of rectangle R be (r_1, r_2). It updates a column k if $r_1 \leq k \leq r_2$. In Figure 4(e), the shaded portion of the rectangle between rows k and r_2 update the column elements between rows k and r_2.

7. Two rectangles update a column

   Let rectangle R_1 have column extent (c_1, c_2) and row extent (r_1, r_2) and let rectangle R_2 have column extent (c_3, c_4) and row extent (r_3, r_4). Let $r_2 < r_3$. Then R_1 combines with R_2 to update a column k if $r_1 \leq k \leq r_2$ *and* (c_1, c_2) intersects (c_3, c_4). In Figure 4(f), the elements of R_1 which are in the row k between the vertical dotted lines combine with the entire shaded rectangle R_2 to update the elements between rows r_3 and r_4 in column k.

8. A rectangle updates a triangle

   Let the row extent of rectangle R_1 be (r_1, r_2) and the row extent of triangle T be (r_3, r_4). The rectangle updates the triangle if (r_1, r_2) intersects (r_3, r_4). In Figure 4(g), the shaded portion of R updates the shaded portion of T.

9. Two rectangles update a triangle

   Let rectangle R_1 have column extent (c_1, c_2) and row extent (r_1, r_2) and let rectangle R_2 have column extent (c_3, c_4) and row extent (r_3, r_4). Let $r_2 < r_3$. Let the row extent of triangle T be (r_5, r_6). Then R_1 combines with R_2 to update T if (c_1, c_2) intersects (c_3, c_4) *and* (r_1, r_2) intersects (r_5, r_6) *and* (r_3, r_4) intersects (r_5, r_6). In Figure 4(h), the shaded portion of R_1 combines with the entire shaded rectangle R_2 to update the shaded rectangular portion of T.

10. Two rectangles update a rectangle

    Let rectangle R_1 have column extent (c_1, c_2) and row extent (r_1, r_2), rectangle R_2 have column extent (c_3, c_4) and row extent (r_3, r_4) and rectangle R_3 have column extent (c_5, c_6) and row extent (r_5, r_6). Let $r_2 < r_3$, $r_2 < r_5$ and $c_4 < c_5$. Then R_1 combines with R_2 to update R_3 if (c_1, c_2) intersects (c_3, c_4) *and* (r_3, r_4) intersects (r_5, r_6) *and* (r_1, r_2) intersects (c_5, c_6). In Figure 4(i), the shaded portion of R_1 combines with the shaded portion of R_2 to update the shaded part of R_3.

3.4 Scheduling

The scheduling process is split up into two parts: allocating unit blocks to processors and ordering the computational work within each processor. In this paper, we are concerned with the first part only and the salient points therein are presented next.

First the independent columns, as identified in the previous step, are allocated to processors in a wrap-around fashion. The remaining clusters are scanned again from left to right. If a cluster is a dependent column, the entire column is allocated to a processor, which is arbitrarily picked from the set of processors which worked on the column's predecessors. If the cluster is not a column, the unit blocks in the triangular part are allocated to processors, followed by the unit blocks in each rectangular block, going top to bottom. For example, in the cluster shown in Figure 3, the six sub-blocks of the triangle would be allocated first, followed by the four sub-blocks of the rectangle below it, finishing up with the three sub-blocks of the bottom-most rectangle.

Allocation within a triangle proceeds by first allocating the triangular units from top to bottom, followed by the rectangular units, going top to bottom and left to right. In the Figure 3 for instance, the sub-blocks in the triangle would be allocated in the order t_1, t_3, t_6, t_2, t_4, t_5. A global set of all processors, P_g, is maintained, with a marker pointing to the first "available" processor. This marker cycles through the global set in a round-robin fashion and is moved up every time a unit block is allocated to the currently available processor. Apart from this, a set of processors, P_a, which have been already allocated to some sub-block in the triangle is maintained. Initially, P_a is empty. The strategy for allocating a processor to a unit rectangle or unit triangle is the same. First, the predecessors of the unit block are scanned. For each predecessor, if the processor p which worked on it is not in P_a, the unit block is allocated to p and p is added to P_a. If all of the processors which worked on all the predecessors of the unit block are already in P_a, the unit block is allocated to the currently available processor in P_g and the marker is moved up to the next processor in P_g.

For allocating the units within a rectangle below the triangle, the choice of processors is restricted to P_t, where P_t is the set of processors to which the unit blocks in the triangle are allocated. Since there is a large amount of communication between a triangle and the rectangles below it, this strategy helps in reducing the communication. First, the processors in set P_t are ordered according to increasing work. Going in round-robin fashion through P_t, the processors are assigned to the unit blocks in the rectangle, going top to bottom and left to right within the rectangle. For example, let processors p_1, p_2 and p_3 be assigned to the unit blocks on the triangle in Figure 3. Assume that the ordering according to work is such that $p_1 < p_2 < p_3$. Then, in the first rectangle below the triangle, r_{11} is allocated to p_1, r_{12} is allocated to p_2, r_{13} is allocated to p_3, r_{14} is allocated to p_1. The set P_t is sorted again and the above

strategy is used to allocate r_{21}, r_{22} and r_{23}.

4 Performance

In this section we present results on the performance of the above described partitioner and scheduler, in terms of the quality of partitioning and allocation that it produces. To quantify the results, we measure the communication overhead in terms of the total data traffic generated and the load balance in terms of a factor that measures the deviation from perfect load balance. We also compare the results with those using the straightforward column wrap assignment scheme. For this purpose, we have used some of the representative test matrices from the Harwell-Boeing package [4]. These test matrices were partitioned and the work units were scheduled as described in the previous section. Using this output, simulations were carried out to get the performance results presented here.

Application	No. of eqns.	No. of non-zeros	No. of non-zeros in factor	Description
BUS1138	1138	2596	3304	Symmetric structure of power system networks
CANN1072	1072	6758	20512	Symmetric pattern from Cannes, Lucien Marro
DWT512	512	2007	3786	Symmetric submarine frame from Naval Ship Research and Development Center
LAP30	900	4322	16697	Symmetric matrix representing 9-point discretization of the Laplacian on the unit square w/ Dirichlet boundary conditions
LSHP1009	1009	3937	18268	Symmetric matrix from Alan George's LSHAPE probs.

Table 1: Selected Harwell-Boeing Test Matrices

For all the results presented in this section, the test matrices were ordered using Liu's modified multiple minimum degree ordering scheme [10]. We used some of the tools

from SPARSKIT [13] and the Wisconsin Sparse Matrix Manipulation System [1] for converting the test matrices into various formats, and for ordering and symbolically factoring the matrices. Table 1 describes the Harwell-Boeing test matrices which were used in our experiments.

In the following, we first quantify the communication and work load distribution aspects of the partitioning schemes. Note that here we are concerned with the quality of the partitioner/scheduler in distributing the work among the processors and hence do not take into account data dependency delays. In practice, the total execution may be affected by the dependency delays as well. However, if the number of processors is relatively small compared to the number of schedulable units, then the allocation scheme described here provides enough parallelism to keep the idle time to a minimum.

The communication cost is parameterized by the total data traffic generated in the system and the mean data traffic per processor. The data traffic is defined as a count of all the non-local data accesses. Accessing a single non-local element constitutes a unit data traffic irrespective of the location from where it is fetched. Once a data element is fetched, that element is stored locally and subsequent usage of that element in the local computations does not add to the data traffic. The total data traffic in the system is the sum of the data accesses by all the processors in the system. This figure represents the volume of the data that must be transmitted by the system during the entire factorization step.

The work load distribution of a partitioning scheme is characterized as follows. The computation cost of updating an element of the matrix by a pair of off-diagonal elements is assumed to be two units; updating the element by the diagonal element is assumed to cost one unit. The computational work assigned to a processor is the sum of the computation costs of all the elements assigned to that processor. The quality of the work load distribution for a partitioning scheme is measured in terms of the load imbalance resulting from the assignment of the work to the processors. The *load imbalance factor* is defined as,

$$\lambda = \frac{(W_{max} - W_{ave}) * N}{W_{tot}},$$

where W_{tot} is the total work, N is the number of processors, $W_{ave} = W_{tot}/N$ is the average work and W_{max} is the maximum work assigned to any processor. Note that when the load is perfectly distributed, W_{max} is W_{ave} and λ is zero. The load imbalance factor can be related to the efficiency e, which is the ratio of speedup to number of processors, where speedup is the ratio of sequential time to parallel time. In the case of zero idle times due to dependency delays, the parallel time is simply the amount of computational work in the processor with the maximum work. The efficiency can then be expressed as,

$$e = \frac{W_{tot}}{W_{max} * N} = \frac{W_{ave}}{W_{max}}$$

which gives us

$$\lambda = \frac{(W_{max} - W_{ave}) * N}{W_{tot}} = \frac{W_{max} - W_{ave}}{W_{ave}} = \frac{1}{e} - 1$$

Table 2 gives the communication traffic in the block scheme for two cases respectively: when the grain size is 4 and when the grain size is 25.

Appl.	P	Total		Mean	
		g=4	g=25	g=4	g=25
BUS	4	1335	1194	334	298
	16	1818	1567	114	98
	32	1910	1649	60	103
CANN	4	47545	40716	11886	10179
	16	138453	80334	8653	5021
	32	171965	89042	5374	2783
DWT	4	5336	3768	1334	942
	16	10328	5482	645	342
	32	11305	5950	353	185
LAP	4	38424	29382	9606	7346
	16	100012	44738	6251	2796
	32	113717	48863	3554	1527
LSHP	4	42044	29899	10511	7475
	16	106973	57773	6686	3611
	32	127612	60243	3988	1883

Table 2: Block mapping communication.

Recall that the grain size is the minimum number of elements in any triangular or rectangular partition. In both cases, total communication increases with the number of processors for all the test problems. However, when the grain size is increased from 4 to 25, there is a significant reduction in communication. For instance, in the LAP30 problem, the $g = 4$ and $g = 25$ columns for total communication in table 2 show that there is more than 50% reduction in the total communication for $p = 16$ and $p = 32$. This is due to the fact that as the block size increases, more work is done in each block with a lot of re-use of data.

Table 3 describes the work distribution in the block scheme for grain sizes 4 and 25. In contrast to the reduction in communication with higher grain size, in most cases, there is an increase in load imbalance. Furthermore, the load imbalance factor λ increases, in general, with the number of processors, as well.

Overall, the larger the grain size, the smaller is the communication, at the cost of larger load imbalance. If the application is run on a system with high communication cost as compared to computation cost, the block-based partitioning can give good performance i.e. the savings in communication will be more than offset the disadvantage of load imbalance. Also, the load balance can be improved by using more sophisticated strategies to allocate blocks to processors.

Appl.	Procs.	Work Distribution		
		Mean	λ	
			g=4	g=25
BUS	4	2791	0.77	0.8
	16	698	3.59	3.59
	32	349	6.3	6.3
CANN	4	151460	0.07	0.122
	16	37865	0.13	0.62
	32	18932	0.38	1.26
DWT	4	11701	0.17	0.18
	16	2925	1.14	1.37
	32	1462	1.48	3.67
LAP	4	108644	0.12	0.16
	16	27161	0.13	1.13
	32	13581	0.48	2.9
LSHP	4	125392	0.06	0.24
	16	31348	0.25	0.74
	32	15674	0.24	2.04

Table 3: Block mapping work distribution.

Apart from grain size, another parameter used in the tests was the minimum cluster width. For instance, if the minimum cluster width is 4, no strip of columns less than four columns wide is acceptable as a cluster - it is broken up into individual columns. The larger the minimum width acceptable, the fewer number of non-single-column clusters there are. For any problem, if the cluster width is set high enough, we end up with all single columns. The results of table 2 and table 3 were obtained using a minimum cluster width of four.

Table 4 shows the variation of communication and load distribution with minimum cluster width for LAP30. The table shows an increase in communication when the width goes from 2 to 4 and then a decrease when the width goes to 8. Load imbalance shows a complementary behavior. It decreases when the width goes from 2 from 4 and then increases when the width goes from 4 to 8. The cluster width has to go in step with the grain size. If the cluster width is too small compared to the grain size,

a large number of skinny clusters would be formed towards the left of the matrix. The blocks would not have enough matrix elements to take advantage of reduction in communication offered by the large grain size.

Width	P	Communication		Work Distr.	
		Total	Mean	Mean	λ
2	4	38936	9734	108644	0.03
	16	96235	6015	27161	0.167
	32	111519	3485	13580	0.54
4	4	38424	9606	108644	0.12
	16	100012	6251	27161	0.13
	32	113717	3554	13580	0.48
8	4	32569	8142	108644	0.62
	16	88408	5526	27161	1.35
	32	101725	3179	13580	2.3

Table 4: Variation with width for LAP30, g = 4.

Table 5 presents the results for the wrap-mapping case. The immediately noticeable property is the consistently uniform load distribution, as seen by the λ column. However, a smaller grain size in the block scheme gives a two-fold advantage of decrease in communication without too much load imbalance as compared to wrap-mapping. For instance, consider the CANN1072 problem with 32 processors. For a grain size of four, the block case provides a 28% saving in communication in going from wrap mapping to the block scheme while the load imbalance factor goes from 0.14 to 0.38, whereas when the grain size is 25, the savings in communication over wrap-mapping is 63% while the load imbalance factor goes from 0.14 to 1.26.

5 Conclusions

In this paper, we have described a block based, automatic partitioning and scheduling scheme for factoring sparse matrices on message passing systems. The primary focus is towards automating the process so that the tedious task of manual parallelization is kept to a minimum. The partitioner makes use of data locality to reduce communication overhead and at the same time attempts to provide the necessary flexibility to the scheduler in manipulating the work allocation so that the load remains balanced. We have used the example of Cholesky factorization to describe the methodology. However, it can very easily be adapted to other factoring methods used in sparse matrix computations. In fact, it can be generalized to computations that can be

Appl.	P	Communication		Work Distr.	
		Total	Mean	Mean	λ
BUS	1	0	0	11164	0
	4	2485	621	2791	0.02
	16	3705	231	698	0.12
	32	3832	120	349	0.35
CANN	1	0	0	605840	0
	4	52363	13090	151460	0.01
	16	171764	10735	37865	0.05
	32	239646	7489	18932	0.14
DWT	1	0	0	46804	0
	4	7599	1900	11701	0.02
	16	17867	1117	2925	0.26
	32	20990	656	1462	0.32
LAP	1	0	0	434577	0
	4	42663	10665	108644	0.01
	16	133720	8357	27161	0.06
	32	177625	5551	13580	0.11
LSHP	1	0	0	501570	0
	4	46347	11586	125392	0.01
	16	146322	9145	31348	0.09
	32	192977	6031	15674	0.24

Table 5: Wrap mapping.

represented as directed acyclic graphs with sufficient information prior to performing the computations.

To analyze the effects on the performance of the partitioning and scheduling technique used, we have compared the communication overhead in the form of total data traffic with that obtained from an implementation where a straightforward column wrap scheme is used. Five representative test matrices from the Harwell-Boeing package were used for this purpose. The comparison shows that the block-based scheme results in a significant reduction in the communication overhead as compared to the wrap-mapping scheme. This is in agreement with our motivation for blocking. On the other hand, the block method results in more load imbalance. Wrap-mappings usually lead to processors communicating with a large number of other processors leading to a large amount of data traffic and possibly to hot-spots. However, in block-based schemes, most of the communication among blocks occur within a cluster and hence can mostly be confined to small groups of processors. Although the increased load imbalance is a serious issue, the provision of the parameters such as the grain size and the cluster widths allows one to minimize the load imbalance for particular applications. Further study of the structure of the sparse matrices is required to optimize these parameters for individual applications. Moreover, in real applications factoring is only a part of the overall solution of the system and other computations such as triangular solves can provide additional flexibility in the balancing the load which is not taken into account here. Finally, more sophisticated scheduling strategies could be used to improve performance. Thus, for systems such as message passing architectures, where communication overhead is much more expensive than computation, automated, block-based methods such as the one described here may prove to be better alternatives.

Acknowledgements

We would like to thank Bob Voigt and Joel Saltz at ICASE for reading the report carefully and making suggestions to improve its presentation.

References

[1] F. L. Alvarado, The Sparse Matrix Manipulation System Users Manual. Technical Report, University of Wisconsin, Madison, 1990.

[2] E. Anderson and Y. Saad, Solving Sparse Triangular Linear Systems on Parallel Computers. CSRD Report No. 794, Center for Supercomputing Development, University of Illinois, 1988.

[3] I. S. Duff, A. M. Erisman, and J. K. Reid, Direct Methods for Sparse Matrices. Oxford Science Publications, Clarendon Press, 1986.

[4] I. S. Duff, R. Grimes, and J. Lewis, Sparse Matrix Test Problems. ACM Transactions on Mathematical Software, Vol. 15, No. 1, pp. 1-14, 1989.

[5] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, Solving Problems on Concurrent Processors: Vol. 1 - General Techniques and Regular Problems. Prentice Hall, 1988.

[6] G. A. Geist and E. Ng, Task Scheduling for Parallel Sparse Cholesky Factorization. Int. Journal of Parallel Programming, Vol. 18, pp. 291-314, 1989.

[7] A. George and J. W. Liu, Computer Solution of Large Sparse Positive Definite Systems. Prentice-Hall, 1981.

[8] A. George, M. Heath, J. W. Liu, and E. Ng, Solution of Sparse Positive Definite Systems on a Hypercube. Journal of Computational and Applied Math., Vol. 27, pp. 129-156, 1989.

[9] G. H. Golub and C. F. Van Loan, Matrix Computations. The Johns Hopkins University Press, 1983.

[10] J. W. H. Liu, Modification of Minimum Degree by Multiple Elimination. ACM Transactions on Mathematical Software, Vol. 11, 1985, pp. 141-153.

[11] V. Naik and M. Patrick, Data Traffic Reduction Schemes for Cholesky Factorization on Asynchronous Multiprocessor Systems. Proceedings of the 1989 International Conference on Supercomputing, ACM, Crete, Greece, 1989. Also available as IBM Research Report RC 14500, 1989.

[12] R. Schreiber and J. J. Dongarra, Automatic Blocking of Nested Loops. Technical Report CS-90-108, Computer Science Department, University of Tennessee, 1990.

[13] Y. Saad, SPARSKIT: a Basic Tool Kit for Sparse Matrix Computations. Technical Report 90-20, RIACS, NASA Ames Research Center, 1990.

[14] P. Sadayappan and S. K. Rao, Communication Reduction for Distributed Sparse Matrix Factorization on a Processor Mesh. Proceedings of Supercomputing'89, pp. 371-379, 1989.

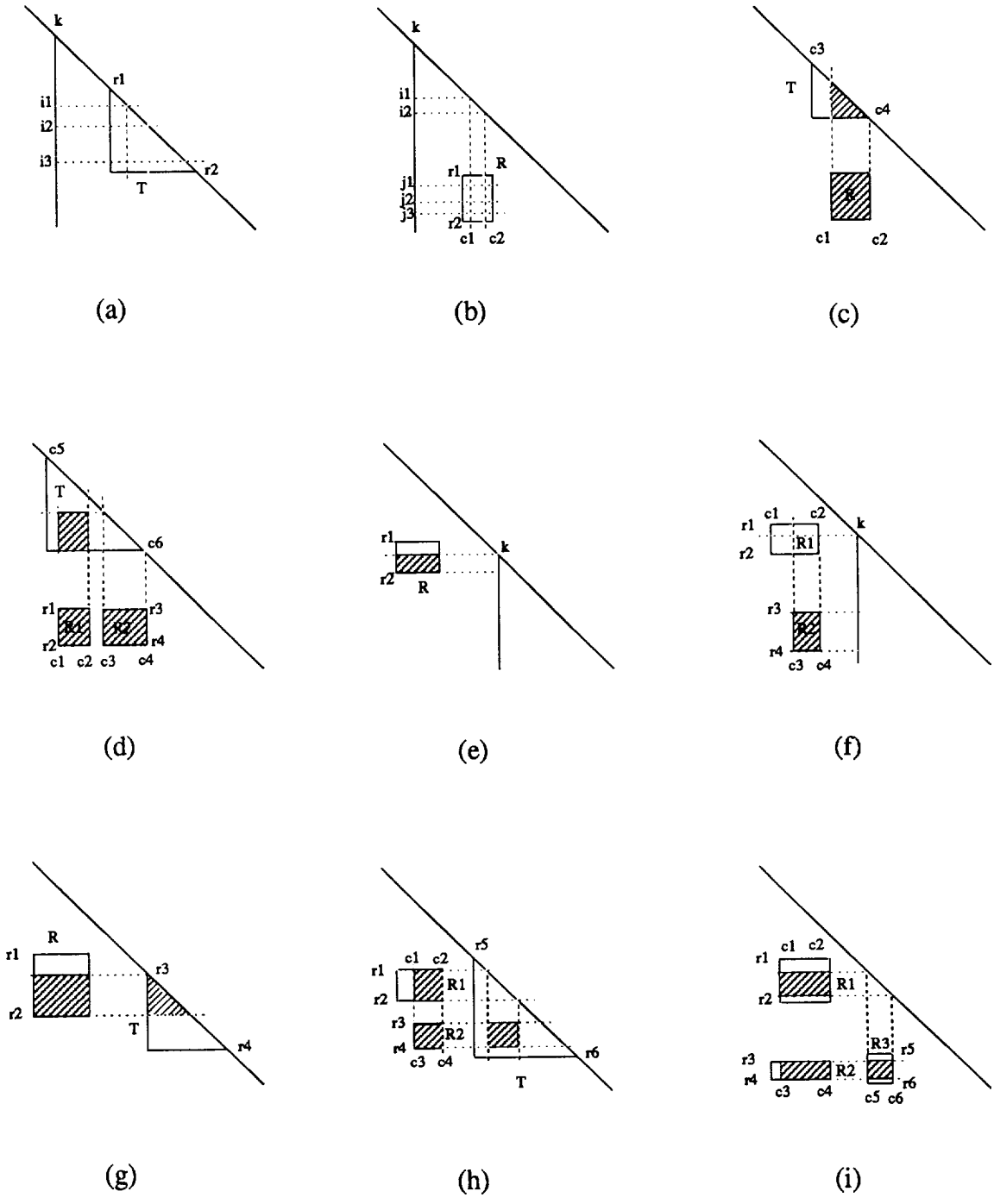

Figure 4: Dependencies

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1991	Contractor Report

4. TITLE AND SUBTITLE
EFFECTS OF PARTITIONING AND SCHEDULING SPARSE MATRIX FACTORIZATION ON COMMUNICATION AND LOAD BALANCE

5. FUNDING NUMBERS
NAS1-18605
505-90-52-01

6. AUTHOR(S)
Sesh Venugopal
Vijay K. Naik

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 91-80

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA CR-189563
ICASE Report No. 91-80

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report

To appear in Proceedings of Supercomputing 1991.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

We present a block-based, automatic partitioning and scheduling methodology for sparse matrix factorization on distributed memory systems. Using experimental results, we analyze this technique for communication and load imbalance overhead. To study the performance effects, we compare these overheads with those obtained from a straightforward "wrap-mapped" column assignment scheme. All experimental results were obtained using test sparse matrices from the Harwell-Boeing data set. The results show that there is a communication and load balance trade-off. The block-based method results in lower communication cost whereas the wrap-mapped scheme gives better load balance.

14. SUBJECT TERMS	15. NUMBER OF PAGES
load balancing; sparse matrices; partitioning; communication	20
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std 239-18
298-102

NASA-Langley, 1991