Proc. 6th Int. Conference on Parallel Architecture and Languages, PARLE'94, Athens, Greece, July 1994, LNCS 817, Springer-Verlag, Berlin, 1994, pp. 313-22

Run-Time Optimization of Sparse Matrix-Vector Multiplication on SIMD Machines

Louis H. Ziantz, Can C. Özturan, and Boleslaw K. Szymanski

Department of Computer Science, Rensselaer Polytechnic Institute
Troy, New York 12180-3590 USA

Abstract. Sparse matrix-vector multiplication forms the heart of iterative linear solvers used widely in scientific computations (e.g., finite element methods). In such solvers, the matrix-vector product is computed repeatedly, often thousands of times, with updated values of the vector until convergence is achieved. In an SIMD architecture, each processor has to fetch the updated off-processor vector elements while computing its share of the product. In this paper, we report on run-time optimization of array distribution and off-processor data fetching to reduce both the communication and computation time. The optimization is applied to a sparse matrix stored in a compressed sparse row-wise format. Actual runs on test matrices produced up to a 35 percent relative improvement over a block distribution with a naive multiplication algorithm while simulations over a wider range of processors indicate that up to a 60 percent improvement may be possible in some cases.

1 Introduction

Sparse matrix operations are difficult to parallelize or optimize at compile time because the structure of the involved computation and communication is not established until execution (i.e., until the positions of nonzero entries in the matrix are known). Yet, in many scientific computations, sparse matrix operations are deeply nested in loops making them a convenient target for parallelization. An example of such an operation is sparse matrix-vector multiplication which is used in iterative solvers for linear equations. Such solvers repeatedly compute a matrix-vector product, and thousands of repetitions may be required to converge to a solution. Linear equations arising in finite element meshes generate sparse matrices. Hence, this computation is often encountered in the finite element method (FEM) implementation of scientific and engineering modeling. In this paper, we describe algorithms for run-time optimization of iterative sparse matrix-vector multiplication on SIMD machines.

Let $\mathbf{A}$ be an $n \times n$ sparse matrix that is block-distributed in a compressed sparse row-wise format [11] over the processors of an SIMD machine. Let $\mathbf{x}$ be a vector aligned with the columns of $\mathbf{A}$ and iteratively updated according to the equation $\mathbf{x}_{j+1} = \mathbf{A}\mathbf{x}_j$. The considered sparse matrix vector multiplication $\mathbf{A}\mathbf{x}_j$ is nested in a loop which is exited when a norm $\|\mathbf{x}_{j+1} - \mathbf{x}_j\|$ satisfies certain convergence criteria. For such a computation, we consider two interrelated issues. First, we investigate different levels of preprocessing that can be done outside the loop on off-processor references to $\mathbf{x}_j$ vector elements. The goal of such preprocessing is to minimize the overhead incurred by off-processor references during the multiplication (i.e., inside the loop). Next, we present a parallel heuristic for redistributing matrix $\mathbf{A}$ and the aligned vector $\mathbf{x}$ among the individual processors to improve the total execution time.

The new generation of parallelizing Fortran compilers, e.g., Fortran D [7], Vienna Fortran [19] and High Performance Fortran [1], provide two classes of directives for the layout of arrays:

- *Alignment directives* that describe how arrays should be aligned with respect to one another, both within and across array dimensions.
- *Distribution directives* that define how arrays should be broken up and distributed onto the processors of the target architecture.

For dense matrices, alignment and distribution of arrays depends on the underlying *fine* grain computation done on the individual array elements. The access patterns of the array elements are dictated by subscript expressions. Hence, optimization of alignments can be done at compile time by analyzing these expressions. Because of the use of indirection in subscript expressions, such an analysis is useless for sparse matrix computations.

The distribution directives **BLOCK** and **CYCLIC** provided by Fortran compilers are not sufficient for effective run-time distribution of sparse matrices. Vienna Fortran [19] and Fortran D [7] employ *mapping arrays* and corresponding routines to support arbitrary distributions of sparse matrices. The mapping array has the same size as the mapped matrix and each location holds the identifier of the processor that stores the matrix element with the corresponding index. Hence, mapping arrays incur a heavy storage cost while still leaving the task of computing the mapping array to the user who has to design a partitioning algorithm.

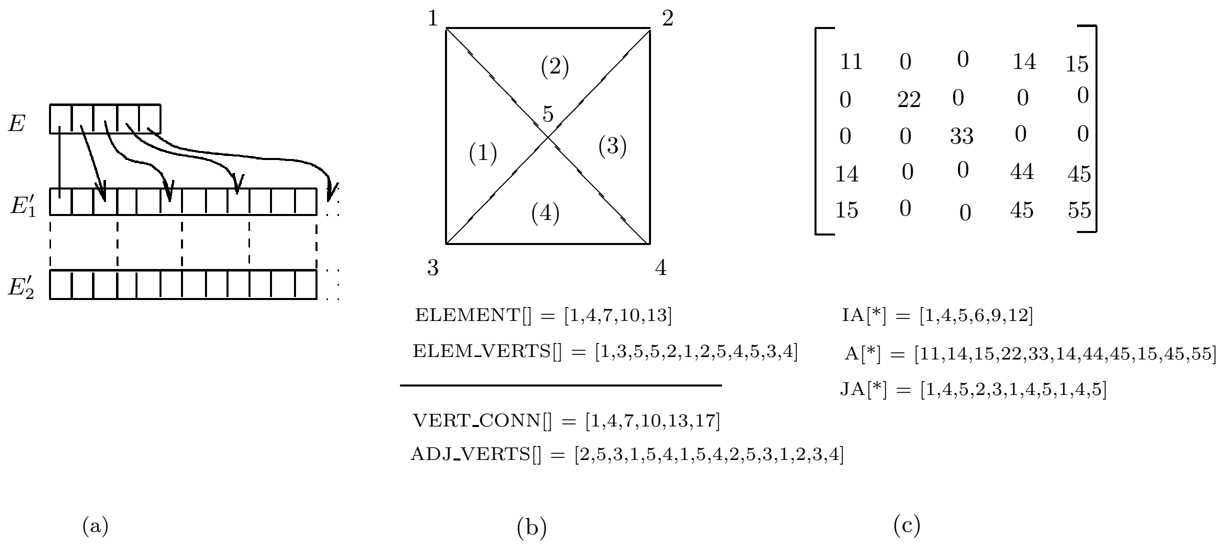
We believe that effective support of run-time optimization for sparse matrices can be provided through libraries as illustrated by the PARTI primitives [16, 18]. We hope that the algorithms presented in this paper will enlarge the repertoire of run-time distribution algorithms available to the users of SIMD machines.

The paper is organized as follows. Section 2 describes array data structures commonly used to represent FEM meshes and sparse matrices in Fortran programs. There is also a discussion of the cost of the sparse matrix vector multiplication as a function of the number of on- and off-processor references and the communication-to-computation ratio. Section 3 compares several implementations of such multiplication on SIMD machines. It also describes a parallel heuristic for redistributing the underlying sparse matrix in order to optimize the performance of the matrix-vector multiplication. The results from implementations and simulations of the multiplication and the heuristic on benchmark test meshes are also given. Concluding remarks are presented in Section 4.

The preprocessing of off-processor references follows the methods proposed in [9] for MIMD machines. The heuristics for run-time redistribution of sparse matrices on SIMD machines is an original contribution of this paper. The total execution time improved by up to 35 percent relative to a naive multiplication method using a uniform block distribution when the heuristic and preprocessing was run on a MasPar MP-1 with 2048 processors. Some simulated runs with a smaller number of processors showed up to a 60 percent relative improvement based on estimated execution times of multiplication using a block distribution versus multiplication using the heuristic distribution.

2 Preprocessing and Data Distribution of Sparse Matrices

Finite element computations involve various mesh related *entities* such as elements, edges, nodes or node connectivities. To represent these entities in an array-based language like Fortran, one-dimensional arrays pointing to other entities via indirection are often used. An example of such a representation is given in Figure 1(a).



$$\begin{bmatrix} 11 & 0 & 0 & 14 & 15 \\ 0 & 22 & 0 & 0 & 0 \\ 0 & 0 & 33 & 0 & 0 \\ 14 & 0 & 0 & 44 & 45 \\ 15 & 0 & 0 & 45 & 55 \end{bmatrix}$$

ELEMENT[] = [1,4,7,10,13]

ELEM_VERTS[] = [1,3,5,5,2,1,2,5,4,5,3,4]

VERT_CONN[] = [1,4,7,10,13,17]

ADJ_VERTS[] = [2,5,3,1,5,4,1,5,4,2,5,3,1,2,3,4]

IA[*] = [1,4,5,6,9,12]

A[*] = [11,14,15,22,33,14,44,45,15,45,55]

JA[*] = [1,4,5,2,3,1,4,5,1,4,5]

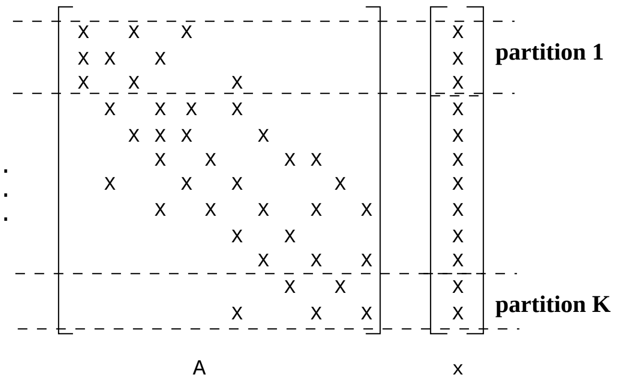
Fig. 1. (a) The data structure used in a typical finite element program; (b) example specifying vertices making up the elements and the adjacency information for the vertices; (c) example from ITPACK showing compressed sparse row-wise format.

With this kind of a data structure, an entity array $\mathbf{E}$ points to one or more entity arrays such as $\mathbf{E}'_i$. For example, Figure 1(b) shows how various pieces of mesh information can be represented in terms of such data structures. Here, $\mathbf{E}$ might be the ELEMENT array and it might point to another entity array $\mathbf{E}'_i$ which stores the element vertices. Other examples of formats for the storage of sparse matrices may be found in [11]. In a compressed sparse row-wise format, a sparse matrix $\mathbf{A}$ is represented as a vector of its nonzero elements $\mathbf{VA}[*]$, the corresponding vector of column indexes of these elements $\mathbf{JA}[*]$, and the vector storing the cumulative number of nonzeroes in each row of the matrix, $\mathbf{IA}[*]$. The range of vector $\mathbf{IA}$ is the number of rows and columns of the sparse matrix, i.e., n. The range of vectors $\mathbf{VA}$ and $\mathbf{JA}$ is the number of nonzero entries in the matrix $\mathbf{A}$ and is denoted by m. Usually, $m \ll n^2$. If for some i, j an inequality $\mathbf{IA}[i] \leq j < \mathbf{IA}[i+1]$ holds, then the j-th nonzero element is equal to $\mathbf{VA}[j]$ and it is located in i-th row and $\mathbf{JA}[j]$-th column of the sparse matrix $\mathbf{A}$.

In this paper, we consider the problem of distributing sparse matrices with the above representation over parallel machines. In Fortran programs, the $\mathbf{JA}$ array is usually accessed via an indirection in a loop. As an example, consider the Fortran code (Figure 2(a)) for matrix-vector multiplication taken from ITPACK [10] which is used in many sparse linear solvers. If both $\mathbf{VA}$ and $\mathbf{JA}$ are distributed by rows and aligned with the multiplied vector $\mathbf{x}$, then each processor is responsible for multiplying the rows allocated to it. Usually matrix-vector multiplication is part of an iterative algorithm where the product $\mathbf{Ax}$ is computed repeatedly with updated values of $\mathbf{x}$. Hence, the off-processor vector elements should be fetched at the beginning of each multiplication step. Depending on the size of a problem being solved, the steps of fetching and computing can be repeated thousands of times before convergence is achieved. Thus, an optimized array distribution capable of reducing both the volume of fetched data and the computing time is important. We are particularly interested in reducing the *combined* cost of fetches and computation for each processor.

```
      SUBROUTINE PMULT (...)
             ....
       DO 30 I = 1,N
          IBGN = IA(I)
          IEND = IA(I+1)-1
          SUM = 0.0D0
          IF (IBGN.GT.IEND) GO TO 20
             DO 10 J = IBGN,IEND
               JAJJ = JA(J)
               SUM = SUM+A(J)*X(JAJJ)
10        CONTINUE
20        W(I) = SUM
30     CONTINUE
       RETURN
```

(a)



(b)

Fig. 2. (a) ITPACK matrix-vector multiplication code and (b) ordered array partitioning.

To formulate the cost function for the optimization we will use the following definitions. We assume that there are k processors. The sparse $n \times n$ matrix $\mathbf{A}$ and n-element vector $\mathbf{x}$ are distributed among them in blocks of rows. Processor i is assigned consecutive sparse matrix rows numbered f_i through $f_{i+1}-1$ and the corresponding vector elements (for notational convenience, we assume that $f_{k+1} = f_{k+2} = n+1$). The number of nonzero elements assigned to the processor is denoted by $E(i)$ and is equal to $\mathbf{IA}[f_{i+1}] - \mathbf{IA}[f_i]$. The number of nonzero elements whose corresponding vector element is off-processor is denoted by $G(i)$. To define $G(i)$, let $\boldsymbol{\alpha}[r_1 : r_2, s]$ be 1 if column s contains any nonzero elements of $\mathbf{A}$ in the rows r_1 to r_2, and 0 otherwise. Then,

$$G(i) = \sum_{l=f_i}^{f_{i+1}-1} \left(\sum_{j=1}^{f_i-1} \boldsymbol{\alpha}[l : l, j] + \sum_{j=f_{i+1}}^{n} \boldsymbol{\alpha}[l : l, j] \right) .$$

Since once the off-processor vector element is fetched, it could be stored and reused without involving communication, the number of distinct off-processor fetches that the processor i has to make is

$$D(i) = \sum_{j=1}^{f_i-1} \boldsymbol{\alpha}[f_i : f_{i+1}-1, j] + \sum_{j=f_{i+1}}^{n} \boldsymbol{\alpha}[f_i : f_{i+1}-1, j] \ .$$

The number of distinct off-processor references that are directed to nearest neighbors is denoted $D_{\mathrm{x}}(i)$, where

$$D_{\mathrm{x}}(i) = \sum_{j=f_{i-1}}^{f_i-1} \boldsymbol{\alpha}[f_i : f_{i+1}-1, j] + \sum_{j=f_{i+1}}^{f_{i+2}-1} \boldsymbol{\alpha}[f_i : f_{i+1}-1, j] \ ,$$

while the number of distinct off-processor references that cannot use nearest neighbor communication is denoted $D_{\mathrm{r}}(i)$ and is given by

$$D_{\mathrm{r}}(i) = D(i) - D_{\mathrm{x}}(i) \ .$$

The execution cost of the sparse matrix-vector multiplication is a function of 1) the amount of computation, 2) the volume and organization of communication needed to resolve non-local data references, and 3) the communication-to-computation ratio, which we will denote by c. Depending on the communication organization, the following two cases can be distinguished for SIMD machines:

1. **SIMD machine with router communication.** Since the communication and computation are synchronized in a lockstep, the processor with the maximum number of data fetches will dictate communication cost and the processor with the maximum number of nonzero elements will define the computation cost:

$$C = c * \max_{1 \le i \le k} D(i) + \max_{1 \le i \le k} E(i) \ . \tag{1}$$

2. **SIMD machine with mesh (xnet) and router communication.** This is a variant of the previous case in which the communication between neighbors uses fast xnet communication:

$$C = c_{\mathrm{x}} * \max_{1 \le i \le k} D_{\mathrm{x}}(i) + c * \max_{1 \le i \le k} D_{\mathrm{r}}(i) + \max_{1 \le i \le k} E(i) \tag{2}$$

   where $c_{\mathrm{x}} < c$ is the xnet communication-to-computation ratio.

For each of the above defined cost functions C, our goal is to find a sequence $1 \le f_1 \le \ldots \le f_k \le n$ for which this function reaches the minimum.

The partitioning problem defined above assumes that the order of rows is preserved in matrix $\mathbf{A}$. Hence, this is a simpler and more restricted problem than an arbitrary partitioning during which the rows can be permuted. This assumption is justified when the sparse matrix has already been permuted by either a straightforward or an advanced scheme like reverse Cuthill and McKee [4]. In the literature (c.f. [3] and [12]), algorithms have been presented for a similar problem involving *only* the computational load as the cost function. The redistribution heuristic presented in this paper generalizes the cost function to include communication costs.

3 Implementation and Results

This section describes different versions of the multiplication implementation and the redistribution heuristics. The versions were run on several benchmarks including meshes originally used by Hammond [8] and test cases from the Harwell–Boeing Sparse Matrix Collection [5, 6]. The characteristics of the tests are given in Table 1.

The first test case is an unstructured triangular mesh around a 3-component airfoil, while the second test is a portion of a larger mesh representing an unstructured tetrahedral mesh about a Lockheed S-3A Viking aircraft. The third test case arises from a mixed kinetics diffusion problem (specifically, the study of ionization in the stratosphere with 38 chemical species). The fourth mesh is derived from a model of a gas-cooled nuclear reactor core, and the fifth case was generated using a package for reservoir modeling.

Table 1. Description of test meshes and matrices.

Test Case	number of rows	number of nonzeroes	max nonzeroes per row	ave nonzeroes per row
Hammond Test Meshes				
1. 3elt	4720	27444	9	5.8
2. wave.6000	6000	73734	21	12.3
Harwell-Boeing Test Meshes				
3. rua-fs_760_1	760	5976	21	7.9
4. rua-nnc1374	1374	8606	16	6.3
5. rua-pores_2	1224	9613	30	7.8

3.1 Preprocessing

The most straightforward implementation of the sparse matrix-vector multiplication shown in Figure 2(a) is to multiply each nonzero element by the corresponding vector element, fetching it through communication, if necessary. Such a solution would be inefficient for SIMD machines because of the tight synchronization of statement execution. Most likely, each elementary multiplication executed on a given processor would be delayed by some processor that has to fetch its off-processor argument. Hence, each step of elementary multiplication would likely include communication overhead. The cost function of such an implementation can therefore be approximated as

$$C \approx (1+c) * \max_{1 \le i \le k} E(i) \ .$$

Thus, it will be higher than the cost functions given by (1) or (2). An improvement over this implementation is to reorder the sequence of elementary multiplications performed by each processor. All elementary multiplications that require communication are executed first. Each off-processor vector value needed by an elementary multiplication is fetched and stored in a local variable. The elementary multiplications of both local arguments are executed next. Hence, only $\max_{1 \le i \le k} G(i)$ elementary multiplications are delayed by the communication. However, the same off-processor vector elements are fetched as many times as they are an argument to an elementary multiplication (i.e., the number of fetches of the same vector element is equal to the number of nonzero elements in the corresponding column section of the sparse matrix). The advantage is the small memory overhead incurred; only an integer vector of the size $G(i)$ is needed. We refer to this implementation as “multiple fetches.” The cost function of this implementation is

$$C = \max_{1 \le i \le k} E(i) + c * \max_{1 \le i \le k} G(i) \ .$$

Preprocessing can eliminate multiple fetches of the same vector element by forcing execution of the needed communication first and storing the received values in a local structure before any elementary multiplication is done. The memory overhead in this case includes an integer vector and a floating point vector, each of the size $D(i)$. The integer vector is needed to define which elements to fetch, and the floating point vector stores the received values. This is the solution characterized by (1). This solution is also used as a basis for the redistribution of the matrix (see the following subsection). We refer to this solution as “distinct fetches.”

Finally, fetches to the neighboring processes can be efficiently executed when fast xnet communication is used instead of a more general but slower router. Therefore, we also implemented a variant of the solution in which the neighboring fetches are executed first using xnet. In such a case, the cost of the communication is dictated by the maximum number of router fetches and the maximum number of xnet fetches to each neighbor destination. Therefore, some of the fetches directed to neighbors might use a router to reduce the total number of communication steps (c.f. formulation of (2)).

The results of benchmarking the discussed alternatives for preprocessing the communication patterns are presented in Table 2.

Table 2. Preprocessing alternatives – execution times of test runs on MP-1 for 1000 iterations.

Method	3elt	wave.6000	rua-fs_760_1	rua-nnc1374	rua-pores_2
	t (s)	t (s)	t (s)	t (s)	t (s)
multiple fetches	33.6	92.8	63.8	59.6	64.5
distinct fetches	32.6	93.4	66.4	42.0	58.9
xnet multiple fetches	33.9	93.2	63.7	42.1	57.7
xnet distinct fetches	32.9	94.2	63.9	41.7	59.5

3.2 Matrix Redistribution

When only the computational load term is present in the cost function, the cost grows monotonically with the partition size. This fact is utilized in [3] and [12] to come up with algorithms of complexity $O(kn)$ and $O(n+(klogn)^2)$, respectively, for the optimal solution. However, when we incorporate the off-processor reference cost into the cost function, this function is no longer monotonical in partition size. A tradeoff between load and communication is included in the formulation. Hence, the cost might decrease as the partition size is increased because of newly localized off-processor references. As a result, the complexity of the solution increases. Based on the algorithm presented in [13], we found an $O(n^2m)$ algorithm for an optimal partitioning of a sparse matrix among k processors for the asynchronous cost function defined as:

$$C = \max_{1 \le i \le k} (E(i) + c * D(i)) \ .$$

The repartitioning must take place at run-time, when the values of a sparse matrix are known. But an algorithm of such complexity is unacceptable for run-time execution of scientific applications in which $m > n$ and n is often in the order of thousands or even millions. The cost functions given by (1) and (2) lead to algorithmically more complex implementations than the algorithm driven by the above cost function. Indeed, suppose that algorithm S solves the problem with the cost function (2). Setting $c_x = c$, we will obtain an algorithm for solving the problem with the cost function (1). If $E(i)$ is replaced by $E(i) + c * G(i)$ and c is set to zero in such a modified algorithm S, then it will solve the case of the asynchronous cost function given above. Hence, we do not expect that an exact solution to any of the discussed cases of the cost functions can be used in practice. Consequently, we developed a fast parallel heuristic for repartitioning a sparse matrix. The heuristic assumes compressed sparse row-wise format and preserves the order of the rows.

Description of Heuristic The heuristic starts with an initial distribution and refines it through parallel processor-pairwise repartitioning. The interconnection network is viewed as a linear array and processors are edge-colored with two colors. One color, which changes from step to step, is selected as active. Each pair of processors connected by the edge of the active color attempts to repartition their rows to improve the cost function using a bi-partition algorithm. Figure 3 gives an example of repartitioning at even and odd stages of the heuristic for six processors. The arrows indicate pairs of processors that are executing the bi-partition algorithm at a particular stage. The dashed lines indicate inter-processor boundaries that cannot be modified during the stage while the dotted lines illustrate boundaries that might be updated due to repartitioning. A single iteration of the heuristic consists of an even stage followed by an odd stage.

Given an arbitrary pair of processors i and $i+1$ with r_f representing the first row of the matrix that is stored on i and r_l representing the last row of the matrix stored on $i+1$, there are $r_\mathrm{l} - r_\mathrm{f} + 2$ possible bi-partitions that preserve the order of the rows within the processor pair. For each possible bi-partition t, the algorithm computes a) $\mathbf{Z}_t[i]$, the number of distinct nonzeroes on i, b) $\mathbf{Z}_t[i+1]$, the number of distinct nonzeroes on $i+1$, c) $\mathbf{L}_t[i]$, the number of distinct nonzeroes that generate local (on-processor) accesses on i, and d) $\mathbf{L}_t[i+1]$, the number of distinct nonzeroes that generate local accesses on $i+1$. This is done as follows (note that $\mathbf{col}[j]$ gives the column of the matrix in which the j-th distinct nonzero element on the processor pair is located):

1. Both the first and the last row in which each distinct element is found in the section of the matrix stored on the processor pair are determined. For the j-th distinct nonzero, these values are given by $\mathbf{first}[j]$ and $\mathbf{last}[j]$, respectively. This computation is done in parallel on i and $i+1$. Values are then merged onto i

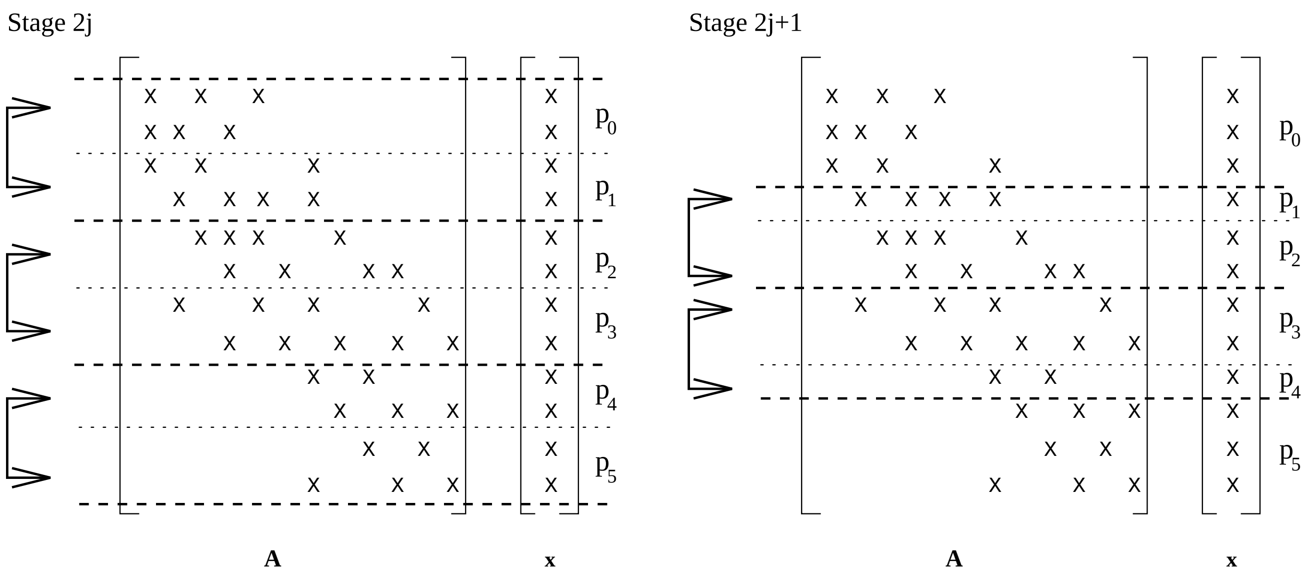


Fig. 3. Example of boundary movements at even and odd stages.

```
    which handles the rest of the bi-partition computation.
2.  For each distinct nonzero $j$,
       add 1 to $\mathbf{Z}_{\mathbf{first}[j]-r_\mathrm{f}+1}[i]$ and to $\mathbf{Z}_{\mathbf{last}[j]-r_\mathrm{f}}[i+1]$
       if $\mathbf{col}[j] \geq r_\mathrm{f}$ and $\mathbf{col}[j] \leq r_\mathrm{l}$ then
          if $\mathbf{col}[j] > \mathbf{first}[j]$ then add 1 to $\mathbf{L}_{\mathbf{col}[j]-r_\mathrm{f}+1}[i]$ else add 1 to $\mathbf{L}_{\mathbf{first}[j]-r_\mathrm{f}+1}[i]$
          if $\mathbf{col}[j] < \mathbf{last}[j]$ then add 1 to $\mathbf{L}_{\mathbf{col}[j]-r_\mathrm{f}}[i+1]$ else add 1 to $\mathbf{L}_{\mathbf{last}[j]-r_\mathrm{f}}[i+1]$
       endif
    end of loop
3.  Cumulatively sum the data in $\mathbf{Z}_t[i]$ and $\mathbf{L}_t[i]$ in order of increasing $t$ values.
4.  Cumulatively sum the data in $\mathbf{Z}_t[i+1]$ and $\mathbf{L}_t[i+1]$ in order of decreasing $t$ values.
```

The number of distinct off-processor references may then be computed for each possible partition t. [1] These are computed as $D_t(i) = \mathbf{Z}_t[i] - \mathbf{L}_t[i]$ and $D_t(i+1) = \mathbf{Z}_t[i+1] - \mathbf{L}_t[i+1]$, respectively. The number of (nondistinct) nonzeroes on i and $i+1$ may also be computed for each possible partition using **IA**. These are denoted by $E_t(i)$ and $E_t(i+1)$.

The actual partition selected is the value of t, say t_p, such that

$$t_p = \max_t\{\min(M_\mathrm{D} - c\max(D_t(i), D_t(i+1)), M_\mathrm{E} - \max(E_t(i), E_t(i+1)))\}$$

where $M_\mathrm{D} = c\max_{1\leq i\leq k} D(i)$ and $M_\mathrm{E} = \max_{1\leq i\leq k} E(i)$ are calculated in the previous step of the heuristic. Note that neither (1) nor (2) can be applied directly in the determination of the bi-partition boundary because both functions are based on data that can only be computed after all of the inter-processor boundaries are set.

The complexity of this bi-partition algorithm on a uniprocessor machine is $O(m)$. Recall that m is the number of nonzeroes in the matrix and, typically, $m \ll n^2$. In a similar fashion, the complexity of a stage of the heuristic is determined by the processor with the heaviest load. Thus, a single stage of the heuristic may be done in $O(M_\mathrm{E})$ steps on a parallel machine. Note that, since the algorithm examines all possible bi-partitions that preserve the order of the rows, it produces an optimal bi-partition for the given pair of processors with respect to a given cost function.

The heuristic iterates and converges to a cost, but there is no guarantee that the resulting cost is a global minimum. It does not perform any probabilistic jump to get out of a possible local minimum.

[1] Note that the value of t ranges from 0 to $r_\mathrm{l} - r_\mathrm{f} + 1$ and corresponds to the number of rows that will reside on i if that t value is selected to set the partitioning boundary.

The node exchange idea can be traced back to Rosen's [15] node ordering algorithm for reducing the *bandwidth* of a sparse matrix. Bokhari [2] used a sequential algorithm based on the same node exchange idea to partition and map FEM meshes onto an array of processors. Hammond [8] used the pairwise exchange heuristic on the massively parallel CM2 system to improve communication.

Execution Results To obtain comparative data we have implemented and timed a matrix-vector multiplication with: (i) block distribution, (ii) computational load balance (assuming consecutive allocation of rows), and (iii) the parallel heuristic. The implementations were done on a MasPar MP-1 with 2048 processors with and without the use of xnet communication. Due to the limited memory size of the available MP-1 configuration and to get an idea about how well the heuristic might work on other SIMD machines, we have also simulated the execution of the heuristic for a small and medium number of processors and for different communication-to-computation ratios.

For each test, we generated a uniform block distribution of the data corresponding to the test and then either directly applied the heuristic or first performed a load-balancing step using a bin-packing algorithm so that each processor would have approximately the same number of nonzeroes. We then calculated a cost for the resulting partition using (1) or (2). This cost was used as the initial cost in the heuristic. The heuristic was then applied iteratively. After each iteration, cost calculation was performed, and the heuristic was repeated until no further reduction in cost was obtained. Although the bin-packing step and the parallel heuristic measured load in terms of the number of nonzeroes, both algorithms only moved entire rows across processors. After the heuristic step was completed, the distribution of the sparse matrix produced was used in the execution of an iterative sparse matrix-vector multiplication routine.

The preliminary results of several runs of the multiplication are given in Table 3. The rows labeled "block" and "load balance" give times for runs of the multiplication with only a block distribution and the load-balancing step performed, respectively. Using the time taken to do a double multiply-and-add as the computation cost and the time to perform a fetch of a double float as the communication cost, the communication-to-computation ratio for the MP-1 was determined to be approximately 3, and this is the c value that was used in the heuristic for all runs. Results from executions presented in Table 3 showed up to a 35 percent relative improvement over the naive multiplication method on the MP-1.

Table 3. Execution times of test runs on MP-1 for 1000 iterations.

Method	3elt	wave.6000	rua-fs_760_1	rua-nnc1374	rua-pores_2
	t (s)	t (s)	t (s)	t (s)	t (s)
Multiplication using router fetches only.					
block, multiple fetches	33.6	92.8	63.8	59.6	64.5
load balance, multiple fetches	28.5	79.8	50.5	52.8	55.3
load balance, distinct fetches	28.6	86.2	52.3	39.6	52.4
heuristic from block	32.5	91.8	67.6	37.9	47.7
heuristic from load balance	28.7	88.1	53.6	37.9	47.7
Multiplication using router and xnet fetches.					
load balance, distinct fetches	29.1	82.6	51.5	40.4	53.3
heuristic from block	33.6	95.8	65.7	39.0	46.7
heuristic from load balance	28.2	84.3	52.5	38.9	47.0
((block-heuristic)/block)*100 %	16.0 %	9.2 %	17.7 %	36.4 %	27.6 %

3.3 Simulation for a Range of Communication Speeds

To further evaluate the performance of the heuristic (and due to memory limitations on our MP-1), it was simulated on a sequential machine using the router fetch SIMD cost function (1) for a range of communication speeds and available processors. For each test case, we first generated a uniform block distribution of the matrix and other information associated with the test. A load-balancing step was then performed to assure that the nonzeroes were more evenly distributed across the processors. An initial cost for the heuristic was generated by calculating the maximum number of nonzeroes and the maximum number of distinct nonlocals on any processor and using (1). As in the implemented version, the parallel heuristic was applied iteratively. After each iteration, cost calculation was performed and the heuristic was repeated until no further reduction in cost was obtained.

The tests were run for communication-to-computation ratios ranging from 3 to 40, the range suggested as the most typical for the the computer architectures of today and the next decade [14, 17]. The number of processors simulated ranged from 16 to 2048 (or to 512 for smaller meshes).

In the following subsections, we discuss various plots from the simulation runs. Note that only results from four of the test cases are shown, and that the independent axis (the number of processors) uses a logarithmic scale for each graph.

Estimated Execution Time vs. Number of Processors Figure 4 gives plots of the estimated execution time versus the number of processors for the last four test meshes. A communication-to-computation ratio of 3 was used in the simulation of the heuristic for these test cases. Estimated times were calculated for each mesh as follows: 1) the actual timings from the MP-1 for the block, load-balance, and heuristic from load-balance distributions (c.f. Tables 2 and 3) were divided by the final costs generated by the simulation (for the appropriate number of processors) for the block, load-balance, and heuristic from load-balance distributions, respectively; 2) the mean of the three resulting values was taken; and 3) this average execution-time-per-unit-cost value was multiplied with the final cost values from the simulation to generate expected execution times across a range of processors. Using the mean of the execution time per unit cost is intended to average out the variations in time per cost unit for different distributions; however, since the number of processors used may also affect this ratio, the values given are only intended to be an approximation of run times.

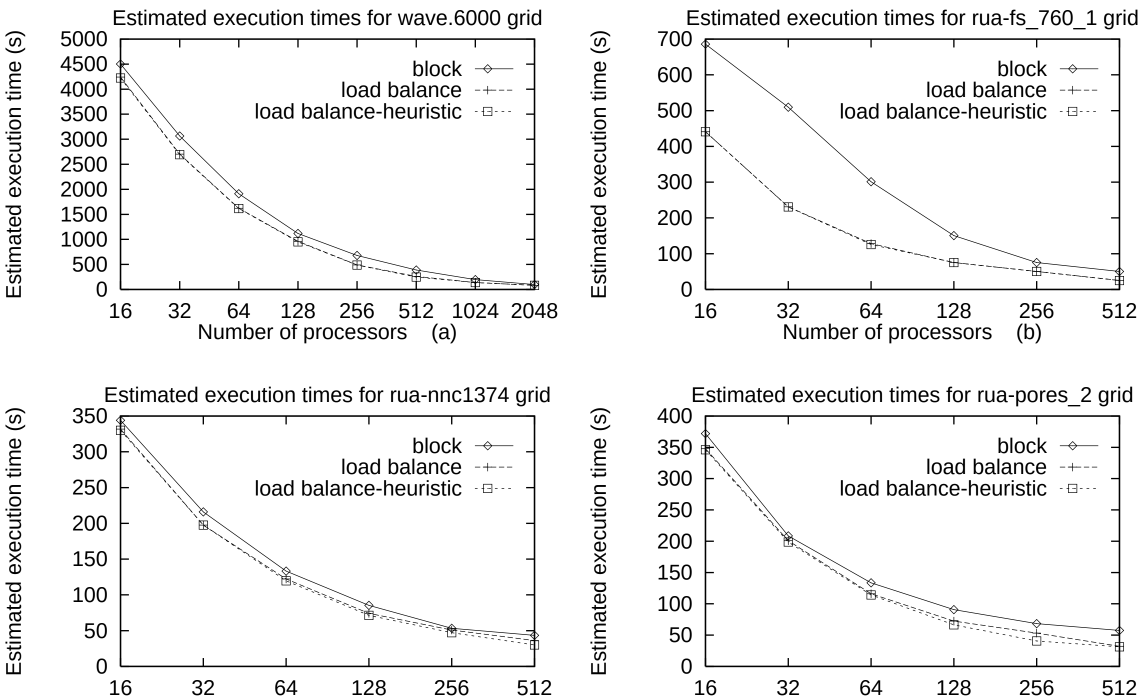


Fig. 4. Plots of estimated times vs. number of processors.

The plots illustrate the estimated reduction in time that would result from balancing the load across the processors and the further reduction that can be achieved using the parallel heuristic afterwards. Simulations of the performance of the heuristic when executed starting with a block distribution indicated that this coupling generally produced results which were sometimes comparable but often inferior to those generated by the heuristic from load balance. Thus, these results have not been included.

Relative Percent Improvements vs. Number of Processors Figure 5 shows the relative percent improvements in estimated execution time over a block distribution for the load balance and heuristic distributions. These values are plotted as a function of the number of processors. These test cases also used a communication-to-computation ratio of 3 in the simulation of the heuristic. The relative percent improvements were found by subtracting the estimated run time of the block distribution by the estimated run time of the appropriate optimizing distribution, dividing this difference by the estimated run time of the block distribution and then multiplying the result by 100 percent. The plots show improvements ranging from 0 to approximately 60 percent.

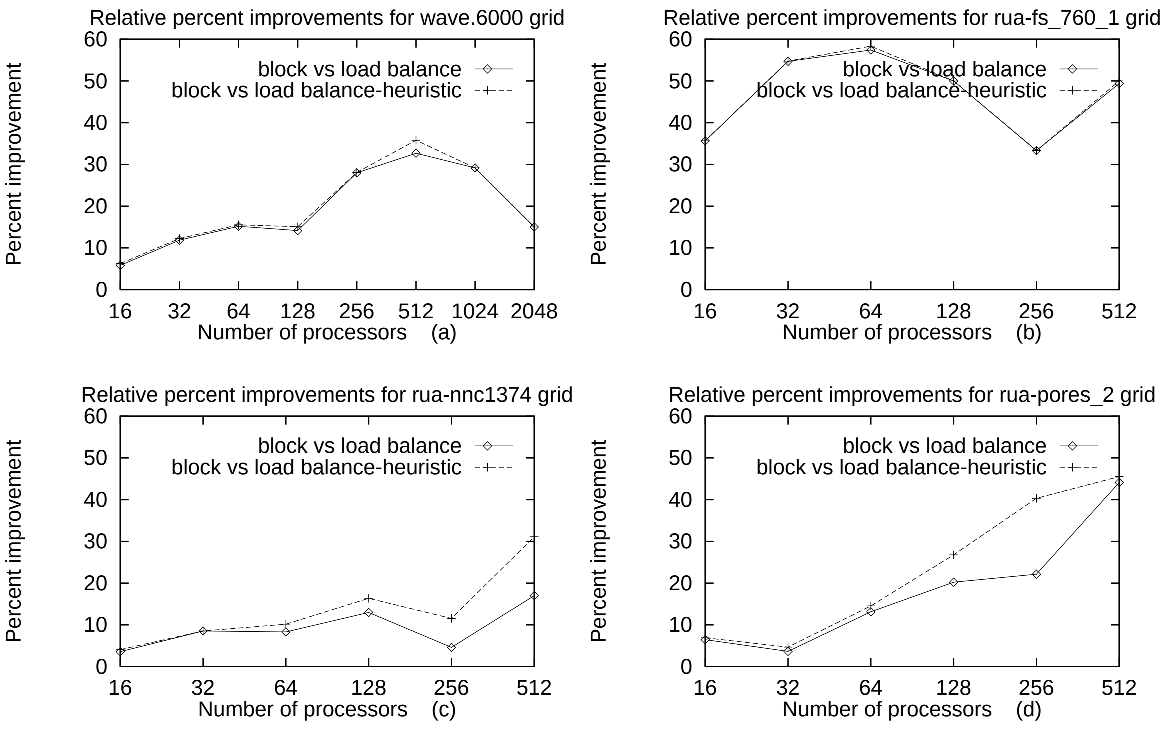


Fig. 5. Plots of relative percent improvements vs. number of processors.

These graphs illustrate the large improvements in execution time that the load-balancing step can produce over a simple block distribution while also showing that the heuristic can often further improve upon these gains. Note that even in the test cases for which Graphs (a) and (b) of Figure 4 implied no difference in the effectiveness of these two methods, we can now see some improvements in the expected times produced by the heuristic over those generated by the load-balancing step. Further, since the number of iterations performed by the heuristic is generally less than 10 (based on both the simulated and actual runs of the algorithm), the added time to perform this extra step is quite small compared to the iterative multiplication itself.

Relative Percent Improvements Across c Values vs. Number of Processors Figure 6 plots relative percent improvements in estimated execution times of the heuristic-from-load-balance distribution over the block distribution against the number of processors for a number of different communication-to-computation ratios. The graphs indicate that the heuristic should produce roughly the same amount of improvement for a wide range of communication-to-computation ratios. Thus, it would be suitable for use on other SIMD machines, even those with communication-to-computation ratios larger than the MP-1.

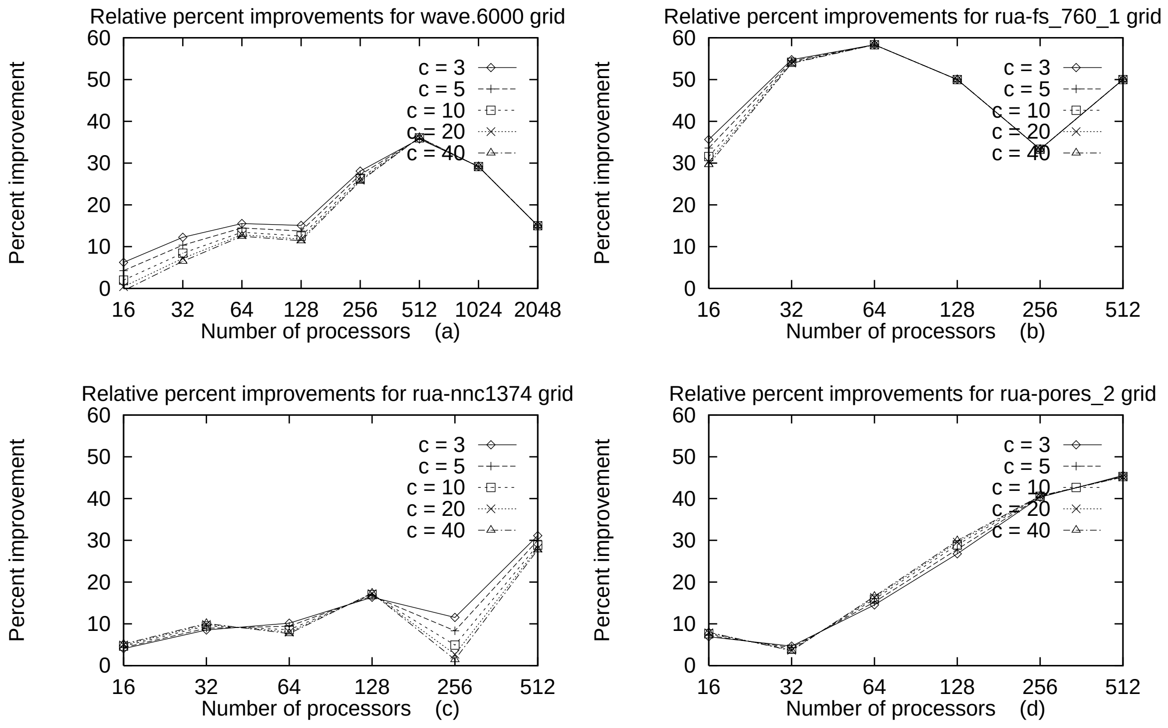


Fig. 6. Plots of relative percent improvements across c values vs. number of processors.

4 Conclusion

This paper starts with a definition of the execution cost of a sparse matrix vector multiplication as a function of local and non-local data references and the communication-to-computation ratio of the machine being used. To minimize these cost functions, we propose algorithms to preprocess off-processor references and to repartition the sparse matrix according to its nonzero entries. Since these algorithms cannot be executed at compile-time, we developed a linear complexity algorithm to optimally bi-partition such matrices and propose a fast parallel heuristic that utilizes such bi-partitions.

Our simulation results for the parallel heuristic showed up to a 60 percent cost improvement for some test cases for a moderate number of processors, while our implementation (combined with other preprocessing) produced up to a 35 percent relative improvement in performance on the MasPar MP-1 parallel computer. The results also indicate that the partition produced by the heuristic is dependent on the communication-to-computation ratio c. Given a fixed number of processors, different partitions could be produced for different values of c. This is in contrast to standard partitioning algorithms which produce only one partitioning for the given number of processors. Another consequence of considering c in a partitioning algorithm is that the best partitions might not utilize all available processors. Some of the simulation results pointed to this effect.

The heuristic and other preprocessing was also implemented on a 16-node IBM SP1 using a cost function appropriate for MIMD machines. However, while the heuristic successfully decreased the execution cost, the net improvement in execution time over simple distributions was small. Designed primarily to support coarse-grained applications, the SP1 has a relatively high communication-to-computation ratio, and communication overhead led to low efficiency of parallel execution with or without the application of the heuristic. The heuristic may produce better results on finer-grained MIMD machines, however.

Our results indicate that the parallel heuristic performs best when there is a medium number of rows assigned to each processor. When the number of assigned rows is small, each row, when relocated, changes the balance drastically, so there is little space for improvement. In the opposite case, when the number of rows is very large, the load on the processors is likely to be even. However, in the most promising cases — problems with moderate granularity and machines with moderate communication latency — the speedup can be significant.

The run-time redistribution of a sparse matrix used repeatedly in a matrix vector multiplication should become a common option in the standard library of sparse matrix operations, such as ITPACK [10]. We hope that the algorithms presented in this paper will enlarge the repertoire of run-time distribution algorithms available to the users of parallel machines.

Acknowledgement

The authors would like to thank Steve Hammond and Iain S. Duff who made their collections of meshes and sparse matrices available for use in benchmark tests in evaluating the algorithm's performance. This research was supported in part by ONR Grant N00014-93-1-0076, NSF Grant CCR-9216053 and an IBM Corporation Development Grant. The content of this paper does not necessarily reflect the position or policy of the U.S. Government — no official endorsements should be inferred or implied.

References

1. High Performance Fortran Language Specification (version 1.0): High Performance Fortran Forum. Center for Reaserch on Parallel Computation, Rice University, Rice 1993.
2. S. H. Bokhari: On the mapping problem. *IEEE Transactions on Computers*, Vol. C-30, 3–30 (1981).
3. H.A. Choi and B. Narahari: Algorithms for mapping and partitioning chain-structured parallel computations. In: *Proc. of the Int. Conference on Supercomputing*, 1991.
4. E. Cuthill and J. McKee: Several strategies for reducing the bandwidth of matrices. In: Rose and Willoughby (eds.): *Symposium on Sparse Matrices and Their Applications.* New York: Plenum Press 1972, pp. 157–166.
5. I. Duff, R. Grimes, and J. Lewis: Sparse matrix test problems. *ACM transactions on Mathematical Software*, 15, 1–14 (1989).
6. I. Duff, R. Grimes, and J. Lewis: *User's Guide for the Harwell-Boeing Sparse Matrix Collection.* CERFACS, Toulouse, France: Cedex 1992.
7. G. Fox, S. Hiranandani, K Kennedy, C. Koelbel, U. Kremer, C. Tseng, and W. Wu': Fortran D language specification. Technical Report COMP TR90079. Department of Computer Science, Rice University, Houston 1991.
8. S. W. Hammond: *Mapping Unstructured Grid Computations to Massively Parallel Computers.* PhD thesis, Department of Computer Science, Rensselaer Polytechnic Institue, Troy 1991.
9. S. Hiranandani, J. Saltz, M. Piyush, and H. Berryman: Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, 12, 315–422 (1991).
10. D. R. Kincaid, J.R. Respess, D.M. Young, and R.G Grimes: ITPACK 2C: A Fortran package for solving large sparse linear systems by adaptive accelerated iterative methods. Technical report, University of Texas at Austin, Austin.
11. V. Kumar, A. Grama, A. Gupta, and G. Karypis: *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Redwood City: Benjamin Cummings 1994.
12. D. M. Nicol: Rectilinear partitioning of irregular data parallel computations. Technical Report 91-55, ICASE, Hampton 1991.
13. C. Özturan, B.K. Szymanski, and J. E. Flaherty: Adaptive methods and rectangular partitioning problem. In: *Proc. Scalable High Performance Computing Conference 1992, Williamsburg.* Washington DC: IEEE Computer Science Press 1992, pp. 409–415.
14. D. Patterson: Massively parallel computer architecture: observations and ideas on a new theoretical model. Technical Report, Department of Computer Science, University of California at Berkeley, Berkley 1992.
15. R. Rosen: Matrix band width minimization. In: *Proc. of the 23rd ACM National Conference.* Brandon Systems Press 1968, pp. 585-595.
16. J. Saltz, R. Crowley, R. Mirchandaney, and H. Berryman: Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8, 303–312 (1990).
17. R. Schreiber: Scalability of sparse direct solvers. Technical Report, RIACS, Moffet Field 1992.
18. J. Wu, J. Saltz, H. Berryman, and S. Hiranandani: Distributed memory compiler design for sparse problems. Technical Report TR91-13, ICASE, Hampton, 1991.
19. H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald: Vienna Fortran - a language specification version 1.1. Technical Report Interim 21, ICASE, NASA, Hampton 1992.