JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **46,** 180–193 (1997)
ARTICLE NO. PC971361

Load-Balanced Sparse Matrix–Vector Multiplication on Parallel Computers

Sorin G. Nastea,*,[1] Ophir Frieder,†,[2] and Tarek El-Ghazawi‡,[3]

*Department of Control and Computers, Polytechnic University of Bucharest, Bucharest, Romania; †Computer Science, Florida Institute of Technology, Melbourne, Florida 32901-6975; and ‡Computer Engineering, Florida Institute of Technology, Melbourne, Florida 32901-6975

We considered the load-balanced multiplication of a large sparse matrix with a large sequence of vectors on parallel computers. We propose a method that combines fast load-balancing with efficient message-passing techniques to alleviate computational and inter-node communications challenges. The performance of the proposed method was evaluated on benchmark as well as on synthetically generated matrices and compared with the current work. It is shown that, by using our approach, a tangible improvement over prior work can be obtained, particularly for very sparse and skewed matrices. Moreover, it is also shown that I/O overhead for this problem can be efficiently amortized through I/O latency hiding and overall load-balancing. © 1997 Academic Press

Key Words: **sparse matrix-vector multiplication; load balancing; parallel computations; greedy allocation; optimized message passing.**

1. INTRODUCTION

Large classes of applications in areas including engineering, military, and medicine heavily rely on matrix tools to represent, store, and process data. In many cases, matrices accommodate data that include a large number of elements that are either null or non-relevant. Such matrices, in which the relevant elements are only a fraction of the total elements, are called *sparse matrices*. Sparse matrices are generally stored in compressed formats that considerably save storage space and avoid unnecessary computations with zero elements.

We consider a typical calculation kernel for many time-consuming computing problems: $Y_i = A \bullet X_i$, where A is a sparse matrix and X_i is a large sequence of dense vectors. The matrix elements are available centrally in the compressed format. The goal is to minimize the overall matrix-vector multiplication time, under the constraint of producing each vector result as soon as possible. In this work, static allocation is selected over dynamic techniques due to the large number of vectors to be multiplied with the balanced work load. Thus, matrix data are allocated once and the cost of the operation is amortized over the repeated matrix-vector multiplication.

As a general solution, a load-balancing static allocation of rows is centrally generated based on their contents of non-zero elements. A greedy allocation to sparse matrix computations is adopted to distribute the matrix rows so that each processor ends up with roughly the same number of nonzero elements. Provisions for skewed distributions are augmented in the basic algorithm, and both communications and I/O are overlapped with computations to minimize the overall execution time.

There exists a number of sparse matrix compression formats that are suitable for different classes of sparse patterns. We have chosen the Scalar ITPACK format [7, 16] that efficiently compresses general sparse pattern matrices and provides easy algorithmic access to matrix elements.

Standard test matrices [6, 7] were used in our experiments. They are available through the Internet to any researcher, thus making our results easy to reevaluate and compare. Additionally, we used a synthetically generated matrix to illustrate that a refinement to our approach enables us to handle special cases, with extremely sparse and very skewed distributions.

The remainder of this paper is structured as follows. In Section 2, we survey the prior work. In Section 3, we describe the computational model, emphasizing the underlying challenges, and define the allocation problem. Section 4 is dedicated to presenting our load-balancing and communications minimizing approaches. In Section 5, we provide a brief description of the parallel platform and of the experimental data. In Section 6, we present our experimental results, organized as two subsections. In Subsection 6.1, we provide comparisons between the non-load-balanced and the load-balanced cases. In Subsection 6.2, we study the impact of I/O on the overall scalability of typical sparse matrix computations such as the compression and the multiplication. We end the paper with our conclusions.

[1]This work is supported in part by the National Science Foundation under Contract IRI-9357785.

[2]Frieder is on leave from George Mason University. E-mail: ophir@cs.fit.edu.

[3]Supported by CESDIS/USRA and NASA GSFC. E-mail: tarek@ee.fit.edu.

0743-7315/97 $25.00

2. PRIOR WORK AND MOTIVATIONS

Prior efforts have predominantly focused on solving dense matrix problems in parallel environments [2–5, 10]. When sparse matrices are used, computations are more difficult to balance. Because of their irregular structure, it is necessary to have good insight into the distribution of non-zero elements.

Load balancing is a key issue in sparse matrix computations given the diversity of sparse matrix non-zero distributions encountered in real-life problems. However, most of the authors focus on minimizing communications costs, neglecting load-balancing. One aspect favoring such an approach is the fact that sparse matrices yield a smaller amount of work, as compared to the dense case. As parallel computations aim at solving very large data set problems, the amount of work and the associated imbalance may become considerable even in the sparse matrix cases in parallel environments, thus turning the load-balancing phase into a necessity.

Ogielski and Aielo [15] presented two load-balancing approaches for matrix multiplication on processor arrays. In the first approach, each matrix row is randomly and independently assigned a processor row, and each matrix column is randomly and independently assigned a processor column. In the second approach, row and column permutations are generated after the matrix is partitioned into equally sized $\lceil M/m \rceil \times \lceil N/n \rceil$ blocks (where $M \times N$ is the size of the matrix and $m \times n$ is the size of the processor array). The construction of row and column mappings is based on probabilistic algorithms. In our case, a permutation of rows (or columns) is generated based on actual knowledge of the non-zero matrix elements distribution in each row.

A different allocation strategy is elected by Rothberg and Schreiber [17] also for a 2-D block mapping. Their goal is to solve the Choleski factorization. Thus, row and column blocks are distributed according to a *greedy* allocation onto processor rows and columns, respectively. Considering our model, described in the next section, a 2-D allocation increases the allocation problem complexity and complicates the result update phase, with no foreseen benefits. In particular, 2-D allocations were shown to cause increased communications and I/O overhead.

Other related work includes the use of a greedy allocation to balance the processing of human genome data [19]. This work, however, greatly differs from ours due to the different application requirements.

An iterative load-balancing algorithm is introduced by Aliaga and Hernandez [1] in a paper that considers also a sparse matrix–vector multiplication problem. Their algorithm generates swaps of matrix rows among processors to gradually smooth minima and maxima of load. This algorithm, henceforth referenced as *Aliaga*, comprises three steps:

Sorting: rows are sorted according to the number of non-zero elements they contain;

Initial allocation: rows are mapped onto processors;

Adjustment: an iterative swapping process between processors with the smallest and the largest buckets is carried out. The data swapping process continues as long as the operation brings both of the buckets toward the average bucket size.

The time complexity of this allocation procedure depends on the actual distribution of data onto processors. A general case would be when the iterative phase stops after each processor bucket was involved in the swapping process only once. In this case, the algorithm complexity is $O(N^2/P)$, where N is the number of rows in the matrix and P is the total number of processing elements in the multiprocessor system.

It will be shown that, unlike the current approach [1], the load-balancing algorithm we used:

(1) has a deterministic performance, as its time complexity is independent of data distribution;

(2) has a better complexity in most cases;

(3) embodies provisions that work well even on very skewed data distributions.

3. THE COMPUTATIONAL MODEL AND PROBLEM DEFINITION

We considered the multiplication model in which:

the size of the sequence of X_i vectors is very large and not *a priori* known;

the resulting Y_i vectors are generated and transmitted on an individual basis.

This model is a direct representation of the real-time response simulation of a linear system stimulated by a large sequence of N-dimensional inputs. Two considerations can be inferred from the model definition: (a) the sparse matrix A remains unchanged for a considerable amount of time; (b) as each vector is broadcast on an individual basis, a large communications overhead must be expected. Considering these aspects, the model emphasizes the typical challenges of sparse matrix problems, namely the necessity of close-to-optimal allocation heuristics and of efficient interprocessor communication strategies.

Given the proposed type of problem and the considered computing model, we focused our efforts in two directions: (a) finding an efficient allocation of rows that would ensure an even distribution of non-zero elements onto processing nodes in a minimum of time; and (b) designing the appropriate message-passing strategy to minimize the idle times of processing elements.

We adapt a load-balancing algorithm based on a *greedy allocation* for general sparse pattern matrices. This load-balancing algorithm is solving the following optimization problem: *given a general sparse matrix, allocate rows so that the function*

$$F = \max_i \left\{ \sum_{j=1}^{M} (nZ_{i_j}) \right\} \tag{1}$$

is *minimized*, where $i = 1, 2, \ldots, P$; $i_j = \{i_1, i_2, \ldots, i_M\}$ represents indices of rows allocated to processor i; and nz_{i_j} is the number of non-zero elements in these rows. By solving this problem, we are actually minimizing the largest bucket size that yields the highest computing time. An optimal solution to this optimization problem is obtained when the largest processor bucket size is equal to the average bucket size. The allocation problem we considered has some similarities with the *bin packing problem* [9, 11, 12] and can be defined in bin-packing terms as follows: given a finite number P of bins and a finite number N of objects with uni-dimensional variable sizes, allocate objects to bins so that the largest bin size is minimized.

Matrices were stored in a compact format to efficiently use the memory space and to avoid unnecessary computations involving zero elements. The Scalar ITPACK format [7, 16] was chosen to generate the matrix compressed representation because it offers a good compact ratio for general sparse pattern matrices and allows ease of handling elements in algorithmic constructions. Also, it is worth noting that the Scalar ITPACK scheme was used to build the Harwell-Boeing collection of benchmark sparse matrices. We illustrate the Scalar ITPACK format through an example in Fig. 1. We define the *sparsity* coefficient as the ratio

$$sp = \frac{nz}{nz + z}, \tag{2}$$

where nz and z represent the number of non-zero and zero elements, respectively, in a matrix.

$$A = \begin{bmatrix} 2 & 0 & 5 & 0 & 0 \\ 8 & 3 & 0 & 7 & 0 \\ 0 & 6 & 2 & 0 & 1 \\ 9 & 0 & 0 & 1 & 0 \\ 0 & 7 & 0 & 0 & 2 \end{bmatrix}$$

$$a[nz] = [2 \;\; 5 \;\; 8 \;\; 3 \;\; 7 \;\; 6 \;\; 2 \;\; 1 \;\; 9 \;\; 1 \;\; 7 \;\; 2]$$

$$ja[nz] = [1 \;\; 3 \;\; 1 \;\; 2 \;\; 4 \;\; 2 \;\; 3 \;\; 5 \;\; 1 \;\; 4 \;\; 2 \;\; 5]$$

$$ia[N+1] = [1 \;\; 3 \;\; 6 \;\; 9 \;\; 11 \;\; 13]$$

FIG. 1. Example illustrating scalar ITPACK format.

4. THE ALLOCATION AND COMPUTATIONAL APPROACH

4.1. Load-Balancing Allocation Algorithm

In the work by Aliaga and Hernandez [1], the initial assumption is that information about the number of non-zero elements in each row is available. To simplify the presentation, we also make this assumption. As the allocation process is mainly sequential, it is run on only one processing node. This allocation is meant to be kept active as long as the matrix elements remain unchanged. We call this allocation procedure the *g*reedy *a*location-based *l*oad-balancing *a*lgorithm (GALA). The general allocation-multiplication algorithm is described in Fig. 2 and the Gala procedure is presented in Fig. 3. The Gala allocation scheme has time complexity independent of data distribution. This complexity is $O(N \cdot P)$ because a search for the lowest bucket size out of P processor-buckets is performed for each of the N rows. This algorithm compares favorably (in

ALGORITHM: allocation and multiplication

Perform on major node:

-***Sort.*** Sort rows in decreasing order according to the number of non-zero values.

-***Allocate according to* GALA.** Allocate the next row to the node with the smallest bucket-size.

-***Send rows to nodes.*** Send rows to nodes according to the allocation obtained after running **GALA**.

Perform in parallel:

-**For each incoming vector X_i:**

· **major node reads X_i;**

· **major node broadcasts X_i;**

· **each node performs the multiplication** between parts of the matrix A, resident in its own memory and the received vector;

· **major node gathers** and **assembles result vectors Y_i;**

· major node writes Y_i to the output port.

FIG. 2. The general (allocation and multiplication) algorithm.

```
Procedure 1: GALA
INPUT:
row_index[N] - row indices sorted in decreasing order w.r.t. the number of non-zero elements;
row_size[N] - contains row sizes (ordered in the same way);

OUTPUT:
alloc[P,N] - a P x N array that contains the index of rows allocated to each node, where P is the number of
processors;
rows_allocated[P] - stores the number of rows allocated to each processor;

ALGORITHM:
bucket_size[P] - stores the size of each processor bucket.
for (i=0, P-1)  /* initialization */
     bucket_size[i]=row_size[i]; /*  initialize the bucket of each processor with one row */
     rows_allocated[i]=1;
    alloc[i][0]=row_index[i];
endfor
for (i=P, N-1)
    size=bucket_size[0];
    greedy_proc=0;
    for (j=1, P-1) /* find the "greediest" processor */
        if (bucket_size[j] < size)
            size=bucket_size[j];
            greedy_proc=j;
        endif
    endfor
    bucket_size[greedy_proc]+=row_size[i]; /* adjust the "greediest" processor bucket size */
    k=rows_allocated[greedy_proc];
    alloc[greedy_proc][ k]=row_index[i]; /* add row index in the list of row indexes */
     rows_allocated[greedy_proc]++;  /* adjust pointer to array alloc[ ] */
endfor
```

FIG. 3. Greedy allocation-based load-balancing algorithm (GALA).

time complexity) with the one introduced by Aliaga and Hernandez [1] in the general case, when each processor bucket is involved in the swapping process at least once.

The multiplication is reviewed in Fig. 2 for completing the discussion.

4.2. Algorithm Enhancement for Highly Skewed Data

Our initial experiments proved that the allocation procedure must be augmented with the capability of handling both very skewed and highly sparse matrices. Under these conditions, it is necessary to split the rows that have significantly larger numbers of non-zero elements into several parts (segments) and allocate them to different processors. These segments participate as any row in the allocation process. They are turned to the compact format as separate rows and allocated according to the same procedure used for complete rows. The benefit is a finer granularity of the allocation problem, thus ensuring better load-balancing results. The price to be paid is not significant: two indices (index of the row from which the segment originated and index of the segment) must be maintained and an extra summation operation for updating the result must be executed. Because of this overhead, the method is recommended only for pathological cases (extreme skewness and sparsity). Therefore, it is critical to select the right threshold to avoid unnecessary overhead associated with over-splitting. In our experiments, the threshold used for the splitting decision is the average bucket size. Whenever the number of non-zero values in a row exceeds this threshold, that particular row is split into one or more parts, as needed. This threshold is selected to produce an optimal balance between obtained performance and incurred overhead, where splitting is performed only if the performance benefit from it exceeds the overhead associated with it. Experimental results, demonstrating the need for the splitting mechanism, are presented in Section 6.

4.3. Communications Optimization

Several options are available for scheduling the multiplication of the sparse matrix by the arriving stream of vectors in real-time. This includes the vector broadcast and results gathering. The broadcast operation can be quite expensive. Hypercube and mesh topologies yield broadcast complexities of $O(\log P)$ and $O(\sqrt{P})$ for each vector, respectively. On the other hand, a software pipelining enables a communications complexity of $O(1 + P/Q)$ per each vector, where P is the number of processing elements and Q is the number of vectors. If $Q \gg P$, $O(1 + P/Q) \rightarrow O(1)$. Another advantage of the vector software pipelining is the distribution of the result update. By vector software pipelining, the update phase presented in the general algorithm (Fig. 2) is now distributed and turned into a scalable operation.

5. EXPERIMENTAL TESTBED

5.1. Parallel Platform

We used an Intel Paragon supercomputer for our experiments. This scalable MIMD machine has 64 processing elements, 56 of which are work nodes. The communications bandwidth supported by the underlying mesh-type topology is up to 160 Mbytes/s. Each of the 64 nodes is based on Intel i860 processors. Large files can be stored into a set of two *r*edundant *a*rrays of *i*nexpensive *d*isks (Raid 3), controlled by two I/O nodes.

5.2. Experimental Data

In our experiments, we used both randomly generated data and selected matrices from the Harwell–Boeing sparse matrix collection [6, 7]. Benchmark matrices offer the advantage of using common data sets, thus making it possible to compare our work with other researchers' achievements. The use of synthetically generated matrices points out some characteristics of the algorithm, such as the enhancement presented in Section 4.2.

5.1.1. Synthetically Generated Data. We evaluated our approach using randomly generated matrices with non-zero elements distributed over rows according to the Zipf distribution. Consider the discrete *probability density function*

$$p(i) = \begin{cases} \dfrac{C}{i^{1-\theta}}, & \text{for } i = \{1, 2, \ldots, N\} \\ 0, & \text{elsewhere,} \end{cases} \tag{3}$$

where

$$C = \frac{1}{\sum_{i=1}^{N} \dfrac{1}{i^{1-\theta}}}, \qquad \theta \in [0, 1]. \tag{4}$$

This discrete distribution is called the *Zipf distribution* and its characteristics strongly depend on the coefficient θ. When θ is close to 0, the data distribution is very skewed. As θ approaches 1, however, the distribution becomes more uniform (Fig. 4). The Zipf matrices were generated as follows:

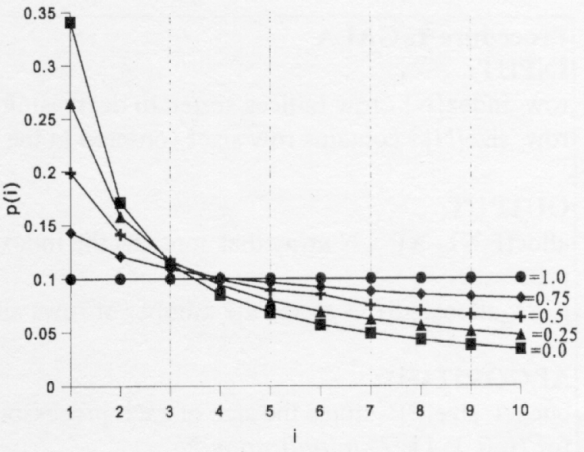

FIG. 4. Zipf distribution for different values of the parameter θ.

(1) associate each row index with a unique number in the sequence $i = \{1, \ldots, N\}$;

(2) split the interval on the real axis [0; 1) into N sub-intervals with length proportional to $p(i)$ defined in expressions (3) and (4);

(3) using a pseudo-random number generator, uniformly generate nz numbers within the interval [0; 1) and count the number of "hits" within each sub-interval;

(4) randomly generate column indices and non-zero values for each row, according to the counter values obtained at Step 3.

We generated a Zipf-distributed matrix ($\theta = 0.1$). The matrix has order 3500 and sparsity 0.004. The order, sparsity, and structure of this matrix are thus chosen to clearly illustrate the benefits of row splitting in some particular cases.

5.1.2. Benchmark Matrices. We selected three matrices from the Harwell–Boeing sparse matrix collection [6, 7] for our experiments. Our selection criteria were large matrices, different sparsity coefficients, and different sparsity patterns. In Fig. 5, we show the sparse pattern and the non-zero values distribution over rows for matrices ORANI678 (Figs. 5a and 5b), PSMIGR1 (Figs. 5c and 5d), and BCSSTK28 (Figs. 5e and 5f). Transposed matrices are represented. The selection criteria enumerated above are clearly illustrated through these graphical representations of the matrices.

An overview summarizing statistical data regarding the synthetic and the selected benchmark matrices is presented in Table I. These data include the average number of non-zero elements per row, the standard deviation, and the coefficient of variation (C.O.V., the ratio between the standard deviation and the average), along with the total number of non-zero elements and the matrix sparsity.

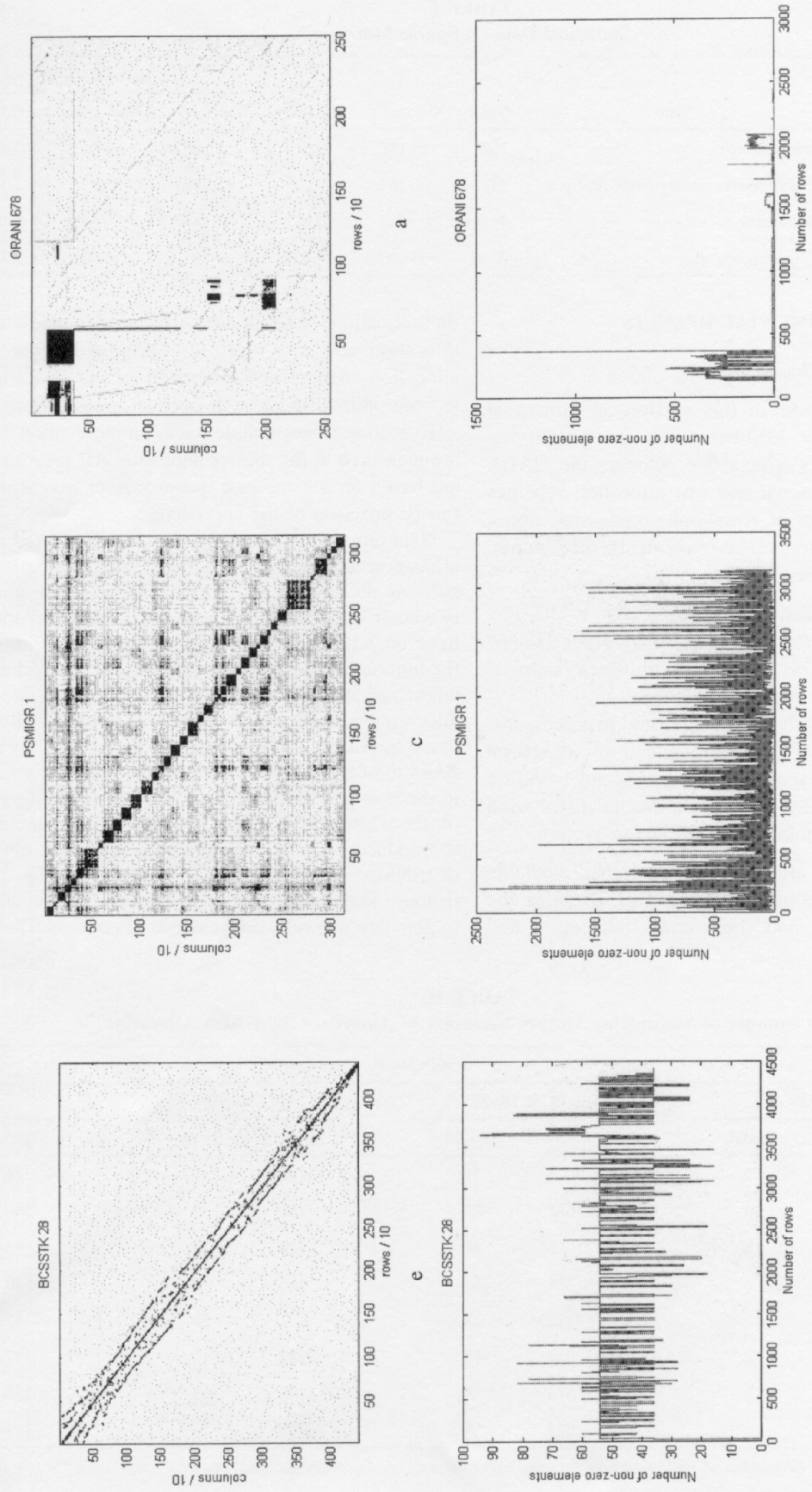

FIG. 5. Sparse patterns and nonzero value distributions of the benchmark matrices.

TABLE I
Statistical Data on Sparse Matrices

Name	Type	Order	Nonzero's	Sparsity	Statistical data		
					Average	Std. dev.	C.O.V.
ORANI 678	Unsymmetric	2529	90158	0.014	35.650	87.962	2.4674
PSMIGR 1	Unsymmetric mostly block-diagonal	3140	543162	0.055	172.981	235.575	1.362
BCSSTK28	Symmetric	4410	219024	0.011	49.665	9.682	0.195
zipf0.1 (synthetic)	Unsymmetric skewed	3500	49000	0.004	14.0	94.594	7.757

6. EXPERIMENTAL RESULTS

6.1. Amortization of Load-Balancing Cost

The experiments discussed in this section are aimed at demonstrating the need for load-balancing and quantifying the benefit from it. Besides Aliaga, we compare the GALA algorithm with two well-known and fast allocation schemes that generate the allocation of rows onto processing nodes without an insight into the non-zero elements distribution. These two allocation schemes are:

The Block allocation: Each processor $p \in \{0, 1, \ldots, P-1\}$ will receive rows $p \cdot \lfloor N/P \rfloor, p \cdot \lfloor N/P \rfloor + 1, \ldots, (p+1) \cdot \lfloor N/P \rfloor - 1$ [18]. We associate this allocation with the *unbalanced* case, because no attempt is made to even the load.

The Cyclic allocation: Each row i is allocated to processor (i mod P), where $i = \{0, \ldots, N-1\}$ [18]. We call this allocation a *naive* attempt to load-balancing, because it is not based on the data distribution, but on the assumption that the distribution of the non-zero elements maintains a continuous pattern.

We compare the Gala algorithm to the other available allocations from several points of view. This includes the speedup of multiplication and the overall (allocation and multiplication) results. We also investigate when and how allocation cost is amortized. Additionally, we compare the allocation costs of both Gala and Aliaga allocations that lead to both optimal or close-to-optimal load-balancing.

Execution times include both communication and work, and are measured at the master node site. All speedup calculations are based on a same best uni-processor execution time, for a fair comparison of the approaches.

Gala employs a relatively more sophisticated algorithm for allocation as compared to the more simplistic approaches, such as the Cyclic and Block. Thus, for a smaller number of vector multiplications, the straightforward methods could have an advantage. However, for practical problems, where the number of multiplications is large, the overhead of Gala is amortized and the overall performance gains due to the better allocation exceed by far those of simplistic allocations. The amortization results are presented in Table II. As expected, the *Block* allocation is amortized very quickly, given its sensitivity to the data permutation. Results in Table II suggest the ability of the *Cyclic* allocation (a naive load-balancing approach) to produce close to optimal allocations in almost constant distributions (as in the case of matrix BCSSTK 28). However, in more skewed distributions, the Cyclic allocation is unable to generate a good load-balanced allocation. Therefore, amor-

TABLE II
The Number of Multiplying Vectors Necessary to Amortize[a] the GALA Allocation

Nodes	Multiplying vectors							
	PSMIGR 1		BCSSTK 28		ORANI 678		zipf0.1	
	Cyclic	Block	Cyclic	Block	Cyclic	Block	Cyclic	Block
5	13	3	1724	34	182	5	228	11
10	16	5	1682	25	188	7	166	11
15	14	3	1305	29	178	7	148	11
20	25	5	1391	34	177	7	140	12
25	34	7	1432	43	221	10	145	12
30	34	8	724	42	232	12	127	12
35	35	7	814	53	173	11	136	14
40	38	9	931	55	179	13	127	14

[a]Amortization results refer to the number of multiplying vectors that make the sum of allocation and multiplication times equal for two different allocation approaches.

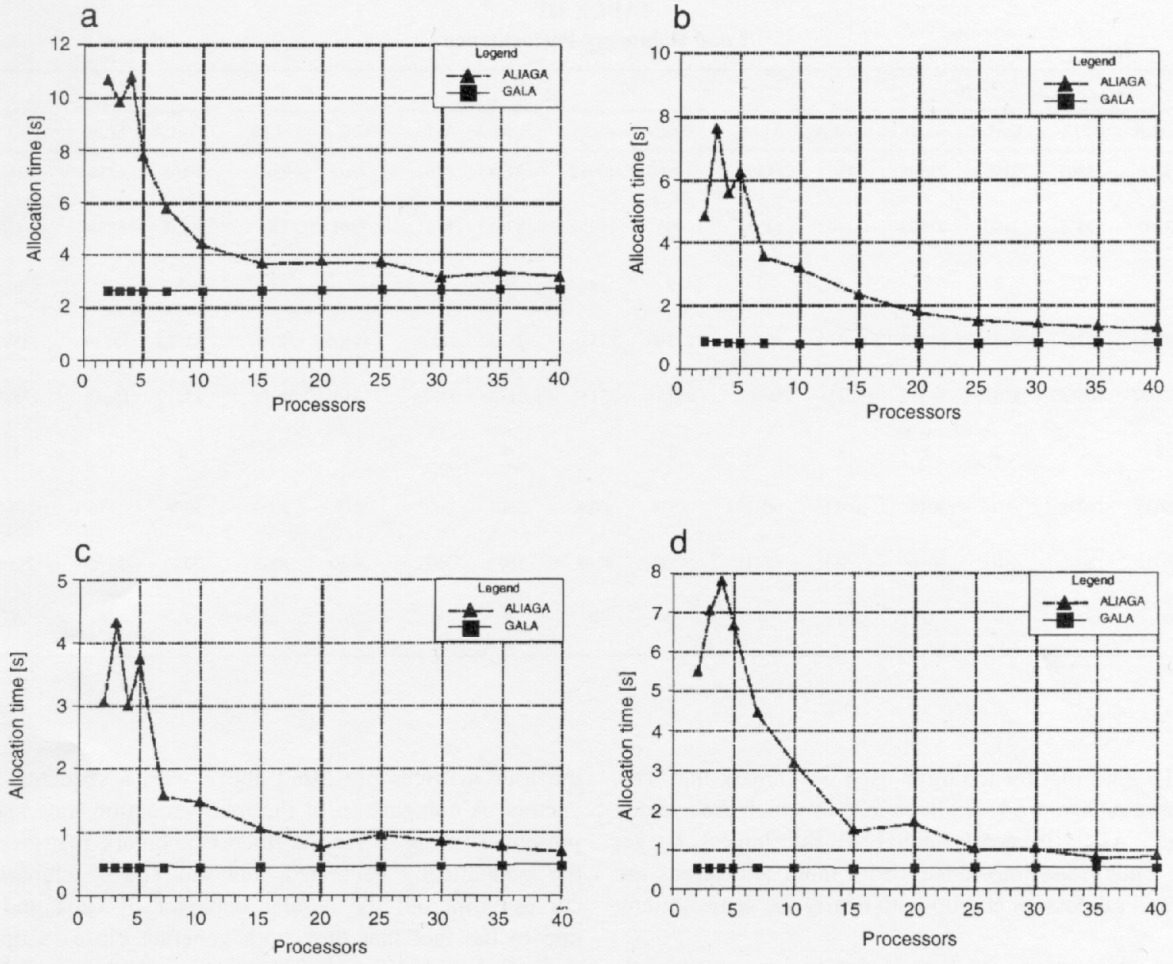

FIG. 6. Allocation timings with GALA and Aliaga allocations for benchmark and synthetically generated matrices.

tization of load-balancing occurs much earlier for the PSMIGR 1, ORANI 678, and zipf0.1 matrices.

A good way to evaluate the degree of skewness of a sparse matrix is to look at the coefficient of variation (C.O.V.) of the distribution (Table I). A large C.O.V. requires a less naive but more time-consuming load-balancing procedure for good overall performance. Additionally, we find that the standard deviation gives quantitative rather than qualitative information regarding the likelihood of the load-balancing amortization speed. It is difficult to evaluate, based on the C.O.V., when a given matrix distribution will trigger the splitting enhancement described in Section 4.2. The difficulty consists of the fact that several distributions may yield the same C.O.V. However, we empirically found that conditions prone to applying the splitting enhancement occur for a number of processors larger than $2\sqrt{N}$/C.O.V.

We compare the allocation performance of the Gala algorithm with the one achieved with the algorithm introduced by Aliaga and Hernandez [1] from two points of view: load-balancing results and time required to allocate the data onto nodes in a balanced way. Both Gala and Aliaga [1] allocation procedures are sequential algorithms. In Fig. 6, we plot allocation time results versus the number of processors. These results include the sorting and allocation (according to both Gala and Aliaga schemes). Results show that the Gala algorithm works significantly faster than Aliaga for small number of processors and they tend to have a similar performance for large number of processors. Also, large differences in allocation speeds in favor of Gala are reported for skewed distributions and large data sets. The unpredictability of the Aliaga allocation is obvious for small number of processors ($P \leq 5$). For a larger number of processors, the allocation time coincides with an exponential pattern.

Load-balancing results obtained with these two allocation procedures are very similar for all practical purposes. They are tabulated in Table III. The following notations were used:

LBS, largest bucket size;
ABS, average bucket size, equal to;
ΔBS, absolute difference between LBS and ABS.

TABLE III
Load Balancing Performance

		5		10		15		20		25		30		35		40	
Nodes		GALA	ALIA	GALA	ALIA	GALA	ALIA	GALA	ALIA	GALA	ALIA	GALA	ALIA	GALA	ALIA	GALA	ALIA
BCSSTK28	LBS	43806	43805	21904	21904	14603	14602	10960	10952	8771	8761	7302	7302	6260	6258	5488	5477
	ABS	43805	43805	21902	21902	14602	14602	10951	10951	8761	8761	7301	7301	6258	6258	5475	5475
	ΔBS	1	0	2	2	1	0	9	1	10	0	1	1	2	0	13	2
PSMIGR1	LBS	108634	108633	54318	54317	36214	36211	27160	27159	21730	21728	18109	18107	15522	15519	13583	13580
	ABS	108632	108632	54316	54316	36211	36211	27158	27158	21726	21726	18105	18105	15519	15519	13579	13579
	ΔBS	2	1	2	1	3	0	2	1	4	2	4	2	3	0	4	1
ORANI678	LBS	18032	18032	9016	9016	6011	6011	4508	4508	3607	3607	3006	3006	2576	2576	2254	2254
	ABS	18032	18032	9016	9016	6011	6011	4508	4508	3607	3607	3006	3006	2576	2576	2254	2254
	ΔBS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table III shows that for the three used benchmark matrices, both Gala and Aliaga produce allocations in which the average bucket size, ABS, is nearly equal to the largest bucket size, LBS. Thus, load imbalances from these allocations are negligible and processors end up with nearly the same amount of work.

In the case of matrix zipf0.1, the benefit of row splitting is clearly illustrated. If the matrix is extremely sparse and highly skewed (which is the case with this matrix), even good load balancing algorithms are unable to produce good results unless this simple heuristic solution is applied (Fig. 7).

We allocated data with the Gala and Aliaga (the load-balanced case), and Cyclic and Block algorithms (the non-load-balanced case), and we performed the multiplication of the four matrices presented above with a sequence of 1000 vectors. A comparison of the total execution time speedup is presented in Fig. 8 for the three benchmark matrices and for the synthetically generated data set. The benchmark matrix curves point out the similar behavior of Gala and Aliaga, due to the fact that they both generate close to optimal or optimal load-balancing. The Cyclic allocation yields good speedup in case of matrix BCSSTK 28, due to the almost continuous matrix distribution (see Table I and Figs. 5e and 5f). However, its performance deteriorates for skewed matrix distributions (matrices PSMIGR 1 and ORANI 678). A special case is represented by the synthetic matrix *zipf0.1*, generated according to the Zipf distribution (parameter $\theta = 0.1$). It is designed to clearly show the benefit of row splitting in some cases with both high sparsity and extreme skewness. Thus, results of the Gala algorithm, provided with the splitting enhancement, are not penalized by this type of matrix distribution as the other allocation approaches are. Therefore, this yields significantly higher speedup.

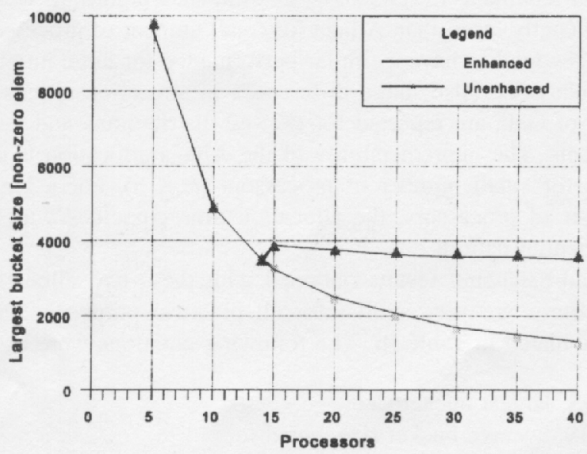

FIG. 7. Load balancing results for the Zipf-distributed matrix.

Large differences in speedup were measured for different data sets. Given the selected message passing strategy, we present an evaluation for total execution time (that includes both work and communication time costs). A simplified evaluation of T_{total} is the following:

$$T_{\text{total}} = T_{\text{COM}}(N, P, Q) + T_{\text{WORK}}\left(\frac{sp \bullet N^2}{P}\right), \quad (5)$$

where T_{com} and T_{work} represent time spent only on communication and work, respectively. T_{com} depends on the total number of processors P, the vector size N, and the number of vectors

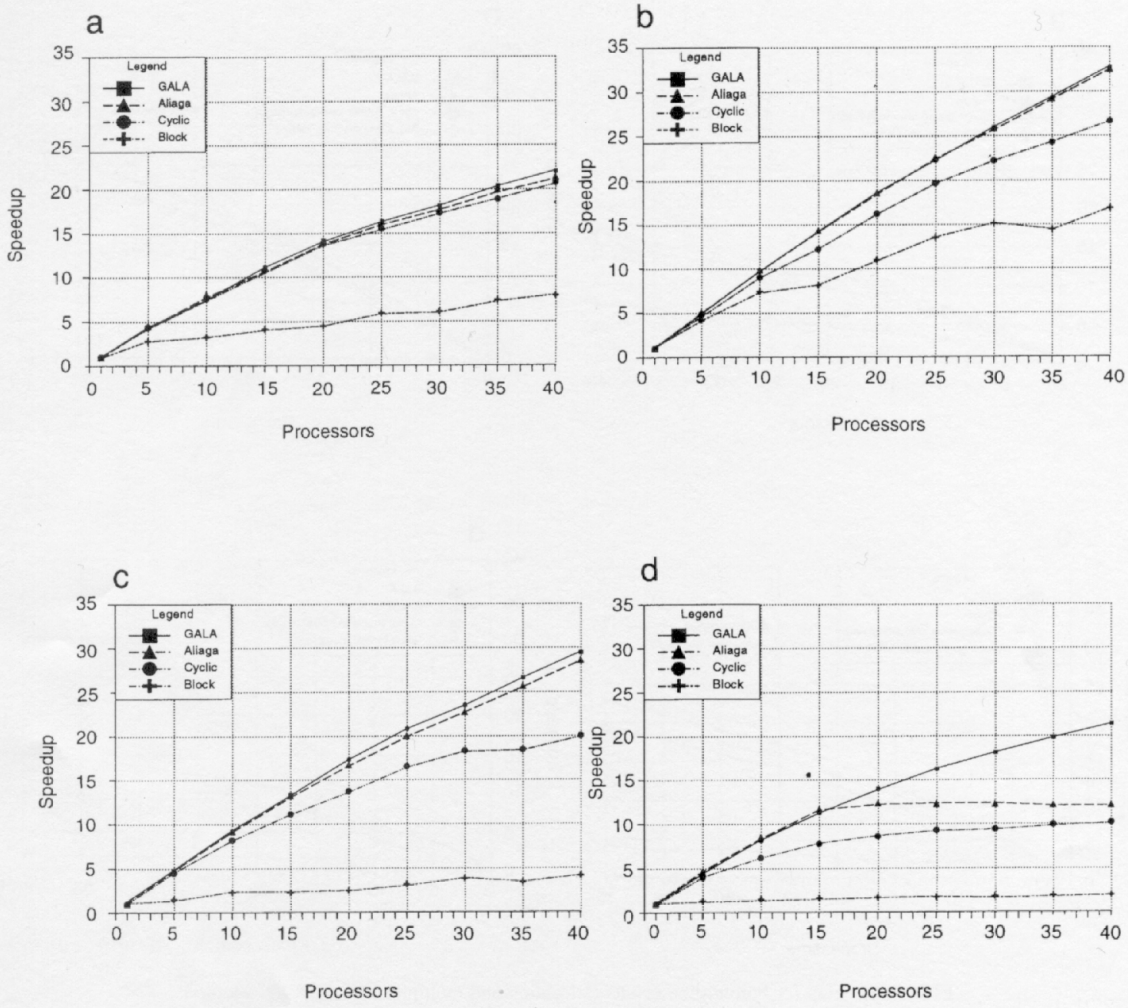

FIG. 8. Comparison of sparse matrix-vector multiplication speedup.

Q. T_{work} depends on the number of non-zero elements allocated to the node with the largest bucket-size and

$$T_{\text{total}} = \max\{T_{\text{com}},\ T_{\text{work}}\} + \text{const} \tag{6}$$

is the number of incoming vectors. On a balanced allocation of significant matrix elements, each processor receives $sp\ N^2/P$ non-zero values. Due to the use of asynchronous message passing in software vector pipelining, T_{com} depends only insignificantly on the number of processor P and a better evaluation for (5) is that saturation of global timings is expected to occur when:

$$T_{\text{work}}\left(\frac{sp \bullet N^2}{P},\ Q\right) \leq T_{\text{com}}(N,\ Q). \tag{7}$$

Thus, the best expectation for global results with the increase of the number of processors is T_{com}.

The obtained total timing results are explained through formulae (6) and (7), and the order and sparsity of the four matrices (Table I) that we used for testing purposes. Compared to the general formulation of the total execution time expressed in Eq. (5), formulae (6) and (7) reflect the communications cost minimizing approach embedded into the implementation. Best global timings scalability is obtained in the case of matrix PSMIGR 1, because the matrix is quite dense and, therefore, provides a large amount of work. Briefly, if the data set generates a large enough amount of computation, the unscalable communication requirements can be successfully hidden behind the scalable computation by using the proper message passing strategy. In Fig. 8, results for matrix PSMIGR 1 show highest achieved scalability. This is because the matrix is relatively more dense than the others which helps amortize communication due to the increased computation. Host–node and node–node communications bandwidth of up to 160 MBytes/s is considered significantly large.

Such communications bandwidth combined with the software pipelining of vectors enables good speedup and scalability.

For matrix BCSSTK 28 and for 400 vectors (Fig. 9), the Cyclic allocation yields best overall (allocation and multiplication) speedup. This is due to the fact that the distribution of the matrix is almost constant and the amount of computation

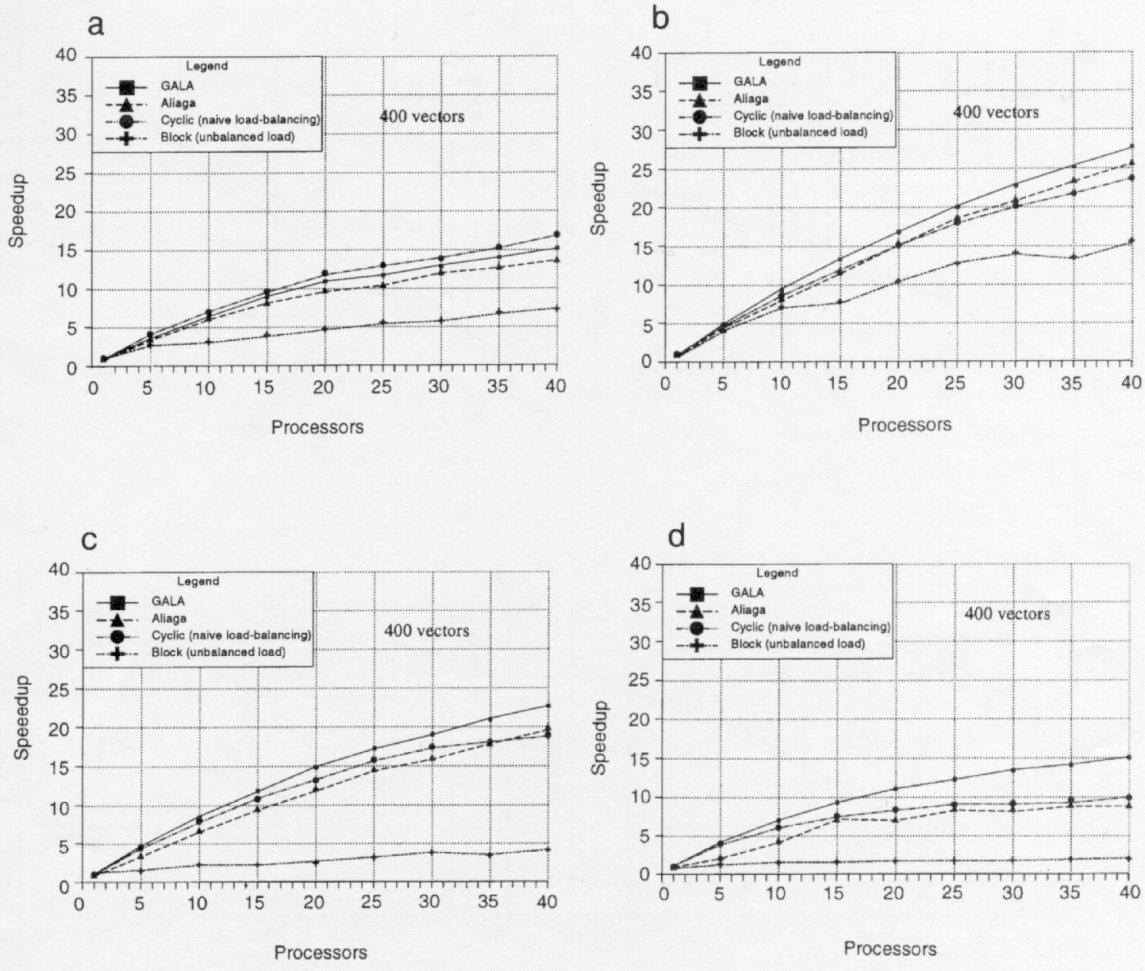

FIG. 9. Speedup for bottom line results (allocation and multiplication with 400 vectors).

is still relatively small to amortize the allocation. For the rest of the matrices, Gala yields best overall performance due to the fast and almost perfect load-balancing. However, for 1000 multiplied vectors (Fig. 10), Gala yields the best performance. Even for matrix BCSSTK 28 (with an even distribution), Gala performs better than the Cyclic allocation for large number of processors (see also Table II). Briefly, Gala and Aliaga yield the best overall performance for a larger number of multiplied vectors due to the good load-balancing they produce. Also, the better allocation cost of Gala is obvious in these measurements.

6.2. Effect of the Multiplication Size on Scalability and Amortization of I/O

One important issue in high-performance computing, which is often neglected, is I/O. In this study, we pay careful attention to the application of I/O requirements. We evaluated the I/O cost and its amortization for the case when the matrices are originally resident on disk rather than at the major node memory. We consider that matrices are stored, in the uncompressed format, on the parallel file system, and vectors were already broadcast to processing nodes. Each node has to read parts of the matrix, compress them, and multiply matrix elements with the resident vectors. The compression phase is considered necessary to avoid the subsequent processing of non-relevant elements and to provide an efficient representation of the compressed matrix. Once again, the chosen compression format was the Scalar ITPACK [7, 16] because of its compression performance and algorithmic suitability of its sparse matrix representation. To minimize the I/O cost, we combined several specific techniques:

— The use of parallel I/O. The Intel Paragon PFS supports 6 PFS file access modes: M_UNIX, M_LOG, M_SYNC, M_RECORD, M_GLOBAL, and M_ASYNC [20]. We selected the M_ASYNC file access mode based on our previous experiments [14] as it is the most suitable for our approach and provides the fastest access to data. M_RECORD, which is another efficient parallel file access mode, is more restrictive due to its synchronization requirements.

— The use of asynchronous I/O read. We use asynchronous I/O read to conveniently overlap I/O and computation. Thus, the unscalable I/O operation is performed as a background process and can be completely hidden behind the scalable computations at the extent allowed by the computations complexity.

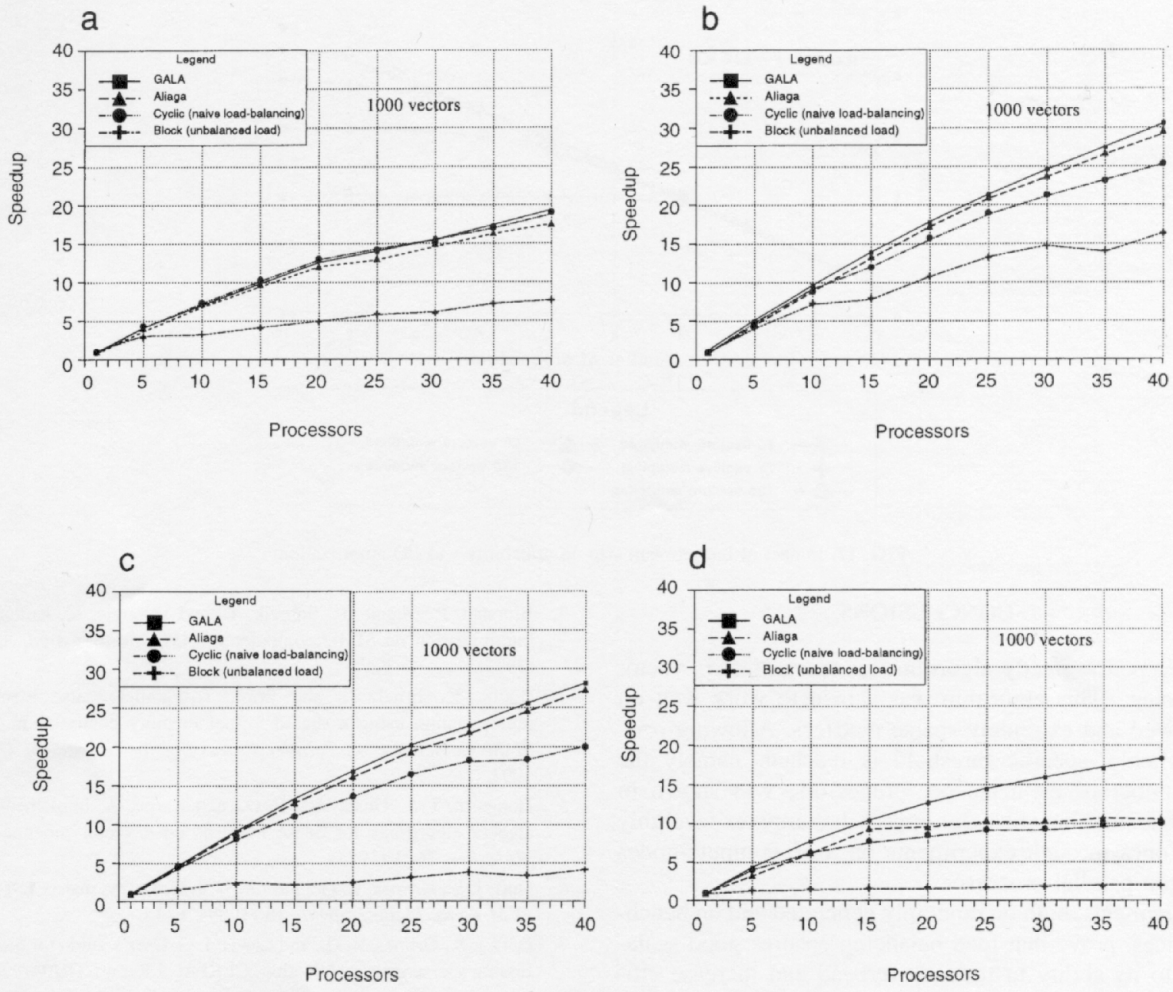

FIG. 10. Speedup for bottom line results (allocation and multiplication with 1000 vectors).

— The use of dynamic allocation rather than static allocation. Thus, after a new I/O session is started in the background, the processing node asks for new work from the master node. The message passing with the master node is also performed through asynchronous communications while work and I/O take place, thus avoiding the extra cost of this operation. This dynamic allocation smoothes the uneven behavior of the I/O nodes or the computational imbalances. On the contrary, a static allocation, in which nodes match their identification numbers against row indices, based on a given permutation method (like the previously described *Block*), is not capable of any load-balancing performance.

Computation with high degree of inherent concurrency scales well, compared to I/O operations. To study the effect of the size of the problem on the overall performance of computation and I/O, we have increased the complexity of the computation part. Thus, we have combined the compression of a sparse matrix with the multiplication of this matrix (in the compressed format) with a variable number of vectors. In Fig. 11, which is based on the PSMIGR 1 benchmark matrix, we summarize the interrelation among overall scalability, I/O, and amortization of I/O with increased computations. As we expected, the overall results scale well as long as the scalable computation part surpasses the I/O part. Each of the curves in Fig. 11 has a scalable segment and a saturated one. Note that the amortization is achieved at a reasonable size of the multiplication problem (125 vectors) for $P = 10$ processing nodes and 2 I/O nodes. This is the direct outcome from the combination of techniques used to increase the performance of the I/O itself, such as asynchronous read and dynamic allocation. The conclusion to be drawn from these results is that, if the I/O operation can be overlapped (by using asynchronous calls) with scalable computation, there exist some problem sizes for which the overall computations become scalable.

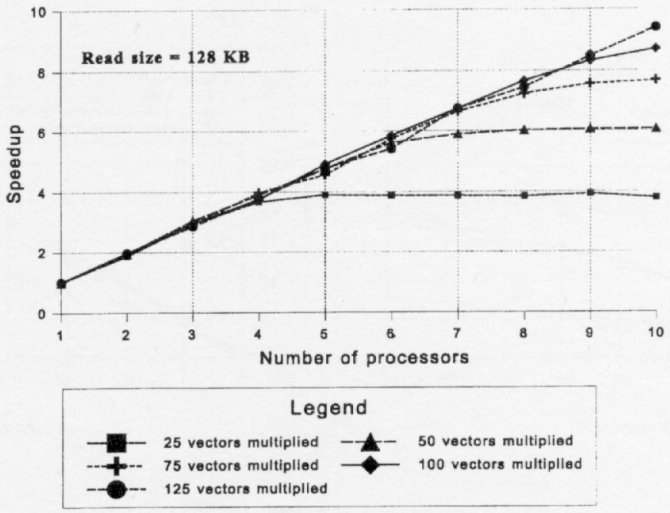

FIG. 11. Impact of the problem size on scalability and I/O amortization.

7. CONCLUSIONS

We developed a greedy algorithm for sparse matrix-vector load-balancing. This algorithm was shown to work well on highly skewed and extremely sparse matrices. Allowing row-splitting when a specific threshold is reached, namely the average number of elements per processors, was shown to provide substantial benefits. Our approach compares favorably to current ones through experiments on a 56-compute-nodes Intel Paragon parallel processor.

Our experiments, both on randomly generated and on benchmark matrices, prove that load balancing ensures good scalability due to its ability to reduce overhead and increase efficiency. We designed our implementation so that it simulates the real-time response of a system to individually-applied vectors, which has a large communications overhead. We showed that, by using an optimized implementation, communications can be successfully hidden through software pipelining. The end-to-end results, in which allocation and multiplication are combined, show that good speedup is obtained for large size problems and that load-balancing pays off in the long run. This is especially true for variably distributed data sets. If statistical data are available on the matrix, we showed that the skewness degree of a matrix can be correctly measured by computing the coefficient of variation, which helps to evaluate the most efficient computational tools for that matrix distribution.

We also analyzed the impact of the multiplication size on I/O cost amortization and overall scalability. We found that, even for reasonably low multiplication sizes, the overall results scale well when I/O is carefully handled.

REFERENCES

1. Aliaga, J. I., and Hernandez, V. Symmetric sparse matrix-vector product on distributed memory multiprocessors. *Conference on Parallel Computing and Transputer Applications,* Barcelona, Spain, 1992.
2. Alonso Sanches, C. A., and Song, S. W. SIMD matrix multiplication on the hypercube, *8th International Parallel Processing Symposium,* April 26–29, Cancun, Mexico, 1994.
3. Bjorstad, P., Manne, F., Sorevik, T., and Vajtersic, M. Efficient matrix multiplication on SIMD computers, *SIAM J. Matrix Anal. Appl.* **33,** 1 (1992).
4. Bodin, F., Erhel, J., and Priol, T. Parallel sparse matrix vector multiplication using a shared virtual memory environment. *Proc. 6th SIAM Conference on Parallel Processing for Scientifical Computing,* 1993.
5. Dongarra, J. J., Gustavson, F. G., and Karpa, A. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine, *SIAM Rev.* **26,** 1 (1984).
6. Duff, I. S., Grimes, R. G., and Lewis, J. G. Sparse matrix test problems, *ACM Trans. Math. Software* **15** (1989), 1–14.
7. Duff, I. S., Grimes, R. G., and Lewis, J. G. User's guide for the Harwell–Boeing sparse matrix collection. CERFACS Report TR/PA/92/86, 1992.
8. Frieder, O., Topkar, V. A., Karne, R. K., and Sood, A. K. Experimentation with hypercube database engines, *IEEE MICRO.* (Feb. 1992).
9. Friesen, D. K., and Langstron, M. A. Variable sized bin packing, *SIAM J. Comput.* **15,** 1 (Feb. 1994).
10. Lewis, J. G., and van de Geijn, R. A. Distributed memory matrix-vector multiplication and conjugate gradient algorithms. *Proc. Supercomputing '93,* Sparse matrix algorithms section, 1993.
11. Kao, C.-Y., and Lin, F.-T. A stochastic approach to the one-dimensional bin-packing problem. *Proc. 9th Conference on Artificial Intelligence for Applications,* Orlando, FL, March 1993.
12. Murgolo, F. D. An efficient approximation scheme for variable-sized bin packing, *SIAM J. Comput.* **16,** 1 (Feb. 1994).
13. Nastea, S. G., Frieder, O., and El-Ghazawi, T. Sparse matrix multiplication on parallel computers. *Proc. 10th International Conference on Control Systems and Computer Science,* Bucharest, Romania, 1995.
14. Nastea, S. G., El-Ghazawi, T., and Frieder, O. Parallel input/output impact on sparse matrix compression. *Proc. IEEE Data Compression Conference,* Snowbird, UT, 1996.
15. Ogielski, A. T., and Aielo, W. Sparse matrix computations on parallel processor arrays, *SIAM J. Scientific Comput.* (1993).
16. Peters, A. Sparse matrix vector multiplication technique on the IBM 3090 VP, *Parallel Computing* **17** (1991).
17. Rothberg, E., and Schreiber, R. Improved load balancing in parallel sparse Choleski factorization. *Proc. Supercomputing '94,* Washington, DC, 1994.
18. Saad, Y., Wu, K., and Petiton, S. Sparse matrix computations on the CM-5. *Proc. 6th SIAM Conference on Parallel Processing for Scientifical Computing,* 1993.

19. Yap, T. K., Frieder, O., and Martino, R. L. Parallel homologous sequence searching in large databases. *Proc. 5th Symposium on the Frontiers of Massively Parallel Computations,* McLean, VA, February 1995.
20. *Paragon OSF/1 User's Manual,* 1995.

SORIN NASTEA received both his B.S. and M.S. in Electrical and Computer Engineering, from the Polytechnic University of Bucharest, Romania, in 1983, and a Ph.D. in Information Technology in 1996 from George Mason University, Fairfax, Virginia. After 1983, Dr. Nastea worked as a researcher in the Telecommunications Research Institute, Bucharest, after becoming a member of the faculty in the Department of Control and Computers, the Polytechnic University of Bucharest, in 1990. In 1993, he enrolled in a Ph.D. program at George Mason University. Dr. Nastea's current scientific interests lie in parallel computing, in general, with emphasis on parallel architectures, optimization of parallel implementations, and parallel I/O. He has published more than 25 scientific papers. Dr. Nastea is an IEEE and ACM member.

OPHIR FRIEDER has received his Ph.D. (1987) in Computer Science and Engineering from the University of Michigan. From 1987 to 1990, Dr. Frieder was a Member of the Technical Staff in the Applied Research Area of Bell Communications Research. In 1990, Dr. Frieder joined George Mason University, where he has been a Professor of Computer Science since 1995. In 1986, Professor Frieder became the Harris Professor of Computer Science at the Florida Institute of Technology. Dr. Frieder has consulted for a variety of organizations, including the FBI, IDA, SAIC, and IBM/Loral Federal Systems, and the Software Productivity Consortium. Dr. Frieder has authored two books, published over 70 refereed publications, was granted two patents, and has received research support from a variety of governmental and industrial grants. In 1993, he was a recipient of the International Information Science Foundation Award from Japan and the NSF National Young Investigator award. Dr. Frieder's research interests include parallel and distributed database and information retrieval systems and biological and medical data processing architectures. In 1993–1994, Dr. Frieder served as the database area editor for IEEE Computer. In 1995–1996, he served as an Associate Editor-in-Chief of IEEE Software. Dr. Frieder was a program Chair of both the 1996 IEEE Fourth International Symposium on the Assessment of Software Tools and the 1996 IEEE Fifth International Conference on Computer & Communications Networks. Dr. Frieder is a member of Phi Beta Kappa and the ASEE, and a Senior Member of the IEEE.

TAREK EL-GHAZAWI received the Ph.D. degree in electrical and computer engineering in 1988 from New Mexico State University. In the fall of 1997 he joined the Computer Engineering program at the Florida Institute of Technology. From 1990 to 1997, he was on the faculty of the Department of Electrical Engineering and Computer Science at George Washington University. Prior to joining GWU, he held positions at the University of Helwan and the Johns Hopkins University. His research interests include high-performance computing, experimental computer architecture, high-performance I/O systems, and experimental performance evaluations. Dr. El-Ghazawi has published over 40 refereed publications in these areas. Dr. El-Ghazawi's research has been supported by NASA HPCC, CESDIS/USRA, NASA GSFC, Hughes Applied Information Systems, and the Army Corps of Engineers through the Computer Science Corporation. Dr. El-Ghazawi has frequently served as a consultant at NASA GSFC and other organizations in the area of high-performance computing. He has served as the workshop chair for Frontiers'95, and as the program co-chair for the International Conference on Parallel and Distributed Computing and Systems, 1991. He is a Senior Member of the IEEE, a member of the ACM, and Phi Kappa Phi.

Received March 13, 1995; accepted July 1, 1997