Parallelization Techniques for Sparse Matrix Applications

MANUEL UJALDÓN,* 2 EMILIO L. ZAPATA,* 2 SHAMIK D. SHARMA,† 3 AND JOEL SALTZ‡ 3

*Computer Architecture Department, University of Malaga, Campus de Teatinos, 29071 Malaga, Spain; and †Computer Science Department, University of Maryland, College Park, Maryland 20742

Sparse matrix problems are difficult to parallelize efficiently on distributed memory machines since data is often accessed indirectly. Inspector–executor strategies, which are typically used to parallelize loops with indirect references, incur substantial runtime preprocessing overheads when references with multiple levels of indirection are encountered—a frequent occurrence in sparse matrix algorithms. The sparse-array rolling (SAR) technique, introduced in [M. Ujaldón and E. L. Zapata, Proc. 9th ACM Int’l. Conf. on Supercomputing, Barcelona, July 1995, pp. 117–126], significantly reduces these preprocessing overheads. This paper outlines the SAR approach and describes its runtime support accompanied by a detailed performance evaluation. The results demonstrate that SAR yields significant reduction in preprocessing overheads compared to standard inspector–executor techniques. © 1996 Academic Press, Inc.

1. INTRODUCTION

Sparse matrices are used in a large number of important scientific codes, such as molecular dynamics, CFD solvers, finite-element methods, and climate modeling. Unfortunately, these applications are hard to parallelize efficiently, particularly using automated compiler techniques. This is because sparse matrices are represented using compact data formats which necessitate heavy use of indirect addressing through pointers stored in index arrays. Since these index arrays are read at runtime, compilers cannot analyze which matrix elements will actually be touched in a given loop, making it impossible to determine nonlocal memory accesses at compile-time.

The standard method of parallelizing loops that use indirect addressing is the inspector–executor strategy [12]. Loops are transformed such that for each indirect reference in a loop, a preprocessing step called the inspector is inserted before the loop. The inspector examines global addresses referenced by the indirect and determines which nonlocal elements must be fetched. The executor uses this information to fetch data and to perform the computation. Such runtime preprocessing techniques have been fairly well-studied and successfully incorporated into compilers [14, 6].

The inspector/executor method incurs a runtime overhead for each inspector stage which are greatly increased when references use multiple levels of indirection. To reduce these overheads in sparse-matrix problems, the sparse-array rolling (SAR) technique was proposed in [15]. This method uses special representations for distributed sparse matrices which allows efficient access to the non-zero elements.

Initial evaluations of the SAR method were promising and led to its integration into the Vienna Fortran compiler using language extensions. For a detailed description of the compilation issues, we refer the reader to [15]. In this paper, we focus on the SAR runtime support and describe how it allows efficient access to the distributed sparse matrix. The performance of SAR depends on a wide range of different parameters such as the data distribution, the locality characteristics of the algorithm, and the sparsity of the input matrix. We have evaluated its performance under many combinations of these choices, and we describe the impact that each choice has on preprocessing costs, executor efficiency, and memory overhead.

The paper is structured as follows. Section 2 provides a general overview of sparse matrix parallelization. Section 3 provides an overview of the SAR technique and its runtime support. Section 4 introduces a sparse-matrix program and illustrates how it is parallelized using the SAR approach. We discuss the impact of distribution choices in Section 5. Section 6 provides a detailed performance evaluation. The last two sections discuss related work and the conclusions drawn from this work.

2. STANDARD SPARSE MATRIX PARALLELIZATION TECHNIQUES

A matrix is called sparse if only a small number of its elements are nonzero. Considerable savings in memory and computation time can be achieved by using sparse-matrix storage formats and algorithms that store and operate on only the nonzero elements [2].
sparse-array rolling technique is based on this strategy, it is worthwhile to review the standard methodology [12].

In the inspector-executor paradigm, loops are transformed so that for each indirect reference to a distributed array, the compiler inserts a preprocessing step before the loop, called an inspector. During program execution, this inspector dereferences the global addresses accessed by the indirection to obtain a (processor, offset) tuple. This tuple describes the location of the element, either in local memory or in a cache-location when fetched. The executor stage uses the preprocessed information to fetch the marked off-processor elements, to place them in their assigned cache locations and to access distributed data using the translated addresses.

An obvious penalty of using the inspector-executor paradigm is the runtime overhead introduced by each inspector stage, which can become significant when multiple levels of indirection are used to access distributed arrays. As we have seen, this is frequently the case for sparse-matrix algorithms. For example, the reference encountered in Fig. 5 requires two preprocessing steps—one to access the distributed array \( \text{Row} \), and a second to access \( \text{Data} \). If the vector \( X \) is distributed across processors, the \( X(\text{Column}(\text{Row}(R)+X)) \) reference will require three preprocessing steps and even if it is replicated, it will still require at least two.

2.1. Storage Formats and Indexing

Numerous storage formats have been proposed in sparse-matrix literature; for our work, we have used the very commonly used CRS (compressed row storage) format, though any format which does not make assumptions of the sparsity structure can be used instead. Figure 1 shows how the sparse matrix \( A \) would be stored using the CRS format. A vector, \( \text{Data} \), stores the non-zero values of the matrix in a row-major order. Another vector, called \( \text{Column} \), stores the column index of each non-zero element while a third vector, \( \text{Row} \), vector marks the beginning of each row in the \( \text{Data} \) and \( \text{Column} \) vectors. \( \text{Row} \) and \( \text{Column} \) are called auxiliary arrays since they do not contain actual data; they only provide information about the structure of the matrix.

Sparse matrix algorithms designed for the CRS format typically use a nested loop, with the outer loop iterating over the rows of the matrix and an inner loop iterating over the nonzeros in that row. Matrix elements are identified using a \((R, X)\) index set, where \( R \) denotes the \( R \)th row of the matrix and \( X \) denotes the \( X \)th non-zero in that row. The matrix element referred to by \((R, X)\) is the one at row number \( R \), column number \( \text{Column}(\text{Row}(R)+X) \) and has the nonzero value stored in \( \text{Data}(\text{Row}(R)+X) \). Note that the row and column numbers could very well be used to index other arrays.

2.2. Parallelization

An obvious disadvantage of using compact sparse storage formats such as CRS is the resulting multilevel indexing required to access data, which becomes a stumbling block when data are distributed across processors.

To parallelize loops that use indirect addressing, compilers typically use an inspector-executor strategy. Since the sparse-array rolling technique is based on this strategy, it is worthwhile to review the standard methodology [12].

In the inspector-executor paradigm, loops are transformed so that for each indirect reference to a distributed array, the compiler inserts a preprocessing step before the loop, called an inspector. During program execution, this inspector dereferences the global addresses accessed by the indirection to obtain a (processor, offset) tuple. This tuple describes the location of the element, either in local memory or in a cache-location when fetched. The executor stage uses the preprocessed information to fetch the marked off-processor elements, to place them in their assigned cache locations and to access distributed data using the translated addresses.

An obvious penalty of using the inspector-executor paradigm is the runtime overhead introduced by each inspector stage, which can become significant when multiple levels of indirection are used to access distributed arrays. As we have seen, this is frequently the case for sparse-matrix algorithms. For example, the \( \text{Data}(\text{Row}(R)+X) \) reference encountered in Fig. 5 requires two preprocessing steps—one to access the distributed array \( \text{Row} \), and a second to access \( \text{Data} \). If the vector \( X \) is distributed across processors, the \( X(\text{Column}(\text{Row}(R)+X)) \) reference will require three preprocessing steps and even if it is replicated, it will still require at least two.

3. SPARSE ARRAY ROLLING

An obvious way to reduce preprocessing costs is to replicate the auxiliary arrays on each processor. Unfortunately, this simple solution, while useful for small matrices, does not scale. For large matrices, replicating arrays whose size is proportional to the number of non-zero elements (such as \( \text{Column} \)) is not feasible.
3.1. Solution Overview

In [15], Ujaldon et al. proposed the sparse-array rolling (SAR) technique, which reduces the preprocessing overheads incurred by the inspector–executor strategy for sparse-matrix applications without imposing high memory overheads. The SAR approach uses a new representation scheme (called d-representation) that encapsulates information of how the matrix is distributed with information about the sparsity structure. They allow us to determine the (processor, offset) location of a sparse-matrix element without having to plot through the distributed auxiliary array data structures. This reduces the preprocessing overheads significantly.

Figure 2 provides an overview of the SAR solution approach. During distribution of the sparse matrix data-structures, a d-representation descriptor is constructed. This descriptor can be indexed using the row number (R) and the nonzero index (X) to locate the processor and offset at which the matrix element is stored. The efficiency of the dereferencing function and the memory overheads of the d-representation descriptor are largely dependant on how the matrix is distributed. In the next section, we describe two distribution strategies, that were adopted from [1] and incorporated with SAR, followed by a detailed description of its runtime support.

3.2. Distribution

Instead of distributing the auxiliary arrays separately, SAR adopts the approach of first mapping the (imaginary) dense matrix across processors followed by assigning each nonzero element to the processor that owns the corresponding index of the dense matrix. This strategy allows a concise representation of the distribution descriptor while still allowing efficient dereferencing.

The distributions that can be used with SAR share the common property of mapping the (imaginary) dense matrix’s index space (1:nrows, 1:ncols) onto a mesh of processors such that each processor is assigned a (possibly noncontiguous) region of the index-space that can be described by the expression: (Rowlo : Rowhi : stride_r, Collo : Colhi : stride_c). Each processor is assigned the nonzeros that lie in its region. Two of the many possible ways of accomplishing such a distribution are:

- The multiple recursive decomposition (MRD) [1] recursively decomposes the sparse matrix over P processors using horizontal and vertical partitions, until the matrix has been decomposed into \( P_1 \times P_2 \) rectangular submatrices (\( P_1 \times P_2 = P \)). At each stage of the partitioning process, the nonzeros in the submatrix of that stage are divided as evenly as possible (see Fig. 3).
The block row scatter (BRS) [1] uses a cyclic mapping of the matrix among \( P \) processors. The matrix is subdivided using a stencil of size \( P_1 \times P_2 (P_1 \times P_2 = P) \), and each processor gets the nonzero elements matching its position in the stencil (see Fig. 4). This is similar to scatter-decomposition distribution schemes and is useful in situations where the concentration of non-zeros may be extremely uneven across the domain, and unpredictable.

The mapping of the nonzero elements defines the distribution used for the Data and Column vectors. The Row vector uses the mapping of the rows of the matrix.

Both these distributions achieve a happy middle ground in terms of memory overheads and load-balance. Unlike completely irregular distributions, they allow a relatively concise description of the mapping of the non-zeros. Unlike strictly regular (i.e., block or cyclic) decompositions the MRD distribution permits unequal sized regions to be assigned to processors, thereby allowing a more balanced distribution of nonzeros.

### 3.3. Language Support

SAR has been successfully incorporated into the Vienna Fortran Compilation System [5] using language extensions that we briefly review here. For a more detailed description of the language issues and compiler support, we refer the reader to [15].

For the compiler to automatically parallelize the code, the user must provide the compiler with the following pieces of information for each sparse matrix: The name of the matrix, its dense index space size (number of rows and columns), the type of its elements, the sparse representation format and the names of the auxiliary arrays and the data distribution method to be used to partition the matrix.

The following set of directives show how language extensions can be used to specify these.

```plaintext
REAL A(N1,N2), SPARSE (CRS(Data, Column, Row)), DYNAMIC 
DISTRIBUTE A::(MRD) 
REAL Y(N1) DYNAMIC, CONNECT (C) WITH A(C,:)
```

These declarations state that matrix \( A \) of size \( N_1 \times N_2 \) is sparse and represented using the CRS arrays \( DA, CO, \) and \( RO \). MRD distribution should be used to distribute this matrix. Similar directives are used to align dense vectors with the matrix in a HPF-like fashion.

### 3.4. Runtime Support for MRD

The MRD distribution maps a rectangular portion of the dense index space \( (N_1 \times N_2) \) onto a virtual processor space \( (P_1 \times P_2) \). The MRD d-representation consists of two parts: A vector \( \text{partH} \) stores the row numbers at which the \( P_1 \) horizontal partitions are made and a two dimensional array \( \text{partV} \), of size \( N_1 \times P_2 \) which keeps track of the number of nonzero elements in each vertical partition for each row.

**Example 1.** For the MRD-distributed matrix in Fig. 3, the corresponding d-representation is the following:

\[
\text{partH}(1) = 8; \text{partV}(1,1:2) = 1,1; \text{partV}(2,1:2) = 0,1; \\
\text{partV}(3,1:2) = 1,2; \text{partV}(4,1:2) = 0,1; \\
\text{partV}(5,1:2) = 1,1; \text{partV}(6,1:2) = 1,1; \\
\text{partV}(7,1:2) = 0,1; \text{partV}(8,1:2) = 0,2; \\
\text{partV}(9,1:2) = 2,3; \text{partV}(10,1:2) = 2,3;
\]

\( \text{partH}(1) = 8 \) denotes that the horizontal partition is made at row 8. Each row has two vertical partitions. The values of \( \text{partV}(9,1:2) = 2,3 \) say that the first section of row 9 has two nonzero elements while the second section has one (\( 3 - 2 = 1 \)).

Given any nonzero element identified by \( (R,X) \) we can perform a dereference to determine the processor that owns the nonzero element. Assuming that processors are identified by their position \( (myR, myC) \) in the \( P_1 \times P_2 \) virtual processor mesh, the values \( myR \) and \( myC \) of the processor that owns the element satisfies the following inequalities:

\[
0 \leq myR < \text{partH}(R) \\
0 \leq myC < \text{partV}(R,myC)
\]

**FIG. 4.** (a) BRS partitioning of a Boeing–Harwell matrix. (b) A smaller BRS matrix partition using a \( 2 \times 2 \) processor stencil; \( P(0,0)'s \) elements underlined. (c) The local submatrix data structures.
Searching for the right myR and myC that satisfies these inequalities can require a search space of size $P_1 \times P_2$. The search is optimized by first checking to see if the element is local by plugging in the local processor’s values for myR and myC. Assuming a high degree of locality, this check frequently succeeds immediately. When it fails, a binary search mechanism is employed. The offset at which the element is located is $X-partV(R, myC)$. Thus the column number of the element $(R,X)$ can be found at $Column((X-partV(R, myC))$ on processor $(myR, myC)$, and the nonzero value can be accessed from $Data((X-partV(R, myC))$ on the same processor, without requiring any communication or additional pre-processing steps.

### 3.5. Runtime Support for BRS

Unlike MRD, the BRS d-representation descriptor is different on each processor. Each processor $(myR, myC)$ has elements from $M/P_1$ rows mapped onto it. The BRS d-representation stores for each local row of the matrix, an entry for every nonzero element on that row, regardless of the fact that element is mapped locally or not. For those elements that are local, the entry stores the local index into $Data$. For off-processor elements, the entry stores the global column number of that element in the original matrix. To distinguish between the local entries and nonlocal entries, we swap the sign of local indexes so that they have negative indexes. The actual data structure used is a CRS-like two-vector representation—a vector called $CS$ stores the entries of all the elements that are mapped to local rows, while another vector, $RA$, stores the indexes at which each row starts.

#### Example 2.
For the sparse matrix $A$ and its partitioning showed in Fig. 4, the values of $CS$ and $RA$ on processor $(0, 0)$ are the following:

$RA(1) = 1; RA(2) = 2; RA(3) = 4;
RA(4) = 5; RA(5) = 6; RA(6) = 9;
CS(1) = 2; CS(2) = -1; CS(3) = 8; CS(4) = 4;
CS(5) = -2; CS(6) = 2; CS(7) = -3; CS(8) = -4;$

$CS(1) = 2$ says that the element $(53)$ is stored on global column 2 and is off-processor. $CS(2) = -1$ signifies that the element $(19)$ is mapped locally and is stored at local index 1. The remaining entries have similar interpretations.

The processor owning the element $R,X$ is identified as follows. First, the local row is identified using the simple formula $r = (R \mod P_1)$. The entry for the element is obtained using $M = CS(RA(r) + X)$. If $M$ is negative, then it implies that the element is local and can be accessed at $Data(-M)$. If it is positive, then we have the global row $R$ and column number $M$ of the element. This implies that the processor owning the element is $Q = (R \mod P_1, M \mod P_2)$. We save the $[R,X]$ indices in a list of indices that are marked for later retrieval from processor $Q$. During the executor, a $Gather$ routine will send these $[R,X]$ indices to $Q$, where a similar dereference process is repeated; this time, however, the element will be locally found and sent to the requesting processor.

### 4. AN EXAMPLE APPLICATION

We now introduce a small but complete application program to demonstrate how different pieces of the SAR runtime support fit together. This program is called $IterSolve$ (see Fig. 5) and it builds upon the $SpMxV$ kernel to implement an iterative solver that solves a sparse system of equations.
4.1. Parallelized IterSolve

Figure 6(left) shows the code parallelized by a compiler using standard inspector–executor support provided by the CHAOS runtime library. There are four preprocessing steps corresponding to the references to the distributed arrays, Row, Data, X, and Y. The Column array is aligned with Data and shares the same preprocessing step. Each of these preprocessing stages involves communication in both translation and gathering phases.

The executor has two loop-nests—a SpMxV operation and a loop that copies Y into X. There are three collective communication stages in the executor. The first communication call gathers all the off-processor elements of X that are referenced in the SpMxV loop-nest. The second communication stage scatters the updated elements of Y to their home processor where they are accumulated. The third communication stage redistributes Y into a temporary array which is aligned with X. After the redistribution, the copy from tmp to X does not require any further communication. It is possible to reduce the executor communication by aligning the vectors X and Y with each other and/or with the rows or columns of the matrix. Two alignment strategies are particularly worth consideration:

- Alignment A: Y and X are aligned with the rows of the matrix. This eliminates the redistribution communication, but the gather communication remains.
- Alignment B: Y is aligned with the rows of the matrix and X is aligned with the columns. This eliminates the gather communication, but requires redistribution.
4.2. SAR Parallelization

4.2.1. MRD Distribution

Figure 6 (right) shows the SAR-parallelized version of IterSolve using MRD distribution. It begins with a call to a distributing function that maps rectangular regions of the matrix to each processor, constructs the d-representation (partH, partV arrays) that encode this distribution, and remaps the sparse matrix arrays accordingly. Since MRD partitions the rows of the matrix, neither alignment choice described earlier can eliminate the accumulation communication. Facing a choice between using Alignment A with its accompanying gather communication and Alignment B with its associated redistribution costs, we choose the latter. Redistributions incur less preprocessing overhead than gathers, they can exploit efficient contiguous block-transfer mechanisms and the resultant collective communication pattern is more balanced. On the other hand, the communication volume can be higher.

After the distribution, we proceed to the inspector stage. Once the loops are partitioned, we proceed to inspect the references that require preprocessing. Since $Y$ is aligned with the rows of the matrix and is accessed directly, references to it do not need any preprocessing. The references to Data and $X$ do require preprocessing, and two inspector phases are added for this purpose. Due to the alignment of these arrays, all references are found to be local and the gather calls do not add to the communication overhead.

In the executor, each processor computes the $Y$ values using its local submatrix. These values are then accumulated with the $Y$ values from other processors using a collective communication step. After the accumulation, $Y$ is redistributed into $X$. The redistribution requires no preprocessing since the communication requirements can be easily (and analytically) determined using information in the d-representation.

4.2.2. BRS Distribution

The BRS parallelized code is similar to the MRD version. Apart from the dereferencing and translation functions, the only significant point of departure is in the calculation of the loop bounds. Since the BRS d-representation does not identify the number of nonzeros in each row of the local submatrix, we need to loop over all the nonzero elements in each row and look up the $CS$ entries to find out how many are local.

4.3. Selective Redistribution

The actual redistribution from $Y$ to $X$ can be implemented in various ways. The simplest method is the conservative strategy of broadcasting the entire $Y$ array to each processor, and then performing a copy into $X$. This does not require any preprocessing but is not efficient in terms of communication volume, which can be reduced by adding a preprocessing step that precisely determines the $Y$ references that lie off-processor in the assignment loop. This optimization is called selective redistribution and its effectiveness depends on the size of the vectors being redistributed.

5. EVALUATION OF DISTRIBUTION METHODS

The choice of distribution strategy for the matrix is crucial in determining performance. It controls the data locality and load balance of the executor, the preprocessing costs of the inspector, and the memory overhead of the runtime support. In this section we discuss how BRS and MRD distributions affect each of these aspects. To account for the effects of different sparsity structures we chose two very different matrices as our input. Matrix characteristics are summarized in Table I.

5.1. Communication Volume in Executor

There are two communication steps in the executor—the accumulation of $Y$ values calculated on each processor.

### Table I

<table>
<thead>
<tr>
<th>Matrix</th>
<th>$N_1$</th>
<th>$N_2$</th>
<th>Nonzeros</th>
<th>Sparsity rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCSSTK29</td>
<td>13992</td>
<td>13992</td>
<td>316740</td>
<td>0.0016</td>
</tr>
<tr>
<td>PSMIGR1</td>
<td>3140</td>
<td>3140</td>
<td>543162</td>
<td>0.0551</td>
</tr>
</tbody>
</table>

### Table II

<table>
<thead>
<tr>
<th></th>
<th>Accumulation</th>
<th>Redistribution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MRD: Min-Max-Avg</td>
<td>BRS: All</td>
</tr>
<tr>
<td>BCSSTK29</td>
<td>6706-7584-6996</td>
<td>6996</td>
</tr>
<tr>
<td>PSMIGR1</td>
<td>1260-1746-1570</td>
<td>1570</td>
</tr>
</tbody>
</table>

Note. The variations across processors are reflected in the Min, Max, and Avg columns.
from its local matrix, and its redistribution for copying into X.

5.1.1. Accumulation Phase

Table II shows the communication volume in the accumulation phase for 16 processors in a $4 \times 4$ mesh. We note two things: first, the relation between communication volume and the processor mesh configuration; second, the balance in the communication pattern (note that comparisons of communication volumes across the two matrices should be relative to their number of rows).

In general, for a $P_1 \times P_2$ processor mesh and an $N_1 \times N_2$ sparse matrix, the communication volume is roughly proportional to $(N_1/P_1) \times \log(P_2)$. Thus an $8 \times 2$ processor mesh will have four times less total communication volume in the accumulation step than a $4 \times 4$ mesh. For BRS, each processor accumulates exactly the same amount of data, while for MRD, there are minor imbalances stemming from the slightly different sizes of the horizontal partitions (see Fig. 7).

5.1.2. Redistribution Phase

Table II also shows the communication volume for the redistribution phase. For BRS, all processors receive the same number of nonlocal $\gamma$ elements, except for those on the leading diagonal of the processor mesh which require fewer nonlocal elements (none at all if it is a square mesh). The volume communicated by other processors is roughly proportional to $(N_2/P_2)$. Note that using a large $P_1$ reduces accumulation costs, while increasing $P_2$ reduces the redistribution volume; it is therefore best to use as squarish a processor mesh as possible.

While the inverse relation to $P_2$ is maintained for worst case communication costs for MRD, the actual communication volume can vary widely. If the horizontal and vertical partitions use the same spacings, then the number of elements to be communicated are roughly the same and the communication balance is optimized. As we increase the mismatch between the partitioning of rows and columns, the communication volume and imbalance also rises. Figure 7 shows the MRD partitions for the two matrices using an $8 \times 4$ processor mesh. BCSSTK29 has a more uneven partitioning (because most of its nonzeros are near the diagonal) resulting in high redistribution communication volume.

5.2. Loop Partitioning and Workload Balance

In the IterSolve application, the bulk of the computation lies in the SpMxV loop nest, which is entirely local. Each iteration is mapped to the owner of the nonzero element (Data) accessed in that iteration. MRD always results in perfect workload balance, since each processor owns an equal number of nonzeros. BRS workload balance relies on the random positioning of the elements, and except for pathological cases, it too results in very good load balance. Table III shows the load balance index for BRS (maximum variation from average divided by its average).

The copying of $\gamma$ into $X$ is a relatively cheap operation, which is fortunate, because the computation in this loop is very unbalanced for MRD. This is because the $X$ vector is aligned to MRD’s column partitions, which can vary widely in size.

5.3. Memory Overhead

Vectors for storing the local submatrix on each processor require similar amounts of memory in both distributions. However, the d-representations used by the runtime support can require substantially different amounts of memory. Table IV summarizes these requirements. The first row indicates the expected memory overhead and the next two rows show the actual overhead in terms of the number

<table>
<thead>
<tr>
<th>TABLE IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Overhead (in Integers) for the d-Representations on $4 \times 4$ Processors</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Matrix</th>
<th>MRD descriptor size</th>
<th>BRS descriptor size</th>
</tr>
</thead>
<tbody>
<tr>
<td>partH</td>
<td>partV</td>
<td>Overhead</td>
</tr>
<tr>
<td>Generic</td>
<td>$P_1 - 1$</td>
<td>$N \times P_2$</td>
</tr>
<tr>
<td>BCSSTK29</td>
<td>3</td>
<td>41496</td>
</tr>
<tr>
<td>PSMIGR1</td>
<td>3</td>
<td>12560</td>
</tr>
</tbody>
</table>

\(^4\) In loops like SpMxV, a REDUCE operator tells the compiler that it can make an exception to the owner-computes rule and allow multiple processors to update an array, followed by an accumulation of partial results.
of integers required. The “overhead” column represents the memory overhead as a percentage of the amount of memory required to store the local submatrix.

Vectors partV and CS are responsible for much of the overhead of its distribution, since they keep track of the positions of the non-zero elements in the MRD and BRS d-representations respectively. This overhead is much higher for the BRS distribution because the CS vector stores the column numbers even for some of the off-processor nonzeros. The length of this vector can be reduced by using processor meshes with $P_1 >>> P_2$.

6..getRuntimeEvaluation

This section describes our performance evaluation of the IterSolve application parallelized using SAR runtime support with both BRS and MRD distributions. Our intent was threefold:

- To study the effect of the distribution choice on inspector and executor performance.
- To evaluate the benefits of selective redistribution.
- To compare the preprocessing overheads with previous techniques.

Both the MRD and BRS versions of IterSolve were parallelized by hand, but by inserting all runtime calls that would be generated by a compiler (as enumerated in Fig. 4). Both these versions were then modified, replacing the redistribution with selective redistribution. Finally, a CHAOS version of the application was used as a baseline to determine the benefits of using SAR.

Our platform was an Intel Paragon using the NXLIB communication library. In our experiments, we do not account for the I/O time to read in the matrix and perform its distribution.

6.1. Inspector Costs

Figure 8a shows the preprocessing costs for the four SAR versions on the two matrices. The preprocessing overheads do reduce with increasing parallelism, though the efficiencies drop at the high end. We also note that while BRS incurs higher preprocessing overheads than MRD, it also scales better.

To understand the relative costs of BRS relative to MRD, recall that the BRS dereference mechanism involves preprocessing all nonzeros in local rows, while the MRD dereferencing requires a binary search through the d-representation data structure only for the local nonzeros. Though it processes fewer elements the size of the MRD search space is proportional to the size of the processor mesh, so as processors are added, each dereference requires a search over a larger space. Though it is not shown in the table, our measurements indicate that the BRS inspector is actually faster than MRD for more than 64 processors.

With respect to the preprocessing overheads for selective redistribution we note that the total additional overhead is low but increases as a percentage of the total preprocessing cost as we add more processors. This poor scaling is because unlike the other preprocessed references (to Data and X) some of the references to Y are off-processor. These references exercise parts of the preprocessing support (e.g., buffer assignment, communication schedule construction) that do not scale as well as the dereferencing mechanism.

6.2. Executor Time

Since both schemes distribute the nonzeros equally across processors we found that the computational section of the executor scaled very well for both distributions until 32 processors, after which the communication overheads start to reduce efficiency. Figure 8b which shows the executor time for all four SAR versions indicates good load balance. In fact, we find some cases of superlinear speedup, attributable to cache effects.

The executor communication time is shown in dark in Fig. 8b. The BRS communication overhead remains essentially invariant across all processor sizes. This suggests that the overhead of the extra communication startups is offset by the reduced communication volume, maintaining the
same total overhead. For MRD, the communication is much more unbalanced and this leads to much poorer scaling of the communication costs. Indeed, this effect is particularly apparent for BCSSTK29, where the redistribution is extremely unbalanced and becomes a severe bottleneck as the processor size is increased.

We also note that selective redistribution provides relatively little benefit for PSMIGR1. This is expected since PSMIGR1 is a small and relatively dense matrix, where the benefits of reducing the redistribution volume can be offset by the penalties of using general gather/scatter communication functions instead of broadcast. For BCSSTK29, we do see benefits of using selective redistribution with BRS. Surprisingly, the MRD version shows relatively little improvement. This can again be attributed to the communication imbalance; the processor with the worst $Y$ to $X$ alignment is already redistributing most of $Y$ and determines the total communication cost.

6.3. Comparison with CHAOS

CHAOS [13], is a runtime library that embodies the standard inspector–executor mechanisms for handling general indirect references. We implemented a parallel version of IterSolve using the CHAOS runtime library to provide a baseline with which to compare the SAR implementations. Note that CHAOS has been targeted towards iterative algorithms, and its design choices are geared toward reducing executor costs at the expense of increasing preprocessing overheads.

The CHAOS implementation uses an irregular distribution of rows of the matrix to processors. To exploit the advantages of row-wise distributions, we use the Alignment A strategy, which aligns the $X$ and $Y$ vectors to the rows of the matrix. This eliminates the accumulation communication as well as the redistribution, resulting in a very efficient executor. On the other hand, the inspector stage has three expensive preprocessing phases, each of which requires communication to dereference global addresses (see Fig. 6).

Figure 9a compares the preprocessing costs of CHAOS with the SAR-MRD and SAR-BRS versions without selective redistribution. For both matrices, the SAR technique consistently reduces the CHAOS preprocessing costs by 40 to 60%. Figure 9b shows the times for the entire IterSolve application (without the initial partitioning) for different number of iterations of the executor. For each choice of iteration, we normalize the time taken by the SAR versions to the time taken by the CHAOS version. SAR’s advantage is greatest when only few iterations are available to amortize the inspector stage; this is the case in many noniterative algorithms and adaptive applications. When the preprocessing costs can be amortized over several iterations of the executor, the preprocessing overheads become insignificant and SAR can provide only marginal benefits at best.

7. RELATED WORK

The MRD distribution used in SAR is an extension of Berger and Bokhari’s BRD partitioning method [3]. Another related mapping technique is Wise’s quadtree distribution which divides the sparse matrix into quads that are either all zero or all nonzero [17]. The MRD approach does not expose as much parallelism but has lower lookup costs.

There have been many efforts aimed at providing compile-time and run-time support for irregular problems such as [11, 10, 14]. Most of the research on irregular problems in Fortran has concentrated on handling single-level indirections, such as the PARTI and CHAOS [13] toolkits.

In practice, irregular application codes have complex access functions that go beyond the scope of current compilation techniques. The first attempt at dealing with multiple levels of indirection inside a compiler was by Das et al. [6], who suggested a technique based on program slicing that transforms the code containing multilevel indirections into code with a single level of indirection but multiple inspector stages. The technique used by SAR is different in that it resolves multiple levels of indirection by exploiting the semantic relations between the index arrays involved in the indirect accesses.

Another approach for parallelizing sparse codes is that...
followed by Bik and Wijshoff [4], who have implemented a restructuring compiler which automatically transforms programs operating on dense two-dimensional matrices into codes that operate on sparse storage schemes. This method simplifies the task of the programmer at the risk of inefficiencies that can result from not allowing the user to choose the most appropriate sparse structures.

8. CONCLUSIONS

The paper presents new techniques for an efficient parallelization of irregular algorithms in data-parallel compilers. The underlying idea consists of supplying compile-time information about the way distributed data is represented in memory with the aim of avoiding part of the overhead needed for its lookup. Our work is targeted toward sparse computations, which are infested using loops with multiple indirections.

We have developed the sparse array rolling (SAR) preprocessing technique, which exploits compile-time information for optimizing data accesses. Semantic information about the indexing arrays is exploited to build direct translation mechanisms which allow access to the final data without accessing the intermediate indexing arrays. The impact of these mechanisms is influenced by a large set of parameters, such as the data distribution, the sparse-matrix representation format, the locality characteristics of the algorithm, and the degree of sparsity of the input matrix. We have implemented and evaluated SAR under many combinations of these choices, and demonstrated the tradeoffs associated with each choice. Overall, we find that SAR significantly reduces the preprocessing costs compared to a standard inspector. Among the two distribution schemes we compared for the SAR implementations, we find that the BRS scheme results in a more efficient executor.

The ideal strategy would be to use SAR-like runtime support from a compiler that analyzes the characteristics of the algorithm to determine which form of runtime support and distribution is best suited for the application. Failing this, the user must specify which distribution mechanisms should be used, and the compiler should determine the best compromise between optimizing the inspector and optimizing the executor.

ACKNOWLEDGMENTS

We thank Rice University and the California Institute of Technology for providing access to their Intel Paragon XP/A4 parallel machines, upon which our timings were obtained.

REFERENCES


Received October 21, 1995; revised June 17, 1996; accepted July 2, 1996


