A Novel Compilation Framework for Supporting Semi-Regular Distributions in Hybrid Applications *

Dhruva R. Chakrabarti † **Prithviraj Banerjee** †

Abstract

This paper explains how efficient support for semi-regular distributions can be incorporated in a uniform compilation framework for hybrid applications. The key focus of this work is in showing how, unlike other existing schemes, our scheme is able to minimize preprocessing overheads and maintain sophisticated communication optimizations (such as reduction of inter-processor communication during schedule generation and sharing of communicated information between regular and irregular accesses) even in the presence of semi-regular distributions. It is only natural that preprocessing overheads associated with semi-regular distributions be intermediate between those involved for regular and irregular distributions. This paper shows how various properties can be inferred for semi-regular distributions. These allow the use of the interval representation which in turn reduces the preprocessing overhead and makes possible compatible code generation for hybrid references. Experimental results on a 16-processor IBM SP-2 for a number of sparse applications using semi-regular distributions show that our scheme is feasible.

1. Introduction

Most applications dealing with sparse structures represent them as dense matrices in order to conserve memory. One such representation is the compressed row storage (CRS) scheme. The *CRS* format allocates contiguous storage in memory for the non-zero elements of the matrix (traversed in a row-wise fashion) in a one-dimensional data array (henceforth denoted by DA) and in the process achieves significant memory savings. However, it requires two auxiliary one-dimensional arrays: the column array (henceforth denoted by CO) which stores the column indices of the corresponding non-zero element in the DA array and the row array (henceforth denoted by RO) which stores the indices in the data array that correspond to the first non-zero element of each row (if non-zero elements exist in that row).

In [1], we presented a uniform run-time scheme for parallelizing mixed regular-irregular applications. In this paper, we will explain how support for semi-regular distributions has been incorporated in our scheme without compromising the basic uniformity in handling hybrid distributions and hybrid accesses. We will also explain in Subsection 3.1 how a part of the analysis for regular references and regular distributions can be moved over to the compilation phase, thus amortizing the total run-time cost. Subsequent sections of the paper illustrate various important properties that can be inferred for semi-regular distributions and show the algorithms used for different distributions. In addition, we explain how, for both regular and semi-regular references, we are able to substantially reduce the inter-processor communication during schedule generation.

The outline of the rest of the paper is as follows. Section 2 discusses some of the problems we have addressed in our work. Section 3 gives an outline of our scheme while Section 4 explains in detail the compilation framework developed by us. Experimental results are presented in Section 5. Sections 6 and 7 discuss related work and present conclusions respectively.

2. Problem Description

It has been outlined in [1] how information sharing between analyses for regular and irregular accesses is important in order to minimize inter-processor communication. It has also been shown that a uniform parallelization approach would be able to generate compatible code for both regular and irregular accesses. In our framework for supporting semi-regular distributions, we maintain both of the above properties. An alternative approach of using multiple data-parallel interoperable libraries has been shown in [2]. This scheme shows how multiple data-parallel run-time libraries can interoperate and exchange data. Using such an approach, the interaction between a structured and an unstructured mesh (shown in Figure 1 taken from [2]) can be handled. Figure 1 contains four loops inside the outermost timestep loop. The four loops implement respectively a sweep over a structured mesh, exchange of boundary in-

*This research was partially supported by the National Science Foundation under Grant NSF CCR-9526325, and in part by DARPA under Contract DABT-63-97-0035.

†Center for Parallel and Distributed Computing, ECE Dept., Tech. Institute, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208-3118, {dhruva, banerjee}@ece.nwu.edu

```
do time = time_start,time_stop
   forall (i=2:n1-1:1,j=2:n2-1:1)
      a(i,j) = a(i,j-1)+a(i-1,j)+a(i+1,j)+a(i,j+1)
   endforall
   forall (i=1:Reg2IrregDim:1)
      x(Reg2Irreg_Irreg(i)) = a(Reg2Irreg_Reg1(i),Reg2Irreg_Reg2(i))
   endforall
   forall (i=1:Nedges:1)
      y(ia(i)) = y(ia(i)) + (x(ia(i))+x(ib(i)))/4
      y(ib(i)) = y(ib(i)) + (x(ia(i))+x(ib(i)))/4
   endforall
   forall (i=1:Reg2IrregDim:1)
      a(Reg2Irreg_Reg1(i),Reg2Irreg_Reg2(i)) = x(Reg2Irreg_Irreg(i))
   endforall
enddo
```

Figure 1. A program containing both regularly and irregularly distributed data

```
do i = 1, N
   forall (j=1,N)
      do k = ar(j),ar(j+1)-1
         r(j) = r(j) + ad(k)*q(ac(k),i)
      enddo
   endforall
   forall (j=1,N)
      a(i) = a(i) + q(j,i)*r(j)
   endforall
enddo
```

Figure 2. A program fragment containing hybrid accesses

formation between the structured and unstructured mesh, sweep over an unstructured mesh and exchange of boundary information between the unstructured and structured mesh. A run-time library specialized for handling regular accesses (like the Multiblock Parti library) can be used to parallelize the first (regular) loop, while a run-time library specialized for handling irregular accesses (like the CHAOS library) can be used to parallelize the other three loops. The library Meta-Chaos [2], which can be used at the interfaces between the Multiblock Parti and the CHAOS library, serves to efficiently copy and communicate data between the other two libraries, which typically use different distributions. While this approach of providing a meta-library to help different specialized libraries to interoperate allows us reap some of the advantages of the individual libraries, the advantages are limited when we consider interaction between structured and unstructured data structures at a finer level, as shown in Figure 2. It may be noted that interaction between regular and irregular structures in Figure 1 is limited to loop boundaries and any individual loop is either only regular or only irregular.

Figure 2 shows a program fragment extracted from Lanczos Algorithm. As is evident, both regular and irregular accesses are present within the same loop and the same array q is accessed both in a regular and irregular manner. Since the array q is not modified between the two inner loops, all the off-processor elements for array q can be aggregated in one message and can be communicated in one step using a unified scheme. However, using multiple interoperable libraries, the off-processor elements corresponding to different loops cannot be aggregated and communicated in a single step because different libraries are used for different loops. Moreover, since hybrid accesses occur within any given loop, none of the libraries alone is best suited for parallelizing it since none of the libraries alone supports the most optimized data distribution and internal data structure representation for hybrid accesses.

3. Outline of Our Scheme

3.1. Compile-time Analysis for Regular References

In the case of regular accesses in irregular applications, we use a technique different from those usually used for regular accesses and regular distributions. At compile-time, we try to build up the data structures that would typically be built by a scheme employing the inspector-executor paradigm. Section 4 describes these structures in more detail. If the array extents and loop bounds are known at compile-time, the compiler generates the trace arrays as well as the set of addresses that need to be communicated. In such cases, the only overhead incurred at run-time is for the division of intervals which may be necessary in order to make the interval structures for the various accesses on the same statement compatible to each other. If the loop-bounds and the number of processors are unknown at compile-time, the compiler generates structures, parameterized by the unknowns, which are eventually evaluated at run-time. All these structures are at first generated in the global address domain. However, if an array is never accessed using an indirection array and if all the usual parameters are known at compile-time, even localization is performed at compile-time and local references are generated by the compiler itself. In the case of arrays accessed in a hybrid manner, localization is postponed to the run-time phase in order to avoid duplicating communication and storage space. If fully resolved, all these structures generated at compile-time are present in the generated code in the form of initialized vectors. The compiler-generated calls to the run-time library build upon these initialized arrays in order to optimize both storage space and inter-processor communication cost. In contrast to usual schemes, the compile-time phase in our framework rarely writes out the actual communication code; on the other hand, it feeds the compile-time-generated communication structures to a run-time routine that finalizes the communication structures at run-time after resolving all compile-time unknowns (irregularities).

3.2. Compile-time Analysis for Irregular and Semi-Regular References

As a part of compile-time analysis, loop and index normalization are performed which help in the detection of

irregularity. In order to compile an irregular application, it needs to be decided whether irregular communication is needed and at what level this communication can be aggregated. Moreover, we determine the outermost loop at which we can lift the computation of the inspector, eliminating unnecessary re-inspections. The analysis that allows us to perform all these optimizations is currently loop-based and uses array data dependence. One of the most important features of our scheme is its ability to have a variety of internal representations of communication patterns. However, the selection of the best representation can sometimes be involved. In order to simplify this task, we have implemented a few heuristics in an access characterization pass. These heuristics rely on the normalization passes and use a combination of dependence analysis and syntactic pattern matching.

Once the analysis is performed, the compiler is able to generate code containing calls to the PILAR run-time library. The compiler inserts calls to build up the preprocessing data structures, known as the inspector. This phase contains calls to the run-time library for generating the translation tables and the trace arrays for every different access pattern. Calls are added for building the communication schedules once the translation tables and the trace arrays are determined. The next phase is usually called the executor where inter-processor communication takes place using the schedules generated in the previous step. The executor phase also consists of actual computation after all off-processor data is obtained. Once the communication point is determined, the compiler simply inserts calls to the run-time library to perform these tasks.

3.3. Run-Time Processing

Run-time processing includes building up of translation information for semi-regular and irregular distributions, generation of trace arrays for the irregular references and computation of communication schedules. The trace arrays and communication schedules built at run-time are aggregated with those built at compile-time if allowed by dependences. Localization of array references is also performed as a part of run-time preprocessing. Finally, inter-processor communication and computation take place.

A significant feature of our run-time library is that it supports multiple internal representations suitable for purely regular, purely irregular and hybrid accesses. Efficient set operations and conversion operations have been implemented for these representations which makes our library suitable for hybrid applications.

4. Details of Our Scheme

This section describes the scheme that we have used for handling applications containing hybrid accesses. The scheme is described in detail in [3]. The scheme uses the inspector-executor paradigm. However, though the analysis for purely regular accesses conforms in general to the inspector-executor paradigm, it is much simpler and does not incur the type of preprocessing overheads associated with irregular accesses.

4.1. Translation Information

4.1.1 MRD: Sparse Matrix Representation

Let us consider a sparse matrix $A(1:N_1, 1:N_2)$. A run-time partitioner examines the sparsity pattern of the matrix A and distributes the three associated vectors, DA, CO and RO according to multiple recursive decomposition. Once the matrix is distributed onto a virtual grid of processors at run-time, the extent of the array A on a processor grid can be represented by a two-dimensional interval descriptor in the following way: $Extent(A,p) = (l_{p_x} : u_{p_x})X(l_{p_y} : u_{p_y})$ where l_{p_x} and u_{p_x} denote respectively the starting and ending row of the partition assigned to processor p while l_{p_y} and u_{p_y} denote respectively the starting and ending column of the partition assigned to processor p. This information for every processor is stored in a translation table. Since the elements owned by a processor are represented in the form of intervals (sets), the translation table is small (length of translation table equals the number of processors) and can usually be replicated on the processors. Assuming that the program references an element of array A as $A(m_1, m_2)$, the processor index owning the reference is obtained by searching the translation table for the processor (p_x, p_y) such that the following conditions are satisfied:
$l_{p_x} <= m_1 <= u_{p_x}$ & $l_{p_y} <= m_2 <= u_{p_y}$
The local offset in a processor p for an element $A(m_1, m_2)$ for the above representation is given by
$O(m_1, m_2) = (m_1 - l_{p_x}, m_2 - l_{p_y})$

Alternative schemes for storing translation information may enumerate all the information as lists or may store contiguous non-zero elements as one-dimensional intervals. Both these schemes will require more memory than the above scheme employed by us; while the time to dereference an access for the processor index requires $O(P)$ time (P is the total number of processors) in our scheme, other schemes may take as much as $O(D)$ where D is the number of non-zero elements in the matrix. However, the dereferencing method described above is applicable only when the sparse structure is not represented as a dense matrix. Most applications store the sparse matrix in a dense compressed format and we show next how dereferencing can still be done efficiently.

4.1.2 MRD: Dense Matrix Representation

We shall now discuss the way we handle the MRD-CRS distribution. The same holds true for MRD-CCS distribution.

We first explain the way we store the data in the case of the MRD-CRS distribution. Our storage scheme for the data is different from the one used in [4]. The latter scheme reconstructs all the three vectors DA, CO and RO

in every individual processor after determining the partitions. While their scheme does not use translation tables explicitly, it uses auxiliary data structures to preserve all the necessary information. Our scheme distributes the DA and CO vectors according to the MRD distribution — however, the CO vector is not reconstructed in the individual processors — the values in the distributed CO vector represent the global address domain. We distribute the RO vector in a one-dimensional blocked way and all the values in the distributed RO vector represent the values present in the original sequential program. This means that, keeping similarity with the CO vector, the RO vector is not reconstructed either in individual processors and the values in the distributed RO vector are in the global address domain.

This way of handling storage of the data vectors for MRD-CRS distribution is significant in a number of ways. This scheme allows us treat the three vectors independent of each other as soon as the translation structures are created. While this kind of generality may not be necessary in most cases, it does give us the flexibility to generate optimized code if there is interaction between only a subset of the three vectors with other arrays distributed in a different way. Since we use a uniform scheme based on the inspector-executor paradigm to handle different distributions in a uniform way, this way of storage allocation is essential in order to generate traces of references in the global address domain. It may be noted that these traces in the global address domain will ultimately be converted to those in the local address domain and all computation will be done ultimately in the local address domain. This allows a uniform way of code generation and potential sharing of communicated data between distinct data accesses. If the arrays were reconstructed in individual processors, the trace method could not be applied. Reconstruction of the RO vector in individual processors leads to a higher usage of memory compared to our scheme, since the former scheme involves some duplication of the information contained in the RO vector among processors.

We now explain the translation tables we maintain for the MRD-CRS distribution. The translation tables store the extent of the vectors DA, CO and RO for every processor. It may be noted that the implementation stores only one set of information for the vectors DA and CO since the translation information are identical for these two vectors. The translation information is always stored in an interval format. This allows regularity to be exploited in the vectors DA and CO even though the corresponding elements may not be contiguous in the original sparse matrix. Such information is always stored in the form of intervals which substantially reduces both memory and processing time requirements. The translation information for the RO vector is equivalent to the one for block-distributed vectors.

The translation tables used by us for the MRD-CRS representation are significant in a number of ways: All information is stored in an interval format. No translation table needs to be maintained for the 1d-block-distributed RO vector since the corresponding translation information can be captured by analytical expressions. It may be noted that this method of distribution of the RO vector deviates slightly from the MRD-CRS distribution. However, it is only natural that the RO vector be distributed in a purely regular fashion since it is never accessed through index arrays. While in this case, we have distributed the RO vector in a 1d-blocked fashion, we consider the distribution of of the accumulating array in deciding on the distribution of the RO vector in order to minimize inter-processor communication. In case of a conflict between different distributions for multiple left-hand-sides, we distribute the RO vector in a blocked fashion. The number of entries in the translation table for DA/CO is bounded by *(number of rows)*(number of row-wise processors)*.

The computation of the index of the owning processor in the case of vectors DA/CO turns out to be an inexpensive one. Since the global index entries are kept sorted in the translation table for DA/CO, binary search is applied in order to find the processor owning a particular element. The time required for this operation is bounded by *lg(number of rows * number of row-wise processors)*. Once the processor owning a particular data element is decided, the translation table for DA/CO is scanned linearly determining the local address. The time required for this operation is thus bounded by *(number of rows * number of row-wise processors)*. The scheme used for one-dimensional generalized block distribution is similar and is explained in more detail in [3].

5. Experimental Results

5.1. Methodology

In this study, whenever we refer to results as obtained by the scheme *Pilar (Enu)*, we mean that the enumerated representation (similar to that used by *CHAOS*) has been used throughout irrespective of the type of accesses. Otherwise, the interval representation is used whenever possible and this will henceforth be referred to as *Pilar (Int)*. We have used the multiple recursive decomposition scheme for distribution of sparse matrices. Since this kind of distribution is outside the purview of HPF and since these are not supported by commercial HPF compilers, we haven't included run-times for commercial compilers (except in one case). However, it has been shown in previous work [1] that our scheme performs better than commercial compilers even for distributions supported in HPF.

The total run-times reported include reading the sparse matrix as well as the initial distribution time. The component run-times are not presented here for lack of space but can be found in [3]. The inspector run-time includes only the preprocessing costs required by our scheme — it does

not include the time required to read in the matrix and its initial distribution. These two components are included in the executor run-times reported.

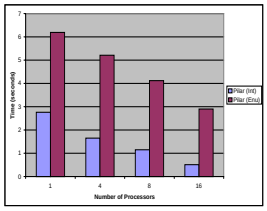
The platform chosen was a 16-processor *IBM SP-2* running *AIX 4.1*. The *SP-2* is a distributed-memory parallel machine and the installation uses sixteen 120 MHz *thin nodes* each with 128 MB of main memory. For all results reported in later sections, the *high performance communication switch* has been used for inter-processor communication. We have selected *MPI* as the communication library to be used by *PILAR*. The MPI version used was IBM's own optimized version of *MPI*. All programs were compiled at an optimization level of 2. All results were taken using the *SP-2 user space library* as the communication subsystem library. All timings reported are wall clock times in seconds.

We have used two sparse matrices. The first is titled CIRPHYS (Circuit Physics Modeling) and identified as JPWH991 in the Harwell-Boeing collection [5]. This is an unsymmetric matrix with variable band characteristics. It is a matrix of order 991 and contains 6027 non-zero entries. The second is titled PSMIGR (Inter-county migration) and identified as PSMIGR1 in the Harwell-Boeing collection. This is a denser matrix compared to the previous one and is of order 3140 and contains 543162 non-zero entries.

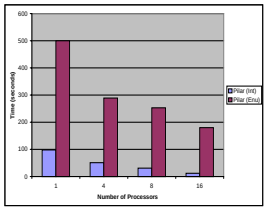
5.2. Benchmarks

5.2.1 Matrix-Vector Multiplication

The accumulating array and the RO vector have been distributed in a one-dimensional blocked manner. The DA and CO vectors are distributed using multiple recursive decomposition. Figure 3(a) and (b) show respectively the total run-times obtained for different number of processors for the matrices, CIRPHYS and PSMIGR.



(a) CIRPHYS

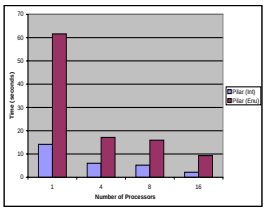


(b) PSMIGR

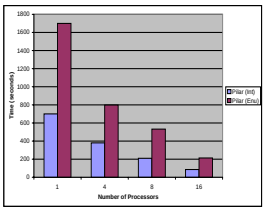
Figure 3. Runtimes for Matrix-Vector Mult.

5.2.2 LU Decomposition

This routine factors a square non-singular matrix A into a lower triangular matrix L and a upper triangular matrix U such that $A = LU$. This code contains the sparse matrix distributed according to multiple recursive decomposition. The program fragment contains a number of regular and irregular accesses in the same statement. An important characteristic which makes this program different from the previous benchmark is that many of the left-hand-sides are irregular since the elements of the sparse matrix have to be modified. Figures 4(a and b) show the total runtimes for the matrices, CIRPHYS and PSMIGR. The run-times scale better for the matrix PSMIGR for this benchmark as well. This is expected since this matrix is denser and is more uniform.



(a) Total RT (CIRPHYS)



(d) Total RT (PSMIGR)

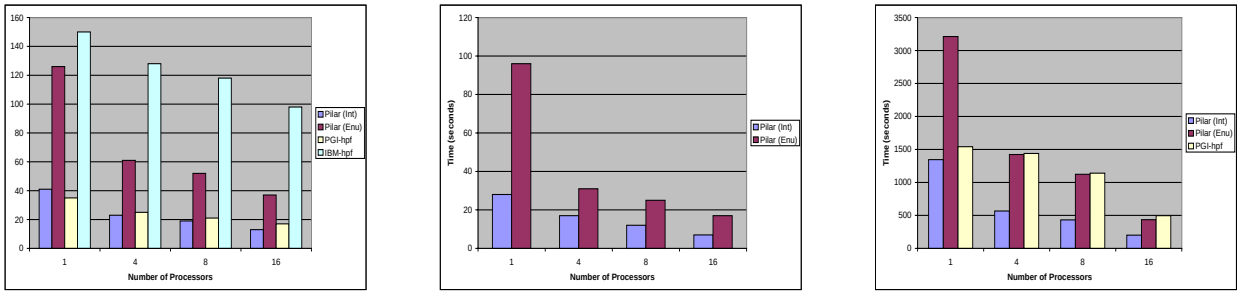
Figure 4. Runtimes for LU Decomposition

5.2.3 Lanczos Algorithm

Lanczos Algorithm [6] contains a number of regular and irregular accesses within the same loop. A couple of scenarios have been considered. In one scenario, in order to compare our scheme with the ones used by commercial compilers, we have distributed all the arrays and vectors in a blocked manner. As Figure 5(a) shows, Pilar (Int) performs better than PGI-hpf [7] and IBM-hpf [8]. In the second scenario, we have distributed the sparse matrices using multiple recursive decomposition. The comparative results are shown in Figure 5(b). In this case, too, Pilar (Int) performs better than Pilar (Enu) and as expected, the run-times are better than the block-distributed case. Figure 5(c) shows comparative results for the second matrix. Pilar (Int) performs better for this matrix since this matrix is more uniform and more dense. The run-times for IBM-hpf are not shown since it did not finish in the one-hour timeout period.

5.3. Significance of the Results

An important feature of the results obtained using our unified scheme is that the preprocessing costs are considerably small compared to the total run-times. We believe that our scheme substantially reduces the inter-processor communication during schedule generation. While we eliminate inter-processor communication totally for regular references and regular distributions, this cost is substantially reduced for semi-regular distributions. For MRD distribution, once a processor determines the global indices it needs from a remote processor p, it sends only the lower and upper indices bounding the concerned region in an interval format. This information is sufficient for the processor p to infer the data elements it needs to send. We try to reuse schedules even across regular and irregular accesses. Contrary to schemes which often approximate accessed references by some smallest regular section and communicate the entire section, our scheme always communicates exactly what is required for computation. We believe that this helps our scheme obtain better results.



(a) Block-CIRPHYS (b) MRD-CIRPHYS (c) Block-PSMIGR

Figure 5. Total Runtimes for Lanczos Algorithm

6. Related Work

CHAOS/PARTI [9], Multiblock PARTI [10] and Meta-CHAOS [2] help in parallelizing irregular applications. Ujaldon and Zapata [11] have proposed an approach to reduce the number of levels of indirect array accesses. Bik and Wijshoff [12] have developed a framework in order to deal with sparsity of data structures at the compilation level. [13] describes a novel approach to sparse and dense SPMD code generation by viewing arrays as distributed relations and parallel loop execution as distributed relational query evaluation. Other related work are described in [3].

7. Conclusions

This paper presents a uniform compilation framework for parallelization of irregular applications that may contain arrays distributed using purely regular, purely irregular or semi-regular distributions like multiple recursive decomposition (MRD) distributions. The key focus of this work is in showing how unlike other existing schemes, this framework handles all the relevant distributions using compatible though specialized structures which enable minimal preprocessing overheads along with sophisticated communication optimizations like reduction of inter-processor communication during schedule generation and sharing of communicated information across regular and irregular accesses. It is only natural that preprocessing overheads associated with semi-regular distributions be intermediate between that involved for regular distributions on the one hand and irregular distributions on the other. However, existing schemes fail to exploit this feature. This work shows how an interval representation can be used for semi-regular distributions which not only reduces the preprocessing overhead but also makes code generation simpler for loops and statements containing arrays with differing distributions. We present methods that try to aggregate messages across regular and irregular accesses, if possible. Experimental results on a 16-processor IBM SP-2 show that our scheme is feasible.

References

[1] Dhruva R. Chakrabarti, Nagaraj Shenoy, Alok Choudhary and Prithviraj Banerjee, *An Efficient Uniform Run-time Scheme for Mixed Regular-Irregular Applications*, Proceedings of The 12th ACM International Conference on Supercomputing (ICS'98), Melbourne, Australia, July 1998.

[2] Guy Edjlali, Alan Sussman, Joel Saltz, *Interoperability of Data Parallel Runtime Libraries with Meta-Chaos*, University of Maryland Technical Report: CS-TR-3633 and UMIACS-TR-96-30 May 1996.

[3] Dhruva R. Chakrabarti, Nagaraj Shenoy, Alok Choudhary and Prithviraj Banerjee, *A Uniform Scheme for Parallelizing Mixed Regular-Irregular Applications*, Technical Report No. CPDC-TR-9802-011, Center for Parallel & Distributed Computing (http://www.ece.nwu.edu/cpdc), Northwestern University, February 1998.

[4] M. Ujaldon, E. Zapata, B. Chapman and H. Zima, *Vienna-Fortran/HPF Extensions for Sparse and Irregular Problems and their Compilation*, Technical Report TR 95-5, Institute for Software Technology and Parallel Systems, University of Vienna (October 1995).

[5] I. Duff, R. Grimes and J. Lewis, *User's Guide for the Harwell-Boeing Sparse Matrix Collection (Release 1)*, CERFACS, France.

[6] Lanczos Algorithm, *http://www.netlib.org/lanczos*.

[7] The Portland Group, Inc., *pghpf* Version 2.2, 1997.

[8] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Y. Wang, W. M. Ching and T. Ngo, *An HPF compiler for the IBM SP-2*, Proceedings of Supercomputing '95, San Diego, CA, December 1995.

[9] Joel Saltz et. al., *A Manual for the CHAOS Runtime Library*, UMIACS, University of Maryland, 1994.

[10] G. Agarwal, A. Sussman and J. Saltz, *Efficient Runtime Support for Parallelizing Block Structured Applications*, Proceedings of Scalable High-Performance Computing Conference, 1994, pp. 158-167.

[11] M. Ujaldon and E. Zapata, *Efficient resolution of sparse indirections in data-parallel compilers*, Proceedings of the 9th ACM International Conference on Supercomputing, pp. 117-126, July 1995.

[12] A.J.C.Bik, *Compiler Support for Sparse Matrix Computations*, Ph.D. Thesis, ftp://ftp.wi.leidenuniv.nl/pub/CS/PhDTheses/bik-96.ps.gz.

[13] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill, *Unified framework for sparse and dense SPMD code generation*, Technical Report 97-1625, Cornell Computer Science Department, 1997.