Towards a Uniform Run-time Scheme for Mixed Regular-Irregular Applications

Dhruva R. Chakrabarti   Nagaraj Shenoy   Alok Choudhary   Prithviraj Banerjee

Technical Report No. CPDC-TR-9801-004

January 1998

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

Towards a Uniform Run-time Scheme for Mixed Regular-Irregular Applications *

Dhruva R. Chakrabarti Nagaraj Shenoy Alok Choudhary Prithviraj Banerjee

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

Corresponding author: Dhruva R. Chakrabarti

Abstract

Almost all applications containing indirect array addressing (irregular accesses) have a substantial number of direct array accesses (regular accesses) too. A conspicuous percentage of these direct array accesses usually requires inter-processor communication for the applications to run on a distributed memory multicomputer. This study highlights how lack of a uniform representation and lack of a uniform scheme to generate communication structures and parallel code for regular and irregular accesses in a mixed regular-irregular application prevent sophisticated optimizations. Furthermore, we also show that code generated for regular accesses using compile-time schemes are *not always* compatible to code generated for irregular accesses using run-time schemes. In our opinion, existing schemes handling mixed regular-irregular applications either incur unnecessary preprocessing costs or fail to perform the best communication optimization. This study presents a uniform scheme to handle both regular and irregular accesses in a mixed regular-irregular application. While this allows for sophisticated communication optimization like message coalescing, message aggregation to be made across regular and irregular accesses, the preprocessing costs incurred are likely to be minimum. Experimental comparisons for various benchmarks on a 16-processor IBM SP-2 show that our scheme is feasible and better than existing schemes.

1 Introduction

Distributed-memory multi-computers have the potential to provide high levels of performance required to handle the Grand Challenge Computational Science problems [1]. In recent years, a significant amount of research has been devoted to the development of source-to-source parallelizing compilers for multi-computers that relieve programmers of the burden of data and computation partitioning and communication generation [3, 4, 5, 6, 7, 8]. A standard language, *High Performance Fortran (HPF)* [2] has been developed for easy specification of these tasks. For the purpose of this study, we refer to regular applications as those which contain array references of the form $A(f)$ where A is an array distributed

*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.

regularly either in a blocked or cyclic(k) fashion among processors and f is an affine function of the enclosing loop variables. A number of commercial compilers for HPF have been made available from the industrial sector by companies such as Applied Parallel Research, Digital Equipment Corporation, The Portland Group, IBM and others. As a part of his *HPF* program, the user can manually specify a preferred data distribution or it can be automatically chosen by the compiler. Once the distribution of data on processors is determined, the compiler can proceed to analyze the data accesses and ultimately generate optimized *SPMD* code. This process involves both partitioning of computation and generation of inter-processor communication. Efficient inter-processor communication is generated by vectorizing and coalescing messages and hoisting them out of loops if allowed by dependences. This development has helped in providing the user with a higher level of programming interface for data-parallel regular applications while passing on some of the complexities to the compiler.

However, many important applications are irregular. For the purpose of this study, we refer to irregular accesses as those array references addressed using one or multiple index arrays. The use of arrays for addressing other arrays makes communication patterns input-dependent, disabling traditional compiler optimizations. Simultaneous use of distribution patterns which are not regular leads to lack of structure in the code presenting difficulties in specifying data and computation distribution while complicating communication generation. However, a large subset of these applications has communication that repeats across multiple iterations. This feature can be exploited by providing a run-time library which analyzes the structure of the irregular accesses before actual computation in a preprocessing step and generates optimized communication. This is typically referred to as an inspector-executor [9] paradigm. The *inspector* phase stores the global to local translation information depending on the data distribution, examines the global irregular references made by a processor and determines the communication requirements, all at run-time. The communication patterns are optimized to generate an efficient *schedule*. The result of this preprocessing step is then used multiple times in the executor phase (which does the communication and computation) thus amortizing its total cost. Runtime libraries like *CHAOS/PARTI* [10] and *PILAR* [13] simplify implementation of this preprocessing step and subsequent inter-processor communication.

We refer to applications containing a mixture of regular and irregular accesses as mixed regular-irregular applications. While most run-time libraries for such applications focus on optimizations for the irregular accesses, the optimizations done for the regular accesses are often not the best and often incompatible with those for irregular accesses. This problem is discussed in more detail in subsections 2.1 and 2.2. This paper presents a unified scheme to handle both regular and irregular accesses in a program by employing the same representation for both of them. This allows more sharing of information and reasonably good optimization for both regular and irregular accesses mixed in a program. While [21] discusses how regular accesses can be efficiently handled, this paper deals with mixed regular-irregular applications.

The outline of the remainder of the paper is as follows. Section 2 explains in detail the problem we are trying to solve while section 3 outlines the scheme used by us. Experimental results are presented in Section 4. Comparisons with other existing schemes show the feasibility of our approach. The last two sections discuss the relevance of this work in the context of some related work and present our conclusions.

2 Problem Description

2.1 Common References among Regular and Irregular Accesses

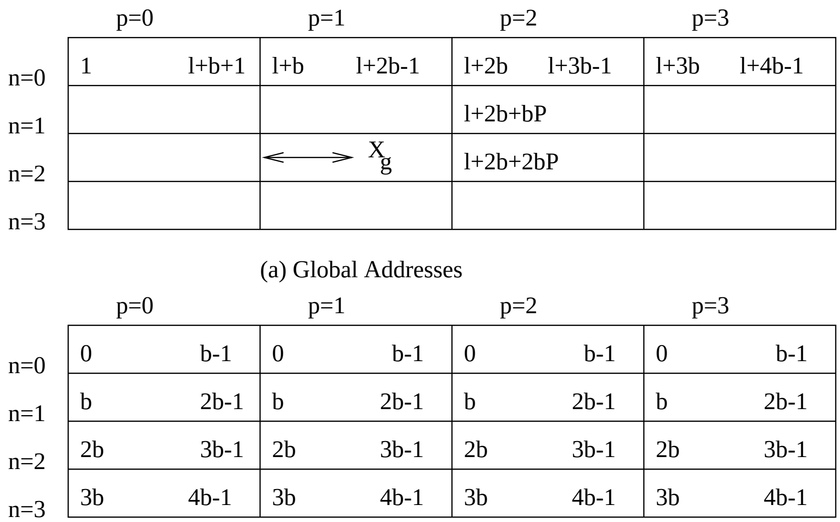
We analyze a typical program fragment that may occur in a mixed regular-irregular application. Figure 1(a) shows a program fragment where a dense array a is accessed both in a regular (line 3) and irregular (line 8) manner. Some examples where this feature is observed are the Lanczos Algorithm and the Conjugate Gradient Algorithm when sparse matrices are involved. A framework which handles regular accesses using compile-time schemes and irregular accesses in the same program using run-time schemes is shown in Figure 1(b). However, since the communication sets for the regular accesses are generated at compile-time and the communication sets for the irregular accesses are generated at run-time, it is not possible to combine them even when there are common accesses among the two. This will potentially lead to redundant communication. On the contrary, a unified run-time scheme for regular and irregular accesses as that shown in Figure 1(c) will be able to generate much less communication decreasing both its number and volume. The above observation holds good also in cases where both the regular and irregular accesses to the same array occur within the same loop.

1: **do** i = 1, n-1 2: **do** j = 1, n 3: b(j,i) = a(j,i+1) 4: **enddo** 5: **enddo** 6: **do** i = 1, n-1 7: **do** j = 0, ar(i+1)-ar(i)-1 8: c(i) = a(i,ar(i)+j) 9: **enddo** 10: **enddo**	Analyze comm. requirements for regular access to a at compile-time Generate communication code at compile-time **do** i = lb, ub **do** j = 1, n b(j,i) = a(j,i+1) **enddo** **enddo** Analyze comm. requirements for irregular access to a at run-time Generate communication code at run-time **do** i = lb, ub **do** j = 0, ar(i+1)-ar(i)-1 c(i) = a(i,ar(i)+j) **enddo** **enddo**	Analyze comm. requirements for regular access to a at run-time Analyze comm. requirements for irregular access to a at run-time Compute union of the two sets at run-time Generate communication code at run-time **do** i = lb, ub **do** j = 1, n b(j,i) = a(j,i+1) **enddo** **enddo** **do** i = lb, ub **do** j = 0, ar(i+1)-ar(i)-1 c(i) = a(i,ar(i)+j) **enddo** **enddo**
(a)	(b)	(c)

Figure 1: Exploiting Common References in a Mixed Regular-Irregular Application

We are not aware of a framework which tries to integrate the analysis for regular and irregular accesses using a representation that suits both. Such a representation must have support for regular sections in order to effectively deal with regular accesses and spatial regularity in midst of irregularity, it must have support for total irregularity and it must have efficient mechanisms for conversion between sections obtained from regular and irregular accesses. It appears to us that the popular run-time library for irregular applications, *CHAOS* [10, 11, 12], has special support for irregular accesses alone. It is not clear to us how this library can be used efficiently for mixed regular-irregular applications; it appears that compilers using the *CHAOS* library create expensive preprocessing structures for *regular off-processor* accesses. In the presence of a run-time library that has special support for mixed regular-irregular applications, much of this preprocessing cost can be amortized. Using such a library *PILAR* [13], it has been shown in [21] how preprocessing costs can be reduced to a minimum. On the other hand, *multi-block PARTI* [25] has support for regular sections but it has not been integrated with *CHAOS* and hence not entirely usable for mixed regular-irregular applications.

2.2 General Compile-time and Run-time Schemes

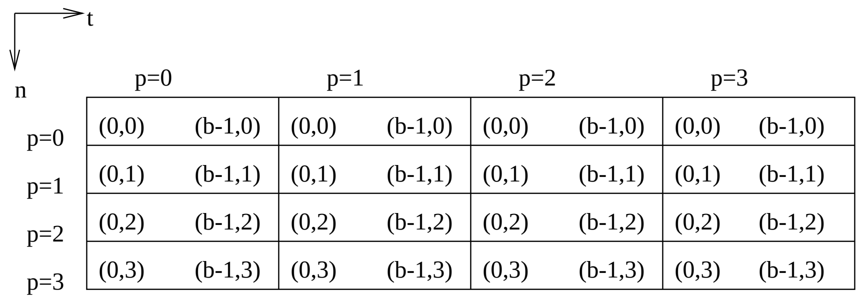


(b) Local Addresses

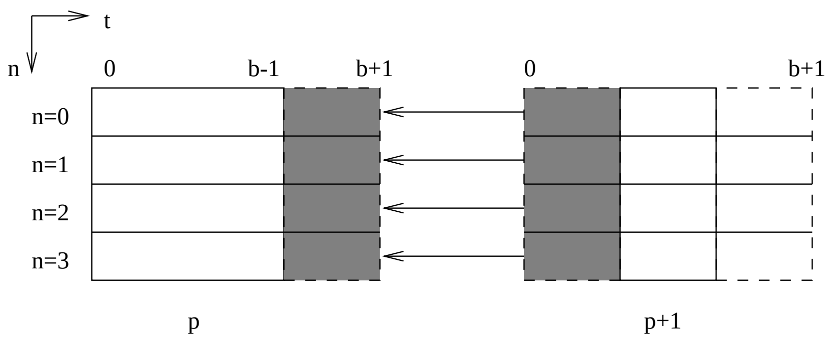
Figure 2: Global to Local Address Translation: Method 1

General compile-time schemes for regular accesses accept arbitrary HPF data alignment and distribution directives, loop iterations and array accesses and ultimately generate distributed code with space-efficient array allocation, tight loop bounds and optimized communication [14, 15, 16, 17, 18]. Using a linear algebra framework [17], the HPF compilation problem can be converted into solving a set of linear equalities and inequalities while polyhedron manipulation and scanning techniques can be used to generate code for a distributed-memory machine from the linear form. In this framework, the structure of loops and the array accesses in the code generated are often different from that in the serial code. For instance, the number of free variables in the set of equalities and inequalities determines the number of loops in the generated code. Typically, the global space in a serial code is mapped into local spaces of processors in the generated code. For example, in block-cyclic distributions, the set of local iterations in the compute set for each processor in the generated code is no more expressed in terms of the iteration variable (present in the serial code) but is expressed in terms of the block size and the cycle count. Change of array dimensions, coordinates and basis is performed in order to minimize the space allocated in local memory and to generate a minimal number of loops. Thus the structure of the original loop is not preserved in the optimized code generated for distributed memory machines.

Moreover techniques which change the dimensionality of a matrix are often used. [18] uses a technique where for cyclic(b) distributions, each dimension may be cast into two dimensions, as shown in Figure 3. A motivation behind using multiple processors lies in the reduction of memory requirements as the number of processors increases. Figure 2 shows the local address allocation for an array A(l:u) distributed as cyclic(b) on $P = 4$ processors. The array declaration on each of the 4 processors changes from $A(l : u)$ to $A(0 : \lceil \frac{u-l+1}{p} \rceil)$. The local index X_l can be written as $X_l = X_g - l - bPn - bp + bn$, where $n = \lfloor \frac{X_g - l}{bP} \rfloor$. Let us now consider shift communication for which overlap areas have to be allocated. Assuming the type of local address allocation shown in Figure 2, one way would be to allocate contiguous buffer space for off-processor data on either side of the local partition.



(a) Local Address mapped to a two-dimensional space



(b) Overlap area using the 2D format of local address

Figure 3: Global to Local Address Translation: Method 2

In the case of simple shift communication, access to data would be expensive because every access has to be characterized first as either local or non-local. If local, the simple translation function derived above can be used; if non-local, the corresponding complicated function depending on the actual storage scheme has to be used. An efficient scheme to buffer non-local data while allowing for simple and inexpensive accesses is difficult to find since for cyclic(b) partitions, the local partition is not contiguous in the original global space. However, if each dimension of the matrix is cast into two dimensions, the access pattern often improves radically along with optimized memory usage. In Figure 3, a global reference A(X_g) is translated into A(t,n). While this allows for compact allocation of non-local space (overlap areas, for instance, can be allocated just by extending the first dimension on either side of the block) and simple access functions, it changes the original structure of the code. A typical change in the structure of the code is shown in Figure 4(a, b).

Distributed memory code generated by run-time schemes often change the structure of the serial code in a way that is not compatible to the change effected by compile-time schemes. This is because irregular accesses and distributions cannot be represented by linear equalities and inequalities. One such scheme has been implemented by the run-time library PILAR (Parallel Irregular Library with Application of Regularity) [19]. This is a C++ library designed to support different types of applications which have

do i = 1, n a(i) = ... **enddo**	**do** i = lbi, ubi **do** j = lbj, ubj d1 = . . . d2 = . . . a(d1,d2) = . . . **enddo** **enddo**
(a) Serial Code	(b) Parallel Code from a Compile-time Scheme
do i = 0, n **do** j = 0, r(i) a(h(i)+j) = . . .	count = 0 **do** i = 0, n **do** j = 0, numInt(i) **do** j2 = 0, newH(count+1)-newH(count)-1 a(newH(count)+j2) = . . . **enddo** count = count + 2
(c) Serial Code	(d) Parallel Code from a Run-time Scheme
do i = 0, n **do** j = 0, r(i) a(i) = b(c(i)+j) **enddo** **enddo**	The schemes shown in (b) and (d) cannot be combined here for lack of compatibility in the loop structure.
(e) Serial Code	(f) Parallel Code

Figure 4: Changes in Code Structure Effected by Compile-time and Run-time Schemes

purely regular accesses, purely irregular accesses and mixed regular and irregular accesses. One of the internal representations used by this library is the interval-based representation which is designed to handle purely regular array accesses as well as spatial regularity in irregular array accesses. Owing to the data distribution, an interval in globals may get decomposed into multiple intervals and sub-intervals. So an extra loop is usually required in the distributed code to enumerate all the sub-intervals. An example of such a change in structure of the loop nests is shown in Figure 4(c, d). Clearly, this change in structure is not compatible to the change in structure generated by compile-time schemes as highlighted in Figure 4(e, f).

As in many existing schemes, it is possible to generate code without changing the structure of the sequential code. In such cases, different paradigms for regular and irregular accesses can coexist. However, in our opinion, it is not possible to perform the best optimization for both regular and irregular accesses in such cases. Compile-time schemes for regular accesses described above try to optimize both time and memory requirements and in the process introduce a change in the sequential structure of the code. Similarly, a scheme for irregular accesses which takes into consideration spatial locality and tries to reduce the time and memory requirements introduces a change in the structure of the sequential code. Unfortunately, such changes for irregular accesses are incompatible to changes for regular accesses.

2.3 Commercial HPF compilers

We have studied two commercial HPF compilers, The Portland Group's (PGI) HPF compiler (version

real a(100,100), c(100,100)
!HPF$ align with a::c
!HPF$ distribute a(*,BLOCK)
!HPF$ independent
    do i = 1, 100
      do j = 1, 100
        a(j,i) = j
        c(j,i) = j
      enddo
    enddo
!HPF$ independent
    do i = 1, 48
      do j = 1, 100
        a(j,i) = c(j,2*i+3)
      enddo
    enddo
    call dummy(a)
	0→1: 20 bytes
	0→2: 20 bytes
	0→3: 20 bytes
	1→0: 5192 bytes
	1→2: 0 bytes
	1→3: 0 bytes
	2→0: 1192 bytes
	2→1: 4400 bytes
	2→3: 0 bytes
	3→0: 392 bytes
	3→1: 4800 bytes
	3→2: 0 bytes
		0→1: 116 bytes
		0→2: 20 bytes
		0→3: 4 bytes
		1→0: 4804 bytes
		1→2: 100 bytes
		1→3: 108 bytes
		2→0: 804 bytes
		2→1: 4404 bytes
		2→3: 4 bytes
		3→0: 4 bytes
		3→1: 4804 bytes
		3→2: 4 bytes

(a)	(b) pghpf's comm. for (a)	(c) Comm. of our scheme

Figure 5: Communication Generated using 4 processors of an IBM SP-2 for a code having a regular access

real a(100,100), c(100,100)
integer d(100,100)
!HPF$ align with a::c,d
!HPF$ distribute a(*,BLOCK)
!HPF$ independent
    do i = 1, 100
      do j = 1, 100
        a(j,i) = j
        c(j,i) = j
        d(j,i) = 2*i+3
      enddo
    enddo
!HPF$ independent
    do i = 1, 48
      do j = 1, 100
        a(j,i) = c(j,d(j,i))
      enddo
    enddo
    call dummy(a)
	0→1: 10020 bytes
	0→2: 10020 bytes
	0→3: 10020 bytes
	1→0: 10392 bytes
	1→2: 10000 bytes
	1→3: 10000 bytes
	2→0: 10392 bytes
	2→1: 10000 bytes
	2→3: 10000 bytes
	3→0: 10392 bytes
	3→1: 10000 bytes
	3→2: 10000 bytes
		0→1: 116 bytes
		0→2: 20 bytes
		0→3: 4 bytes
		1→0: 4804 bytes
		1→2: 100 bytes
		1→3: 108 bytes
		2→0: 804 bytes
		2→1: 4404 bytes
		2→3: 4 bytes
		3→0: 4 bytes
		3→1: 4804 bytes
		3→2: 4 bytes

(a)	(b) pghpf's comm. for (a)	(c) Comm. of our scheme

Figure 6: Communication Generated using 4 processors of an IBM SP-2 for a code having an indirect access with the same regularity as in Figure 5(a)

2.2) [20] and IBM's HPF compiler (version 1.1.0.0) [7]) in order to observe the amount of communication since reduction of communication is one of the focus of our work. We present here a few simple code fragments and the amount of communication generated by the PGI HPF compiler and our scheme. More details can be found in [22]. Our scheme is described in detail in subsequent sections. The code fragments were run on 4 processors of an IBM SP-2 and the amount of communication was obtained from the Visualization Tool (vt).

Figure 5(b & c) illustrate the amount of communication generated for a simple regular code fragment shown in Figure 5(a) by the PGI HPF compiler and our scheme. For all practical purposes, the amount of communication shown in Figure 5(b & c) are identical and the slight mismatch is due to the difference in protocol among processors used by the schemes.

Figure 5(a) and Figure 6(a) are semantically equivalent code fragments, the only difference being the use of an indexing array in the latter case. The communication generated can be expected to be the same in both the cases. However, as is shown in Figure 6(b), the communication generated for the code shown in Figure 6(a) by pghpf is more than the first one. It appears from the amount of communication and the intermediate code (not shown here) that while appropriate sections are communicated in the case of the code fragment shown in Figure 5(a), an all-to-all communication of entire off-processor elements takes place in the case of the code fragment shown in Figure 6(a). However, as shown in Figure 6(c), our scheme communicates only the required data.

```
real a(100,100), c(100,100)
integer d(100,100)
!HPF$ align with a::c,d
!HPF$ distribute a(*,BLOCK)
!HPF$ independent
        do i = 1, 100
            do j = 1, 100
                a(j,i) = j
                c(j,i) = j
                d(j,i) = 2*i+3
            enddo
        enddo
!HPF$ independent
        do i = 1, 48
            do j = 1, 100
                a(j,i) = c(i,d(j,i))
            enddo
        enddo
        call dummy(a)
```

(a)

(b) pghpf's comm. for (a)	(c) Comm. of our scheme
0→1: 2320 bytes	**0→1**: 116 bytes
0→2: 20 bytes	**0→2**: 20 bytes
0→3: 20 bytes	**0→3**: 4 bytes
1→0: 2892 bytes	**1→0**: 52 bytes
1→2: 0 bytes	**1→2**: 100 bytes
1→3: 0 bytes	**1→3**: 108 bytes
2→0: 2892 bytes	**2→0**: 12 bytes
2→1: 2300 bytes	**2→1**: 48 bytes
2→3: 0 bytes	**2→3**: 4 bytes
3→0: 2892 bytes	**3→0**: 4 bytes
3→1: 2300 bytes	**3→1**: 52 bytes
3→2: 0 bytes	**3→2**: 4 bytes

Figure 7: Communication Generated using 4 processors of an IBM SP-2 for a code having an irregular access

```
real a(100,100), c(100,100)
integer d(100,100)
!HPF$ align with a::c,d
!HPF$ distribute a(*,BLOCK)
!HPF$ independent
        do i = 1, 100
            do j = 1, 100
                a(j,i) = j
                c(j,i) = j
                d(j,i) = 2*i+3
            enddo
        enddo
!HPF$ independent
        do i = 1, 48
            do j = 1, 100
                a(j,i) = c(j,d(j,i))
                           + c(i,d(j,i))
            enddo
        enddo
        call dummy(a)
```

(a)

(b) pghpf's comm. for (a)	(c) Comm. of our scheme
0→1: 12320 bytes	**0→1**: 116 bytes
0→2: 10020 bytes	**0→2**: 20 bytes
0→3: 10020 bytes	**0→3**: 4 bytes
1→0: 12892 bytes	**1→0**: 4804 bytes
1→2: 10000 bytes	**1→2**: 100 bytes
1→3: 10000 bytes	**1→3**: 108 bytes
2→0: 12892 bytes	**2→0**: 804 bytes
2→1: 12300 bytes	**2→1**: 4404 bytes
2→3: 10000 bytes	**2→3**: 4 bytes
3→0: 12892 bytes	**3→0**: 4 bytes
3→1: 12300 bytes	**3→1**: 4804 bytes
3→2: 10000 bytes	**3→2**: 4 bytes

Figure 8: Communication Generated using 4 processors of an IBM SP-2 for a code having scope for message coalescing

Figures 7(b & c) show the communication incurred by the PGI HPF compiler and our scheme respectively for the irregular code shown in Figure 7(a). While Figure 6(a) has the access to array c as c(j,d(j,i)), Figure 7(a) accesses array c as c(i,d(j,i)). Though the PGI HPF compiler does not do an all-to-all communication in this case, the amount of communication is still more than what is incurred by our scheme. Figure 8(a) shows a program fragment where the messages can be potentially coalesced. Accesses in Figure 8(a) are present individually in Figure 6(a) and Figure 7(a). In the actual compute loop of Figure 6(a) and Figure 7(a), the i loop is distributed while the j loop is not distributed. In the actual compute loop, since the access in Figure 7(a) has i in the first dimension and the access in Figure 6(a) has j in the first dimension and the functions are the same in both the cases in the second dimension, the accesses in Figure 7(a) are a subset of those in Figure 6(a). Thus the communication requirement for the code in Figure 7(a) is also a subset of that for the code in Figure 6(a). However, the communication generated for the code in Figure 8(a) by the PGI HPF compiler is the sum of those generated for Figure 6(a) and Figure 7(a). The slight mismatch is due to the communication required for protocol. So it is clear for the code in Figure 8(a), that while *only one* of the r.h.s. irregular reference leads to communication of the entire non-local portions, the messages have not been coalesced in this case too. The scheme used by us does an exact communication analysis, does message coalescing and communicates exactly what is required. It may be noted that the communication generated by our scheme for the code shown in Figure 8(a) is the same as that for the code in Figure 6(a).

Figure 9(a) shows a code fragment where common elements have been accessed by the references on the r.h.s. and the messages can be potentially coalesced. One of the accesses is regular (same as that shown in Figure 5(a)) while the other is irregular (same as that shown in Figure 6(a). The amount of communication illustrated in Figure 9(b) shows that the messages have not been coalesced by the PGI HPF compiler. It is effectively the sum of communication generated for Figure 5(a) and Figure 6(a). The scheme used by us generates the minimum amount of communication.

```
real a(100,100), c(100,100)
integer d(100,100)
!HPF$ align with a::c,d
!HPF$ distribute a(*,BLOCK)
!HPF$ independent
        do i = 1, 100
            do j = 1, 100
                a(j,i) = j
                c(j,i) = j
                d(j,i) = 2*i+3
            enddo
        enddo
!HPF$ independent
        do i = 1, 48
            do j = 1, 100
                a(j,i) = c(j,d(j,i))
                          + c(j,2*i+3)
            enddo
        enddo
        call dummy(a)
```

(a)

(b) pghpf's comm. for (a)	(c) Comm. of our scheme
0→1: 10020 bytes	**0→1**: 116 bytes
0→2: 10020 bytes	**0→2**: 20 bytes
0→3: 10020 bytes	**0→3**: 4 bytes
1→0: 15192 bytes	**1→0**: 4804 bytes
1→2: 10000 bytes	**1→2**: 100 bytes
1→3: 10000 bytes	**1→3**: 108 bytes
2→0: 11192 bytes	**2→0**: 804 bytes
2→1: 14400 bytes	**2→1**: 4404 bytes
2→3: 10000 bytes	**2→3**: 4 bytes
3→0: 10392 bytes	**3→0**: 4 bytes
3→1: 14800 bytes	**3→1**: 4804 bytes
3→2: 10000 bytes	**3→2**: 4 bytes

Figure 9: Communication Generated using 4 processors of an IBM SP-2 for a code having regular and irregular accesses of the same array and having scope for message coalescing

Figures 5, 6, 7, 8 and 9 show a few cases where the communication generated by the PGI HPF compiler is more than what is required. It is, however, unclear to us whether the PGI HPF compiler can optimize communication in a better manner in other cases (different distributions, accesses etc.). In our opinion, derived from our study of various code fragments with different distributions and accesses (not everything is shown here), though the type of memory allocation and communication used by the PGI HPF compiler may simplify the array accesses, excess communication is likely to lead to a substantial increase in run-time.

In the experiments we have performed (not shown here for lack of space), we have observed that the HPF compiler from IBM on the SP-2 (xlhpf90) obtains a copy of the entire portion of the non-local array from other processors whenever an indirect access is encountered. Though this avoids preprocessing costs, this is likely to give rise to lots of unnecessary communication for most irregular applications.

2.4 Solution for Mixed Regular-Irregular Applications

We advocate using a uniform run-time scheme for handling both regular and irregular accesses in mixed regular-irregular applications. Quite frequently, spatial regularity exists in irregular accesses and a dense array is accessed both in a regular and irregular manner [24, 32]. All such accesses should be handled using an interval representation [19]. Irregular accesses devoid of any regularity whatsoever need to be handled using an enumerated representation. However, the interval representation has to be *compatible* to the enumerated representation in the sense that efficient conversion between the two representations should be possible depending on various factors. This will enable sophisticated communication optimization in the form of schedule reuse, incremental scheduling etc. irrespective of the type of accesses for which the schedules are originally generated. This will also generate uniform code for both regular and irregular accesses allowing them to coexist. As shown in [21], even in the worst case where all accesses and distributions are regular, our scheme would perform comparably with compile-time schemes. This is expected since even in a real-life *regular* application, the loop bounds and the processor size will be unknown at compile time and so most of the analysis and communication set generation will take place only at run-time even for compile-time schemes.

3 Outline of Our Scheme

This section describes the scheme that we have used for handling applications containing both regular and irregular accesses. The scheme is described in detail in [22]. The scheme uses the inspector-executor paradigm to handle irregular as well as regular accesses. 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. The scheme has been implemented in the *PARADIGM* compiler [6] and the run-time library *PILAR* (Parallel Irregular Library with Application of Regularity) [13]. The basic irregular compiler support required for irregular applications has been reported in [19]. More details are being reported in a separate study.

We will use the synthetic benchmark (Figure 10) as a running example to illustrate our scheme. The code is written in HPF version 2.0 [36] format. We wish to show how communication can be optimized across regular and irregular accesses. This is why we have used a dense array x accessed both in a regular and irregular manner. It may be noted that a frequently occurring access pattern in sparse code, $data(ind(i) + j)$, is present in the code too. This will show the effect of spatial regularity in midst of irregularity. We also have a totally irregular access to array d and we show in later sections how this example is handled by our scheme.

3.1 Supported Array Distributions

The current implementation supports block, cyclic, block-cyclic and an indirect distribution. The indirect distribution is used for specifying an irregular distribution where an integer mapping array is used to specify the target processor of each individual element of the array dimension being distributed. The i_{th} element of the mapping array specifies the processor number owning the i_{th} element of the data array. The mapping array is typically distributed in a blocked fashion among processors. All these distributions are supported in any number of dimensions.

```
        real a(8), x(16), d(16)
        integer map(16), ind(8), e(16)
        parameter (map=/1,2,2,3,1,4,3,1,4,4,4,3,3,2,1,2/)
!HPF$ processors p(4)
!HPF$ align a(i) with ind(i)
!HPF$ distribute ind(BLOCK) onto p
!HPF$ align x(i) with e(i)
!HPF$ distribute e(CYCLIC) onto p
!HPF$ distribute map(BLOCK) onto p
!HPF$ distribute d(INDIRECT(map)) onto p
        do k = 1, 100
!HPF$ independent
          do i = 1, 7
            do j = 1, ind(i+1)-ind(i)
              a(i) = x(i) + x(ind(i)+j-1) + d(e(ind(i)+j-1))
            enddo
          enddo
        enddo
```

Figure 10: A Mixed Regular-Irregular Program Fragment

3.2 Supported Internal Representations

PILAR is a C++ library designed to support different types of applications which have purely regular accesses, purely irregular accesses and mixed regular and irregular accesses. *Communication-pattern-internal-representation* (CPIR) is an abstract base class that unifies the different communication representations. Object-oriented techniques are used to encapsulate this internal representation from the rest of the library, thus simplifying the implementation of new representations and the interaction between objects with different internal representations. One of these representations is the interval-based representation, referred to as *CPIRInterval*, which is specially tuned for regular applications. This essentially consists of a collection of intervals (start and stop values for a regular section of an array) acted upon by basic operations like union, intersection, difference, etc. The second representation is *CPIREnumerated*, targeted towards communication patterns with almost no regularity and is very similar to what *CHAOS/PARTI* provides. Each memory access is represented by an entry in an index array. Operations like union and intersection are typically implemented by pre-sorting the arrays. The third representation, *CPIRCyclic* is targeted towards strided access when the stride is a small constant. This case is common in Fortran where a single array is sometimes used for storing data records, with multiple entries per record, the size of the record becoming the stride. This representation can in turn be represented in a compressed or expanded form. The functionality provided by a CPIR object includes functions to pack or unpack itself for network transmission, combine with other CPIR using basic operations, map from one array to another using the pattern encoded and a predefined operation, transform to a different internal representation and so on. These operations are, however, implemented in different ways depending on the actual *CPIR* type.

3.3 Translation Tables

A translation table encodes information about the home processor and the local address in the home processor for array elements. For block and cyclic(k) distributions, *no translation tables* are required and all translation information are encoded by simple analytical functions. In the case of an indirect distribution, each entry of the translation table stores the element index, the processor to which the element is allocated and its local offset in that processor. Figure 11 shows the values of the index arrays and the translation information for the arrays in the example program shown in Figure 10. It may be noted that the translation information shown for the arrays *a, ind, x, e* and *map* are encoded by analytical functions and are shown in a table form in Figure 11 only for clarity purposes. Typically, the translation table for array *d* would be distributed across processors. However, without loss of generality, for simplicity and for the purposes of this example, we maintain a replicated translation table for array *d*.

a

Indices	Proc
1:2	1
3:4	2
5:6	3
7:8	4

ind

Indices	Proc
1:2	1
3:4	2
5:6	3
7:8	4

x

Indices	Proc
1:1	1
2:2	2
3:3	3
4:4	4
5:5	1
6:6	2
7:7	3
8:8	4
9:9	1
10:10	2
11:11	3
12:12	4
13:13	1
14:14	2
15:15	3
16:16	4

e

Indices	Proc
1:1	1
2:2	2
3:3	3
4:4	4
5:5	1
6:6	2
7:7	3
8:8	4
9:9	1
10:10	2
11:11	3
12:12	4
13:13	1
14:14	2
15:15	3
16:16	4

map

Indices	Proc
1:4	1
5:8	2
9:12	3
13:16	4

d

Proc
1
2
2
3
1
4
3
1
4
4
4
3
3
2
1
2

ind

Values
1
3
4
7
7
8
12
14

e

Values
4
2
5
1
3
9
14
11
15
16
13
5
8
14
2
1

Figure 11: Translation Information and Values of Index Arrays: Note that the implementation uses analytical expressions and not tables for arrays a, ind, x, e and map

Several important considerations are linked to the design of the translation table. All information associated with the translation table are stored as sets and not as lists. This not only reduces the memory requirement substantially but also improves run-time performance. Table lookup is another primary consideration. Since only the bounds of array segments are stored, the size of the translation table is small in most cases and we can often afford to have it replicated in each processor's local memory.

3.4 Trace collection

Trace collection refers to the collection of the array references in the global address domain. For regular accesses, the interval representation is used to collect the start and end values of a contiguously accessed set of elements and these are stored in a trace array. For *totally irregular accesses*, the enumerated representation is used where every element has to be stored individually in the trace array owing to lack of contiguity. However, many irregular applications contain spatial regularity where the first element accessed is something random but the next few elements are regularly spaced among themselves. This corresponds to the pattern $(data(ind(i) + j))$ in code segments where *data* and *ind* are the data array and index array respectively. Though this pattern may not be explicitly present in real-life applications, it manifests itself once *index normalization* is performed. Let us consider the code segment extracted from *SPARSKIT* [24] and shown in Figure 12(a).

```
do ii=1, nrow
    do ka=ia(ii), ia(ii+1)-1
        jj = ja(ka)
        do kb=ib(jj),ib(jj+1)-1
            jcol = jb(kb)
            ...
        enddo
    enddo
enddo
```

(a) Original Code

```
do ii=1, nrow
    do ka=1, ia(ii+1)-ia(ii)
        jj = ja(ia(ii)+ka-1)
        do kb=1, ib(jj+1)-ib(jj)
            jcol = jb(ib(jj)+kb-1)
            ...
        enddo
    enddo
enddo
```

(b) Index-normalized Code

Figure 12: Explicit Spatial Regularity in Index-normalized code

As illustrated in Figure 12(b), spatial regularity is present even in the case of unstructured code. This feature is exploited by the *PILAR* run-time library by using intervals for trace collection in such cases. For instance, the trace intervals for the array reference *jb* in Figure 12 will be bounded by *ib(jj+1)* and *ib(jj)* for appropriate values of *jj*. Figure 13 shows the global references (GlobRef) made by the processors. x_1 and x_2 denote respectively the regular and irregular reference on the r.h.s. of the statement shown in Figure 10. ind_1 and ind_2 denote the references $ind(i+1)$ and $ind(i)$ respectively.

a		ind_1		ind_2		x_1		x_2, e		d	
GlobRef	Proc	GlobRef	Proc	GlobRef	Proc	GlobRef	Proc	GlobRef	Proc	GlobRef	Proc
1:2	1	2:3	1	1:2	1	1:2	1	(1:2),(3:3)	1	(4,2,5)	1
3:4	2	4:5	2	3:4	2	3:4	2	4:6	2	(1,3,9)	2
5:6	3	6:7	3	5:6	3	5:6	3	(7:7),(8:11)	3	(14,11,15,16,13)	3
7:7	4	8:8	4	7:7	4	7:7	4	(12:13)	4	(5,8)	4

Figure 13: Global References made by processors

Several important considerations are associated with the type of trace collection. Collecting traces in the form of intervals reduces the memory requirement for storing global accesses and improves performance owing to less number of assignments. This also has important implications for other phases of the program in the sense that these result in fewer local accesses stored, more regularity in schedules (and hence better communication optimization) and even better executor performance. This also allows for better sharing of information since the structures used for regular and irregular accesses become compatible. It has been shown in [21] that preprocessing costs (which subsume trace generation costs) are very low compared to the total execution time for completely regular accesses. In our opinion, though no trace generation cost is incurred by a compile-time scheme, the advantages derived in inter-processor communication in a uniform scheme are likely to outweigh the overheads associated with run-time trace generation when appropriate representations are used for different accesses. This has been verified by the study reported in [21] and the results shown in this paper.

3.5 Localization of Array References and Schedule Generation

Primitives are provided to automatically compute local addresses corresponding to given global addresses. Buffer space for references owned by a processor is allocated contiguously in the local memory of the processor irrespective of whether they are contiguous or not in the global address domain. Buffer space for external references is allocated contiguously at the end of the local buffer. All global references in the trace array are localized using the processor and offset information stored in the translation table. The local addresses are stored in an interval format corresponding to the global addresses, if possible.

a		ind_1		ind_2		x_1	
LocalRef	Proc	LocalRef	Proc	LocalRef	Proc	LocalRef	Proc
1:2	1	2:3	1	1:2	1	(1:1),(5:5)	1
1:2	2	2:3	2	1:2	2	(5:6)	2
1:2	3	2:3	3	1:2	3	(5:6)	3
1:1	4	2:2	4	1:1	4	(5:5)	4

x_2		e		d	
LocalRef	Proc	LocalRef	Proc	LocalRef	Proc
(1:1),(5:6)	1	(1:1),(5:6)	1	(5:6),(2:2)	1
(6:7),(2:2)	2	(5:6),(2:2)	2	(5:5),(2:2) (6:6)	2
(2:2),(7:9) (3:3)	3	(2:2),(5:7) (3:3)	3	(5:8),(4:4)	3
(3:3),(6:6)	4	(3:3),(5:5)	4	(5:6)	4

Figure 14: Local References generated from Global References and Translation Information

In the current implementation, the compiler uses a simplified version of the algorithm proposed in [35] to determine the placement of communication. The basic idea of this algorithm is that the communication level of an array reference is the maximum of the nesting levels of all incoming flow dependences to that array reference. The nesting level of a dependence is defined as the innermost level that carries the dependence, or if the dependence is loop-independent, the common nesting level of the statements involved. Once the communication points for a certain array are determined, the compiler generates run-time library calls to analyze the relevant schedules and coalesce messages, if possible. For any communication point C_i, all relevant schedules corresponding to references to the same array can potentially be combined irrespective of the type of access. When schedules are combined, the corresponding local references need to be modified simultaneously so that the correct element is accessed by the executor. At an implementation level, efficient set operations are used to manipulate intervals. Conversion from interval representation to enumerated representation and vice-versa is effected whenever required. We now show how local references and schedules are generated when multiple references to the same array exist. Let us consider the references X_1, X_2, ..., X_k to an array X. We denote the global references made by reference X_i as $GlobRef(X_i)$. The following steps are carried out:

- First the union of all the global references made for array X is computed as $GlobRef(X_1X_2\ldots X_k) = \cup_{i=1}^{k} GlobRef(X_i)$
- The corresponding local references are computed as $LocalRef(X_1X_2\ldots X_k) = (Localize(GlobalRef(X_1X_2\ldots X_k)))$
- $I_{X_i} = SearchIndices(GlobRef(X_1X_2\ldots X_k), GlobRef(X_i))$ I_{X_i} gives the indices of $GlobRef(X_1X_2\ldots X_k)$ if the corresponding values of $GlobRef(X_1X_2\ldots X_k)$ are present in $GlobRef(X_i)$.
- Values of $LocalRef(X_1X_2\ldots X_3)$ corresponding to indices I_{X_i} are copied to $LocalRef(X_i)$.

Figure 14 shows the local references for the arrays in the case of different processors for the example code shown in Figure 10.

A schedule stores a communication pattern that encodes information about the data to be sent and received by a processor. The following steps are carried out for the generation of schedules.

- A processor computes the indices of the off-processor elements it must receive and the processors it must receive from. This is determined from the global traces already collected for the reference and the translation information. $GlobRecvData(X)$ refers to the indices of the off-processor data in the global address domain associated with the references $X_1, \ldots, X_k$ while $Own(X)$ refers to the indices of X owned by the processor. Thus
  $GlobRecvData(X) = GlobRef(X_1 X_2 \ldots X_k) - Own(X)$
- The addresses of the remote processors owning $GlobRecvData(X)$ are obtained by a dereference request which uses the translation information to compute the owning processor address. Let the indices (in the global domain) of the data to be received from processor p be denoted by $GlobRecvDataProc(X, p)$.
- The addresses of the local buffers where off-processor elements should be received are determined in conjunction with localization of off-processor global references and are given by
  $Localize(GlobRecvDataProc(X, p))$.
- Once a processor knows the addresses of the processors it needs to receive data from and the local buffer addresses where the data have to be received, the global indices ($GlobRecvDataProc(X, p)$) are exchanged among processors. This enables all processors to know the addresses of the processors it needs to send data to and the local addresses of such data. The exact scheme for doing this may differ with implementation; our implementation uses incremental schedules, derived datatypes, message aggregation etc. before actual communication.

Figure 15 shows the schedule (in brief) generated for the example program shown in Figure 10. In Figure 15, $ArrayName \leftarrow$ indicates that the corresponding processor has to receive data for array $ArrayName$ in its local buffer listed under LRef (LocalRef) from the processor listed under RProc (Remote Processor). Similarly, $ArrayName \rightarrow$ indicates that the corresponding processor has to send data for array $ArrayName$ from its local buffer listed under LRef to the processor listed under RProc.

Processor 1

Array	LRef	RProc
a	Null	Null
$ind\leftarrow$	3	2
$\leftarrow$	5	2
$\leftarrow$	6	3
$x\rightarrow$	2	2
$\rightarrow$	2	3
$\rightarrow$	3	3
$\rightarrow$	4	4
$\leftarrow$	5	2
$\leftarrow$	6	3
$e\rightarrow$	2	2
$\rightarrow$	3	3
$\rightarrow$	4	4
$\leftarrow$	5	3
$\leftarrow$	6	2
$d\rightarrow$	1	2
$\rightarrow$	4	3
$\rightarrow$	2:3	4

Processor 2

Array	LRef	RProc
a	Null	Null
$ind\leftarrow$	3	3
$\rightarrow$	1	1
$\leftarrow$	5	3
$\leftarrow$	6	4
$\leftarrow$	7	1
$x\rightarrow$	1	1
$\rightarrow$	2	3
$\rightarrow$	3	3
$\leftarrow$	5	4
$\leftarrow$	6	1
$e\rightarrow$	1	1
$\rightarrow$	3	3
$\leftarrow$	5	1
$\leftarrow$	6	4
$d\rightarrow$	1	1
$\rightarrow$	3	3
$\rightarrow$	4	3

Processor 3

Array	LRef	RProc
a	Null	Null
$ind\leftarrow$	3	4
$\rightarrow$	1	2
$\leftarrow$	5	1
$\leftarrow$	6	2
$\leftarrow$	7	4
$x\leftarrow$	8	1
$\leftarrow$	9	2
$\rightarrow$	1	1
$\rightarrow$	1	2
$\rightarrow$	2	4
$\leftarrow$	5	4
$e\leftarrow$	6	1
$\leftarrow$	7	2
$\rightarrow$	1	1
$\leftarrow$	5	2
$\leftarrow$	6	4
$d\leftarrow$	7	1
$\leftarrow$	8	2
$\rightarrow$	1	1

Processor 4

Array	LRef	RProc
a	Null	Null
$ind\rightarrow$	1	3
$\leftarrow$	5	3
$x\leftarrow$	6	1
$\rightarrow$	1	2
$\rightarrow$	2	3
$\leftarrow$	5	1
$e\rightarrow$	1	2
$\rightarrow$	2	3
$\leftarrow$	(5:6)	1
$d\rightarrow$	2	2
$\rightarrow$	4	3

Figure 15: Schedules for the arrays in the case of example code fragment

3.6 Schedule Optimizations

Various communication optimization support present in *CHAOS* are described in [11]. All of these optimizations are supported by *PILAR* too. Efficient set operations are used to compute the various information required for schedules whenever the global references are presented as intervals. All optimizations are possible irrespective of the regularity of accesses since both the interval and enumerated representations are employed by the same paradigm and conversion from one to the other is effected depending on the granularity of regularity. In order to perform optimizations on schedules spanning different references, efficient set operations have been defined. The operation *Merge* merges schedules and removes duplicates while *Concat* merges schedules without removing duplicates. The operation *Intersection* finds duplicates among schedules. The *Difference* operation is used to obtain an incremental schedule. Operations are provided to expand a schedule, replicate a pattern in a schedule with or without an offset, invert a pattern in a schedule etc.

Compile-time schemes incur similar overhead while generating send/receive and in/out sets as that involved in schedule generation in an inspector-executor scheme. However, we feel that for irregular applications, our scheme would incur a lower communication cost since the number of communication startups and the total communication volume are likely to be less.

3.6.1 No Redundant Communication

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. This is ensured since all references are initially collected as traces and localized. Though this may mean communication of elements non-contiguous in memory, we use derived data types whenever support is available (for instance, when *PILAR* is running over MPI). This is likely to amortize the communication cost substantially. Exact communication also leads to reduced allocation of memory for off-processor elements. Duplicate global references are always removed to ensure that no element is communicated more than once as long as its value remains the same. Since we use sets for representation of all information, this feature is always ensured.

3.6.2 Incremental Scheduling

In *PILAR*, the instantiation of a schedule is decoupled from its creation. Decoupling saves the overhead of instantiation when the same schedule is used with different elementary types or when simple schedules are only used for building complex ones. Every schedule is associated with a particular distribution and a particular access function. This allows use of the same schedule for different arrays having potentially different data types. Derived data types are created for every pair of processors that need to communicate among themselves. This allows message aggregation and message coalescing. Schedules in *PILAR* are different from those in *CHAOS/PARTI* in that they are fully symmetric. Typical gather or scatter operations implemented in the latter assume that one end of the communication is contiguous in memory. In contrast, the communication framework in *PILAR* implements a more general scheme in which both ends can be non-contiguous. This has simplified the implementation of the library since only one data exchange primitive needs to be implemented.

3.6.3 Reuse of Schedules Across Regular and Irregular Accesses

We are not aware of a scheme which reuses schedules efficiently across regular and irregular accesses in a mixed regular-irregular application. In our opinion, efficient reuse of schedules in a mixed regular and irregular application is possible if support is available for efficient representation of both regular and irregular accesses and support is available for derived data types. This is possible in our scheme since compatible representations are used throughout the application. It may be observed that in many applications (like the Lanczos algorithm [32]), the same array is accessed in both regular and irregular manner, either inside the same loop or inside different loops. In the case of such applications, schedules generated for regular accesses can be reused for irregular accesses. A code fragment containing such a scenario is present in Figure 10. Array x is accessed both in a regular and irregular manner. The two schedules for array x can be combined and the non-local data for array x needs to be communicated only once. The fact that there is likely to be a substantial amount of spatial regularity in the irregular accesses made to array x contributes largely to the efficiency of the scheme.

3.6.4 Global communication primitives

PILAR supports both blocking and non-blocking global communication primitives [23]. Before any data exchange can take place, a *communication buffer* (CB) is created. CB is a template class that performs a type instantiation of the schedule and, if required, allocates internal buffers. Each CB also has associated with it a scope of communication. CBs simplify the implementation of non-blocking exchange primitives because internal buffers can be reused in a safe manner as long as there is only one outstanding communication per CB. In *PILAR*, the call to data exchange is split up into two calls — the first implements the first step of a conventional communication call, but instead of blocking for incoming messages, it immediately returns a data structure with pending message identifiers and their buffers. The second blocks until all messages arrive. CBs store the state of the pending communication, blocking the process until all messages arrive. This way of implementation usually results in better performance than blocking communication since it allows overlap of computation and communication.

3.7 Executor

The executor tries to exploit the features of interval representation and is different from traditional executors found in many run-time libraries. The actual computation loop tries to preserve regularity in the subscript structure (of the form $data(ind(i)+j)$) as illustrated in Figure 16. The call *MergeLocalInterval* makes the intervals (for the arrays mentioned) compatible (the same number of local intervals of the same size). A reasonable sequential compiler will pull out of the innermost loop the invariant portion of the access to the array leaving a regular access in the innermost loop. On the contrary, loops not possessing this structure may incur the cost of twice as many loads assuming that the innermost loop has a reasonable average trip count. Moreover adding accesses to a large index array in the innermost loop may increase cache conflicts further degrading single-node performance.

Such an executor changes the loop structure since an extra loop has to be inserted. When we transform a contiguous sequence of accesses from global to local addresses, a sub-sequence of accesses might not be local to the processor and this breaks the original interval into sub-intervals. The extra loop is needed to enumerate sub-sequences for each global sequence. Assuming that the data distribution across processors does not fragment too much the address space, the added loop will have on the average a few iterations and after traditional sequential compiler optimizations, the single-node performance of this code would

```
Call MergeLocalInterval(a,x1)
Call MergeLocalInterval(x2,d)
do iter = 1, 100
   count_11 = 0
   count_12 = 0
   do i = 1, NIntervals_1
      do j = 0, refLocOff_a(count_11+1) - refLocOff_a(count_11)
         do k = 1, NIntervals_2
            do l = 0, refLocOff_x2(count_12+1) - refLocOff_x2(count_12)
               a(j+refLocOff_a(count_11)) = x(j+refLocOff_x1(count_11)) +
                                            x(l+refLocOff_x2(count_12)) +
                                            d(l+refLocOff_d(count_12))
            enddo
            count_12 = count_12 + 2
         enddo
      enddo
      count_11 = count_11 + 2
   enddo
enddo
```

Figure 16: Executor for the Example Program

be similar to the sequential code. However, since both regular and irregular accesses may be present in the same statement, intervals can get broken down to produce as many subsequences as the actual number of array accesses in the worst case. However, given that irregular accesses tend to have spatial locality (as illustrated earlier), this is likely to be rare. Even if the worst case happens, the computation performance would be equivalent to that in other run-time libraries without an interval representation; however, other phases of the program like trace collection, scheduling and communication would still be more optimized since intervals in these phases need not be broken down into individual elements.

3.8 Interval-based Pilar and Enumerated-based Pilar

In our experiments detailed in later sections, we have used the *PILAR* library with different representations, where the enumerated-approach is effectively equivalent to the *CHAOS/PARTI* approach.[1] Figure 17 highlights the difference between the interval-based and enumerated-based approaches in handling of array accesses.

Though in both cases, the global array references are picked up and stored in a data structure, the interval-based representation fares better since it stores only the *start* and *stop* values of regular sections. Translation of global references to local ones, removal of duplicates to optimize the communication schedule and computation of permutation of the local array elements need to be done in both the approaches. Although both libraries are performing similar operations at each step, all these operations are performed in PILAR using an interval-based representation. This compact representation can dramatically reduce the memory overhead and the time required for these operations.

[1] In a separate study [13, 19], we have demonstrated that the run-time performance of enumerated-based *PILAR* is actually better than that of *CHAOS/PARTI* library primarily due to different implementation styles. Hence we believe that this is a fair comparison.

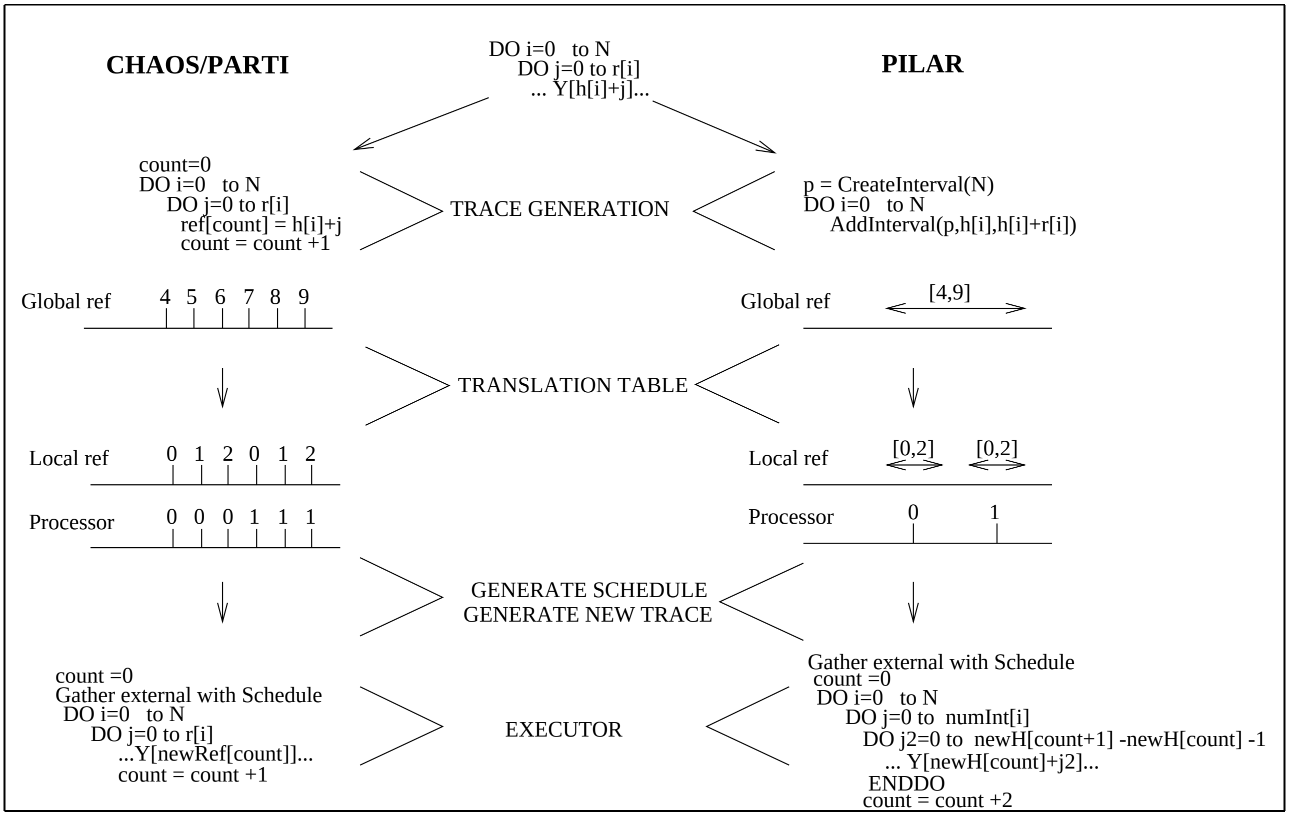


Figure 17: Differences between CHAOS/PARTI and PILAR with interval representation

4 Experimental Results

4.1 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)*. A number of benchmark kernels have been used in comparing the various schemes, code generated employing calls to the *PILAR* library (both Pilar (Int) and Pilar (Enu)), code generated by the IBM HPF compiler (IBM-hpf) and code generated by the PGI HPF compiler (PGI-hpf). The benchmarks are a synthetic benchmark shown in Figure 10, subroutines to compute the infinity norm of a matrix (extracted from ITPACK [31]), a subroutine implementing the Lanczos algorithm [32] and a subroutine implementing the Conjugate Gradient Algorithm [34, 33]. All these benchmarks contain a mixture of regular and irregular accesses and are representative of real-life irregular applications.

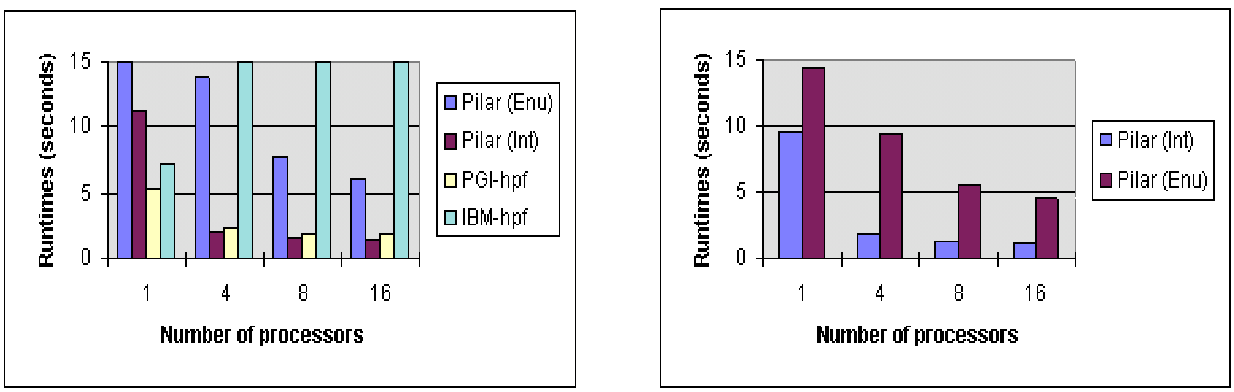
The platform chosen was a 16-processor *IBM SP-2* running *AIX 4.1*. The *SP-2* is a distributed-memory parallel machine and the installation uses sixteen 120 MHz *thin nodes* each with 128 MB of main memory. For all results reported in later sections, the *high performance communication switch* has been used for inter-processor communication.

PGI's HPF compiler (version 2.2) for PGI-hpf and *IBM's Standard XL High Performance Fortran* *compiler (version 1.1.0.0)* for IBM-hpf have been used. We have selected *MPI* [30] 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.

Though we present run-times for the various schemes, it may be noted that the run-times alone may not be enough to judge the efficiency of the schemes. It needs to be emphasized that while the implementation framework for *Pilar (Int)* and *Pilar (Enu)* are the same, those for *PILAR, PGI-hpf* and *IBM-hpf* are likely to be vastly different. More precisely, efficient implementation of low-level routines, intelligent transformations etc. can produce substantial difference in the run-times. This is why we also present the communication volume which in our opinion is one of the important yardsticks for comparison.

4.2 Synthetic Benchmark

The synthetic benchmark shown in Figure 10 has both regular and irregular accesses. Arrays a and *ind* are always accessed in a regular manner, arrays d and e are always accessed in an irregular manner while array x has both regular and irregular accesses made to it. We have observed comparative run-times for this benchmark in different scenarios.



(a) Regular Distributions

(b) Mixed Regular-Irregular Distributions

Figure 18: Comparison of Runtimes for the Synthetic Benchmark

In the first scenario, we have observed comparative run-times for *Pilar (Int)*, *Pilar (Enu)*, *PGI-hpf* and *IBM-hpf*. All arrays except d have the same distributions as that shown in Figure 10. The distribution of array d has been changed to a regular one (BLOCK, in this case) since commercial HPF compilers currently handle only regular distributions. Figure 18(a) shows the comparative run-times for the various schemes. The array size chosen is 1024 and the outer loop iterates 1000 times. Though *Pilar (Int)* takes more time for a single processor, it performs the best as the number of processors is increased. A timeout of 15 seconds was set; *Pilar (Enu)* in the case of a single processor and *IBM-hpf* for 4, 8 and 16 processors failed to complete within this time period.

Figure 19(a & b) show the amount of inter-processor communication for *Pilar (Int)* and *PGI-hpf*. It should be noted that the communication volume shown includes that required for both data and protocol. This implies that some of the communication volume shown for *Pilar (Int)* is for generating schedules. It may be noted that the communication required for *Pilar (Enu)* will be approximately equal to that for *Pilar (Int)* with a little difference arising out of difference in schedule representations. It seems that *PGI-hpf* is doing some redundant communication. However, since all communication for this example program can be hoisted out of the outermost loop, the communication cost is negligible and does not affect the total run-times substantially. However, in most real-life applications, the effect of inter-processor communication will be an important factor. This is indicated by the results reported for other benchmarks. It is also interesting to note that the difference in bytes communicated between our implementation and the *PGI-hpf* scheme doesn't always get reflected in the runtimes; in our opinion, highly tuned PGI libraries is the reason. Figure 19(c & d) show the component costs for *Pilar (Int)* and *Pilar (Enu)* respectively for the distributions specified in Figure 10. As expected, the component costs for *Pilar (Int)* are less than those for *Pilar (Enu)*.

(a) Pilar (Int)	(b) PGI-hpf	(c) Pilar (Int)		(d) Pilar (Enu)	
0→1: 1380 bytes	**0→1**: 3348 bytes	Pilar (Int)	4 proc	Pilar (Enu)	4 proc
0→2: 1404 bytes	**0→2**: 3348 bytes	Total time	1.87	Total time	9.31
0→3: 1404 bytes	**0→3**: 3348 bytes	Inspection time	1.71	Inspection time	6.32
1→0: 1244 bytes	**1→0**: 7720 bytes	Scheduling time	1.36	Scheduling time	4.34
1→2: 1036 bytes	**1→2**: 3328 bytes	Communication time	0.04	Communication time	1.84
1→3: 820 bytes	**1→3**: 3328 bytes	Computation time	0.12	Computation time	1.12
2→0: 1288 bytes	**2→0**: 3720 bytes				
2→1: 424 bytes	**2→1**: 7328 bytes				
2→3: 536 bytes	**2→3**: 3328 bytes				
3→0: 1260 bytes	**3→0**: 3720 bytes				
3→1: 249 bytes	**3→1**: 3328 bytes				
3→2: 1204 bytes	**3→2**: 7328 bytes				

Figure 19: Communication volume and Library Costs for the Example Program

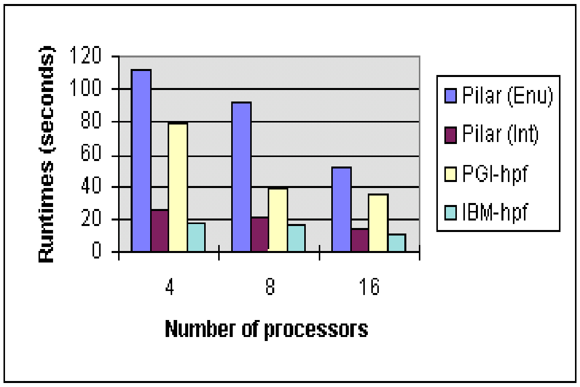
In the second scenario, we have observed comparative run-times for *Pilar (Int)* and *Pilar (Enu)* for the distributions specified in Figure 10. It may be noted that this scenario is likely to be more realistic than the previous one. As Figure 18(b) shows, the run-times in this scenario improve for both *Pilar (Int)* and *Pilar (Enu)* compared to the previous one. This is because of reduction in inter-processor communication.

4.3 ITPACK Subroutines

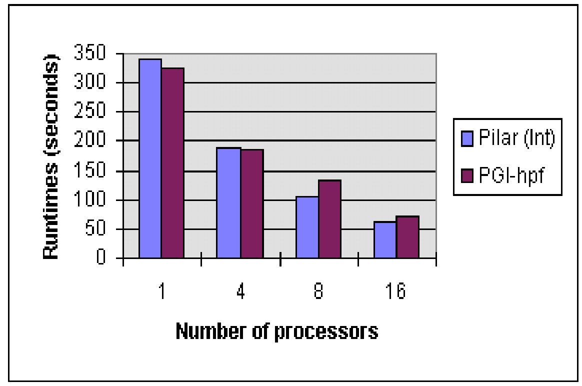
ITPACK [31] contains subroutines for solving large sparse linear systems by iterative methods. Two kernels were extracted from ITPACK subroutines. The first of them calculates the infinity norm of a matrix using the *ellpack* data structure whereas the second calculates the infinity norm using the *symmetric diagonal data structure*. Figure 20(a) gives comparative run-times for different schemes in the case of ellpack data structure. We have used the following array sizes: 50000x1024 for the matrix representation array (of type real) and 50000 for the workspace array (of type real). The benchmark is too big to run on a single processor and this is why only run-times for 4, 8 and 16 processors are presented. This benchmark does not contain any irregular accesses.

Figure 20(b) gives comparative run-times for different schemes in the case of symmetric diagonal data structure. Each dimension size was 1024 in this benchmark. This benchmark contains a number of irregular accesses. A timeout of 400 seconds was set; IBM-hpf and Pilar (Enu) failed to finish within this period and their runtimes are not shown in Figure 20(b).

We don't show the number of bytes communicated for the ITPACK kernels since the communication



(a) Runtimes for Itpack kernel 1



(b) Runtimes for Itpack kernel 2

Figure 20: Comparison of Runtimes for Itpack Subroutines

Pilar (Int)	1 proc	4 proc	8 proc	16 proc	Pilar (Enu)	1 proc	4 proc	8 proc	16 proc
Total	341	192	105	61	Total	1197	651	607	435
Inspection	2.56	0.51	0.46	0.39	Inspection	32.19	16.92	11.43	9.01
Schedule gen.	0.15	0.45	0.41	0.42	Schedule gen.	20.91	12.34	8.75	6.12
Executor	338	191	104	60	Executor	1164	634	595	426

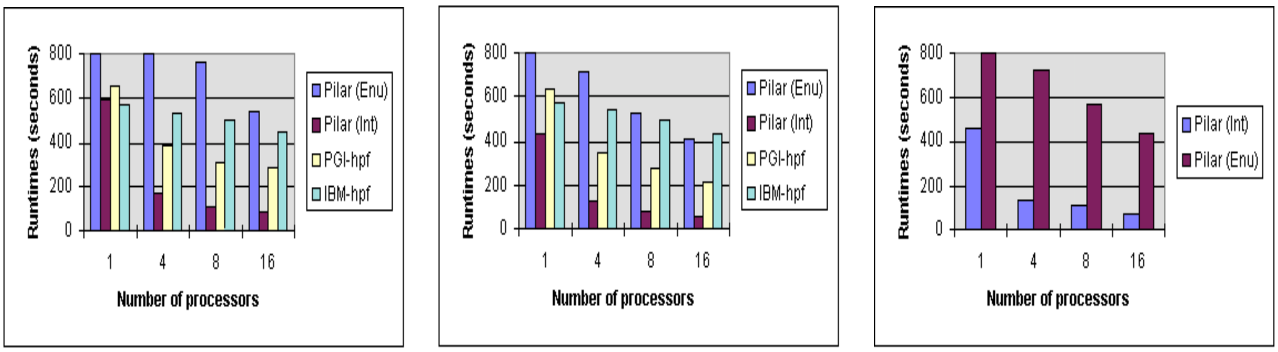
Table 1: Profiling of Itpack kernel 2 using Pilar

requirement is not very high in this case and both our scheme and the PGI HPF compiler generate similar amounts of communication. We have taken an execution profile of the code (kernel 2) parallelized with interval-based PILAR in order to understand the component costs involved. The inspection cost subsumes the schedule generation cost. As the results in Table 1 show, the inspection time forms a small portion of the total time in the case of Pilar (Int). This is owing to the use of the interval representation. While this makes trace collection much faster, this also facilitates efficient set operations in a wide range of preprocessing computations like building of schedules. This program accesses the *workspace* array in both regular and irregular manner. This allows a wide range of optimizations to be performed as already described in Section 3. The *Executor* cost includes time required for communication as well as computation. As Table 1 shows, the executor time dominates the preprocessing time. It may be noted that the executor time is likely to be the same for PGI-hpf and Pilar (Int) since communication and actual computation take place at run-time irrespective of the paradigms used. As expected, all the component costs in Pilar (Enu) are much higher than those in Pilar (Int).

4.4 Lanczos Algorithm

We have extracted a kernel from the Lanczos algorithm for determining eigen systems from codes available in [32]. All sparse matrices are represented in compressed sparse row (CSR) [34] format. The kernel used here is similar to the one outlined in [28]. We have observed comparative performance for a number of scenarios. The benchmark used by us consists of a two-dimensional dense matrix and a number of one-dimensional vectors. Every dimension is of size 2048. The dense array is accessed both in a regular and irregular manner. Some of the one-dimensional vectors are accessed in an irregular manner.

Figure 21(a) gives comparative run-times when regular distributions are used for all the arrays. However, the index arrays have been initialized in a way so that there is very little amount of spatial regularity. As shown in Figure 21(b), the run-times improve once the index arrays take into account the program semantics and there is spatial regularity. Figure 21(c) presents run-times for irregular distributions. In all the cases, *Pilar (Int)* fares better than other schemes.



(a) Regular Distributions (b) Simulated Irregular Distributions (c) Irregular Distrib.

Figure 21: Comparison of Runtimes for the Lanczos Algorithm with Different Data Distributions

Figure 22 shows the communication volume for *Pilar (Int)* and *PGI-hpf* in different scenarios. This was obtained with the size of every dimension as 128 (since otherwise the trace arrays become too big) using the Visualization Tool (vt) on the IBM SP-2. Figure 22(a & b) correspond to cases when the values of the index arrays are initialized in a way so that there is very little amount of spatial regularity. We observed the inter-processor communication volume for this case in order to roughly estimate the potential worst-case communication in the case of *Pilar (Int)*. As seen from Figure 22(a & b), the communication in the case of *Pilar (Int)* is around hundreds of kilobytes less for every pair of processors in a total of 4 processors. We wanted to simulate an irregular distribution which tries to partition data according to the semantics of the program (thus reducing interprocessor communication). We have chosen index array values so that there is some amount of spatial regularity. It may be noted that currently available commercial compilers don't support irregular distributions directly. The total runtimes for this scenario are given in Figure 21(b). Figure 22(c & d) show the inter-processor communication for the second scenario where intuitively more of the accesses should be local and consequently reduced inter-processor communication. In this case, the inter-processor communication in the case of *Pilar (Int)* is around 1 megabyte less than that of *PGI-hpf* for every pair of processors in a total of 4 processors.

4.5 Conjugate Gradient Algorithm

We have used the core computation loops from the Conjugate Gradient Algorithm [34, 33] for comparing the different schemes. The code consists of sparse matrices represented in CSR format. The code used

0→1: 2319684 bytes	**0→1**: 2621716 bytes	**0→1**: 1064564 bytes	**0→1**: 2601236 bytes
0→2: 2519636 bytes	**0→2**: 2621716 bytes	**0→2**: 1064564 bytes	**0→2**: 2601236 bytes
0→3: 2103252 bytes	**0→3**: 2605332 bytes	**0→3**: 1030724 bytes	**0→3**: 2584980 bytes
1→0: 2524760 bytes	**1→0**: 2624136 bytes	**1→0**: 1033816 bytes	**1→0**: 2603640 bytes
1→2: 1923780 bytes	**1→2**: 2621696 bytes	**1→2**: 1032820 bytes	**1→2**: 2601216 bytes
1→3: 2507348 bytes	**1→3**: 2605312 bytes	**1→3**: 1000004 bytes	**1→3**: 2584960 bytes
2→0: 1523732 bytes	**2→0**: 2622088 bytes	**2→0**: 1032788 bytes	**2→0**: 2601608 bytes
2→1: 2524760 bytes	**2→1**: 2623744 bytes	**2→1**: 1033880 bytes	**2→1**: 2603248 bytes
2→3: 2507396 bytes	**2→3**: 2605312 bytes	**2→3**: 1000052 bytes	**2→3**: 2584960 bytes
3→0: 2418036 bytes	**3→0**: 2622088 bytes	**3→0**: 1032788 bytes	**3→0**: 2601608 bytes
3→1: 2317396 bytes	**3→1**: 2621696 bytes	**3→1**: 1032772 bytes	**3→1**: 2601216 bytes
3→2: 1800664 bytes	**3→2**: 2623744 bytes	**3→2**: 1033800 bytes	**3→2**: 2603248 bytes
(a) Our scheme	(b) PGI-hpf	(c) Our scheme	(d) PGI-hpf

Figure 22: Communication volume for Lanczos Algorithm for Different Index Values

here consists of a number of single-dimensional arrays with every dimension of size 2048. All index arrays are initialized with random values.

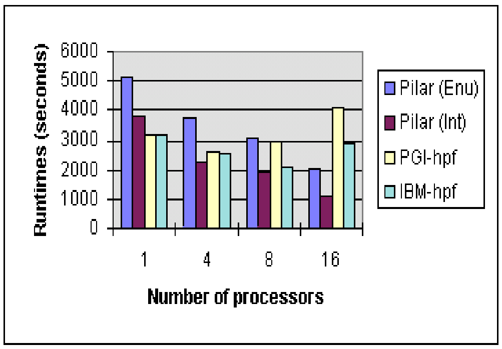


Figure 23: Comparison of Runtimes for the Conjugate Gradient Algorithm

Figure 23 shows the comparative run-times when all the arrays are distributed in a blocked fashion. Our scheme (Pilar (Int)) outperforms all the other schemes.

4.6 Summary of Results

- Experimental results presented for some mixed regular-irregular applications show that *Pilar (Int)* performs better than other schemes. This also means that a uniform run-time scheme performs well for mixed regular-irregular applications.

- Experimental results corroborate the ideas presented in previous sections that show that inter-processor communication can be greatly reduced if a uniform scheme is used to handle both regular and irregular accesses in a mixed regular-irregular application.

- The preprocessing costs and executor costs along with the total costs are very high for *Pilar (Enu)* (as shown in this study and in [21]). This shows the need for a representation similar to the interval representation that takes into account the regularity of accesses and distributions. It appears to us that a run-time library having multiple internal representations for handling different types of accesses is the most suitable one.

- Appropriate irregular distributions need to be used for arrays accessed in irregular ways. This is likely to reduce inter-processor communication. Many existing schemes have already shown this need [10, 28].

- The commercial compilers use run-time schemes but there are differences between their and our scheme. It appears to us that in the case of commercial compilers, the support for handling irregular accesses is implemented as a generic run-time library; so there is no way they can handle the sparsity which is specific to a given irregular access unless this information is passed on to the run-time library. This is however not done or probably cannot be done without complicating the design of the library. This results in overestimating a sparse region by an enclosing dense region which leads to redundant communication. In contrast to this, we try to handle a sparse region as a collection of intervals or elements by generating *appropriate inline code*. We can avoid overestimating communication by communicating only the required elements and not the region enclosing them. This further allows us better communication optimization since we are performing the set operations at a finer granularity.

5 Related Work

The *CHAOS/PARTI* [10] library has developed methods for parallelizing applications that are irregular. The library uses the *inspector-executor* paradigm. Since the development of this library was initially motivated by irregular applications, the representation used is an enumerated one.

Multiblock PARTI [25] provides support for block-structured applications with regular decompositions. The design of this library was motivated by multiblock and multigrid applications and can be used for supporting regular applications also. A typical multiblock application consists of an outer sequential loop and an inner parallel loop. The inner loop iterates over the blocks of the problem which also consists of applying boundary conditions to all the blocks. Each block constitutes a regular computation. Two types of communication are required in these applications : interaction between different blocks requiring the movement of regular array sections and interaction among the same block if it is partitioned across processors. The functionality and effectiveness of this library is thus very similar to that of *interval-based PILAR* if regular accesses and distributions are considered alone. However, in contrast to our scheme, this scheme may not be very suitable for mixed regular and irregular accesses. This library is a stand-alone library in its current implementation and has not been integrated with the *CHAOS/PARTI* library. The development of *PILAR* was mainly geared towards irregular applications while having efficient support for regular structures also. *PILAR* has efficient in-built operations encapsulated within a common interface to handle both regular and irregular accesses. Conversion between representations (regular and irregular) is possible and is done transparently by the library depending on the fineness of regularity or sizes of intervals.

Ujaldon and Zapata [26] have proposed an approach to reduce the number of levels of indirect array accesses. This method is based on enforcing alignment restrictions of common sparse data structures using compiler directives and taking the help of the compiler to reduce the preprocessing levels. This has been implemented as an extension to the *Vienna Fortran Compilation system* [27]. This method is, however, restricted to a specific set of applications only. In [28], they have proposed new methods (for instance, the *multiple recursive decomposition*) for the representation and distribution of sparse data on distributed memory parallel machines and have shown how compiler and run-time techniques can achieve storage economy and reduce communication overhead. Their representation exploits the locality of sparse computations, preserves a compact representation of matrices and vectors and tries to exploit the semantic information present in specific applications. They have also shown that this distribution is suitable for mixed regular-irregular accesses. However, they don't take into consideration the type of regularity exploited by our work. We feel that our methods can be integrated into their scheme.

LPARX [29] is a domain-specific run-time C++ library that provides support for dynamic irregular problems in a variety of platforms. However, its main target is not the fine grained blocks obtained from unstructured grids but the larger ones that come from finite difference methods. Thus it is different from libraries like *CHAOS/PARTI* and *PILAR*.

6 Conclusions

This study highlights how lack of a uniform representation and lack of a uniform scheme to generate communication structures and parallel code for regular and irregular accesses in a mixed regular-irregular application prevent sophisticated optimizations. Furthermore, we also show that code generated for regular accesses using compile-time schemes are *not always* compatible to code generated for irregular accesses using run-time schemes. This study presents a uniform scheme to handle both regular and irregular accesses in a mixed regular-irregular application. Using multiple internal representations, it has been shown how this scheme not only allows for sophisticated communication optimization like message coalescing, message aggregation to be made across regular and irregular accesses, the preprocessing costs incurred are also likely to be minimum. Experimental comparisons for various benchmarks on a 16-processor IBM SP-2 show that our scheme is feasible and better than existing schemes.

References

[1] Office of Science and Technology Policy, *Grand Challenges 1993 : High Performance Computing and Communications*, Washington, National Science Foundation, 1992.

[2] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr. and M. Zosel, *The High Performance Fortran Handbook*, Cambridge, MA, The MIT Press, 1994.

[3] S. Hiranandani, K. Kennedy and C. Tseng, *Compiling FortranD for MIMD distributed memory machines*, Communications of the ACM, vol. 35, No. 8, pp. 66-80, Aug. 1992.

[4] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka and My. Y. Wu, *Compiling Fortran90D/HPF for distributed memory MIMD computers*, Journal of Parallel and Distributed Computing, vol. 21. No. 1, pp.15-26, April 1994.

[5] B. Chapman, P. Mehrotra and H. Zima, *Programming in Vienna Fortran*, Scientific Programming, vol. 1, No. 1, pp. 31-50, August 1992.

[6] P. Banerjee, J. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy and E. Su, *The PARADIGM compiler for distributed-memory multi-computers*, IEEE Computer, vol. 28, No. 10, pp. 37-47, Oct. 1995.

[7] 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.

[8] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam, *Maximizing Multiprocessor Performance with the SUIF Compiler*, IEEE Computer, December 1996.

[9] R. Mirchandaney, J. Saltz, R. M. Smith, D. M. Nicol and Kay Crowley, *Principles of Run-time Support for Parallel Processors*, Proceedings of the 1988 ACM International Conference on Supercomputing, pages 140-152, July 1988.

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

[11] R. Das, M. Uysal, J. Saltz, Y. S. Hwang, *Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures*, Journal of Parallel and Distributed Computing, vol. 22, no. 3, pages 462-479, September 1994.

[12] Ravi Ponnusamy, Joel Saltz, Alok Choudhary, *Runtime-Compilation Techniques for Data Partitioning and Communication Schedule Reuse*, Proceedings Supercomputing '93 pages 361-370, November 1993.

[13] A. Lain and P. Banerjee, *Exploiting Spatial Regularity in Irregular Iterative Applications*, Proc. of the 9th International Parallel Processing Symposium, pp. 820-827, Santa Barbara, CA, 1995.

[14] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber and S. H. Teng, *Generating Local Addresses and Communication Sets for Data-Parallel Programs*, Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming, San Diego, CA, May 1993, pages 149-158.

[15] S. P. Midkiff, *Local Iteration Set Computation for Block-Cyclic Distributions*, Proceedings of the 24th International Conference on Parallel Processing, Oconomowoc, WI, 1995.

[16] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C. H. Huang and P. Sadayappan, *On Compiling Array Expressions for Efficient Execution on Distributed Memory Machines*, Proc. of the 22nd International Conference on Parallel Processing, IL, 1993.

[17] C. Ancourt, F. Coelho, F. Irigoin and R. Keryell, *A Linear Algebra Framework for Static HPF Code Distribution*, Proceedings of the Fourth Workshop on Compilers for Parallel Computers, Delft, The Netherlands, Dec. 1993.

[18] Ernesto Su, *Compiler Framework for Distributed-Memory Message-Passing Multicomputers*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1997.

[19] Antonio Lain, *Compiler and Run-time Support for Irregular Computations*, PhD thesis, University of Illinois at Urbana-Champaign, 1995.

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

[21] Dhruva R. Chakrabarti, Antonio Lain and Prithviraj Banerjee, *Evaluation of Compiler and Runtime Library Approaches for Supporting Parallel Regular Applications*, To appear in The International Parallel Processing Symposium, Orlando, Florida, March 1998.

[22] Dhruva R. Chakrabarti, Nagaraj Shenoy, Alok Choudhary and Prithviraj Banerjee, *Towards A Uniform Run-time Scheme for Mixed Regular-Irregular Applications*, Technical Report No. CPDC-TR-9801-004, Center for Parallel & Distributed Computing, Northwestern University, January 1998.

[23] A. Lain and P. Banerjee, *Techniques to Overlap Computation and Communication in Irregular Iterative Applications, Proceedings of the 8th ACM International Conference on Supercomputing*, pp. 236-245, Manchester, UK, July 1994.

[24] Yousef Saad, *SPARSKIT: a basic tool-kit for sparse matrix computations (Version 2)* , http://www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html.

[25] 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.

[26] 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.

[27] M. Ujaldon, E. Zapata, B. Chapman and H. Zima, *New data-parallel language features for sparse matrix computations*, Proceedings of the 9th International Parallel Processing Symposium, April 1995.

[28] 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).

[29] Scott R. Kohn and Scott B. Baden, *A robust parallel programming model for dynamic non-uniform scientific computations*, Proceedings of SHPCC, 1994.

[30] W. Gropp, E. Lusk and A. Skjellum, *Using MPI : Portable Parallel Programming with the Message Passing Interface*, Cambridge, MA, The MIT Press, 1994.

[31] ITPACK, *http://www.netlib.org/itpack/index.html*.

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

[33] Conjugate Gradient Algorithm, *http://www.netlib.org*.

[34] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing*, Benjamin-Cummings, 1994.

[35] M. Gupta, *Automatic Data Partitioning on Distributed Memory Multicomputers*, Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, IL, September 1992.

[36] *High Performance Fortran Language Specification, Version 2.0*, High Performance Fortran Forum, January 31, 1997.