An Efficient 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

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 require interprocessor 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 optimizations such as 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

A significant amount of research has been devoted to the development of compilers which automatically parallelize regular applications for distributed memory multi-computers. 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 regularly either in a blocked or cyclic(k) fashion among processors and f is an affine function of the enclosing loop variables. Irregular applications are another important class. 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 [1] paradigm. Runtime libraries like *CHAOS/PARTI* [2] and *PILAR* [3] 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.

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

*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.

ICS 98 Melbourne Australia

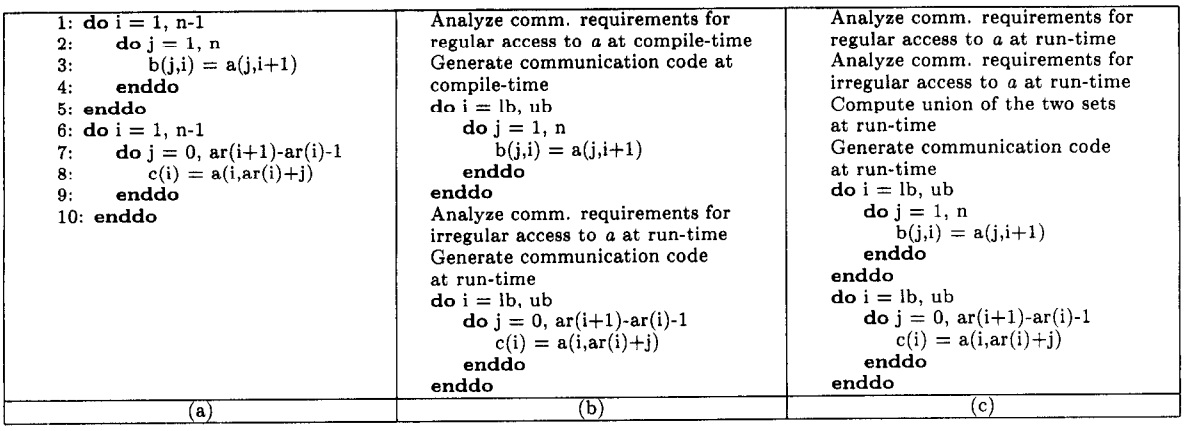


Figure 1: Exploiting Common References in a Mixed Regular-Irregular Application

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.

There exists no current 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. The popular run-time library for irregular applications, *CHAOS* [2, 4, 5], has special support for irregular accesses alone. This library cannot be used efficiently for mixed regular-irregular applications; 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* [3], it has been shown in [6] how preprocessing costs can be reduced to a minimum. On the other hand, *multi-block PARTI* [7] 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

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 [8, 9, 10, 11, 12, 13]. Using a linear algebra framework [12], the HPF compilation problem can be converted into solving a set of linear equalities and inequalities while polyhedron manipulation

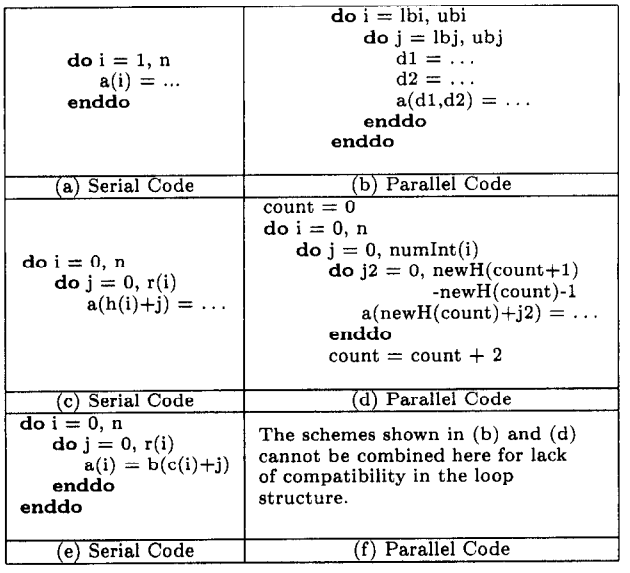


Figure 2: Lack of Compatibility of Code Structure for Regular and Irregular Accesses

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. Moreover techniques which change the dimensionality of a matrix are often used. [13] uses a technique where for cyclic(b) distributions, each dimension may be cast into two dimensions. More details can be found in [13] and [14]. A typical change in the structure of the code in the case of regular applications is shown in Figure 2(a, b).

Distributed memory code generated by run-time schemes for irregular accesses often changes 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) [15]. Owing to the data distribution, an interval in globals may get decomposed into multiple intervals and sub-intervals. Hence 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 in the case of irregular applications is shown in Figure 2(c, d). Clearly, this change in structure is not compatible to the change in structure generated by compile-time schemes as highlighted in Figure 2(e, f).

2.3 Commercial HPF compilers

We have studied two commercial HPF compilers, The Portland Group's (PGI) HPF compiler (version 2.2) [16] and IBM's HPF compiler (version 1.1.0.0) [17]) in order to observe the amount of communication since reduction of communication is one of the focus of our work. We present here a simple code fragment and the amount of communication (in bytes) generated by the PGI HPF compiler and our scheme. More details of our experiments can be found in [14]. 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).

```
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)

```
0→1: 10020
0→2: 10020
0→3: 10020
1→0: 15192
1→2: 10000
1→3: 10000
2→0: 11192
2→1: 14400
2→3: 10000
3→0: 10392
3→1: 14800
3→2: 10000
```

(b)

```
0→1: 116
0→2: 20
0→3: 4
1→0: 4804
1→2: 100
1→3: 108
2→0: 804
2→1: 4404
2→3: 4
3→0: 4
3→1: 4804
3→2: 4
```

(c)

Figure 3: Communication for 4 processors of IBM SP-2

Figure 3(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 while the other is irregular. The amount of communication, incurred by the PGI HPF compiler, illustrated in Figure 3(b), and other experiments performed by us show that the messages have not been coalesced by the PGI HPF compiler. Our scheme generates the minimum amount of communication and this is illustrated in Figure 3(c).

In our opinion, based on 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 [18, 19]. All such accesses should be handled using an interval representation [15]. 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.

3 Details of Our Scheme

The following subsections describe the scheme that we have used for handling applications containing both regular and irregular accesses. The scheme is described in detail in [14]. 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 [20] and the run-time library *PILAR* (Parallel Irregular Library with Application of Regularity) [3]. The basic irregular compiler support required for irregular applications has been reported in [15]. More details are being reported in a separate study.

We will use the synthetic benchmark (Figure 4) as a running example to illustrate our scheme. The code is written in HPF version 2.0 [21] 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 the 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.

```
      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 4: A Mixed Regular-Irregular Program Fragment

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.

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. This has been made possible by having support for multiple internal representations including an interval-based representation and an enumerated representation.

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. 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 runtime 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.

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

(a) Original Code | (b) Index-normalized Code

Figure 5: Spatial Regularity in Index-normalized code

Let us consider the code segment extracted from *SPARSKIT* [18] and shown in Figure 5(a). As illustrated in Figure 5(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 5 will be bounded by *ib(jj+1)* and *ib(jj)* for appropriate values of *jj*.

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.

3.5 Localization of Array References and Schedule Generation

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 performed 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 $GlobRef(X_i)$.

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

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. Each 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 $GlobRecvData(X) = GlobRef(X_1X_2...X_k) - Own(X)$. 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 know the addresses of the processors they need 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. The schedule for the example program is not shown here for lack of space.

3.6 Schedule Optimizations

Various communication optimization support present in *CHAOS* are described in [4]. 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.

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. 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. Derived data types are created for every pair of processors that needs to communicate among themselves. This allows message aggregation and message coalescing.

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 [19]), 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. *PILAR* supports both blocking and non-blocking global communication primitives. 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 6. 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.

```
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 6: Executor for the Example Program

4 Experimental Results

4.1 Methodology

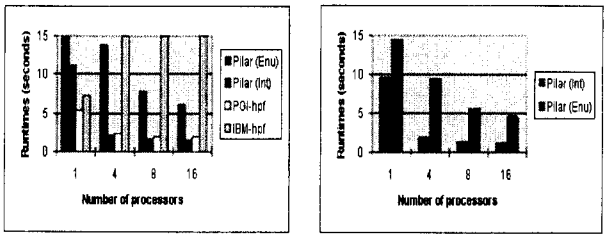
In this study, whenever we refer to results as obtained by the scheme *Pilar (Enu)*, we mean that the enumerated representation (similar to that used by *CHAOS*) has been used throughout irrespective of the type of accesses. [1] Otherwise, the interval representation is used whenever possible and this 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 4, subroutines to compute the infinity norm of a matrix (extracted from ITPACK [22]), a subroutine implementing the Lanczos algorithm [19]. All these benchmarks contain a mixture of regular and irregular accesses and are representative of real-life irregular applications.

[1] In a separate study [3, 15], we have demonstrated that the runtime 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.

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* 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.



(a) (b)

Figure 7: Runtimes for the Synthetic Benchmark

4.2 Synthetic Benchmark

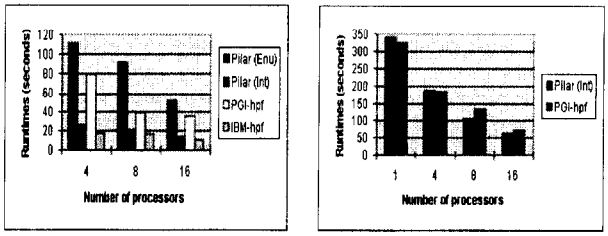
We have obtained comparative run-times for the synthetic benchmark shown in Figure 4 in different scenarios. In the first scenario, we have observed comparative run-times for *Pilar (Int)*, *Pilar (Enu)*, *PGI-hpf* and *IBM-hpf*. All arrays except d have the same distributions as that shown in Figure 4. 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 7(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.

In the second scenario, we have obtained comparative run-times for *Pilar (Int)* and *Pilar (Enu)* for the distributions specified in Figure 4. It may be noted that this scenario is likely to be more realistic than the previous one. As Figure 7(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 [22] 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 8(a) gives comparative run-times for different schemes in the case of the 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 8(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 8(b).



(a) (b)

Figure 8: Runtimes for Itpack Subroutines

We don't show the number of bytes communicated for the ITPACK kernels since the communication requirement is not very high in this case and both our scheme and the PGI HPF compiler generate similar amounts of communication. An execution profile of the code (kernel 2) parallelized with different versions of PILAR showed that the component costs in Pilar (Enu) are much higher than those in Pilar (Int). These results are not shown here for lack of space and they can be found in [14].

4.4 Lanczos Algorithm

We have extracted a kernel from the Lanczos algorithm for determining eigen systems from codes available in [19]. All sparse matrices are represented in compressed sparse row (CSR) format. We have obtained 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 9(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 9(b),

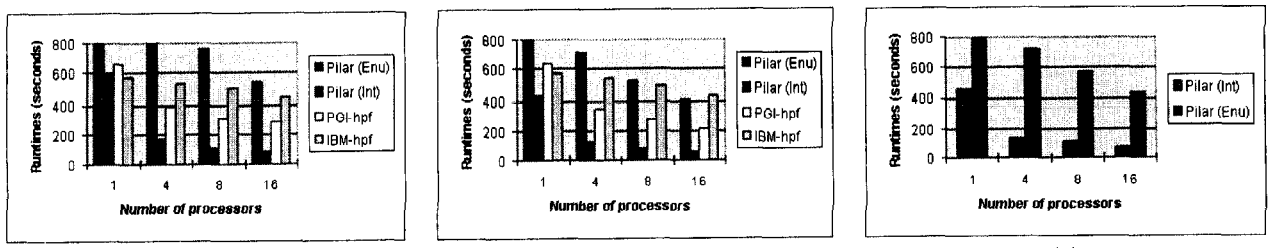


Figure 9: Comparison of Runtimes for the Lanczos Algorithm with Different Data Distributions

the run-times improve once the index arrays take into account the program semantics and there is spatial regularity. Figure 9(c) presents run-times for irregular distributions. In all the cases, *Pilar (Int)* fares better than other schemes.

Figure 10 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 10(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 10(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 9(b). Figure 10(c & d) show the inter-processor communication for the second scenario where intuitively many 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 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 [6]). 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 [2].
- *The commercial compilers use run-time schemes but* there are differences between their and our scheme. In case of commercial compilers, the support for handling 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* [2] 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* [7] provides support for block-structured applications with regular decompositions. The functionality and effectiveness of this library is very similar to that of *interval-based PILAR* if regular accesses and distributions are considered alone. However, in contrast to our scheme, this scheme is not 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.

Ujaldon and Zapata [23] have proposed an approach to reduce the number of levels of indirect array accesses. This method is, however, restricted to a specific set of applications only. 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

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 10: Communication volume for Lanczos Algorithm for Different Index Values

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 [24] 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 performed 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] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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, Northwestern University, February 1998.

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

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

[17] 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.

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

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

[20] 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 multicomputers*, IEEE Computer, vol. 28, No. 10, pp. 37-47, Oct. 1995.

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

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

[23] 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.

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