Evaluation of Compiler and Runtime Library Approaches for Supporting Parallel Regular Applications[*]

Dhruva R. Chakrabarti   Prithviraj Banerjee
Center for Parallel and Distributed Computing,
Northwestern University, Evanston, IL 60208
{dhruva, banerjee}@ece.nwu.edu

Antonio Lain
Hewlett Packard Labs.,
Barcelona, Spain
lain@crons1.hpl.hp.com

Abstract

Important applications including those in computational chemistry, computational fluid dynamics, structural analysis and sparse matrix applications usually consist of a mixture of regular and irregular accesses. While current state-of-the-art run-time library support for such applications handles the irregular accesses reasonably well, the efficacy of the optimizations at run-time for the regular accesses is yet to be proven. This paper aims to find out a better approach to handle the above applications in a unified compiler and run-time framework. Specifically, this paper considers only regular *applications and evaluates the performance of two approaches, a run-time approach using* PILAR *and a compile-time approach using a commercial* HPF *compiler. This study shows that using a particular representation of regular accesses, the performance of regular code using run-time libraries can come close to the performance of code generated by a compiler. We also determine the operations that usually contribute largely to the run-time overhead in case of regular accesses. Experimental results are reported for three regular applications on a 16-processor IBM SP-2.*

1. Introduction

Distributed-memory multi-computers have the potential to provide high levels of performance required to handle the Grand Challenge Computational Science problems. 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 [1]. A standard language, *High Performance Fortran (HPF)* has been developed for easy specification of these tasks. However, at present, this standard is limited to regular applications only. Unfortunately, many important applications are irregular and this makes communication patterns input-dependent, disabling traditional compiler optimizations. Lack of structure in the code presents difficulties in specifying data and computation distribution while complicating communication generation. However, a large subset of these applications has input-dependent 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. The result of this preprocessing step is then used multiple times during actual computation, thus amortizing its total cost. Runtime libraries like *CHAOS/PARTI* [2] and *PILAR* [3] simplify implementation of this preprocessing step and subsequent inter-processor communication.

Figure 1 shows a typical code segment encountered in irregular applications. Arrays *A* and *B* are accessed in a

```
do i = 1, m
    A(i) = 0.0
    do j = 1, R(i+1) - R(i)
        A(i) = A(i) + B(i+2) + C(R(i)+j)*D(R(i)+j)
    enddo
enddo
```

Figure 1. An Example Program

regular manner while accesses to arrays *C* and *D* are irregular. So intuitively, a scheme which handles optimizations for arrays *A* and *B* at compile-time [1] and those for arrays *C* and *D* at run-time is the most appropriate. However, implementation of such a non-unified scheme at the level of element access may be infeasible since the ultimate structure of the parallel code generated by compile-time and run-time techniques are often different. Also, implementation of two orthogonal schemes may result in redundant analysis which may in turn lead to increased run-time. This work aims to find out whether an efficient run-time analysis of regular accesses (by a library having support for irregular applications also) can match the performance of code generated by

[*]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.

a compiler. In such a case, a unified framework that handles both regular and irregular accesses efficiently can be developed. Support for such mixed regular and irregular accesses by a run-time library will be reported in a future study.

The rest of the paper is organized as follows. Section 2 gives an overview of the features of the run-time libraries used in our experiments. Section 3 discusses the methodology employed in the comparison. Experimental results are presented in Section 4. The last two sections discuss the relevance of this work in the context of some related work and present our conclusions.

2. Overview of Run-time Support

2.1. Different Internal Representations

PILAR (Parallel Irregular Library with Application of Regularity) [4] 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 by incorporating 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. The third representation, *CPIRCyclic* is targeted towards strided access when the stride is a small constant. 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.

2.2. Preprocessing Structures

In the spirit of the inspector-executor paradigm, the array references made by each processor are collected and stored in trace arrays. For regular accesses, all contiguous accesses are denoted by the lower bound and the upper bound. This not only requires less memory but also amortizes the trace collection cost.

2.3. Schedule

A *schedule* is an object storing a communication pattern. CPIR objects are used to represent permutations in the local address space. 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.

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. When *PILAR* is running over *MPI*, a schedule is instantiated by creating derived data types for every pair of processors that communicate. It has been shown in [4] that some *MPI* implementations can take advantage of derived data types to improve performance.

Mechanisms are provided in *PILAR* to automatically map global addresses to local ones and generate appropriate permutations in the local address space. Off-processor elements are accumulated at the bottom of the communication array in a contiguous manner. These facilities are similar to those provided in *CHAOS/PARTI*, the only difference being the use of specialized representations and different kinds of implementation of the same operation depending on the particular representation.

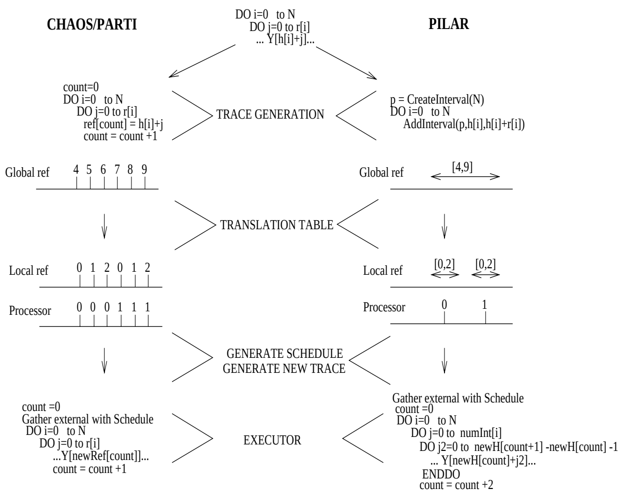
2.4. Global communication primitives

PILAR supports both blocking and non-blocking global communication primitives [5]. 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.

2.5. Interval 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 2 highlights the dif-

[1] In a separate report [4], 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.


```
DO i=0   to N
    DO j=0 to r[i]
      ... Y[h[i]+j]...
```

```
count=0
DO i=0   to N
    DO j=0 to r[i]
      ref[count] = h[i]+j
      count = count +1
```

```
p = CreateInterval(N)
DO i=0   to N
    AddInterval(p,h[i],h[i]+r[i])
```

```
count =0
Gather external with Schedule
 DO i=0   to N
     DO j=0 to r[i]
         ...Y[newRef[count]]...
         count = count +1
```

```
Gather external with Schedule
 count =0
 DO i=0   to N
     DO j=0 to  numInt[i]
        DO j2=0 to  newH[count+1] -newH[count] -1
           ... Y[newH[count]+j2]...
        ENDDO
        count = count +2
```

Figure 2. Differences between CHAOS/PARTI and PILAR with interval representation

ference 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. The resulting executor in Figure 2 for our approach is very different from the one obtained with a conventional technique. First, it tries to preserve the original array subscript structure as much as possible. For instance, in the example in Figure 2 we still have a regular access pattern carried by the innermost loop, regularity that is lost in the case of CHAOS/PARTI. This implies that the back-end compilers can generate more efficient code [4]. Secondly, we are sometimes forced to add extra loops in order to maintain this subscript structure. In our example in Figure 2, an extra loop enumerates all the sub-intervals in which an interval in globals is decomposed due to the data distribution.

3. Methodology

Three benchmark kernels have been used in comparison between the three schemes, code generated by the IBM HPF compiler, version 1.1.0.0. (IBM-hpf), code using the *interval-based PILAR* (Pilar (Int)) and code using the *enumerated-based PILAR* (Pilar (Enu)). The three benchmark kernels are *Jacobi's Iterative Method* [6], *2-D Explicit Hydrodynamics (Expl, from Livermore kernel 18)* [7] and *Adi Integration (ADI, from Livermore kernel 8)* [7]. These benchmarks have been widely used by the compiler community and being small enough, an extensive evaluation of various test cases has been possible. 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. *IBM's Standard High Performance Fortran* compiler has been used for compiling the above three benchmarks written in *HPF*. We have selected *MPI* [8] 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.

Pilar (Int)	1 proc	4 proc	8 proc	16 proc
Total	29.89	7.54	3.83	1.87
Inspection	0.272	0.187	0.149	0.02
Scheduling	0.106	0.027	0.028	0.012
Communication	0.001	0.001	0.001	0.001
Computation	29.62	7.35	3.68	1.85

Table 1. Component Runtimes: Jacobi(*,B)

Pilar (Enu)	1 proc	4 proc	8 proc	16 proc
Total	95.8	25.29	12.49	7.95
Inspection	51.69	12.55	6.23	2.95
Scheduling	47.92	12.46	6.15	2.92
Communication	0.003	0.54	0.58	0.23
Computation	44.11	12.22	5.68	4.77

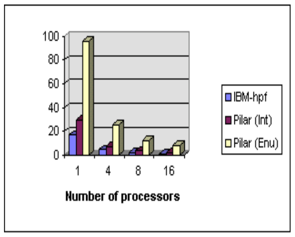
Table 2. Component Runtimes: Jacobi(*,B)

4. Experimental Results

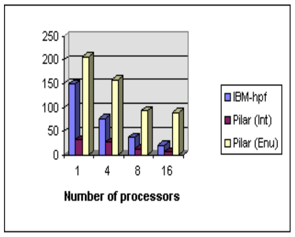
4.1. Jacobi's Iterative Method

This is a simple numerical computation often referred to as the Jacobi finite difference computation. In the benchmark that we have used, a two-dimensional grid is repeatedly updated by replacing the value of each point with some function of the values at points surrounding it. The code consists of two two-dimensional arrays and the sizes of the arrays used for the experimental results are 1024x1024 unless otherwise mentioned. Figure 3(a) gives the performance results for one-dimensional blocked distribution (by columns). The performance of *Pilar (Int)* is almost as good as that of *IBM-hpf* when the arrays are distributed blockwise. This is especially the case as the number of processors is increased upto 16. However, *Pilar (Enu)* performs quite poorly when compared to the other two.

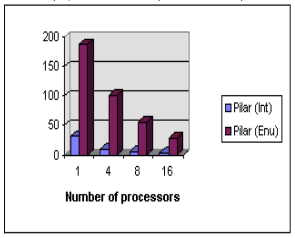
We have taken an execution profile of the code parallelized with *interval-based PILAR* and *enumerated-based*



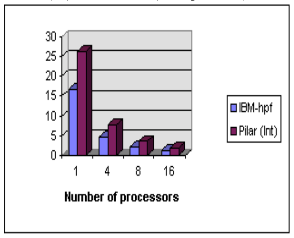
(a) Jacobi(*,block)



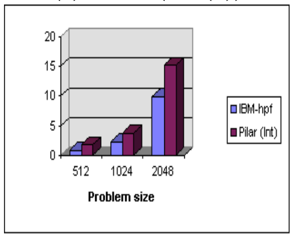
(b) Jacobi(*,cyclic)



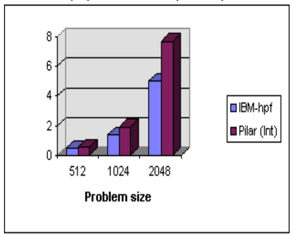
(c) Jacobi(*,C(8))



(d) Jacobi(B,B)



(e) 8 processor case



(f) 16 processor case

Figure 3. Comparative Runtimes for Jacobi

Pilar (Int)	1 proc	4 proc	8 proc	16 proc
Total	31.63	26.04	13.63	6.47
Inspection	0.301	0.097	0.192	0.03
Scheduling	0.130	0.048	0.016	0.016
Communication	0.003	17.88	9.51	4.51
Computation	31.33	8.06	3.93	1.93

Table 3. Component Runtimes: Jacobi (*,C)

Pilar to understand the component costs involved. The inspection cost subsumes the schedule generation cost. As the results in Tables 1 and 2 show, while the inspection time in *Pilar (Int)* is a very small percentage of the total time, the inspection time in *Pilar (Enu)* forms a sizeable portion of the total time. The executor in *Pilar (Int)* is very different from the one in *Pilar (Enu)*. The structure of the accesses along with the nesting level of loops may get changed differently in the two schemes. This explains the difference in computation costs shown in Table 1 and 2.

When the distribution of the arrays is changed to one-dimensional cyclic (by columns), the runtimes increase for all the schemes (Figure 3(b)). This is mainly owing to the disruption of the locality of the accesses and subsequent increase in the inter-processor communication.

The current *IBM-hpf* compiler does not support block-cyclic distribution. As Figure 3(c) shows, the performance of *Pilar (Int)* for one-dimensional block-cyclic distribution with a block size of 8 is intermediate between its performance for one-dimensional blocked and one-dimensional cyclic distributions.

Pilar (Int)	1 proc	4 proc	8 proc	16 proc
Total	26.36	6.81	3.45	1.91
Inspection	0.301	0.147	0.076	0.086
Scheduling	0.137	0.091	0.046	0.058
Communication	0.002	0.126	0.115	0.229
Computation	26.06	6.54	3.26	1.60

Table 4. Component Runtimes: Jacobi (B,B)

The runtimes for a two-dimensional block-block distribution are presented in Figure 3(d). Again, the performance of *Pilar (Int)* comes very close to the performance of *IBM-hpf*. The run-times in this case for each of the schemes is less than the corresponding one for a single-dimensional blocked distribution. An explanation for this is that better cache performance for a smaller working set more than compensates for a higher inter-processor communication. This is also indicated by the results in Tables 1 & 4 where the communication cost of two-dimensional blocked case is more than that for one-dimensional blocked distribution but the computation cost is lower in the former. The complete set of component costs for two-dimensional block-block distribution is shown in Table 4.

We have studied the changes in runtimes with changes in problem size. Figures 3(e & f) highlight the runtimes for *IBM-hpf* and *Pilar (Int)* for 8 and 16 processors respectively where the arrays are distributed blockwise. Both the schemes scale effectively and the performance of *Pilar (Int)* is close to that of *IBM-hpf*.

We have also observed the changes in runtimes with changes in the distribution of the arrays (Tables 1, 3 & 4). The cost of inspection other than scheduling is largely independent of the distribution if the translation table is replicated in each of the processors. The scheduling cost may slightly vary depending on the distribution; the scheduling cost is the least for block distribution owing to least executor communication generation. The scheduling costs increase slightly for block-cyclic and two-dimensional block-block distribution since for more complex executor communication, the complexity of formulating the derived data types among other scheduling optimizations is more. The communication cost also increases substantially for one-dimensional block-cyclic and two-dimensional block-block distributions. The computation costs remain approximately the same across distributions as expected.

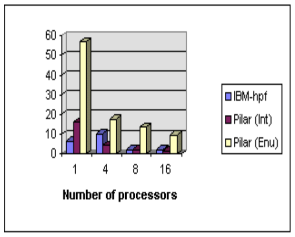
4.2. Explicit Hydrodynamics (EXPL)

2-D Explicit Hydrodynamics (EXPL, from Livermore kernel) has nine 1024x7 arrays accessed inside doubly perfectly nested loops, all enclosed within the same non-partitionable outer loop. The doubly nested loops contain multiple statements, each performing a non-trivial computation. The communication pattern is regular and always oc-

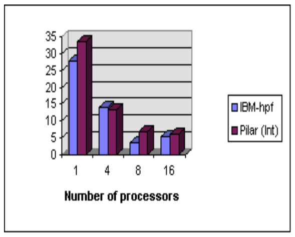
Benchmark	Compiler	1 proc	4 proc	8 proc	16 proc
Expl512(*,B)	IBM HPF	3.47	5.75	1.05	1.06
Expl512(*,B)	Pilar (Int)	8.00	2.33	1.36	0.90
Expl1024(*,B)	IBM HPF	6.75	10.15	2.11	2.04
Expl1024(*,B)	Pilar (Int)	16.14	4.39	2.39	1.41
Expl2048(*,B)	IBM HPF	13.74	20.03	19.30	19.40
Expl2048(*,B)	Pilar (Int)	33.10	8.74	4.44	2.47

Table 5. Variation of Runtimes for Expl

curs among neighbors. Though some of the inter-processor communication can be vectorized and moved out of the outer loop, most of the communication occurs at points between the outer loop and inner loop nests. The *PILAR* library has been successful in optimizing the communication.



(a) Expl(*,B)



(b) Expl(*,C)

Figure 4. Comparative Runtimes for EXPL

Figure 4(a) gives the performance results for one-dimensional blocked distribution of the arrays. Interestingly, *Pilar (Int)* performs better than *IBM-hpf*. The runtimes for *Pilar (Int)* are better than *IBM-hpf* for 4 and 16 processors and the former scheme obtains a better relative speedup. The runtimes for *IBM-hpf* show a sudden rise with increase in the number of processors but then tend to come down gradually. We have tried to investigate this unexpected behavior by studying the intermediate code generated by the compiler. However, the loops were found to have been parallelized appropriately and the exact cause of poor runtimes could not be ascertained. Without an in-depth performance analysis with tracing or profiling, it is not clear to us from the intermediate code generated whether inter-processor communication, which forms a major part of the runtimes, has been properly optimized by the compiler. Assuming that the ratio of communication to computation is very high for *IBM-hpf*, we have observed runtimes for various array sizes hoping that the compute power of processors will be utilized more effectively. As Table 5 shows, the runtimes for *IBM-hpf* consistently follow a certain pattern : an increase in runtimes with increase in number of processors from 1 to 4, a steep drop in runtimes (for array sizes of 512 and 1024) with increase in number of processors from 4 to 8 and almost no changes with increase in number of processors from 8 to 16. For array size of 2048, the performance of *IBM-hpf* is particularly inexplicable since it obtains fractional speedup even for 16 processors. In contrast, *Pilar (Int)* achieves consistent speedups and good runtimes.

With the change in distribution to one-dimensional

Pilar (Int)	1 proc	4 proc	8 proc	16 proc
Total	33.62	13.18	7.01	6.14
Inspection	1.36	0.427	0.358	0.381
Scheduling	1.25	0.377	0.323	0.312
Communication	0.039	6.377	4.22	4.19
Computation	32.22	6.38	2.43	1.57

Table 6. Component Runtimes: Expl (*,C)

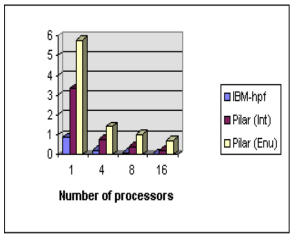
Pilar (Int)	1 proc	4 proc	8 proc	16 proc
Total	32.38	14.81	9.22	9.54
Inspection	1.32	0.350	0.211	0.150
Schedule	1.22	0.325	0.172	0.093
Communication	0.044	8.34	6.78	8.55
Computation	31.02	6.12	2.23	0.84

Table 7. Component Runtimes: Expl (*,C(16))

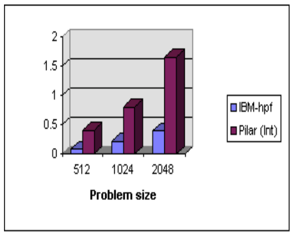
cyclic, the runtimes increase for all the schemes. As shown in Figure 4(b), both *IBM-hpf* and *Pilar (Int)* obtain similar runtimes and achieve similar speedups. The component costs for one-dimensional cyclic and one-dimensional block-cyclic distributions (block size of 16) are shown in Table 6 and 7 respectively.

4.3. ADI Integration

ADI (from Livermore kernel) features a triply nested loop enclosing six assignment statements (three non-trivial RHS computations and three simple ones), all of which give rise to communication that can be vectorized out of the loop. So inter-processor communication needs to be done only once. *IBM-hpf* correctly detects this feature and performs extremely well. *Pilar (Int)* detects this feature also in the sense that it generates empty schedules and the calls to communicate don't actually communicate. However, these empty calls do lead to some overhead and this gets reflected in the runtimes.



(a) Adi(*,B)



(b) Prob. Size Variation

Figure 5. Comparative Runtimes for ADI

For a one-dimensional blocked distribution, *Pilar (Int)* performs comparably with *IBM-hpf* (Figure 5(a)) and achieves almost linear speedup. The extent of difference in performance between *Pilar (Int)* and *Pilar (Enu)* as compared to the other two benchmarks is worth noticing (Table 8 & 9). The size of intervals in this benchmark is very small and there is not much scope to exploit regularity. In fact, profiling of *Pilar (Int)* and *Pilar (Enu)* shows that inspection and scheduling costs are actually lower for

Pilar (Int)	1 proc	4 proc	8 proc	16 proc
Total	3.34	0.80	0.41	0.21
Inspection	0.52	0.129	0.070	0.044
Scheduling	0.52	0.126	0.068	0.043
Communication	0.00	0.00	0.00	0.00
Computation	2.82	0.67	0.34	0.17

Table 8. Component Runtimes: Adi (*,B)

Pilar (Enu)	1 proc	4 proc	8 proc	16 proc
Total	5.78	1.43	1.02	0.72
Inspection	0.51	0.104	0.085	0.063
Scheduling	0.43	0.10	0.09	0.06
Communication	0.00	0.00	0.00	0.00
Computation	5.27	1.33	0.93	0.66

Table 9. Component Runtimes: Adi (*,B)

the latter in some cases. However, the communication costs are lower in the former, probably due to better optimization. This is a case where run-time methods are likely to perform badly since inter-processor communication takes place only once and all preprocessing and optimizations do not make much difference. This is reflected in Figure 5(b) which highlights the results for different problem sizes on 4 processors.

4.4. Summary of Results

The performance of traditional run-time libraries (for example, *CHAOS/PARTI*), emulated here by *PILAR* with enumerated representation, is much inferior to that of compiler approaches for regular applications. Hence we conclude that run-time approaches using standard irregular support should not be used for regular accesses in applications.

The performance of a run-time library with appropriate representations (for example, *PILAR* with interval representation) and sophisticated optimization can come close to the performance of code generated by a compiler even for regular applications. Hence we conclude that sophisticated run-time approach using interval support can be used for regular accesses in applications.

A major reason behind the above feature is that with a representation similar to the interval representation, the runtime preprocessing costs can be reduced to a minimum. This enables total runtimes to be comparable since communication and computation costs are similar in both compile-time and run-time approaches.

In cases of regular distributions and accesses, compilers can help determine the communication requirement. For instance, in *ADI*, communication needs to be done only once. While all run-time methods will generate empty schedules and execute empty communication calls inside the loops, a unified scheme should be able to remove any communication calls and hence reduce overheads.

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 motivated by irregular applications, the representation used is an enumerated one. Consequently, the library does not perform well for regular applications.

Multiblock PARTI [9] 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. 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 *PILAR*, this library is not suitable for irregular accesses. This library is a stand-alone library in its current implementation and has not been integrated with the *CHAOS/PARTI* library.

6 Conclusions

This study shows that using a particular representation of regular accesses, the performance of regular code using run-time libraries can come close to the performance of code generated by a compiler. We have also determined the operations that usually contribute largely to the run-time overhead in the case of regular accesses. The results indicate that a run-time library incorporating multiple representations for efficient resolution of both regular and irregular accesses will perform better than conventional techniques in case of real-life irregular problems.

References

[1] 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.

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

[5] 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.

[6] S. Golub and J. M. Ortega, *Scientific Computing : An Introduction with Parallel Computing*, San Diego, CA, Academic Press, 1993.

[7] F. McMohan, *The Livermore Fortran kernels : A computer test of the numerical performance range*, Lawrence Livermore National Laboratory, Livermore, CA, Tech. Rep. UCRL-53745, Dec. 1986.

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

[9] 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.