COMPILER AND RUN-TIME SUPPORT FOR IRREGULAR COMPUTATIONS

BY

ANTONIO LAIN

Ingeniero, Universidad Politecnica de Madrid, 1990
M.Sc.,University of Edinburgh, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

Abstract

There are many important applications in computational fluid dynamics, circuit simulation and structural analysis that can be more accurately modeled using iterations on unstructured grids. In these problems, regular compiler analysis for Massively Parallel Processors (MPP) with distributed address space fails because communication can only be determined at run-time. However, in many of these applications the communication pattern repeats for every iteration. Therefore, equivalent optimizations to the regular case can be achieved with a combination of run-time support (RTS) and compiler analysis.

Moreover, many real applications have irregular access patterns with certain structure. This structure is needed to improve sequential performance, better exploiting the memory hierarchy. In many cases this structure is reflected in the array subscript expressions in the form of hybrid accesses, that is, affine combinations of array indirection and regular accesses. We will show that when we get advantage of this structure, we can dramatically reduce pre-processing overheads, both in terms of memory and time, and improve communication and single node performance.

In order to achieve these goals we need a combination of run-time and compiler support. Unfortunately, existing irregular run-time support cannot get advantage of regularity and therefore, our first requirement was to develop a new run-time support for irregular computations, that we called PILAR (Parallel Irregular Library with Application of Regularity). Our second requirement was to automate the task of exploiting regularity in irregular applications through compiler support. For this task we extended the PARADIGM compiler, developed at the University of Illinois, to handle irregular problems. Experimental results will validate the efficacy of our approach.

Dedication

A mis padres, Antonio y Pilar

Acknowledgments

I would like to thank my advisor, Professor Prithviraj Banerjee, for his constant support, encouragement and advice that made possible this research. I will always remember our weekly meetings as one of the most intellectually challenging experiences of my life. I would also like to thank the members of my committee, Professors David Padua, Constantine Polychronopoulos and Michael Heath, for their valuable comments and suggestions on my work. Further thanks to Santosh Abraham and Alexandre Eichenberger for their help with the Indico program.

Thanks also to the PARADIGM team (Buck, Shankar, Ernesto, Dan and Amber) for creating a very enjoyable working environment that was crucial to the success of my research. In particular, Shankar and our coffee breaks were essential to remember that life is not only computers. Discussions with Lawrence Rauchwerger made me understand what run-time parallelization really means.

Victor and Luis made a very pleasant experience my early years in USA. Juliana, Pilar, Eduardo, Joan, Nuri, Luis, Ricardo and the rest of the "Spanish gang" made much easier to "survive" in Urbana. In particular, my friend Isabel, with her constant support and understanding, really improved my "quality of living" during my stay in Urbana.

I would like to thank my family for their love and support during my long years abroad. I would have not stand it without them. Thanks also to my grandfather Joaquin. I really missed our long and always interesting conversations during my stay in Urbana.

Table of Contents

List of Tables

Table **Page**

List of Figures

Chapter 1

INTRODUCTION

The demands of computing power in many important applications does not seem to be satisfied by improvements in basic computer technology. For the last twenty years, parallel processing is being explored as a viable option to further increase this performance. Distributed-memory multi-computers seem to have a hardware cost/performance and scalability advantage over other parallel architectures. However, this is at a cost of a greater software complexity.

Recent efforts have tried to tackle the problem of software complexity in distributed-memory multi-computers. MPI [45], an important standardization effort of message passing software, provides a common interface of a relatively low-level programming paradigm. HPF [34], an extension to Fortran that provides directives to distribute data, implicitly distributing the computation using the "owners-compute" rule [1], gives a higher-level programming interface for data-parallel applications, typically regular ones, shifting the complexity into the HPF compiler.

Unfortunately, a large proportion of "real-world" applications are irregular [63]. The "software problem" in multi-computers become much worse for these applications. Communication patterns are input dependent, disabling traditional HPF compiler optimizations. The need of aggregating communication to reduce network latency overhead complicates communication generation. Also, in many cases good computation and data distributions cannot be represented in simple ways, as in the case of regular computations, and are critical to obtain performance.

An important subset of these irregular applications have input dependent communication patterns that repeat across multiple iterations. In those cases run-time compilation techniques [62] can be used. The dependency structure of the problem is analyzed before the actual computation in a preprocessing step. Then, the information extracted during the preprocessing can be used to improve the performance of the computation. Moreover, this pre-processing can be re-used across multiple iterations, amortizing its cost. In practice, irregular run-time support libraries are used to simplify the implementation of this pre-processing and to communicate irregularly accessed arrays.

In this thesis we want to take this idea even further. Many real applications have irregular access patterns with certain structure. This structure is needed to improve sequential performance, better exploiting the memory hierarchy. In many cases this structure is reflected in the array

[1] the processor that "owns" a value is responsible for computing and updating that value

subscript expressions in the form of hybrid accesses, that is affine combinations of array indirection and regular accesses. Figure 1.1 compares the different array subscript types. We will show that when we get advantage of this structure, we can dramatically reduce pre-processing overheads, both in terms of memory and time, and improve communication and single node performance.

DO I=1, N ... A(I)... ENDDO	DO I=1, N ... A(B(I))... ENDDO	DO I=1, N ... A(I+B(J))... ENDDO
Regular	Irregular	Hybrid

Figure 1.1 Array references types

In order to achieve these goals we need a combination of run-time and compiler support. Unfortunately, existing irregular run-time support cannot get advantage of regularity and therefore, our first requirement was to develop a new run-time support for irregular computations, that we called PILAR (Parallel Irregular Library with Application of Regularity). Our second requirement was to automate the task of exploiting regularity in irregular applications through compiler support. For this task we extended the PARADIGM compiler, that we will briefly describe next, to handle irregular problems.

The PARADIGM (PARAllelizing compiler for DIstributed memory General-purpose Multicomputers) project at the University of Illinois provides an automated means to parallelize programs, written in a serial programming model, for efficient execution on distributed-memory multicomputers. The compiler accepts either a sequential FORTRAN 77 or HPF program and produces a parallel program optimized for a target machine.

PARADIGM uses *Parafrase-2* [51] as a preprocessing platform to parse the sequential program into an intermediate representation, to analyze the code and generate flow, dependence, and call graphs, and to perform transformations such as constant propagation and induction variable elimination.

In addition to the traditional compiler optimizations to distribute computations and to reduce communication overheads, PARADIGM's capabilities include:

- Optionally parse High Performance Fortran (HPF) directives [39].
- Perform automatic data distribution for regular computations, conventionally only specified through user directives.
- Generate high-level communication primitives.

- Optimize communication for regular computations.
- Exploit functional and data parallelism simultaneously.
- Generate multi-threaded message-driven code to tolerate communication latencies.

In this dissertation we describe an extension to PARADIGM to provide support for irregular computations using a combination of compile-time analysis and run-time support.

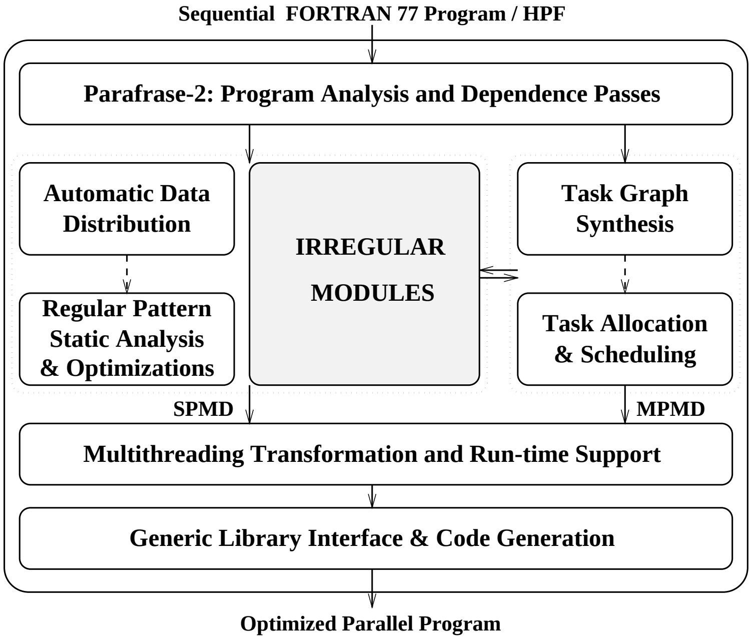

Figure 1.2 PARADIGM compiler overview

Figure 1.2 shows the different components of PARADIGM, high-lighting how the new irregular modules described in Chapter 4 fit in the general framework of PARADIGM.

The rest of this dissertation is organized as follows. Chapter 2 describes some related work in the areas of compiler and run-time support for irregular problems. Chapter 3 describes PILAR, a new run-time support for irregular computations. Chapter 4 describes the modifications to PARADIGM that are needed to handle irregular computations using PILAR. Chapter 5 presents experimental results for some Fortran programs compiled with PARADIGM on two multi-computers. Finally, Chapter 6 presents our conclusions and discusses future work in this area.

Chapter 2

RELATED WORK

In this chapter we will briefly describe related work in the areas of compiler and run-time support for irregular computations. There has been a significant amount of research on this topic. For this reason, we will further restrict our descriptions to work that is closer to our research.

2.1 Irregular Run-Time Support

The CHAOS/PARTI library [53], and in particular, the original PARTI library [54], had a significant impact in the design of PILAR. We borrowed and extended some basic concepts like storing an optimized communication pattern in a *Schedule* and reusing it with the help of global exchange primitives, or the translation from global indices to local indices and processor ids using *Distributed Translation Tables*. However, the fact that we wanted to get a better match with MPI, that we support different internal representations of communication patterns, that we provide non-blocking global exchange primitives and that everything is given in an object-oriented framework distinguish PILAR from PARTI. Chapter 3 describes in more detail the differences and similarities between them.

Multiblock PARTI [2] provides support for block-structured applications with regular decompositions. Unknown problem sizes may disable compile-time analysis for regular decompositions and this library can optimize communication at run-time for those cases. In contrast to PILAR, this library is not suitable for indirect accesses.

LPARX [41] is a C++ library that provides run-time support for dynamic, block-structured, irregular problems in a variety of platforms. Its main target is not the fine grained blocks obtained from unstructured grids but the larger ones that come from finite difference methods.

A related area of research, critical for the success of an irregular RTS, is the development of efficient algorithms to partition unstructured grids across processors. The goal of this partitioning is to minimize communication across partitions while maintaining a reasonable load balance. Recently, there has been an enormous research effort in this area [50, 24, 33, 25, 37, 10] that has lead to the development of scalable and fast parallel algorithms to perform this task.

An interesting approach pioneered by Ranka et al [50] provides an architecture-independent transformation that improves locality. After this transformation, that may include array re-indexing, it is possible to assume that a block-based decomposition would minimize communication across partitions. This method can simplify load balancing in adaptive problems, and it also reduces the compiler support needed. We apply some of these ideas in Chapter 5.

2.2 Compilation of Irregular Applications

Kali [40] was the first compiler system to provide support for regular and irregular applications in multi-computers. A difference with Fortran-D is that the owner computes rule was not implied. This requires that both the data and the computation partitioning need to be specified. Support for irregular computations was given by the inspector/executor model, although the data structures used to keep external references were different from PARTI.

ARF [72] was the first effort to automatically insert calls to PARTI in programs written in a small subset of Fortran. It also follows the inspector/executor model.

Hanxleden [30] developed the irregular support for the Fortran-D compiler at Rice. The irregular run-time support that he used was CHAOS/PARTI. An important contribution of this work is a dataflow framework called Give-N-Take [69], based on partial redundant elimination [47], that is used for placement of communication statements in arbitrary points in the program. He also proposed using value-based decompositions that use the contents of an array to decide on data distribution, as opposed to index-based decompositions commonly used for regular problems. His implementation used symbolic analysis techniques to further eliminate redundant inspectors. However, he did not exploit regularity in irregular problems, and his work was more focused on communication optimization rather than inspector generation/placement, a non-trivial task when a complex communication placement scheme is assumed. Also, his experiments did not demonstrate the actual performance advantage of using a complex communication placement scheme over a simpler one.

Das et al [18] developed program transformation techniques, based on code slicing, to flatten multiple levels of indirection, even in the presence of conditionals. They ensure that dependences between slices are preserved by performing a topological sort of a slice graph. A limitation of their approach is that they treated communication placement as an orthogonal problem to inspector generation, losing opportunities to further optimize inspection. Also, they remove multiple levels of non-locality by removing multiple levels of indirection, and this has some consequences, discussed in Section 4.4.6. Their basic irregular infrastructure is also based on Fortran-D and the CHAOS run-time support.

Bozkus et al [15] developed inspectors for Fortran90 *forall* statements with assignments containing indirection arrays. They also used PARTI as the run-time support.

Eswar et al described in [20] a scheme for checking feasibility of inspector extraction. Our approach is based on this scheme and further discussed in Section 4.4.5.

Ruhl [60] was the first one to handle multiple levels of non-locality, even in the presence of conditionals. His scheme did not remove multiple levels of indirection, only multiple levels of non-locality. A comparison between his scheme and the scheme by Das et al can be found in Section 4.4.6. He also developed his own run-time support, using hash tables to store external references. Recent extensions [48] can further increase the accuracy of slicing by generating slices dynamically.

Ujaldon and Zapata [67] proposed a different approach to reduce the number of levels of non-locality. Their method is based on enforcing alignment restrictions of common sparse data structures by using compiler directives. Later, the compiler can take advantage of this information, reducing the complexity of inspection and communication. This is being implemented as an extension to the Vienna Fortran Compilation System [68].

Wijshoff and his research group at Leiden University proposed a sequential irregular compiler [11] that would transform a program with regular data structures into one that uses a sparse representation. They also proposed a scheme in [38] to improve data locality in sparse computations through a combination of run-time search of patterns in the sparse matrix, suggested by user directives, and multi-version code for these patterns automatically generated by their compiler.

In this thesis we are not considering the problem of extracting parallelism at run-time, relying instead on compiler directives when static analysis cannot identify parallel loops. Interesting work in this area, mainly for shared address space multiprocessors, is described in [46, 74, 16, 52]. Recently, Rauchwerger [59, 58] proposed a very promising scheme to detect fully parallel loops. This scheme is being implemented in the context of the Polaris compiler project [12] at the University of Illinois.

Chapter 3

RUN-TIME SUPPORT FOR IRREGULAR COMPUTATIONS

3.1 Overview

In order to parallelize applications that have input-dependent communication patterns on a Massively Parallel Processor (MPP), run-time compilation techniques [54] can be used. The dependency structure of the problem is analyzed before the actual computation in a preprocessing step. If a certain structure of the computation is maintained across several iterations, this preprocessing can be reused, amortizing its cost. In practice, this concept is implemented with two sequences of code: an inspector code for the preprocessing and an executor code for the actual computation.

The preprocessing step can be very complex. It partitions the unstructured grid, optimizes the resulting communication pattern and translates global into local indices. Also, during the execution of the computation, data is exchanged based on this preprocessing analysis. In order to simplify the implementation of inspectors and executors, irregular run-time support libraries (RTS) like CHAOS/PARTI [53] or PILAR, a new RTS described in this chapter, provide primitives for these operations.

Fortunately, in many irregular applications access patterns have certain structure. This structure is used to obtain reasonable sequential performance in modern architectures by reducing the indexing overhead and improving cache performance. This structure is sometimes reflected in the subscript expressions of irregular array references in the form of combinations of affine expressions of the loop variables and array indirection. An important observation is that the same structure that is used for improving sequential execution can also be used to ease the parallelization of an irregular application.

Preliminary experiments presented in [43] showed that state of the art run time support for irregular computations on MPP like CHAOS/PARTI [53] fail to exploit this regularity, producing unnecessary time and memory overheads. PILAR (Parallel Irregular Library with Application of Regularity) was designed at the University of Illinois to fill the gap between existing irregular RTS and to provide a basic irregular framework for PARADIGM [7], a HPF/Fortran-77 compiler. PILAR is a C++ library that supports different internal representations of communication patterns. This allows the efficient handling of an entire range of applications, from fully irregular to regular, using a common framework. For example, an interval based representation is used to exploit spatial regularity, and an enumerated representation, similar to PARTI, is used for patterns with little regularity. Object oriented techniques are used to encapsulate this internal representation from the rest of the library, simplifying the implementation of new representations and the interaction between objects with different internal representations.

MPI [45] is an emerging message passing standard that has already attracted much attention in industry and academia. MPI adds new features like derived data types and communicators that are not typically used in current systems. PILAR has been designed to take full advantage of these new features, making it easy to integrate PILAR with future MPI libraries.

Non-blocking global communication primitives are important for certain irregular problems [42]. They relax an implicit barrier synchronization present in blocking irregular communication primitives, allowing to overlap computation and communication. These primitives are supported in PILAR in a very efficient way.

Also, we are proposing mechanisms to combine the results of several pre-processing steps using local operations. These operations benefit from our efficient internal representations and simplify the implementation of communication primitives.

To summarize, some important features of PILAR are:

- Reduces memory and time overhead using structure of irregular codes.
- Gives full support to MPI advanced features.
- Provides non-blocking global data exchange primitives to overlap computation and communication.
- Uses a compositional approach to pre-processing, regardless of the internal representation used during this pre-processing.
- Provides all of the above in an object oriented framework.

3.2 Exploiting Regularity in Irregular Applications

There are many important applications in computational fluid dynamics, circuit simulation and structural analysis that can be accurately modeled using iterations on unstructured grids [63]. Figure 3.1 shows an example of an unstructured grid. Typical applications associate values with

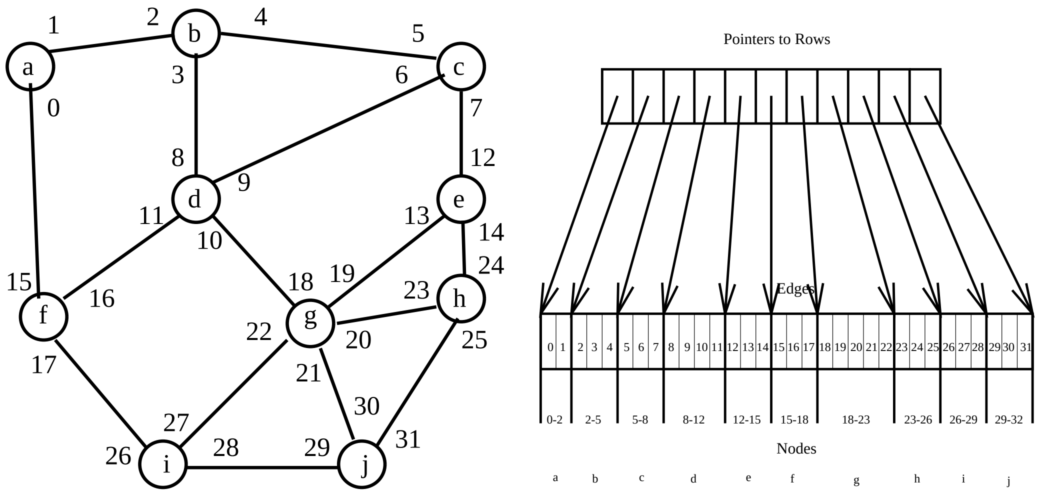

Figure 3.1 An example of an unstructured grid and its contiguous layout

edges or nodes or both and update the values of the nodes with a function of the values of their neighbors.

It is well known that the performance of these operations in a sequential computer depends heavily on the internal representation of these grids [19]. For this reason, representations that layout contiguously in memory all the edges associated with a particular node are commonly used. Figure 3.1 shows an unstructured grid and its contiguous representation. The advantage of this layout comes into play when the iterative algorithm traverses all the edges of a particular node in sequence. In this case, the spatial locality of the accesses improves the performance of the memory hierarchy. A typical algorithm using spatial locality can be seen in Figure 3.2. Here, references to the array y will first have an irregular access $y[index[i]]$ followed by a sequence of $numEdges[i]$ contiguous accesses.

```
for i = 0 to numNodes-1
  for j = 0 to numEdges[i]-1
    x[i] = x[i]+ f(y[index[i]+j],...)
```

Figure 3.2 Sweep with spatial locality

It is also common in many irregular codes to associate more than one value with each edge or node in the unstructured grid. In those cases, it also tends to improve cache behavior to layout contiguously in memory all the elements of a particular edge or node. This implies that array accesses are regularly strided after a starting irregular access, and the stride is usually small, typically related to the amount of information associated with each node or edge. Figure 3.3

shows an example with a 3-strided unstructured sweep. These strided accesses are also found when multi-dimensional arrays are only irregularly accessed in some dimensions but not others.

```
for i = 0 to numNodes-1
  for j = 0 to numEdges[i]-1
    x[i] = x[i]+ f(y[index[i]+3 j],y[index[i]+3 j+1], y[index[i]+3 j +2],...)
```

Figure 3.3 Sweep with small constant stride

In practice, most successful irregular codes in current microprocessors use spatial locality in one way or another to improve first level cache hit ratio. Most current and future MPP are built with off-the-shelf microprocessors. Therefore, only these irregular applications will obtain good raw performance on MPP after parallelization. This fact motivates our work: designing a RTS that takes advantage of the subscript structure of irregular array references, present in many sequential applications, to further reduce the overhead involved in parallelization.

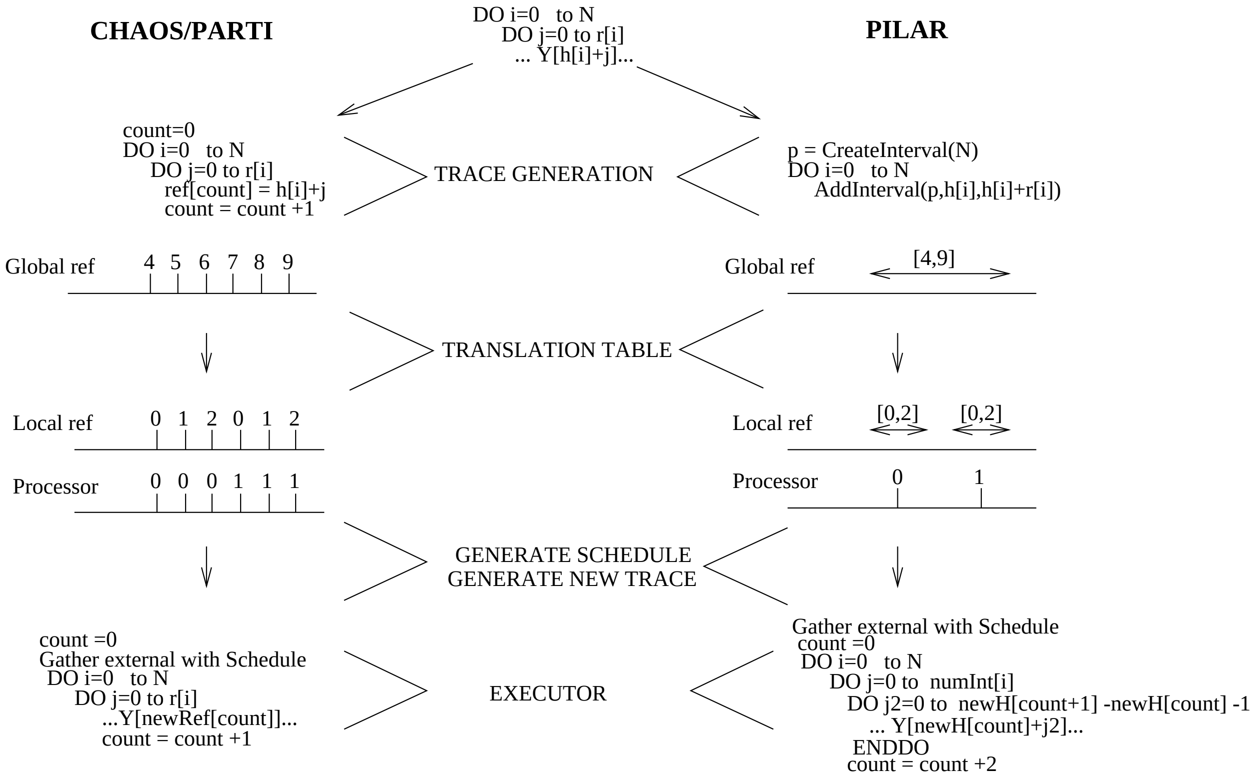

Figure 3.4 Using spatial regularity

An important subset of irregular applications have spatial locality. Figure 3.4 shows the different handling of an irregular reference with spatial locality in PILAR and CHAOS/PARTI. In both cases we need to generate the stream of global references and store it in a data structure. Then, we have to translate these global references into processor location and local offsets in this processor using a translation table. Later, we can filter the local references and optimize the communication pattern of external references storing this pattern in a *Schedule*. Also, we need to obtain the new local references that will be used during the executor. During the actual computation we first collect the external references using the *Schedule* and a global communication primitive and then perform the computation using the new local indices. 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 time for these operations, as will be shown in Chapter 5.

Also, the resulting executor in Figure 3.4 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 3.4 we still have a regular access pattern carried by the innermost loop, regularity that is lost in the case of CHAOS/PARTI. Second, we are sometimes forced to add extra loops in order to maintain this subscript structure. In our example in Figure 3.4, an extra loop enumerates all the sub-intervals in which an interval in globals is decomposed due to the data distribution. In Section 4.8 we will discuss in more detail the code transformations needed and how to automate them. Also, in Chapter 5 we will show that in many cases there is a performance advantage of using this type of executor.

Clearly, if the original access pattern has no structure, or this structure is destroyed by the data distribution, we will not be able to benefit from a more complex representation of access patterns. For those cases we rely on an enumerated representation (similar to PARTI). Also, in other cases we have a fuzzy situation in which there is some structure but it is very fine grained. Moreover, we will show in Chapter 5 that in many applications we have array references with a clear structure mixed with others that have none. These two observations lead to a general design principle on PILAR representations: make representations simple enough so that they can easily be combined and can also benefit from a fine-grained structure. In [43] we measured the threshold of regularity at which our new representations start to pay-off, finding that this threshold was very small. This should also be clear from the results in Chapter 5.

In some cases the irregular access pattern has structure, but the array subscript expressions do not reflect this structure. In fact, this is not a problem only for parallelization but also for sequential performance of the original code. In the context of sequential performance, Knijnenburg et al [38] proposed a combination of run-time search of patterns in the sparse matrix, specified by user directives, and a compiler that would generate multi-version code to take advantage of these patterns. Our approach is orthogonal to this problem because we could take as input the multi-version code, where patterns are explicit, and optimize it for execution on multi-computers.

3.3 Irregular Computations using MPI

MPI [45] is an emerging standard of message passing that has attracted much attention in the industrial and academic community.

MPI does not provide explicit support for irregular computations, but it has enough hooks to make an irregular run-time support built on top of it very efficient. In this section we discuss these hooks and how PILAR uses them.

MPI supports derived data types (DDT) in point-to-point and collective communications. A DDT is a type obtained from simpler ones through a sequence of combining operations. The combining operation *MPI_Type_indexed()* is of particular importance for irregular problems, allowing the specification of blocks of simpler data types with an arbitrary number of elements and offset. Derived data types do not impose message packing on the underlying implementation. Therefore, sending and receiving non-contiguous messages can be as efficient as possible.

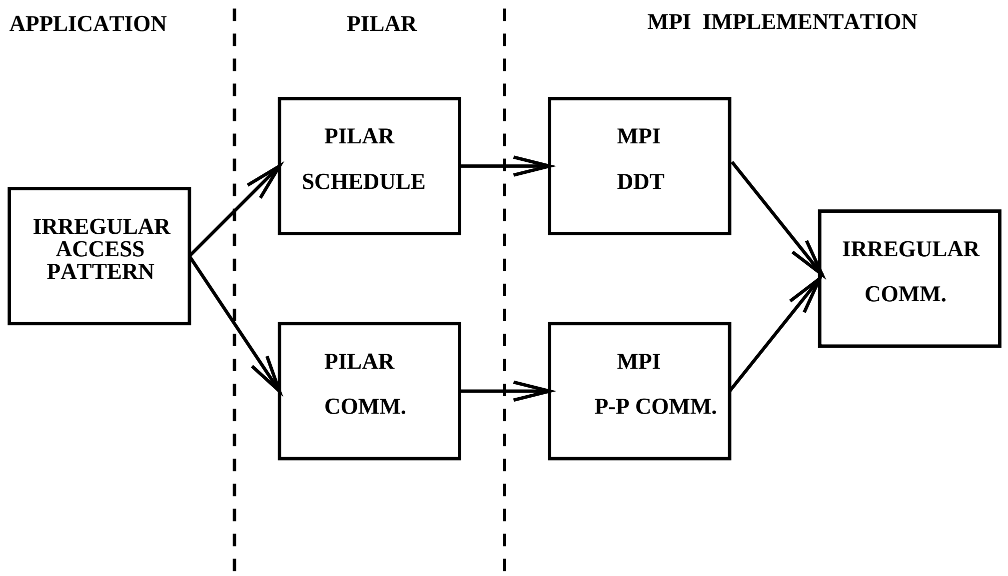

Figure 3.5 PILAR and MPI

Figure 3.5 shows how PILAR uses MPI derived data types: a *Schedule* object summarizes a communication pattern. When PILAR is running under MPI, it instantiates a *Schedule* generating a MPI derived data type for every pair of processors that communicate. The similarities between *MPI_Type_indexed()* and our representations make this translation straightforward. Later, a global exchange in PILAR is implemented using point-to-point MPI communication primitives using these pre-computed derived data types and **no extra buffering** is imposed on the MPI implementation.

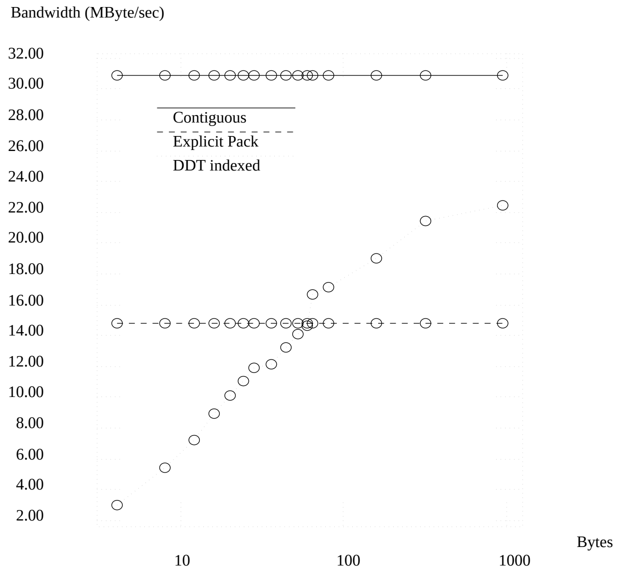

Figure 3.6 MPI derived data types

Figure 3.6 shows that some MPI implementations, in particular IBM MPIF, can actually get advantage of DDT to improve performance. We measured on an IBM-SP1 the round-trip time of sending a large message between two processors by encoding a message of fixed size using multiple *MPI_Type_indexed()* DDT and variying the size of the innermost block in the DDT. We also measured the time when an explicit copy is required and a best case, although unrealistic for irregular applications, where the message is assumed to be contiguous in memory and we do not need a copy. The explicit copy case is the one typically used by an irregular RTS (CHAOS/PARTI). We can see that, provided that the innermost block in the DDT is large enough, we can obtain with this method significantly better bandwidth than using an explicit copy. We also can see that due to the overhead of the DDT implementation, we are better off using an explicit copy when the blocks are small. PILAR actually checks at run-time, when it builds a *Schedule*, that blocks are bigger than a certain threshold, selecting dynamically the best method.

We can also experiment with different mappings of our internal representations to MPI DTT. In particular, when strided accesses are involved, we could use *MPI_Type_vector()* combined with *MPI_Type_indexed()*, providing more information on the nature of the access pattern to the underlying MPI implementation. Unfortunately, preliminary experiments using IBM MPIF did not show performance benefits with this technique, although this result is clearly MPI implementation dependent. The benefit of using PILAR is that it can be parameterized depending on which MPI implementation it is running. Therefore, applications that use PILAR do not need to be explicitly tuned for a particular MPI implementation.

A question that comes to the mind is why not use MPI collective communications for PILAR data exchanges instead of MPI point-to-point. Unfortunately, the most flexible MPI collective primitive *MPI_Alltoallv()* is not flexible enough. *MPI_Alltoallv()* forces the **same** derived data type for all the sends in a particular node and we need **different** data types for every pair of processors.

The second important feature of MPI that PILAR uses is communicators. A communicator defines a scope of communication: a group of processes involved and a unique context that isolates this scope from others. A communicator allows the restriction of global communication to a subset of physical processors, and it also encourages the development of libraries using message passing because interference between message tags is avoided using different contexts. In PILAR a communicator parameter is added to calls that require network access. This allows the integration of PILAR with other MPI libraries or the exploitation of functional parallelism in irregular applications [57].

To summarize, PILAR can be viewed as a tool to simplify the non-trivial task of building efficient MPI derived data types for irregular computations, together with high-level collective communication primitives that are more flexible than the ones supported in the MPI standard.

3.4 Non-Blocking Communication

Irregular RTS like CHAOS/PARTI provides support for gathering and scattering globally indexed data elements of irregularly distributed arrays using pre-computed communication schedules. One shortcoming of these RTS is that gather (scatter) operations are blocking. This means that, in a gather (scatter) operation, a processor has to wait until it has received all its messages. In practice, the majority of the processors are involved in a communication exchange, forcing an implicit barrier synchronization. Blocking forces successful applications using these libraries to have:

- Bursty traffic with low average utilization of the network.
- Good load balance between communication operations.

There are irregular applications that do not necessary follow these requirements. For example, multicoloring sweeps on unstructured grids tend to put more stress on the network than regular grids or explicit methods. Moreover, as it was shown in [42], it is difficult to obtain perfect load balance for each color across partitions. In practice, it is much simpler to obtain good load balance for each sweep across partitions.

It is possible to relax the previous limitations in a simple manner. In PILAR, the functionality of a non-blocking communication primitive is split up in two calls: the first call is similar to a conventional communication primitive but, instead of blocking for incoming messages, it immediately returns a data structure with pending messages identifiers. The second one simply blocks until all the pending messages arrive. Then, it performs the necessary bookkeeping and returns.

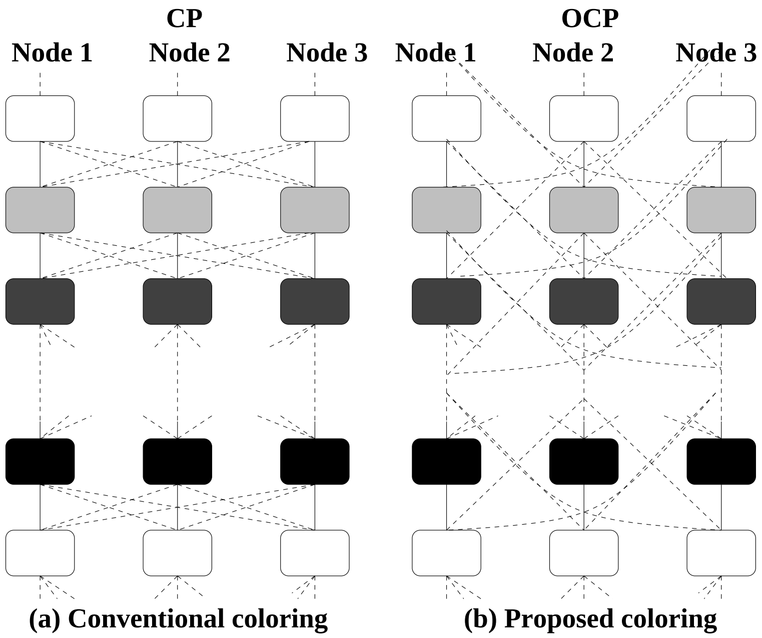

Figure 3.7 Conventional coloring versus coloring for overlapping

Our example is a multicolored sweep over the edges of an unstructured grid. We have two implementations of this sweep as shown in Figure 3.7. The first one assumes a normal coloring and therefore, there is no overlapping. It generates a *Schedule* for every color that is used to gather external references before the update of nodes with a particular color. In the second one, we are assuming an OCP coloring described in [42], otherwise data dependences are not respected. The idea is that the gather operation of a color begins after the computation of another that is two colors ahead ends. Implementation details and performance results on an Intel Paragon are given in [42].

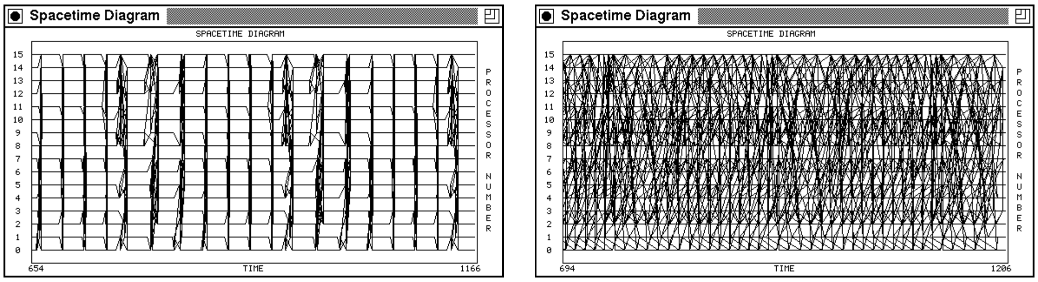

Figure 3.8 Space-time diagram: CP versus OCP on a 16 processor Intel iPSC/2

Figure 3.8 illustrates the different behavior of blocking and non-blocking execution on a 16 processor Intel iPSC/2. Tracing information was obtained using PICL [23, 31] and graphs were obtained using ParaGraph [31]. The graphs represent Space-Time communication diagrams of the executor. The vertical axis is the processor number and the horizontal axis time. Horizontal lines represent processor activity (busy or idle) and slanted lines represent messages between processors. As it was expected, the blocking version of the code (left) enforces an implicit barrier synchronization after each color and communication is not overlapped. On the contrary, the non-blocking code has communication patterns that are more irregular in time, and communication gets overlapped with computation.

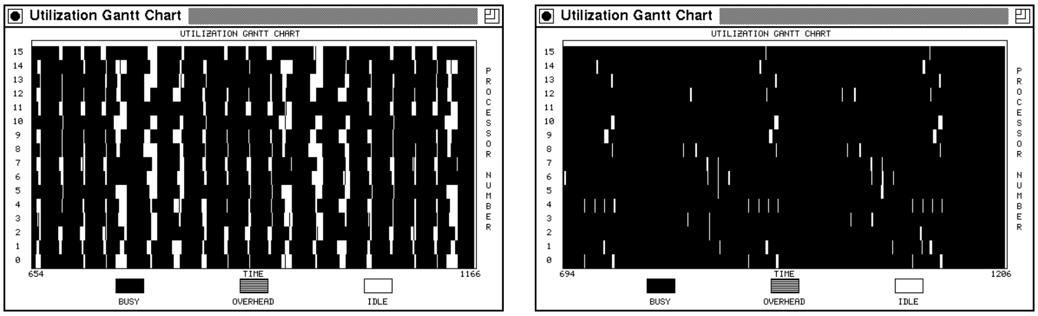

Figure 3.9 Gantt chart: CP versus OCP on a 16 processor Intel iPSC/2

Figure 3.9 is a Gantt chart representing activity of processors in time. White means idle time and black means busy time. We can intuitively see that there is a significant amount of idle time in the blocking code due to synchronization and communication. Non-blocking reduces this idle time using overlapping techniques and looser synchronization.

The effect that we are obtaining is somewhat similar to block pre-fetching [26] in shared memory architectures. The important difference here is that blocks are obtained from *irregularly* distributed data arrays and cannot be determined at compile time. Therefore, we need to combine the ideas of runtime compilation and block pre-fetching.

3.5 PILAR Implementation

PILAR represents a significant implementation effort with roughly 10000 lines of C++ code. The multi-representation nature of the library makes it very amenable to object-oriented techniques but at the same time it has very strong performance constraints. The final purpose of parallel processing is to save time and this is difficult to achieve with slow run-time support. For these reasons the implementation of PILAR is a clear compromise between speed and flexibility. For example, we do not allow hybrid representations in which each individual element might use a different representation. This would imply a virtual method dispatch in almost every innermost loop in the library. Preliminary experimental results showed that, with existing C++ compilers, performance would not be acceptable. Instead, our representations are based on collections of objects with the same type, and we can amortize the cost of a virtual method dispatch over the whole collection.

3.5.1 Communication Pattern Internal Representation (CPIR)

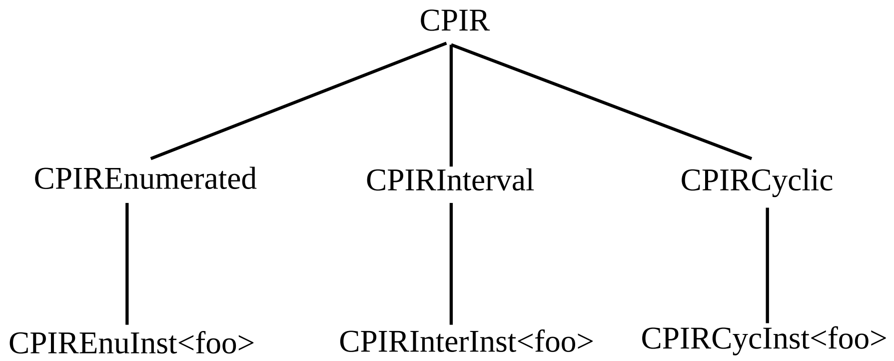

Figure 3.10 CPIR hierarchy

CPIR is a C++ abstract base class that unifies the interface of our representations of communication patterns. Figure 3.10 shows the class hierarchy of these representations. The rest of the library uses pointers to objects of this abstract base class and invokes the correct functions using the C++ dynamic method dispatching. This encapsulation simplifies the addition, debugging or tuning of new representations. We use a template class at the leaf of each representation to implement an instantiation mechanism that will be described in Section 3.5.2.

3.5.1.1 Functionality

Functionality	**Methods**
Marshall for network transmission	pack(), unpack()
Pattern information	getNumPoints(), getArrayNumPoints(), amISorted(), ...
Change representation	convertEnumerated(), convertInterval(), convertCyc()
Combine with other CPIR	Union(), Difference(), Expansion(), Concat(), Intersection(), ...
Instantiate type	Instantiate(), getMPIType()
Mapping arrays	gatherCopy(), scatterCopy(), scatter-Add(), scatterMul(), ...
Search	Align(), Extract(), Find()
Remote search	RemoteExtract(), RemoteFind()

Table 3.1 Functionality of CPIR methods

Table 3.1 gives a flavor of the functionality provided by a CPIR object. This includes functions to pack or unpack itself for network transmission, combine with other CPIR using a set of operations (union, intersection, concatenation, difference ...), map from one array to another using the pattern encoded and a predefined operation (copy, summation, multiplication,...), transform to a different internal representation and so on. The key idea is that these operations are implemented in very different ways depending on the actual CPIR type. In Chapter 5 we will show that this is crucial to improve performance. Also, CPIR objects are almost immutable, this means that most of these operations do not change the objects involved. This fact has greatly simplified the development of PILAR.

When we try to combine two CPIR objects with different internal representations, the object in which the method was invoked will decide the representation of the final object. Then, it will invoke the corresponding conversion method on the other object, obtaining a temporary object of a compatible type. When the operation finishes the temporary object is automatically deallocated. Clearly, there is an overhead in these conversions, and the compiler/programmer should try to minimize them by selecting adequate representations.

3.5.1.2 CPIREnumerated

This representation is targeted toward communication patterns with no spatial regularity and it is very similar to what CHAOS/PARTI provides. Each memory access is represented by an entry in an index array. Operations like union, intersection, and others, are typically implemented by pre-sorting the entries. We minimize the number of times an object is sorted by keeping track of the "already-sorted" property inside the object. When a CPIREnumerated object contains a large number of elements, a hashing based scheme would be preferred because it has better complexity than sorting. In practice, the number of elements at which the hashing scheme wins is large because we use highly tuned sorting routines and we do not require dynamic allocation of memory.

3.5.1.3 CPIRInterval

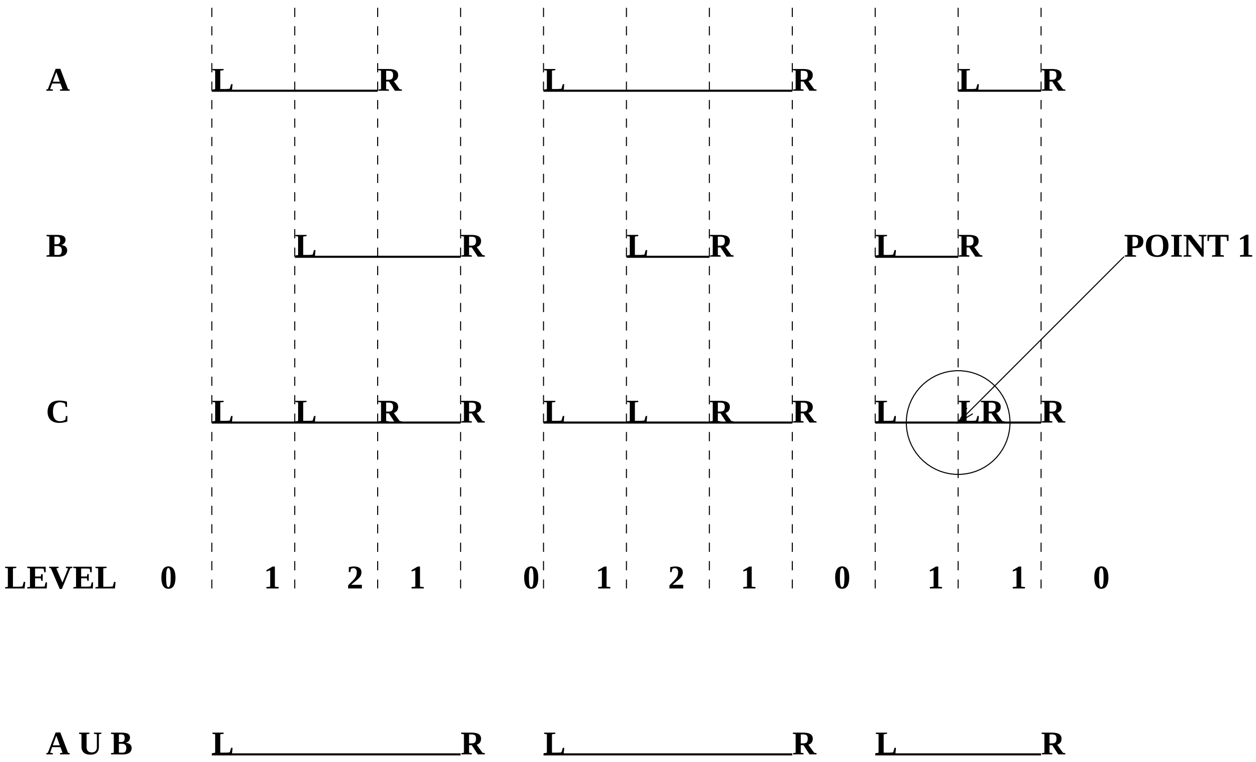

Figure 3.11 Union with intervals example

Collections of intervals are used to represent irregular communication patterns with spatial locality in a compact manner. In particular, we use an array of integers to store the collection, where two integers *start* and $stop + 1$, represent each interval.

```
Union-Intervals-Collection()
A ← collection of intervals
B ← collection of intervals
Tag endpoints of each interval LEFT or RIGHT
C ← Sort together endpoints of A and B keeping tags (increasing)
    ensure in sorting (point, LEFT) < (point, RIGHT)
LEVEL ← 0
for each endpoint P in C
do if TAG(P) = LEFT
      then if LEVEL = 0
              then Record P start of interval
           LEVEL ← LEVEL + 1
      else if LEVEL = 1
              then Record P end of interval
           LEVEL ← LEVEL − 1
```

Figure 3.12 Algorithm for unioning intervals

Efficient computational geometry algorithms [55] are used to perform basic operations like union, intersection and difference of array of intervals. In particular, Figure 3.12 describes a basic algorithm to perform the union of two collection of intervals. Also, Figure 3.11 shows an example of the application of this algorithm. The sweep moves from left to right and our intervals are open to the right. For this reason, we need to find left endpoints before right endpoints in order to avoid "glitches" that would otherwise increase the number of intervals. Point 1 in Figure 3.11 is a clear example of this situation. The structure of algorithms for performing intersection, difference, and others, is similar to the one in Figure 3.12.

3.5.1.4 CPIRCyclic

This representation is targeted toward strided accesses when the stride is a small constant. Those cases are common in Fortran programs 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 benefits from recent work on handling block cyclic distributions in PARADIGM [56]. It has two internal representations: the compressed representation stores a collection of strided intervals, where all share the same stride, as a stride variable and a single CPIRInterval. The expanded representation has as many CPIRInterval objects as the size of the stride and we store all the intervals such that $lower\_bound mod stride = k$ in the kth CPIRInterval object. The benefit of this expanded representation is that it can be shown by a simple application of the GCD test [8] that two strided intervals in different CPIRInterval objects will never collide. Therefore, in order to perform an operation on two expanded CPIRCyclic with the **same** stride, we just simply perform the same CPIRInterval operation to each of the corresponding components.

```
MULTIPLY-STRIDE()
K ← constant stride factor
A ← collection of strided intervals with stride STR
for j ← 0 to K − 1
do for  each strided interval strInt in A
    do if RIGHT(strInt) > LEFT(strInt) + STR ∗ j
          then Add interval (LEFT(strInt) + STR ∗ j, RIGHT(strInt)) to result
Set stride of result to STR ∗ K
return result
```

Figure 3.13 Algorithm for multiplying the stride by a constant K

```
COMBINE-STRIDEDINTERVALS()
A ← Collection of strided intervals in compact form with stride strA
B ← Collection of strided intervals in compact form with stride strB
lcm ← LCM(strA, strB)
newA ← multiplyStride(A, lcm/strA)
newB ← multiplyStride(B, lcm/strB)
expNewA ← expandForm(newA)
expNewB ← expandForm(newB)
for i ← 1 to lcm
do elemA ← get ith element of expNewA
    elemB ← get ith element of expNewB
    elemC ← elemA combined with elemB using CPIRInterval op
    set ith element expResult to elemC
result ← compressForm(expResult)
return result
```

Figure 3.14 Algorithm for combining intervals with stride

When two collections of strided intervals have different stride we cannot directly combine them. We first find the LCM (least common multiple) of both strides and further expand each of them until they have the same stride, using the algorithm shown in Figure 3.13. Figure 3.14 shows a generic algorithm for performing an operation using two collections of strided intervals. Figure 3.15 shows an example when this operation is the union of both collections.

When the stride is large, the expanded representation might have many empty slots. However, the efficiency of this representation could be easily improved by adding a hashing scheme to the

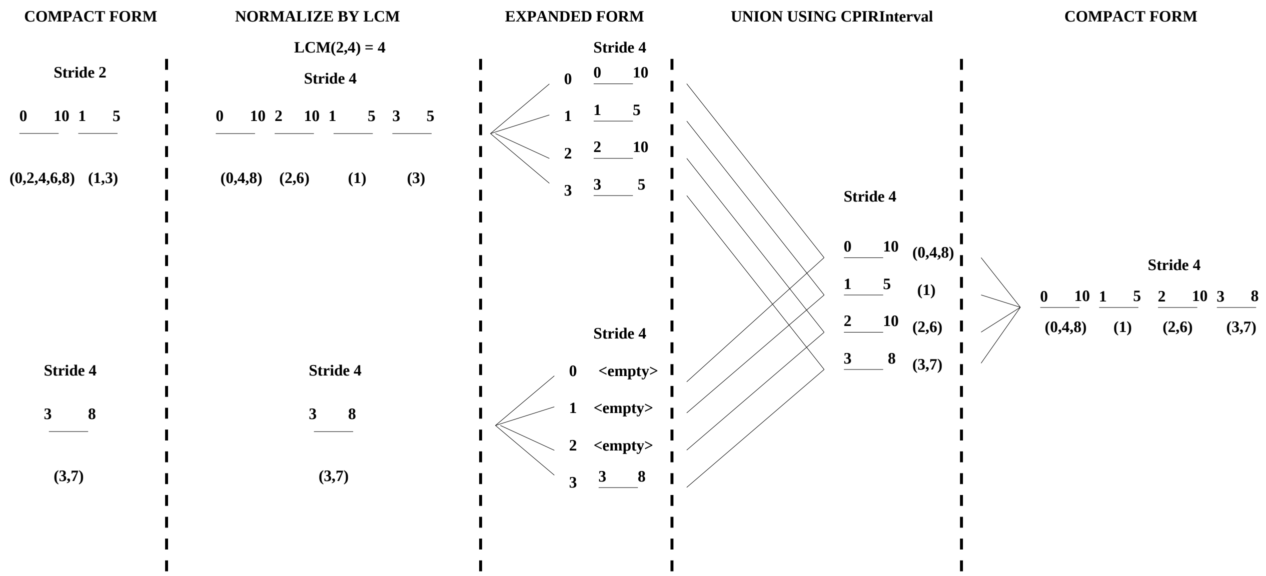

Figure 3.15 Union with strided intervals

expanded representation. In this case we no longer need as many CPIRInterval objects as the size of the stride, but only as many as the non-empty ones.

3.5.2 Schedules

A *Schedule* is an object that represents a communication pattern. Figure 3.16 shows the structure of a Schedule in PILAR, where CPIR objects are used to represent permutations in the local address space. Schedules in PILAR are different from schedules in the CHAOS/PARTI library in that they are fully symmetric. Typical gather or scatter operations assume that one end of the communication is contiguous in memory. In contrast, our communication framework implements directly a more general scheme in which both ends can be non-contiguous [1]. This symmetry has simplified the implementation of the library because only one data exchange primitive needs to be implemented. Also, the special cases of contiguous ends are handled efficiently with our intervals representation.

[1] This generality can be implemented in CHAOS using scatter non contiguous calls at the cost of an extra level of indirection

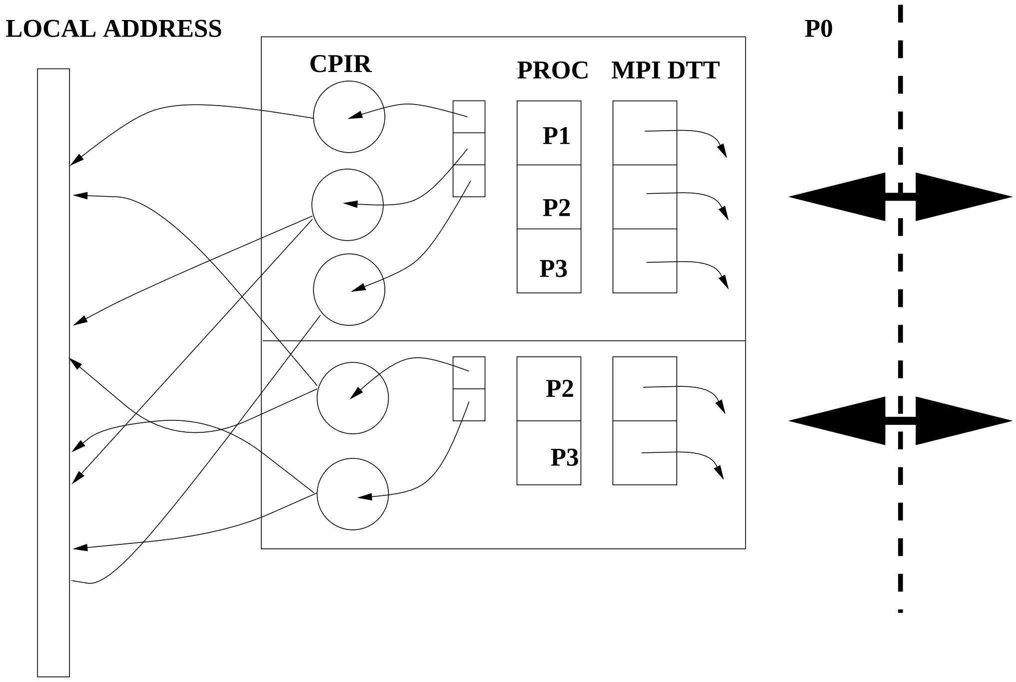

Figure 3.16 PILAR schedule

3.5.2.1 Schedules instantiation

In PILAR we decouple the creation of a Schedule from its instantiation. This is important because we sometimes want to use the same Schedule with different elementary data types. Also, sometimes Schedules are temporary objects used to build more complex ones but never actually needed to communicate. In those cases, decoupling saves the overhead of instantiation.

During instantiation we select the elementary type, generating and committing MPI derived data types. Also, in cases in which we perform explicit buffer copies, we instantiate the CPIR objects contained in the Schedule. To instantiate a CPIR object we just simply return its corresponding template leaf class for that elementary type.

3.5.2.2 Building Schedules from scratch

Similar to *localize* primitives in CHAOS/PARTI, we are also providing a high-level mechanism to automatically build schedules and compute local addresses, given a mapping from global to locals and the collection of global accesses that will be needed in a particular node. We assume that external references will be allocated at the bottom of the communicated array in a contiguous buffer. The only difference between our approach and the one in [18] is that we can also use interval-based representations to describe global/local addresses, mappings and schedules.

3.5.2.3 Composition of Schedules

Operation	**Functionality**
Union	Merge and remove duplicates
Concat	Merge without removing duplicates
Intersection	Find duplicates
Difference	Obtain an incremental schedule
Expansion	Increase size of elementary type
Replication	Replicate the pattern with an offset
Reverse	Perform the inverse pattern

Table 3.2 Operations to combine schedules

In PILAR, schedules that are compatible can be composed to obtain more complex ones using a set of **local** operations. Two schedules are compatible when, in a particular processor, a global index is always mapped to the same local index. When this is the case, comparing local indices tells us if they represent the same value. Table 3.2 summarizes the operations currently supported. In particular, **Expansion** and **Replication** can be used to obtain, respectively, multi-column and multi-row Schedules assuming row major order.

The approach that we take in PILAR is orthogonal to the one in CHAOS. In CHAOS more complex data exchange operations are provided to support multi-schedule or multi-column gather. In PILAR, we get the same effect performing a sequence of union or expansion of schedules and then a simple exchange. If the internal representation of these schedules is compact enough, these operations are performed very efficiently and with no significant increase in memory requirements. The advantage of this approach is that we only have to optimize one simple exchange primitive and the creation of the new schedule gets amortized over all the data exchange calls that use it.

3.5.2.4 Specialized schedules

It is easy to extend the functionality of the basic Schedules through derivation. For example, Distributed Translation Tables (DTT) with intervals can be implemented more efficiently using two new schedules: *ScheduleSort* and *ScheduleVariableSize*. A *ScheduleSort* combines sorting the entries with the creation of the schedule. A *ScheduleVariableSize* is used when the size of the elements that will be exchanged is not known a priori, as it is the case with interval queries in DTT.

3.5.3 Network Interface

A *Network* object encapsulates all the machine dependent code of the library, providing basic access to communication and timing routines. It also provides wrappers to the MPI C interface, defining a MPI communicator.

3.5.4 Global communication primitives

PILAR supports blocking and non-blocking global communication primitives [42] using a pre-computed *Schedule*. Before any data exchange takes place, it is necessary to create a CB (Communication Buffer) object. CB is a template class that will perform a type instantiation of the schedule and, if needed, allocate internal buffers. Also, each CB has associated a communicator that defines 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. Also, they store the state of the pending communication, blocking the process until all the messages arrive.

3.5.5 Translation Tables (TT)

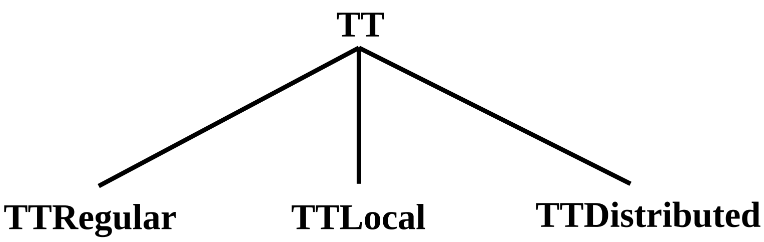

Figure 3.17 Translation table hierarchy

In the same spirit of CHAOS/PARTI TT, PILAR TT maps global indices to local indices and processors. A TT class in PILAR is an abstract base class that defines an interface. Figure 3.17 shows the hierarchy of translation tables derived from it. The advantage of using derivation is that the rest of the library is independent of the actual implementation of the table (distributed, replicated, regular ...) and this allows significant code reuse.

Also, the initialization and query of these tables is done in terms of CPIR. This means that we can query a translation table with a set of intervals of global indices (CPIRInterval) or a set of enumerated global indices (CPIREnumerated). The case when the actual type of the query does not match the internal representation of the table can also be handled with an extra overhead due to internal conversions.

3.5.5.1 Regular Translation Tables

They support regular HPF block, cyclic and block-cyclic mappings with analytical expressions [39]. They can also be trivially extended to handle user-defined analytical mapping functions.

3.5.5.2 Local Translation Tables (LTT)

They are used for tables that need to be only locally consistent or they are replicated in every processor. Figure 3.18 illustrates the implementation of a query to a LTT using intervals. The interval space of globals is sorted and is aligned with the interval space of locals, this means both have the same number of intervals and corresponding intervals have the same number of elements. This alignment is forced when the table is created using the CPIR method *Align*. To perform an interval query, we find the intervals in global space that correspond to the left and right boundaries of the query. This is done efficiently using binary search because globals are sorted. Then, we compute the offsets inside those boundary intervals to further restrict the search. Later, we use the fact that global and local intervals are aligned to extract the local intervals and processor mappings that correspond to the query. Finally, we adjust the intervals of the first and last local interval to reflect the pre-computed offsets.

3.5.5.3 Distributed Translation Tables (DTT)

In some applications, even with an interval-based translation table, this table might be too big to be replicated. In that case we partition the table across processors and we use another table to represent this partitioning, typically using a LTT replicated in each processor, although any other type could be used. Figure 3.19 shows the increase in the complexity of the implementation when intervals are used in queries. We first dereference the replicated TT locally, to find to which processors to forward the query. The problem is that a single interval query could involve several processors, as in the example in Figure 3.19. In this case, we partition the query into sub-queries, forward them to the corresponding processors, and after a dereference in the remote processors, similar to the one explained in the previous section, we assemble locally the received sub-queries results, obtaining the answer.

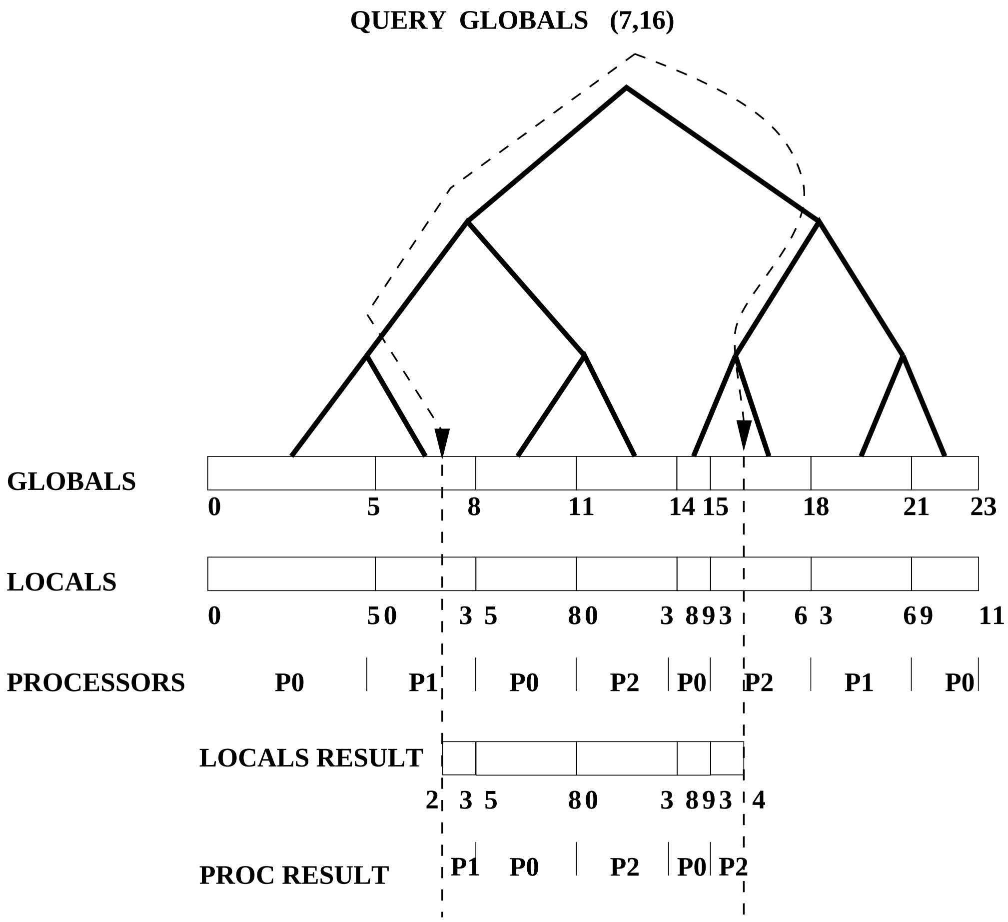

Figure 3.18 Local translation tables with intervals

Clearly, if we have multiple interval queries in one processor we do not want to process each of them at a time because network latency would degrade performance. Instead, we collect all the sub-queries to a particular processor in a single request, ensuring first that we have enough information to assemble the results. In practice, this is performed in two steps, in the first step we build a *ScheduleVariableSize* at the same time that we perform the remote dereferences. In the second step we communicate back the results of the sub-queries using this Schedule, assembling locally the result.

The reason why we need a new type of schedule, *ScheduleVariableSize*, is that the processor requesting a range does not know how many intervals are used to describe that range in the remote processor. This is needed to allocate space locally for external references and requires an extra communication step during the construction of the schedule.

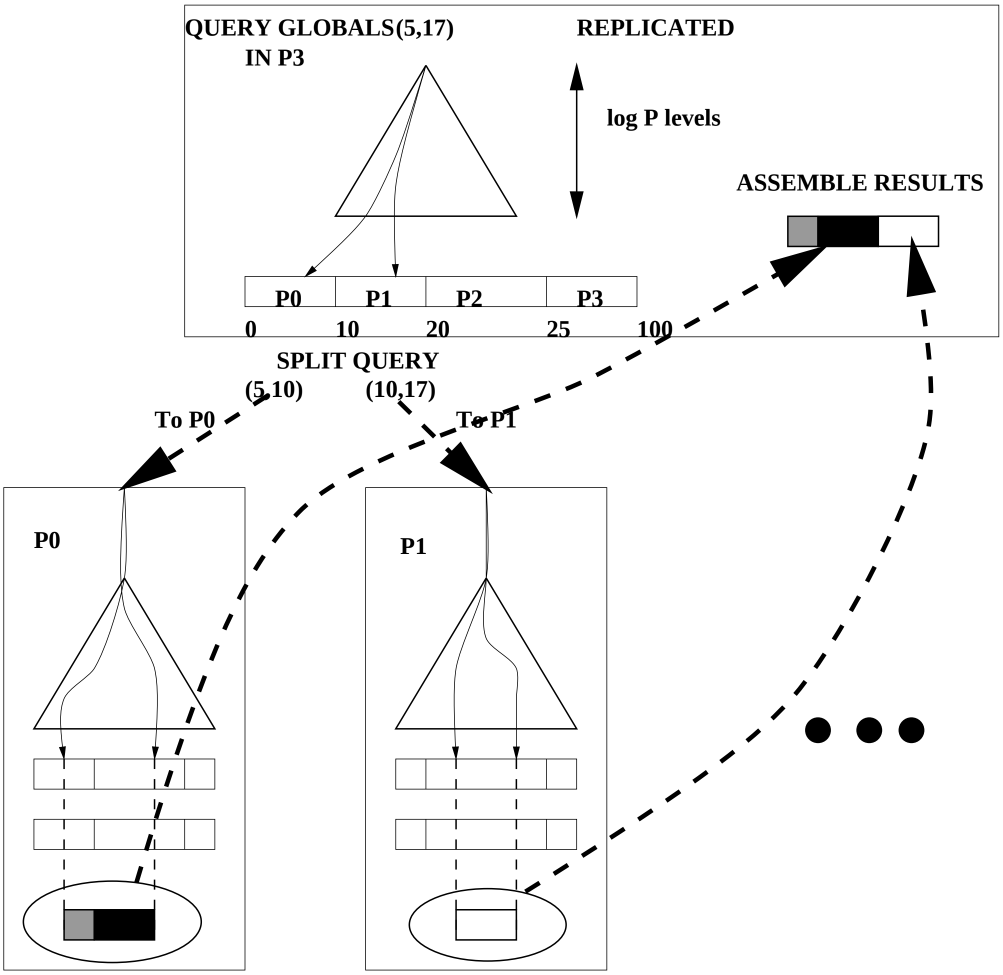

Figure 3.19 Distributed translation tables with intervals

3.6 PILAR Interface

PILAR provides two interfaces, one is object-oriented and is meant to be used from C++ programs, typically manually parallelized. The other one, used by our compiler, PARADIGM (see Chapter 4), is a Fortran interface implemented as a set of wrappers to the object-oriented interface.

Figure 3.20 shows a simple example of the C++ interface usage. The code gathers two ranges of elements using a schedule. We first initialize the network interface (MPI), then we create a TT with a regular mapping. Later, we construct an interval query in globals of possibly remote elements and build a schedule for it. Using this schedule and the network interface we construct a communication buffer of integers (the type of the array to be communicated). Finally, we communicate the array elements using this CB and shutdown the communication interface.

```
int
main(int argc, char *argv[])
{
  // Initialize MPI, construct Network Interface
  Network * Net = new Network(argc,argv);
  int foo[100];

  // Describe a regular block mapping
  TTRegular TT(100,Net->Size(),BLOCK);

  Interval value [] = {Interval(1,10),Interval(20,40)}
  CPIRInterval(value,2);

  // Build Schedule using global addresses
  Schedule * SC = BuildSchedule(Net,&value,&TT);

  // Build Communication buffer of type int
  CommBuffer <int,INT> cB(Net,SC);

  // communicate data modifying elements foo(1..10) and foo(20..40)

  cB.XChange(foo,foo)

  // Finish MPI, destroy Network Interface
  delete Net;
}
```

Figure 3.20 Example of PILAR C++ interface

Examples of the Fortran interface are shown in the appendix. This interface uses Fortran integer variables to encode pointers to C++ objects. It also provides some support for dynamic allocation of arrays from Fortran.

3.7 Summary of Contributions

A new run-time support library for irregular computations called PILAR has been developed. PILAR has the following unique features:

- Multiple representations of communication patterns to exploit regularity in irregular accesses.
- Full integration with MPI, taking advantage of communicators and derived data types.

- Non-blocking irregular communication.
- Local combining of Schedules to reduce pre-processing overhead.
- Interval-based Distributed Translation Tables.
- Implementation complexity managed by an object-oriented design.

Chapter 4

COMPILER SUPPORT FOR IRREGULAR COMPUTATIONS

4.1 Introduction

PILAR provides a relatively high-level and machine independent programming interface that greatly simplifies the efficient parallelization of irregular applications on a distributed-memory multi-computer. Unfortunately, the task of generating inspectors and executors with embedded PILAR primitives is still involved for non-trivial programs. There are several reasons to automate this task:

- Inspectors quickly become complex. This is particularly true when multiple levels of non-locality and different representations of access patterns are used.
- Different representations of access patterns lead to radically different inspectors and executors. The effort involved in code rewriting is a clear deterrent for the programmer to experiment with new features that PILAR provides.
- Keeping consistent versions of inspector and executor code is a maintenance nightmare.
- Readability of the final code is also reduced by complex array subscripts, added counters, modified loop structure and a significant code expansion.

In this chapter we will describe how to automate some of the tasks involved in the parallelization of an irregular application on a distributed address space. In particular we will investigate:

- Detection and placement of irregular communication.
- Feasibility check and placement of inspectors.
- Selection of internal representations for irregular access patterns.
- Selection of ad-hoc types of inspectors that can reduce the inspector and/or executor times.
- Generation of inspector code that may involve multiple levels of non-locality
- Transformation of array subscripts from a global into a local address space

Unfortunately, some tasks are not automated yet:

- Identifying loops as parallel when this cannot be detected with static analysis.
- Selection of an adequate data and loop iterations partitioning strategy.

The feasibility of extending this framework to automate these tasks will be discussed in Chapter 6.

4.2 Overview

4.2.1 PARADIGM

The PARADIGM (PARAllelizing compiler for DIstributed memory General-purpose Multicomputers) project at the University of Illinois provides an automated means to parallelize programs, written in a serial programming model, for efficient execution on distributed-memory multicomputers. The compiler accepts either a sequential FORTRAN 77 or HPF program and produces a parallel program optimized for a target machine.

PARADIGM uses *Parafrase-2* [51] as a preprocessing platform to parse the sequential program into an intermediate representation, to analyze the code and generate flow, dependence, and call graphs, and to perform transformations such as constant propagation and induction variable elimination.

In addition to the traditional compiler optimizations to distribute computations and to reduce communication overheads, PARADIGM's capabilities include:

- Optionally parse High Performance Fortran (HPF) directives [39].
- Perform automatic data distribution for regular computations, conventionally only specified through user directives.
- Generate high-level communication primitives.
- Optimize communication for regular computations.
- Exploit functional and data parallelism simultaneously.
- Generate multi-threaded message-driven code to tolerate communication latencies.

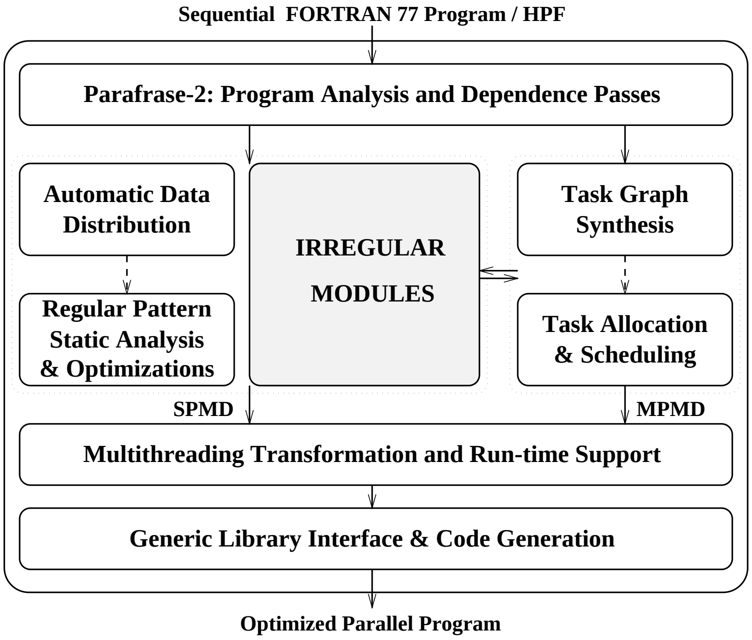

Figure 4.1 PARADIGM compiler overview

In this dissertation we describe an extension to PARADIGM to provide support for irregular computations using a combination of compile-time analysis and run-time support.

Figure 4.1 shows the different components of PARADIGM, high-lighting how the new irregular modules described in this chapter fit in the general framework of PARADIGM.

4.2.2 Irregular modules

Figure 4.2 shows the new irregular compiler passes. A brief description of these passes follows:

Normalization Performs traditional loop normalization and a new index normalization that simplifies irregular subscripts.

Basic irregular support

- Detection of irregularity produced by either the data distribution or the nature of the array access pattern.
- Placement of irregular communication to maximize message aggregation
- Feasibility check of inspector extraction
- Code slicing

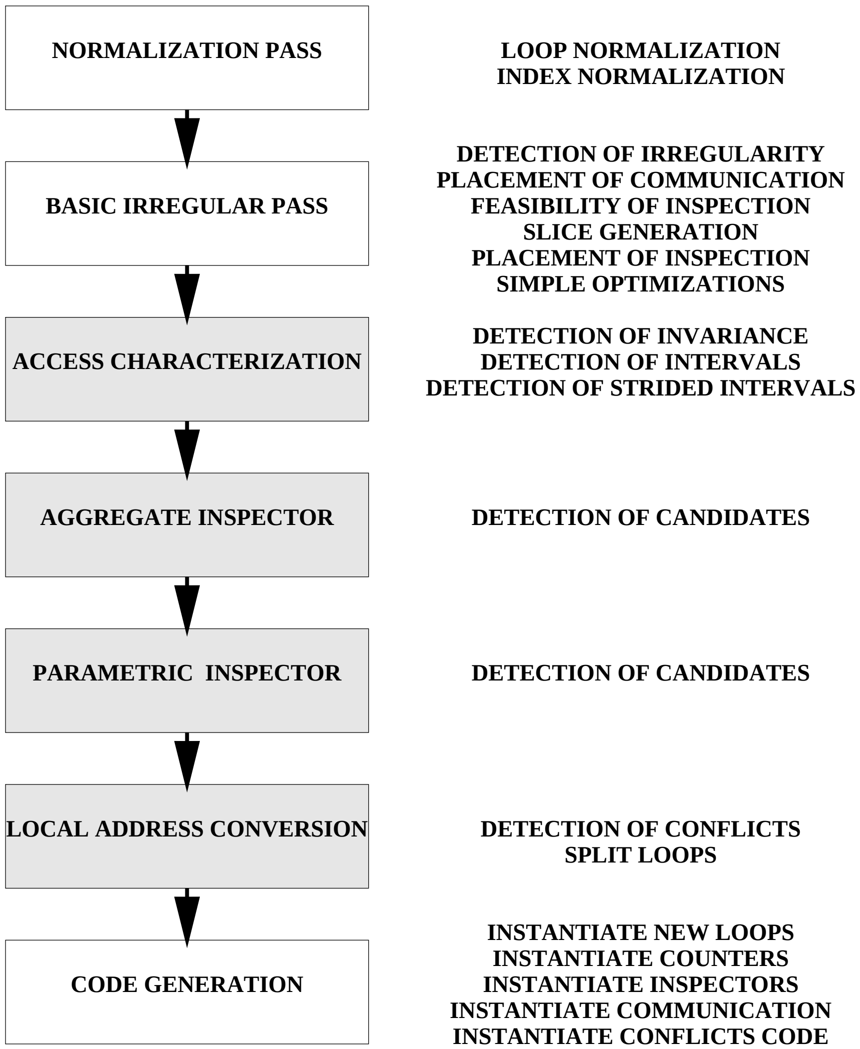

Figure 4.2 Irregular passes

- Placement of inspectors to maximize reuse
- Redundant communication and/or inspection elimination

Access characterization Exploit properties of the array access pattern in order to select PILAR internal representations.

Aggregate inspectors Further optimize communication by taking advantage of multiple references to the same array.

Parametric inspectors Further reuse of inspectors by realizing the multi-phase nature of certain applications.

Local address conversion Transforming array subscripts from global to local addresses, even in the presence of interval based representations.

Code generation Generation of communication and inspection code.

The modularity of our design allows for the optional bypassing of certain passes (shadowed boxes in Figure 4.2). This gives the freedom to, i.e., generate code using local or global addresses and/or force all accesses to be enumerated.

4.2.3 Limitations of the irregular passes

Two major limitations of this work are: first, that we use slicing techniques that are restricted to structured code [66] [1] and second, that we are not contemplating extra levels of non-locality indirection caused by conditionals. Moreover, decisions on where to place communication and inspection are loop based, using data dependence analysis, and that restricts the optimizations possible in the presence of conditionals. These simplifications are key to reduce the complexity of the compiler and it is unclear whether more complex analysis techniques [70] would pay-off in terms of performance of real applications.

4.3 Code Normalizations

The normalization of loop bounds and array subscripts is crucial to reduce the complexity of the rest of the compiler. The implementation of this pass was greatly simplified by the use of *Mathematica*. A description of this implementation is given in Section 4.10.2.

4.3.1 Loop normalization

Figure 4.3 shows a well-known algorithm to perform loop normalization. Loop normalization is useful in the context of irregular compilation to expose indirect accesses hidden by the loop bounds. For example, in the sparse CSR matrix- dense vector product code shown in Figure 4.4, considering only the subscript index expressions, it is not clear that arrays a and ja are irregularly accessed and array x is accessed with two-levels of indirection. After loop normalization shown in Figure 4.5 this fact becomes clear.

[1] we are not aware of any work that uses a more general slicing technique for the generation of irregular inspectors

```
Loop Normalization()
for each loop lp in pre − order
do IVAR ← loop variable of lp
    BODY ← statements in the body of lp
    loop header in lp is DO I = lower, upper, step
    newUpper ← Simplified((upper − lower + step)/step)
    Replace loop header in lp by DO newI = 1, newUpper
    Replace in BODY IVAR by lower + (newI − 1) ∗ step
    Simplify BODY
    if IVAR is used outside lp
       then Insert "IVAR = lower + (newI − 1) ∗ step" after lp
```

Figure 4.3 Algorithm for loop normalization

```
do i = 1, n
  t = 0.0d0
  do k=ia(i), ia(i+1) -1
    t = t + a(k)*x(ja(k))
  enddo
  y(i) = t
enddo
```

Figure 4.4 Sparse CSR matrix - dense vector product

4.3.2 Index normalization

Unfortunately, a side effect of loop normalization is that it can complicate other necessary tasks, i.e., slicing, by increasing the complexity of array subscript expressions. The index normalization pass transforms every irregular subscript of the form $A(f(\ldots))$, where $f(\ldots)$ is a non-linear expression in the enclosing loop variables, to the simplified form $A(ITEMP)$ where $ITEMP$ is an auxiliary integer variable. Moreover, during the normalization process we also perform common sub-expression elimination at the statement level. This means that, if two array references share

```
do i = 1, n
  t = 0.0d0
  do knew=1, ia(i+1) -ia(i)
    t = t + a(knew+ia(i)-1)*x(ja(knew+ia(i)-1))
  enddo
  y(i) = t
enddo
```

Figure 4.5 Loop normalization: sparse CSR matrix - dense vector product

```
IndexNormalization()
for each statement S
do AUXIVAR = EMPTY
    IVAR = loop variables of loops enclosing S
    for each irregular array ref A(b(...)) in S (post − order)
    do repeat
               subB ← subexpression in b(...) of the form C(d(...))
                 where d(...) is linear in IVAR ⋃ AUXIVAR
               if subB ≠ NIL
                 then Declare ITEMP a new unique integer variable
                        Insert statement "ITEMP = subB" before S
                        Replace in A(b(...)) subB by ITEMP
                        Replace in S subB by ITEMP
                        Add ITEMP to AUXIVAR
          until (subB = NIL)
        if b(...) is NOT a variable in AUXIVAR
          then Declare ITEMP a new unique integer variable
               Insert statement "ITEMP = b(...)" before S
               Replace in A(b(...)) b(...) by ITEMP
               Replace in S b(...) by ITEMP
               Add ITEMP to AUXIVAR
```

Figure 4.6 Algorithm for index normalization

a subscript expression, they will also share the same auxiliary variable $ITEMP$. The importance of this will become clear in Section 4.4.7, where the elimination of redundant communication and inspection is addressed.

An algorithm to perform index normalization is shown in Figure 4.6. After applying this algorithm to the loop normalized example in Figure 4.5, we obtain the code shown in Figure 4.7. Note that arrays ja and a share the same index variable $itemp1$ and all the irregular subscripts have been normalized.

4.4 Basic Irregular Compilation Support

In order to compile an irregular application we first decide whether irregular communication is needed and at what loop level this communication can be aggregated. After that, in certain cases we can compute an inspector that will mimic and record the irregular access pattern. Moreover, we can determine the outermost loop at which we can lift the computation of the inspector, eliminating

```
do i = 1, n
  t = 0.0d0
  do knew=1, ia(i+1) -ia(i)
    itemp0= ia(i)
    itemp1 = knew+ itemp0-1
    itemp2 = ja(itemp1)
    t = t + a(itemp1)*x(itemp2)
  enddo
  y(i) = t
enddo
```

Figure 4.7 Loop and index normalization: sparse CSR matrix - dense vector product

unnecessary re-inspections. In certain cases several array references can share the same inspector, further reducing the cost of inspection.

The analysis that allows to perform all these optimizations is loop based and uses array data dependence. This means that we cannot place communication and inspection in arbitrary points in the program, but only at the header, or just after, a loop. Unfortunately, this limits the applicability of certain optimizations, in particular when the application has complex control flow.

Nevertheless, there are certain irregular problems in which array data dependence can be more useful than a data-flow framework that treats the whole array as a single variable. For example:

- When the irregularity of the array reference comes from the data distribution of the array and not from the array subscripts (i.e., stencil-based computations with irregular boundaries).
- When multi-dimensional arrays are only accessed irregularly in some dimensions but not others.
- When most of the array references contained in the inspector slice of an irregular reference are regular.

Previous work focussed on data flow frameworks to either place irregular communication [70] or generate inspectors with multiple levels of indirection [18]. Unfortunately, these studies treat communication and inspection placement as two independent problems, and the issues of a non-trivial integration are not discussed. Our approach is to consider these problems together but in a more simplified framework. In particular, we use the irregular communication placement information to further reduce the scope of inspection. This is important because we may be able to reduce memory overhead and further improve opportunities to re-use inspection.

In Chapter 5 we will show experimentally the effectiveness of this analysis. It remains to be seen whether the complexity of a data-flow analysis framework pays-off in terms of performance of real applications.

4.4.1 Detection of irregularity

Before the irregular passes we typically run an induction variable elimination pass provided by Parafrase-2 [51] that greatly simplify array accesses. After this pass, we assume during index normalization that any array subscript expression that is not an affine expression of the enclosing loop variables is irregular. Clearly, it is trivial to identify irregular references after index normalization by comparing subscript expressions with temporary variables.

```
  real a(1000),b(1000), c(2000)
  integer TTab, TTc
!dir$ irregular a,b with TTab
!dir$ irregular c with TTc
  do i = 1, 1000
    c(i) = a(i) +b(i)
  enddo
```

Figure 4.8 Irregular data distribution with regular accesses

In certain cases we want to treat array references with regular subscript expressions as irregular ones. Using a directive, as shown in Figure 4.8, we can force the compiler to do so. For example, in Figure 4.8 we declare irregular distributions for arrays a and b as using the abstract irregular distribution TTab and for array c using TTc. TTab and TTc are integer variables that encode pointers to PILAR internal distribution objects. Later, the compiler generates irregular communication and inspection for all the references to arrays a, b and c. It also takes advantage of the fact that a and b are equally distributed for further optimization. Unfortunately, the generation of code to create PILAR distribution objects has not been automated, but in our experience this was not a major drawback.

4.4.2 Placement of irregular communication

An algorithm for placing regular communication was proposed by Gupta [28]. Here we use a simplified version of this algorithm that does not contemplate loop transformations, such as loop interchange or loop distribution, that could change the communication point of an array reference. 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. In the context of irregular accesses, we can conservatively apply this algorithm assuming that any common nesting loop could carry the dependence.

In the presence of irregular references, the static analysis for placing communication might become too conservative. For those cases we provide a compiler directive to lift communication of references to an array over a particular loop.

In some cases we need to override the owners-compute rule. From a communication point of view, this implies the need to "write-back" certain array elements after being modified. We can obtain the level of that communication by reversing the direction of flow-dependences and using the previous algorithm. Then, we have to check that no output or anti dependences further restrict this communication level. Finally, we insert communication at the bottom, instead of at the top, of the particular loop.

4.4.3 Slicing techniques

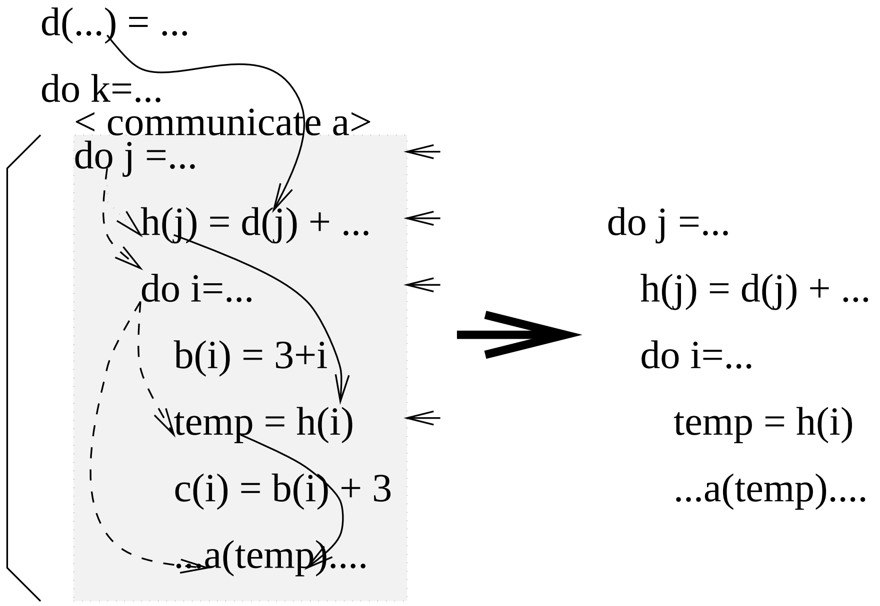

Figure 4.9 Slicing example

A program slice consist of the parts of a program that could potentially affect the value computed at a particular point in the program [66]. In the context of irregular compilation, slices are computed

```
SLICING()
Q ← NIL
S ← statement that contains A(INDEX)
for each data dependence DEP into S
do if (DEP is flow dependence)AND(DEP is due to variable INDEX)
      then S2 ← source statement of DEP
            if (S2 is inside comm. region)AND(S2 not already visited)
               then Add S2 to Q
                     Mark S2 as visited
for each control dependence DEP into S
do S2 ← source statement of DEP
    if (S2 is inside comm. region)AND(S2 not already visited)
      then Add S2 to Q
            Mark S2 as visited
while Q ≠ NIL
       do Pick Statement S from Q
           for each control or flow dependence DEP into S
           do S2 ← source statement of DEP
               if (S2 is inside comm. region)AND(S2 not already visited)
                  then Add S2 to Q
                        Mark S2 as visited
Dead − code non − visited statements
```

Figure 4.10 Algorithm for slice generation

to regenerate in the inspector the sequence of accesses to a particular array. In our case, we further restrict slices to the scope defined by the level of communication.

There are many techniques for program slicing. We have chosen a simple scheme based on Ottenstein and Ottenstein [49] that maps slicing to a reachability problem in a program dependence graph (PDG). The PDG is a directed graph with vertices corresponding to statements or conditions and edges corresponding to control and data dependences between them. Figure 4.10 shows an algorithm to compute the slice for a particular array access $A(INDEX)$ restricted by the communication level of array A. Figure 4.9 shows the application of this algorithm to a simple example.

4.4.4 Feasibility of inspection

A necessary condition to extract an inspector is that it is possible to decouple the generation of the addresses from the actual computation. For instance, in the example shown in Figure 4.11

```
do i = 1, n
S1  ii = ix(ia(i-1))
S2  ia(i) = ii
enddo
```

Figure 4.11 Example of infeasible inspection

```
do i = 1, n
S1  ii = ix(ia(i))
S2  ia(i) = ii
enddo
```

Figure 4.12 Example of feasible inspection using data dependence

there is a flow dependence cycle between the addresses $ia(i-1)$ and the actual array values ix. This dependence cycle makes infeasible inspector extraction.

A simple method to verify the feasibility of inspection is to use the algorithm in Figure 4.10 and then check that the statement that contains the sliced array reference has not been visited (does not belong to the resulting slice).

The example in Figure 4.12 shows the benefit of using a data dependence versus a data flow framework. In this example we are always using old values of array ia to compute the index into array ix. Data dependence analysis would detect this, asserting that there is no flow dependence from S2 to S1 due to array ia, therefore inspection is feasible. However, using a data flow framework we have to conservatively assume a similar case to the example in Figure 4.11, where inspection is infeasible.

In the way we have defined inspection feasibility, depending on the communication point, inspection of an array reference may or may not be feasible. Intuitively, this reflects that there is a trade-off between message aggregation and the difficulty of inspection.

4.4.5 Placement of inspectors

Even if inspection is feasible, it is usually of little practical use unless it can be re-used several times. The problem of reusing inspectors is related to loop invariant computation detection but less strict. Inspector reuse requires that a sequence of computations is loop invariant, even though each of these computations in isolation might not be.

Eswar et al [20] proposed an iterative data flow algorithm to extract inspectors. The main idea of this algorithm is that if a variable has reaching definitions from inside the region that defines the candidate inspector **and** from outside, its value might change across inspector executions,

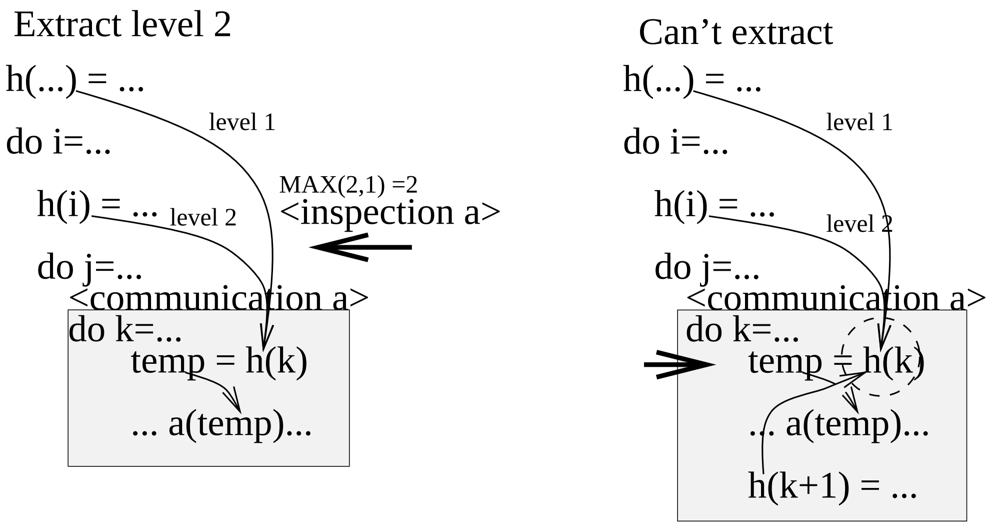

Figure 4.13 Placement of inspectors example

therefore we cannot reuse inspection. This algorithm first computes this "variance" property for all the variables in the region. Then, it propagates this property iteratively using def-use chains and control dependences, tagging a node as variant if **any** incoming value is variant. The process stops when no new variables are tagged as variant in a complete iteration of the algorithm.

We have modified the original algorithm by Eswar et al in several ways. We eliminated the need of iteration by using a pre-computed slice. We used data dependence instead of def-use chains that further increase accuracy when regular array accesses are part of the slice. Moreover, if the inspector can be re-used across multiple loops, the original algorithm would need to be applied several times while one application of ours suffice. Also, we are not allowing loop carried dependences inside the slice that could otherwise complicate the parallelization of the inspector or its reuse. Figure 4.14 shows the modified algorithm and Figure 4.13 an example of its use.

4.4.6 Multiple levels of non-locality

In many real codes more than one level of non-local indirection need to be resolved before the actual computation can proceed. In those cases the code slices computed by algorithm in Figure 4.10 may contain irregular references that require communication.

Ruhl [60] was the first to deal with multiple-levels of indirection in irregular codes. In his approach he first implicitly forms a graph with nodes representing the array references needing communication and edges denoting the control and data dependences between them. Then, he levelizes this graph assigning a depth number to each node, where a smaller number means it is

```
InspectorPlacement()
Compute slice SL restricted by level L
inspLevel ← 0
for each statement S in SL
do IN = NIL; OUT = NIL
    for each flow dependence DEP into S
    do S2 ← source statement of DEP
        VAR ← variable of the dependence
        if S2 is in SL
          then Add VAR to IN
                if (DEP is loop carried)OR(VAR is in OUT)
                  then error "Cannot extract inspector"
          else  Add VAR to OUT
                if VAR is in IN
                  then error "Cannot extract inspector"
                  else  inspLevel ← MAX(nesting level of DEP, inspLevel)
```

Figure 4.14 Algorithm for placement of inspectors

at a lower level in the graph. Later, he generates as many slices as levels in the graph, each of them containing all references with a particular depth number or smaller. Finally, he orders slices by their greatest depth number and inserts communication between them. Recently, a variation of this scheme was proposed in [48] that could produce more general slices dynamically.

Das [18] also proposed a method that eliminates multiple levels of non-locality by transforming the initial code into one that has at most one level of array indirection. A code slice is computed for every irregular reference that needs pre-processing. Then, a slice graph is computed with a node for every slice and an edge for every dependence between slices. Finally, during instantiation he orders the slices using a topological sort of the slice graph and adds communication to preserve the dependences.

One difference between Das's and Ruhl's methods is that the latter does not eliminate array indirection levels, only non-locality levels. In cases with low subscript reuse, eliminating array indirection levels could actually improve single node performance. On the other hand, with high subscript reuse, memory requirements would force the use of hash tables in order to eliminate array indirection, degrading sequential performance. In those cases it is sometimes better to replicate the global address space in each processor and use global addresses with multiple levels of array indirection. This technique would also be better for issuing irregular memory pre-fetches on a shared-memory architecture.

Another difference between both methods is that Ruhl's method increases the granularity of slices by aggregating all the references at the same level. This aggregation could limit the re-use of inspectors in some cases. On the other hand, Das's method generates fine-grained slices and relies on loop fusion to increase granularity, potentially having a greater control over it.

Our approach is closer to Das's method in that a code slice is generated for each reference and we also rely on topological sort to ensure that dependences are preserved. On the other hand, slices are restricted by communication points and we do not stop slicing when we find another irregular access. An implication of this is that we can generate code with multiple levels of indirection (similar to Ruhl's method). In practice we do not explicitly form a slice graph, we simply perform a depth-first traversal of the slices and when we find an irregular access inside one slice, we recursively start traversing the slice of that irregular access. During this traversal we number the nodes in topological order and, using a standard algorithm [17], we check that no dependence cycles involving non-locality accesses are present. Note that these cycles can always be broken by replicating some of the arrays involved.

Similarly to Das's method, we can also generate code with at most one level of indirection by running a simple dead code elimination pass after the conversion to local addresses pass.

4.4.7 Eliminating redundant communication and inspection

The current implementation has a rather simplistic approach to eliminate redundant inspection. We use the information given by the common expression elimination during index normalization to identify equivalent subscripts in a given statement. Then, we ensure that their data distribution is consistent and they share the same communication level. Finally, we eliminate one of the inspectors, updating the communication primitives that use them.

Similarly, to eliminate redundant communication we ensure that both communication primitives will be placed at the same loop level, they share the inspector and they will communicate elements from the same array.

This approach can only remove redundancy from irregular references in the same statement, but its extension to the scope of a basic block is straightforward. However, the generalization to eliminate redundancy from irregular references in arbitrary points in the program is complex and it may require a global value numbering scheme as in [70].

4.5 Access Characterization

One of the most important features of PILAR is its ability to have a variety of internal representations of communication patterns. We will show experimentally in Chapter 5 that, selecting the most adequate representation for a particular array reference, it is possible to improve performance and reduce memory overhead. However, this selection can sometimes be involved. In order to simplify this task we have implemented a few heuristics in an access characterization pass. These heuristics rely on the normalization passes, to simplify their complexity and increase their effectiveness. Also, they use a combination of dependence analysis and syntactic pattern matching as shown in Figure 4.15. Also, Figure 4.16 shows the application of this heuristic to the reference $x(itemp2)$ in the example shown in Figure 4.7.

```
AccessCharacterization()
LP ← innermost loop enclosing access A(INDEX)
IVAR ← loop variable of LP
Compute slice SL for access A(INDEX) restricted to LP
if SL contains cross iteration dependences
  then return ENUMERATED
if SL is NOT a sequence of assignments
  then return ENUMERATED
for each assignment S_i in SL (execution order)
do LHS ← left hand side of S_i
   RHS ← right hand side of S_i
   Replace LHS by RHS in assignments S_{i+1} ... S_N ∪ INDEX
if IVAR not in INDEX
  then return INVARIANT
if IVAR not in Simplify(INDEX − IVAR)
  then return INTERVAL
TARGET ← Find substring of the form "constant * IVAR" in INDEX
if IVAR not in Simplify(INDEX − TARGET)
  then return STRIDED with constant
return ENUMERATED
```

Figure 4.15 Heuristics for access characterization

After this analysis is performed, the selection of internal representation is trivial:

ENUMERATED Default case. Use CPIREnumerated in PILAR.

INVARIANT The array access is constant in the innermost loop. We can eliminate the innermost loop during inspection. Use CPIREnumerated for the other loops.

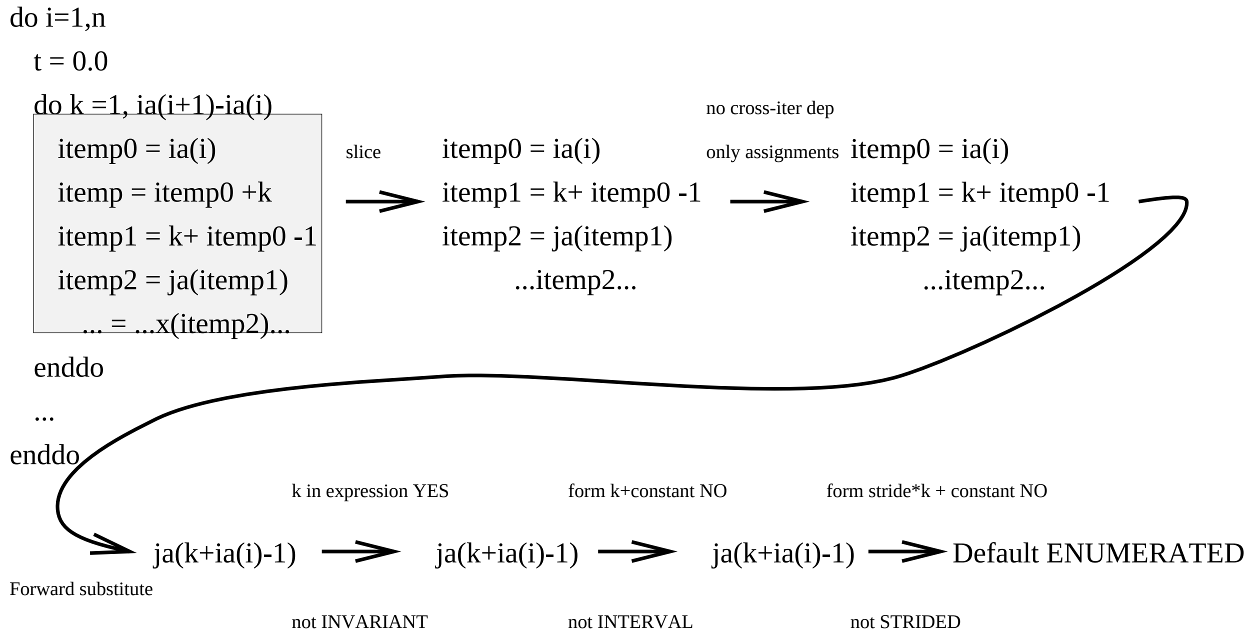

Figure 4.16 Example of the application of the access characterization heuristics

INTERVAL Array accesses in the innermost loop are contiguous in memory. Use CPIRInterval in PILAR.

STRIDED with K Array accesses in the innermost loop have a constant stride. Use CPIRCyclic in PILAR.

Even though the heuristics are very simple, they are powerful enough in most cases as it will be shown experimentally in Chapter 5. Moreover, their implementation was greatly simplified by the built-in pattern matching functionality in *Mathematica*.

Our characterization pass assumes that the original code was written in such a way that the pattern is explicit in the program. Sometimes this is not the case, and the pattern occurs in the sparse matrix but it is not explicit in the code. An interesting and somewhat complementary approach was taken by Knijnenburg et al [38] in the context of improving data locality in sparse computations. Their method involved a combination of run-time search of patterns in the sparse matrix, suggested by user directives, and multi-version code for these patterns automatically generated by the compiler. Our compiler could be applied to this sequential, multi-version code generating parallel code for a distributed-memory multi-computer.

Another limitation of our current approach is that we are only considering the innermost loop to identify a pattern. Unfortunately, this cannot take advantage of patterns that span several loops, and this could be important to achieve performance in certain codes. Also, the current implementation of the code generation pass can only get advantage of intervals.

4.6 Aggregate Inspectors

```
DO i =1,100
  DO i =1,100
    b(i,j) = (a(i-1,j)+a(i+1,j)+a(i,j+1)+a(i,j-1))/4.0
  ENDDO
ENDDO
```

Figure 4.17 Stencil computation

In many applications it is crucial to eliminate redundant communication across multiple array references to the same array. For example, in the stencil computation in Figure 4.17, four times as much communication is needed unless we can eliminate redundancy across multiple references to array a. Moreover, in the context of irregular communication, we typically allocate one buffer for external references per array variable and we need to have a consistent use of this buffer across multiple references, otherwise we could overwrite useful data.

```
DetectingAggregation()
for each loop LP
do AGGR ← NIL
    for each irregular ref IR with communication at loop LP
    do FOUND ← FALSE
        for each aggregate AG in AGGR
        do if (array variable of AG = array variable of IR) AND
                (inspection level of AG = inspection level of IR) AND
                (inspector type of AG = inspection type of IR) AND
                (IR not contained in any slice in AG)
                then Add IR to AG
                     FOUND ← TRUE
                     BREAK
        if NOT(FOUND)
          then Create new aggregate NEWAG containing IR
               Add NEWAG to AGGR
    Delete from AGGR aggregates with one element
    Aggregate inspectors in AGGR
```

Figure 4.18 Detecting candidates for aggregation

Aggregation can be achieved in two ways in PILAR: either we generate a single *Schedule* from scratch by concatenating the trace arrays of all the array references involved or we generate a *Schedule* for each reference, ensuring a consistent mapping from global to local addresses, and then merge them using a union operation. The former method is preferred when the computation of the *Schedule* is communication bound, the latter method is useful when individual *Schedules* can be re-used for other references. For simplicity, we are only handling the former method in our compiler.

Figure 4.18 shows an algorithm to classify references into sets that could be aggregated. The conditions that we impose for aggregation are that communication and inspection can be hoisted to the same loops, and the candidate irregular reference is not included in the inspector slice of other reference in the set. Also, in order to simplify code generation, we do not aggregate references with different access characterization.

4.7 Parametric Inspectors

An important set of irregular problems are multi-phased. By multi-phased we mean problems that have different behaviors at different steps, but after certain step the sequence of behaviors repeats. For example, a multi-colored sparse solver would have a different communication pattern for each color, being typically invoked many times in a real application for the same sparsity structure.

```
do t=1, nsteps
  do color =1, ncolors
    low = begin(color)
    high = end(color)
    Communicate x
    do i= low, high
      ... = ... x(g(i))...
    enddo
    x(... ) = ...
  enddo
enddo
```

Figure 4.19 Example of multi-phase computation

An important observation is that in many of these problems the number of phases before repetition is also input dependent. In order to handle those cases with PILAR we typically need to:

```
do t=1, nsteps
  do color =1, ncolors
    low = begin(color)
    high = end(color)
    compute schedule SC for x
    Communicate x using SC
    do i= low, high
      ... = ... x(g(i))...
    enddo
    x(... ) = ...
  enddo
enddo
```

Figure 4.20 Example of parametric inspector

- Generate code to count the number of phases (T)
- Dynamically allocate arrays of size T
- Add a new phase counter that keeps track of the current phase
- Add an extra level of indirection to access most PILAR objects, using the phase counter to index in the previous arrays.

```
do i =1, ncolors
  Compute Schedule SC_i for elements of x in color i
enddo
do t=1, nsteps
  do color =1, ncolors
    low = begin(color)
    high = end(color)
    Communicate x using SC_color
    do i= low, high
      ... = ... x(g(i))...
    enddo
    x(... ) = ...
  enddo
enddo
```

Figure 4.21 Example of parametric inspector with new placement

Unfortunately, applying the algorithm for placing inspectors described in Section 4.4.5 to a multi-phase irregular computation as the one in Figure 4.19, we will not be able to reuse a schedule (Figure 4.20). An important observation is that even though each individual schedule cannot be

```
PlacementParamInspector()
CL ← compute communication level for reference x(INDEX)
IL ← compute inspector level for reference x(INDEX) assuming CL
if (CL = IL)AND(CL > 1)
  then CLNEW ← CL − 1
       ILNEW ← Compute inspector level for x(INDEX) assuming CLNEW
       if ILNEW < CLNEW
         then Found candidate for parametric inspector
              Set inspector level to ILNEW
              Set communication level to CL
```

Figure 4.22 Algorithm for placing parametric inspectors

reused, a loop of schedules might be. Using this observation, and the fact that the original algorithm depends on the communication level, a modified algorithm is shown in Figure 4.22. Figure 4.21 shows that after applying this algorithm to the original example, we can safely reuse a loop of inspectors.

4.8 Local Address Conversion

In many cases the memory requirements of replicating the global address space of the original application in each processor is unacceptable. In those cases we have to transform array accesses from global to local addresses in each processor. Irregular computations may compute this mapping at run-time using PILAR, but certain modifications to the original code are also needed. In the rest of this section we assume that this mapping uses trace arrays [18] as opposed to other methods, like hashing-based schemes, that trade time by memory overhead.

```
do i=1,n
  do j = 1, ia(i+1)-ia(i)
    ... x(ia(i)+j) ...
  enddo
enddo
```

Figure 4.23 Example in globals

In Figure 4.23 we can see a typical example of an irregular array access with spatial locality. The result of applying a conventional technique to transform this access into locals is shown in Figure 4.24. We added a new indirection array (*local*) that was initialized at run-time by PILAR. The size of this array is the number of accesses to x in the innermost loop, potentially much larger

```
count =0
do i=1,n
  do j = 1, ia(i+1)-ia(i)
    ... x(local(count)) ...
    count = count+1
  enddo
enddo
```

Figure 4.24 Example in locals with enumeration

than the size of array x. We also needed a counter to index into this array. This counter was initialized at the communication level and incremented every access.

It is important to note that during the local conversion we have lost the original structure of the array access. In the program of Figure 4.23 a reasonable sequential compiler will pull out of the innermost loop the invariant part of the access to array x (ia(i)) leaving a regular access in the innermost loop. On the contrary, the access in Figure 4.24 cannot be simplified and is likely to require almost twice as many loads[2]. Moreover, adding accesses to a large array like *local* in the innermost loop may increase cache conflicts with accesses to the array x, further degradating single node performance. Also, if there is a significant reuse among accesses to x, the size of the trace array *local* could become too large. In that case, substituting a trace array access by a hash table look up is very likely to produce an even larger degradation of single node performance. We will give an empirical evaluation of these issues in Chapter 5.

4.8.1 Handling local intervals

A problem with the previous transformation into locals is that it loses the structure of the array subscript. This structure can be preserved using the access characterization facilities provided in the compiler, together with the capabilities of PILAR to handle multiple internal representations.

The new program that preserves subscript structure can be seen in Figure 4.26. Unfortunately, the loop structure needed to be modified by adding an extra loop. Figure 4.25 explains the need for this extra loop. 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 2 or more subintervals. The extra loop is then needed to enumerate sub-sequences for each global sequence.

[2] Assuming the innermost loop has a reasonable average trip count

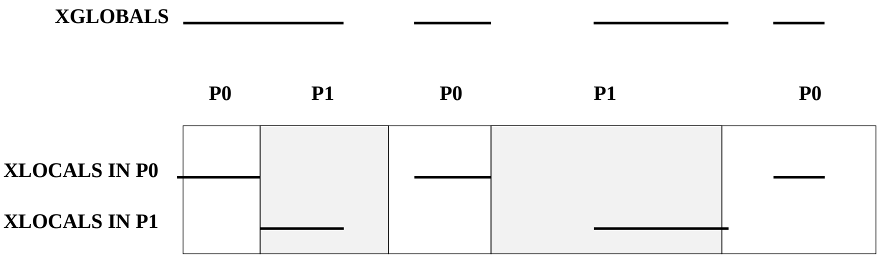

Figure 4.25 Intervals in locals

```
count =0
do i=1,n
  do j1 = 1, nIntervals(i)
    do j2 = 1, localInter(count+1)-localInter(count)
      ... x( j2 +localInter(count)) ...
    enddo
    count = count+2
  enddo
enddo
```

Figure 4.26 Example in locals with intervals

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 be equivalent to the one in globals shown in Figure 4.23. Moreover, the memory overhead of maintaining trace arrays could be drastically reduced, avoiding the need of a hash table in most cases.

In a worst case scenario a totally random data distribution could produce as many subsequences as elements in the array, defeating the purpose of the transformation. However, a data distribution that breaks the access locality is likely to produce unnecessary communication anyway. For example, given an unstructured grid with an arbitrary distribution of grid nodes across processors, we typically want the majority of the edges incoming to a grid node mapped to the same processor that owns the grid node. Nevertheless, even in this worse case, we could always force a redistribution followed by a local re-indexing if this is required.

Unfortunately, we can have added complications when multiple references with spatial locality have different access functions or data distribution, and share the same innermost loop, like in the example in Figure 4.27. The diagram in Figure 4.28 helps to explain the situation: even though both references have the same number of intervals and these intervals have the same size in global addresses, when we transform them into locals, each original segment might break in a different

```
do i=1,n
  do j = 1, ia(i+1)-ia(i)
    ... x(ia(i)+j) + y(ia(i)+j +k) ...
  enddo
enddo
```

Figure 4.27 Example in globals with multiple references

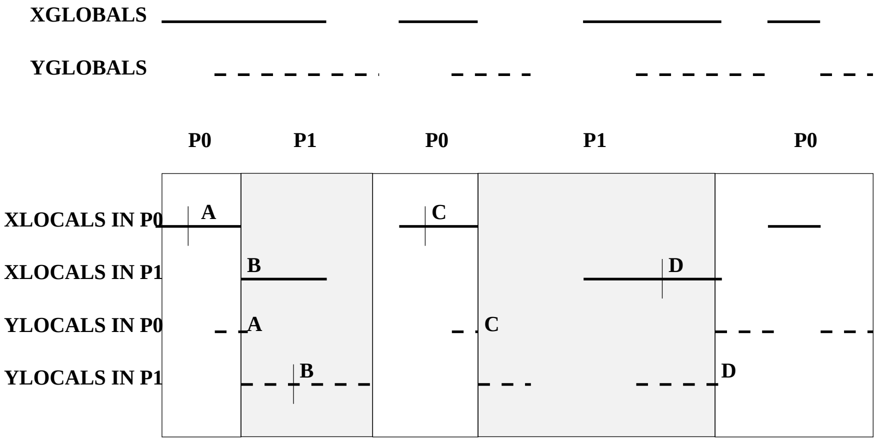

Figure 4.28 Conflicts with intervals in locals

way for each reference, leading to families of segments that cannot easily be enumerated by a single nested set of loops.

```
Merge(n,nIntervalsX,localInterX,nIntervalsY,localInterY,newNIntervals,
    newLocalInterX,newLocalInterY)
count =0
do i=1,n
  do j1 = 1, newNIntervals(i)
    do j2 = 1, newLocalInterX(count+1)-newLocalInterX(count)
      ... x( j2 +newLocalInterX(count)) + y( j2 +newLocalInterY(count))...
    enddo
    count = count+2
  enddo
enddo
```

Figure 4.29 Example in locals with intervals and multiple references

Our solution to this problem is to extend PILAR by providing functionality to further break two sequences of intervals so that: the resulting sequences are compatible (same number of intervals and of the same size) and the number of resulting intervals is minimized. The result of this can be seen in Figure 4.28, where we have tagged with the same label a natural break, produced by the data distribution, and its corresponding induced break, introduced to ensure that sequences are compatible. Figure 4.29 shows the result of applying this technique to the example in Figure 4.27[3].

```
INTERVALSCONFLICTS()
for each loop lp
do CONFLICTS ← NIL
   for each irregular reference ir directly enclosed by lp
   do if TYPE(ir) = INTERVAL
         then TOADD ← TRUE
              for each ir2 in CONFLICTS
              do if inspector of ir2 = inspector of ir
                    then TOADD ← FALSE
              if TOADD
                then Add ir to CONFLICTS
   if SIZE(CONFLICTS) > 1
     then Find irregular reference lastIr in CONFLICTS with
             larger inspection level and topological ordering number
           Insert merge code after inspector of lastIr to deal with CONFLICTS
```

Figure 4.30 Algorithm for handling interval conflicts

In order to automate the detection and handling of conflicts across multiple references with locality we use in our compiler the algorithm described in Figure 4.30. In summary, after identifying possible conflicting references in one loop, we found the reference with the inspector that will be computed last, adding code to the end of this inspector to merge the conflicting interval traces.

Even though we have restricted our explanations to intervals with stride one, the generalization to intervals with constant stride should be simple using PILAR support.

4.9 Code Generation

We tried to minimize the modifications to the parse tree in the previous passes, so that the analysis of the program previously performed by Parafrase-2 (control-flow graph, data dependence graph, call graph ...) did not have to be repeated after each pass. In the code generation pass we do not perform any analysis and simply modify the parse tree using information gathered in other passes. Some of the tasks performed in this pass are:

- Generate counting slices that compute the number of accesses or the number of intervals that encode those accesses. Figure 4.31 shows a counting slice for the example in Figure 4.23

[3]We could also have chosen newLocalInterY for the loop bounds of j2

```
count =0
do i=1,n
  if (1 .LE. ia(i+1)-ia(i))
    count = count +1
  endif
enddo
```

Figure 4.31 Example of a counting slice

```
count =0
do i=1,n
  if (1 .LE. ia(i+1)-ia(i))
    collect(count) = ia(i) +1
    collect(count+1) = ia(i+1) +1
    count = count +2
  endif
enddo
```

Figure 4.32 Example of a collecting slice

assuming an interval-based representation. A counting slice replicates the loop structure, adding a counter incremented inside the innermost loop. In some cases we can optimize this slice by transforming the innermost loop into a masking statement, avoiding zero-trip loops, and incrementing its counter by the number of iterations of the innermost loop.

- Insert calls to dynamically allocate PILAR data structures.
- Generate collecting slices to store the global accesses. Collecting slices repeat the access pattern and store it in a data structure for further processing. Figure 4.32 shows a collecting slice for the example in Figure 4.23, assuming an interval-based representation.
- Instantiate all the counters needed in the program.
- Insert the extra loops that local conversion might require
- Change the subscripts of every array reference that was transformed into local accesses.
- Add PILAR library calls to build schedules and communication buffers, inserting communication primitives when required.
- Add initialization code to some PILAR objects.

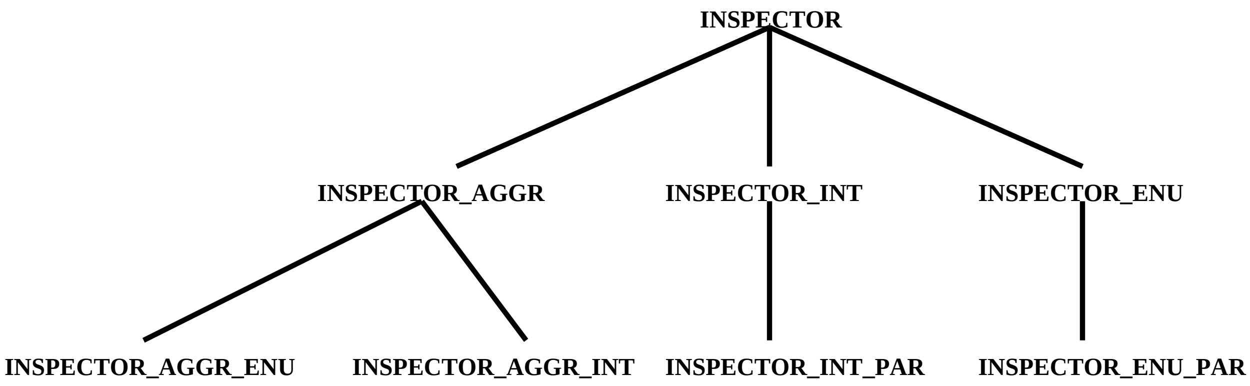

Figure 4.33 Inspector class hierarchy

A difficulty in this pass is the large number of possible cases that we have to consider. For example, an inspector could simultaneously use intervals, be aggregated with other inspectors, parameterized with respect to a certain loop and require conversion to locals. Very quickly an exponential growth of possible cases makes code generation unmanageable. We are trying to manage this complexity in two ways: first using object-oriented techniques, and in particular inheritance, to re-use as much functionality as possible (Figure 4.33 shows a simplified class hierarchy for inspector code generation). Second, trying to prune certain cases that we have found less useful in practice.

4.10 Basic Implementation Techniques

The irregular passes of our prototype compiler represent a significant implementation effort with roughly 10000 lines of C++ code and 1500 lines of *Mathematica* [4] code, even though they extensively use functionality provided by Parafrase-2, other passes in PARADIGM and off-the-shelf *Mathematica* packages.

Most of the implementation relies on two building blocks that have greatly simplified the rest of the implementation. One is an object-oriented shell that hides C-style Parafrase-2 data structures, allowing to easily integrate Parafrase-2 in an object-oriented framework. The other is an interface from Parafrase-2 internal data structures to *Mathematica*, allowing complex symbolic manipulations of expressions to be performed using built-in functionality in *Mathematica*.

[4] *Mathematica* is a registered trademark of Wolfram Research, Inc.

4.10.1 P2++

It is widely accepted that object-oriented techniques can reduce the implementation effort of large programs by promoting re-use and modularity. For this reason we used C++ as our main programming language in both PILAR and the compiler. Parafrase-2, the underlying infrastructure of PARADIGM, is a large and complex software effort mainly written in C, but with a modular design and a reasonable encapsulation of the main data structures. We wanted to further extend these features by adding C++ inheritance, constructors, templates and static type checking. The task of rewriting in C++ such system is unthinkable in the scope of this dissertation. Fortunately, we can get advantage of the existing encapsulation to obtain a compromised solution.

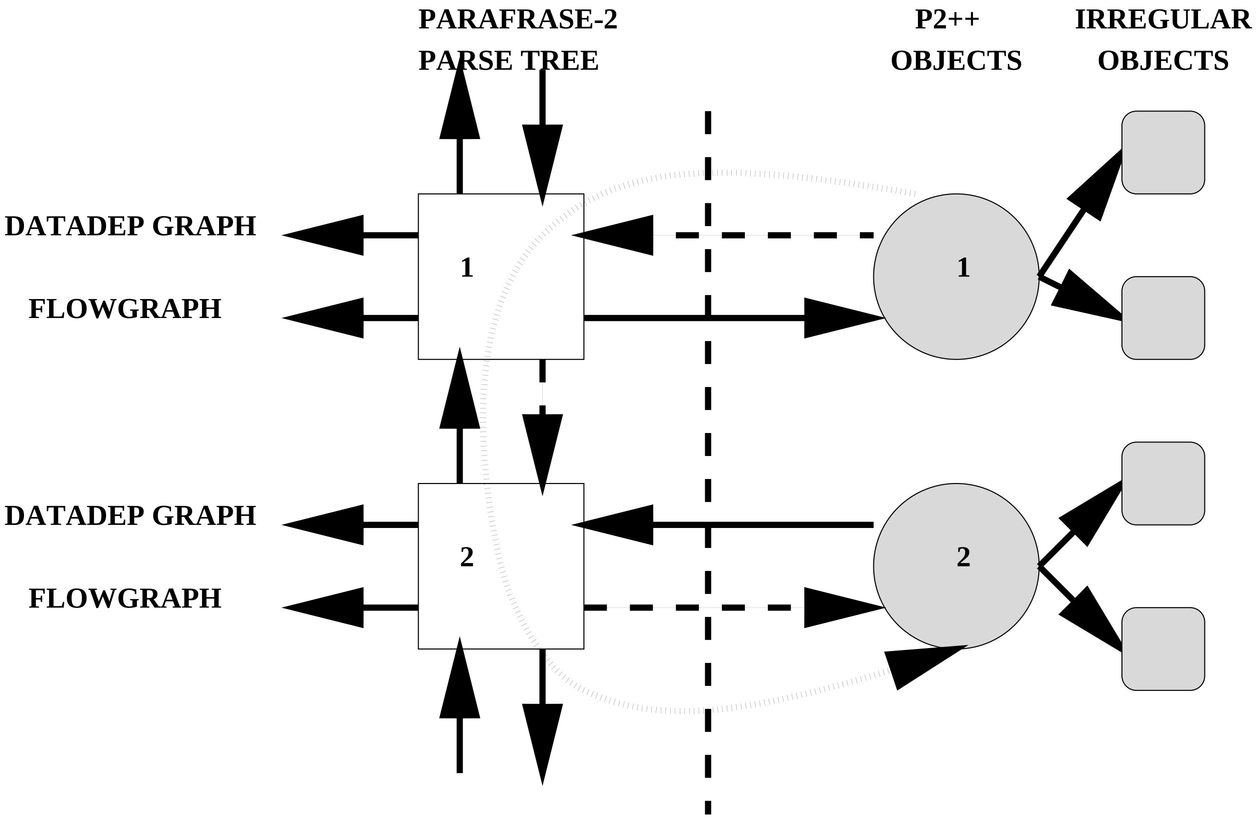

Figure 4.34 Hiding Parafrase-2 data structures

Figure 4.34 shows how the integration with C++ is achieved. Shaded objects are written in C++ and the rest are the **unmodified** Parafrase-2 data structures. Each P2++ object has a pointer to its corresponding statement, and the statement itself points back to the P2++ object using the user pointer. The key idea is that we do not need to add direct pointers among the C++

objects. Instead, we create **virtual** pointers that use the underlying Parafrase-2 infrastructure. For example, if we invoke the method *Next* in the P2++ object 1, we are actually traversing a pointer to Parafrase-2 statement 1, using the *Next* pointer in that statement to reach Parafrase-2 statement 2, and then returning its corresponding P2++ object. In a similar manner we are currently supporting most of the original functionality of Parafrase-2 data structures access macros, so that we never have to deal with them directly.

The motivation for using **virtual** pointers is mainly implementation simplicity. Direct pointers add redundancy that may produce consistency problems. Moreover, the overhead of extra-levels of indirection do not significantly degrade performance because, in most cases, statement traversal is a small component of the overall compilation time.

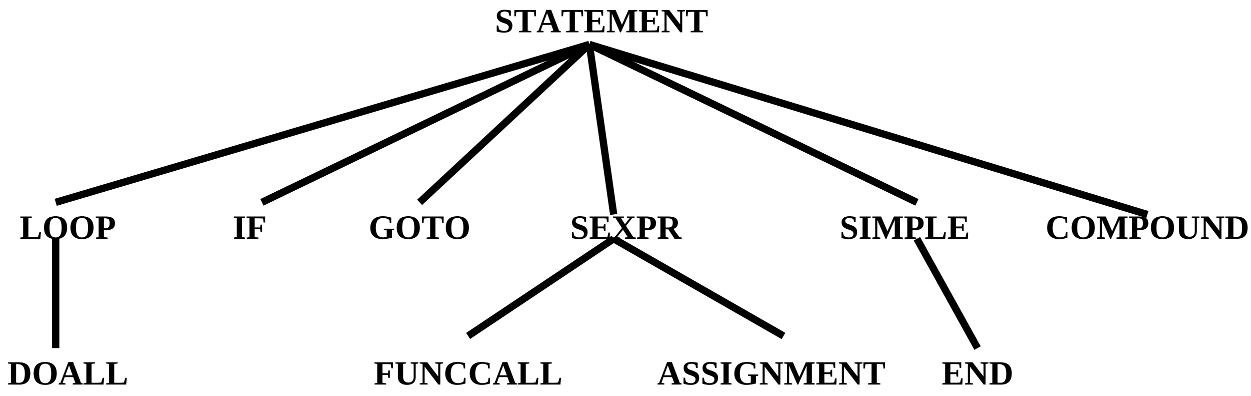

Figure 4.35 P2++ class hierarchy

An important benefit of using P2++ objects is that they form a class hierarchy. Figure 4.35 shows a simplified version of this hierarchy. Virtual methods of P2++ objects allow different behavior depending on the actual statement type, limiting the need of explicit type checks in the code. Also, C++ static type-checking makes typical errors like assuming an incorrect statement type to be detected at compile-time. Object constructors and destructors simplify the creation and deletion of new statements and their insertion in Parafrase-2 data structures.

A powerful abstraction in Parafrase-2 is a set of generic parse-tree traversal functions. We have extended their functionality by conditionally masking certain statements (P2++ objects), and by invoking virtual methods on non-masked objects. In fact, most of the irregular passes use two traversals, the first one masks certain objects and the second one invokes a virtual method on non-masked objects.

Also, we are using the C++ template facility to make even easier to add new passes to Parafrase-2. Many Parafrase-2 passes share a significant amount of code. The idea is that we have defined a generic pass that is automatically instantiated for common cases, eliminating this redundant code.

4.10.2 Mathematica

Almost all the irregular passes previously described use *Mathematica* at some point, although part of the functionality described in this section is only required by the regular passes of PARADIGM [64].

A novel aspect of our approach is to meet the increasing demands for more sophisticated symbolic manipulations by exploiting the well-established functionality of existing powerful symbolic software, in our case, *Mathematica*.

Just as most parsers today are implemented using stable and wide-spread tools like *yacc* and *lex* instead of being built from scratch, we argue that many parallelizing compilers in the future, in particular research prototypes developed in academic environments, will rely on off-the-shelf symbolic packages for most of their symbolic manipulations. Several reasons account for this:

- New compilers are required to perform increasingly complex symbolic tasks [14, 29, 65].
- As compiler algorithms become more complex, there is a need to focus more on their design rather than on their implementation details.
- Many powerful symbolic packages are readily available (e.g., *Mathematica*, Maple™), and some provide external interfaces (e.g., *MathLink*) with reasonable performance and robustness.

Clearly, the performance is not comparable with a C implementation optimized for special cases, but compilation time has not been a major problem in our experience. Furthermore, it is not a good practice to fully tune compiler features until one has tested their impact on optimizing real codes [13]. The approach presented allows quick prototyping and testing of new algorithms with real programs. Once the feasibility of an algorithm is verified, then the critical features affecting efficiency and compilation time can be identified and rewritten in C.

4.10.2.1 Linking PARADIGM with Mathematica

Figure 4.36 shows an overview of the interactions between PARADIGM and *Mathematica*. This follows a typical client/server model in which the server (*Mathematica*) waits for a “symbolic

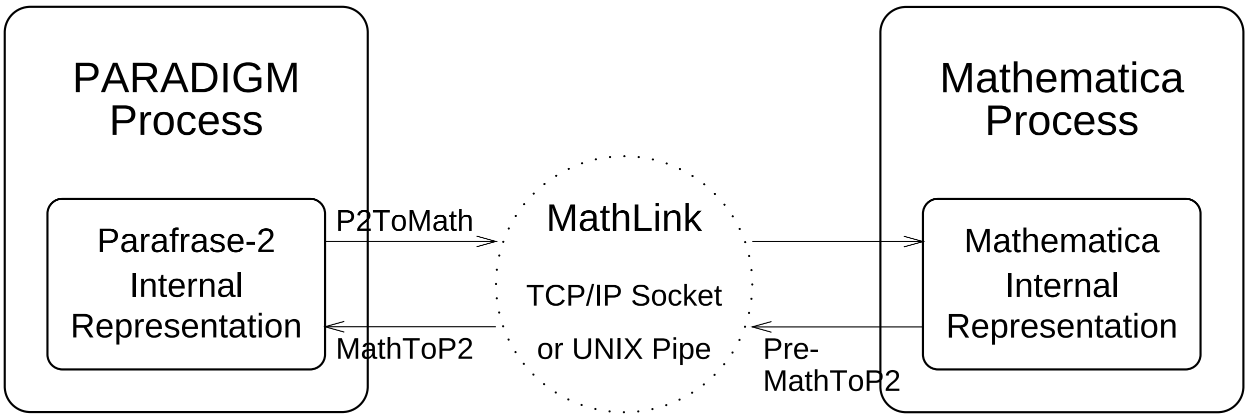

Figure 4.36 Linking PARADIGM with *Mathematica*

request," processes it, and returns the result to the client. Both run as separate UNIX processes and communicate with each other using *MathLink*. The underlying communication mechanism can be either UNIX pipes, when they are running on the same machine, or TCP/IP sockets for remote execution.

Efficient forward and reverse conversion routines transfer data between PARADIGM's internal representation (based on Parafrase-2) and *Mathematica* expressions (*MathToP2* and *P2ToMath* in the figure). Conversion time is kept to a minimum using several techniques:

- The structure of the expression is never lost. We use *MathLink* API function calls to send and receive a tokenized stream.

- Variable names are transformed so that they encode a pointer to the symbol table. This eliminates symbol table look-ups when expressions return from *Mathematica*.

- *Mathematica* also pre-processes the result (*PreMathToP2* in the figure) so that functions have the correct number of arguments and easy-to-decode names following a preset convention.

An Example: Simplifying Loop Bounds To illustrate the interactions between PARADIGM and *Mathematica* we use a simple example, namely the simplification of loop bounds. After certain loop transformations, such as loop normalization, bound expressions can become very complex, and in particular, when arrays with multiple levels of indirection are involved, it is difficult for conventional compilers to "clean up the mess."

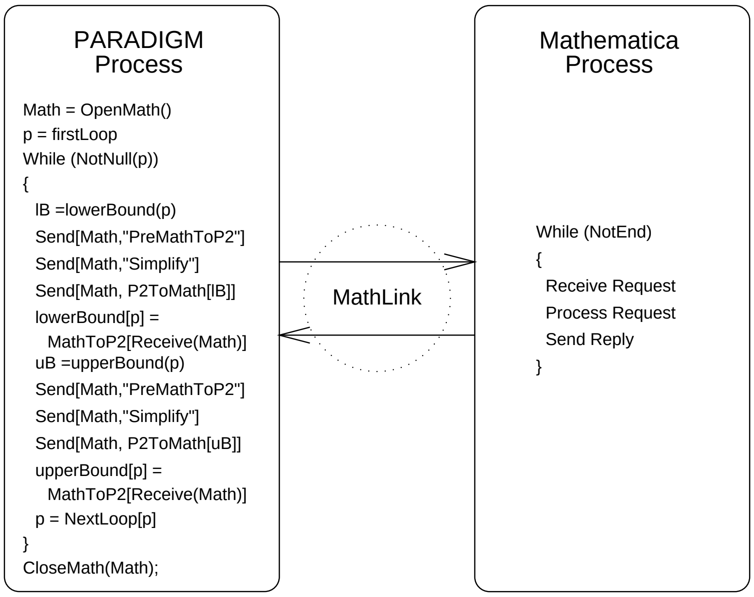

Figure 4.37 Simplifying loop bounds

Figure 4.37 shows our solution to this problem using *Mathematica*. The PARADIGM process first opens a connection with the *Mathematica* process and then traverses the hierarchical representation of the loop nest and finds the lower and upper bound expressions. Each of these expressions is transformed into a *Mathematica* expression using *P2ToMath*, and a *Mathematica* built-in function, *Simplify*, is applied to them by the server. The result is sent back to the client where it is transformed into a PARADIGM expression by *MathToP2* and inserted into the appropriate data structures.

A simple optimization that reduces the number of interactions between the processes is to generate a list with all the lower and upper bound expressions and send it for simplification as one stream. This is particularly important when the granularity of the *Mathematica* operations is small as it is in this case.

4.10.2.2 Mathematica tools in PARADIGM

Using the approach described above we have written a set of tools that we briefly describe next. For the sake of completeness we also describe tools that are only used in the regular part of the compiler.

Symbolic Fourier-Motzkin Elimination The Fourier-Motzkin method solves systems of linear inequalities with real variables using an elimination method. In the context of compilers it has important applications, including data dependence testing [44], inter-procedural analysis and, of particular importance to us, loop transformations [9] and automatic generation of communication for multi-computers [3, 4].

Our implementation quite closely follows the algorithm described by U. Banerjee [8] but also includes some of the conditions stated by M. Ancourt [5] to ensure that the integer projection is the same as the real projection. Our Mathematica implementation has full symbolic capabilities[5]. The basic functionality takes hardly 100 lines of *Mathematica* code, making extensive use of built-in functionality. Symbolic comparisons at every stage of the elimination process significantly reduce the number of redundant constraints and improve performance. This is particularly efficient when the compiler can provide information about the signs of symbolic constants, which is usually the case. Other redundant constraints are eliminated using traditional methods [5].

Enhanced simplification capabilities We have extended the built-in simplification rules of *Mathematica*, providing extensive support for *floor*, *ceiling*, *min* and *max* operations. In particular, declaring unknown symbolic variables as integer and using the fact that integers are closed under certain operations $(+, -, \times)$, we can eliminate many unnecessary *floor* and *ceiling* operations. In addition, using a public domain package[6], expressions like $\frac{3a+4b+2}{3}$ can be determined as non-negative if a and b are first declared to be non-negative. This is used to eliminate operands in *min* and *max* operations by just subtracting operands and checking non-negativity of the result.

Generic loop transformations The functionality of a loop transformation pass is split between the PARADIGM process and the *Mathematica* process in the following way: the PARADIGM process identifies the loops involved in the transformation, checks that the transformation requested is valid using the data dependence and control flow graphs, establishes a connection with the *Mathematica* process, and sends it the relevant portions of the loop such as the loop bounds, array subscripts, and/or the entire loop body. The *Mathematica* process computes the new loop bounds after the transformation and a set of replacement rules to map the old loop variables into the new ones. It then returns the new loop bounds and loop body after applying the replacement

[5]Symbolic coefficients with unknown signs give rise to multi-version code

[6]Thanks to Stephan Kaufmann at the Swiss Federal Institute of Technology for providing the NonNegativeQ package

rules. The PARADIGM process converts the results back into its internal representation. The loop normalization pass discussed in Section 4.3 was implemented using this technique.

Unimodular loop transformations Unimodular loop transformations provide a uniform view to many important transformations [8] (e.g. loop interchange, loop skewing, and loop reversal) and are a powerful tool to increase parallelism [9] and/or locality [71]. A unimodular loop transformation is described by a unimodular matrix, which is an integer matrix with determinant ± 1. In our *Mathematica* implementation, unimodular matrices are allowed to have symbolic terms. In particular, the unimodularity of a symbolic matrix is verified before its inverse is computed using built-in functions. Then, the corresponding loop transformations are applied to a set of loops by computing the new loop variables with the inverse unimodular matrix, generating a set of inequalities from the loop bounds and the new loop variables, and finally applying Fourier-Motzkin elimination to obtain the new loop bounds. Roughly 20 lines of *Mathematica* code in addition to the Fourier-Motzkin package were required for performing unimodular loop transformations.

Regular redundant access elimination transformations We want to ensure that each element of an array is only transmitted once. Based on the early work of Gallivan et al [22], later refined by Ancourt [5], we can obtain a new set of loops that access each element at most once. The main idea behind this transformation is the decomposition of the integer matrix that describes the access pattern into the product of other matrices with special characteristics. In our implementation, we used a *Mathematica* package[7] to obtain the Smith Normal Form and Hermite Column or Row Form of a matrix. Then, we can use the unimodular package described above to compute the new loop bounds.

Integer area estimation For applications like automatic data partitioning or static buffer allocation it is crucial to have compile-time estimates of computation and/or communication requirements. This problem can be reduced to the computation of the integer volume of a polyhedron, but this is in general too complex. Fortunately, in most practical cases we only need an estimate of this volume, and heuristics like the one proposed in [1] are sufficient. We have a preliminary 2-D implementation to estimate the integer area, which extends the work in [1] by handling affine loop bounds. The integer area is approximated by the "real" area plus half the points in the boundary.

[7]Thanks to Brian L. Evans at Georgia Tech for providing the LatticeTheory package

Using a Mathematica package [8], we obtain all the vertices of the polyhedron from the loop inequalities. Then, using built-in functionality, we order the vertices clock-wise and obtain the "real" area through triangulation. Finally, the perimeter is computed by applying the GCD rule to consecutive points.

Graphic visualization The built-in plotting functions in *Mathematica* are used to visualize iteration space and array access patterns in two or three dimensions. By merging plots for different processors (using different colors for each processor), the RECEIVE and SEND sets for a particular communication or the load distribution after applying owner computes rule can be shown. Masks in plotting functions can be used to represent non-convex regions or filter certain types of points. By combining these plotting functions with Fourier-Motzkin Elimination, any convex set defined by linear inequalities can be visualized. This tool proved to be invaluable for debugging the compiler.

C++ code generation We have extended *Mathematica*'s capabilities to output structured C/C++ loops, conditionals, local declarations, and function headers and calls from *Mathematica* expressions. Together with the built-in function *Splice* and a template file, it is relatively easy to generate simple C++ code.

4.11 Summary of Contributions

- Implement and in certain cases extend existing algorithms for basic irregular compiler tasks such as communication and inspector placement, inspector extraction and code slicing.
- Automatic characterization of array access patterns used to select PILAR internal representations.
- Support for multi-phased irregular computations in the form of parametric schedules.
- Support for inspectors involving multiple references to the same array using aggregate schedules.
- Local address space conversion in the presence of interval based representations.

[8] Thanks to Komei Fukuda and Ichiro Mizukoshi of University of Tsukuba, Tokyo for providing the N-dimensional vertex enumeration package

- Implementation methodology based on linking existing traditional compiler platforms with off-the-self symbolic packages and object-oriented environments.

Chapter 5

EXPERIMENTAL RESULTS

In this section we want to show experimentally that we can extend the applicability of irregular compilation techniques to a wider range of problems. This is achieved by a combination of a run-time support with multiple internal representations of communication patterns (PILAR) and an irregular compiler capable of array access characterization. We have selected applications that cover a wide spectrum of regularity:

JACOBI Regular array access pattern and regular data distribution.

AMUX Irregular array access pattern/distribution with very fine grained regularity in some array references

BLOCKAMUX Irregular array access pattern/distribution with coarser grained regularity

LDSOLL Multi-phase irregular array access pattern/distribution with very fine grained regularity in some array references

TINY_INDICO Regular array access pattern and irregular data distribution

5.1 Methodology

Our experiments use an Intel Paragon with 32 MBytes of memory per node and an IBM SP-1 with an upgraded network interface (similar to SP-2) and 128 MBytes per node. In both cases we are using MPI as our underlying communication library, but we are using IBM MPI-F native implementation [21] in the SP-1 and MPICH implementation [27] in the Intel Paragon.

When the application requires an irregular distribution, we follow the approach described in [50] of using a graph partitioner, in our case ML-RSB [10], to obtain an architecture-independent transformation that improves locality. In summary, the graph partitioner is invoked assuming a much larger number of partitions than the maximum number of processor that will ever be used. The output of this partitioner is used to find a permutation vector that, after being applied to the original matrix, will obtain one in which contiguous blocks of rows preserve locality. The benefits from this approach are that after the transformation we can assume a block distribution and we do not need to recompute the partitioning information every time the number of processors changes. The drawbacks are that the quality of the partition is not as good as when it is tuned for a particular number of processors and array re-indexing is necessary. Fortunately, none of these drawbacks were important in our applications. Also, partitioning and re-indexing is performed sequentially and not counted in the inspector times.

All the examples were automatically compiled using PARADIGM and linked with PILAR. Unless stated otherwise, the internal representations of PILAR for each access were automatically chosen by the compiler. In all cases we tried to aggregate communication for multiple references to the same array using an aggregate schedule. Directives were added in some cases to assert that certain loops were parallel or that a particular array has been distributed irregularly. In all cases we were able to re-use inspection across one loop or more.

5.2 Performance Measures

When inspection is re-used multiple times, its relative importance becomes a function of the inspection time, the executor time and the number of times the executor is invoked. In particular, the bottom line speed-up for n iterations would be:

$$S_n = \frac{n * T_{seq}}{T_{insp} + n * T_{exec}}$$

where T_{seq} is the time for sequential execution, T_{insp} is the time for inspector execution, T_{exec} is the time for one invocation of the executor and n the number of times the executor is invoked.

That can be written as a function of executor speed-up $S_{exec} = \frac{T_{seq}}{T_{exec}}$:

$$S_n = \frac{S_{exec}}{1 + \frac{T_{insp}}{n * T_{exec}}}$$

Drawing an analogy with vector processing [35] we can try to give a more meaningful quality measure of inspector overhead. In particular, considering the start-up time of a vector pipe equivalent to inspector time, the vector length equivalent to the number of times the executor is invoked and the processing rate of the pipe equivalent to a function of executor time, we can try to compute $n_{\frac{1}{2}}$, the number of iterations needed to obtain half the executor performance [35]. In particular:

$$n_{\frac{1}{2}} = \frac{T_{insp}}{T_{exec}}$$

$$S_n = \frac{S_{exec}}{1 + \frac{n_{\frac{1}{2}}}{n}}$$

Unfortunately, $n_{\frac{1}{2}}$ is a function of T_{exec}, and it is difficult to compare inspectors with different executor performance, as in most experiments in this section. Instead, we can assume a perfect executor $T_{exec} = \frac{T_{seq}}{N_{proc}}$ and compute $n_{opt\frac{1}{2}}$ in a similar manner:

$$n_{opt\frac{1}{2}} = \frac{N_{proc} T_{insp}}{T_{seq}}$$

What intuitively $n_{opt\frac{1}{2}}$ tells us is that no matter how good our executor would be after inspection, unless we have more than $n_{opt\frac{1}{2}}$ iterations to amortize the inspector, we will only get half the performance of the machine. It is probably a point in which we should worry more on speeding up the inspector at a cost of a slower executor.

Also, $n_{opt\frac{1}{2}}$ is related to $n_{\frac{1}{2}}$ in a simple manner:

$$n_{\frac{1}{2}} = n_{opt\frac{1}{2}} \frac{T_{seq}}{N_{proc} T_{exec}} = n_{opt\frac{1}{2}} \frac{S_{exec}}{N_{proc}} = n_{opt\frac{1}{2}} Eff_{exec}$$

For this reason, given $n_{opt\frac{1}{2}}$, N_{proc} and S_{exec} we can trivially derive S_n for any n using the previous expressions. Therefore, we will use $n_{opt\frac{1}{2}}(N_{proc})$ and $S_{exec}(N_{proc})$ for characterizing performance independent of the number of iterations.

5.3 Performance Results

5.3.1 JACOBI

JACOBI is an iterative method to solve a system of linear equations. It is a regular, stencil-type computation that can be nearly optimally handled by distributed-memory compilers for regular computations, like PARADIGM [6], using a block based distribution. The motivation for this experiment is to measure the performance penalty of treating a regular computation as irregular for the purpose of computing local addresses and generating optimized communication. We also want to show what is the benefit of exploiting regularity at run-time in cases where there is plenty of regularity to exploit.

```
      subroutine jacobi(a,b,niter,low,high,tt)
      parameter(n=1001)
      double precision a(0:n,0:n), b(0:n,0:n)
      integer tt,n,niter,low,high
!dir$ irregular a with tt
!dir$ irregular b with tt
      do k = 1, niter
        do j = low, high
          do i = 1, n-2
            a(i,j) = (b( i - 1,j) + b(i + 1,j) + b(i,j - 1) + b(i,j + 1)) / 4.0
          enddo
        enddo
        do j = low, high
          do i = 1, n-2
            b(i,j) = a(i, j)
          enddo
        enddo
      enddo
      end
```

Figure 5.1 JACOBI

Figure 5.1 shows our input program. Arrays A and B are distributed using a (*,BLOCK) HPF distribution. An irregular directive forced the compiler to treat references to array A and B as irregular ones. A matrix size of 1001 by 1001 was chosen for A and B to minimize cache effects.

Figure 5.2 shows an important improvement in executor performance by using an interval-based representation versus an enumerated one. This is mainly due to improved single node performance. The reason is that enumeration adds an extra-level of indirection for every memory access to obtain local addresses and, in particular, on machines with fast floating point hardware like the SP-1, indirection degrades performance significantly [19]. On the other hand, after conventional optimizations performed by the sequential compiler, the single-node performance of the interval-based program is roughly the same as the original code. This is due to the local conversion transformation in the presence of intervals described in Section 4.8 that avoids extra-levels of indirection in the innermost loop. When intervals are being used the speed-up is almost linear and comparable to the one obtained by regular compilation techniques [6].

Moreover, the memory overhead of enumeration due to trace arrays increased the overall memory requirements by a factor of five with respect to the original program. On the contrary, the memory overhead of the interval-based scheme was almost negligible.

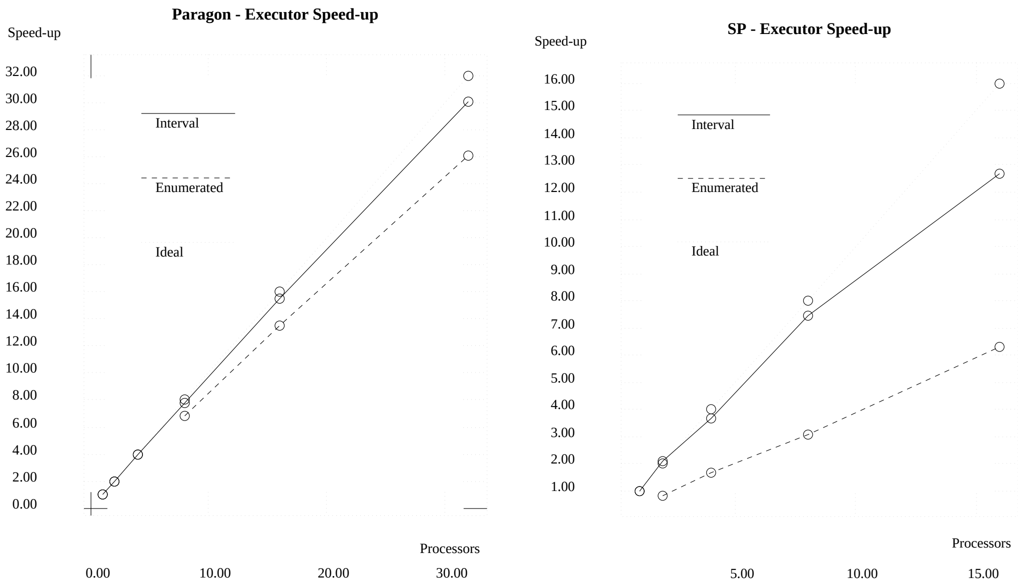

Figure 5.2 Executor speed-up of JACOBI

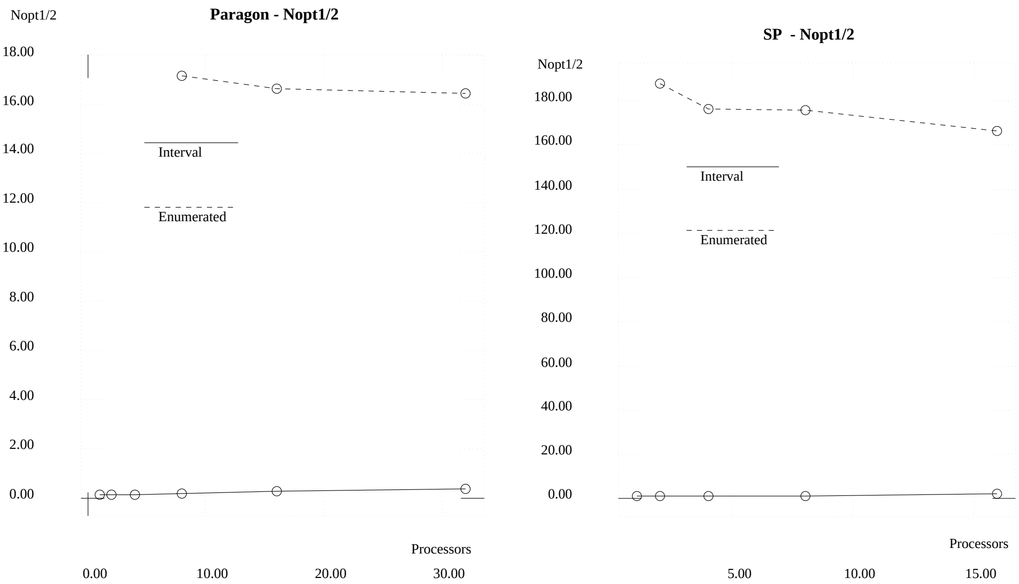

Figure 5.3 Inspector performance: $n_{opt\frac{1}{2}}$ of JACOBI

Figure 5.3 shows that we are obtaining two orders of magnitude of improvement in inspector execution times by using intervals. Moreover, when intervals are being used, $n_{opt\frac{1}{2}}$ is always less than 0.4 in the Paragon so we do not even need to extract an inspector to obtain good performance. In the SP-1 this value is less than 2, still much smaller than the typical number of iterations of the method.

The expected conclusion is that traditional methods for handling irregular problems are not suited for regular problems. Nevertheless, when run-time analysis techniques are extended with more powerful representations, they can even compete with compile-time analysis for regular computations. This example shows clearly that, if regularity is available, dramatic improvements can be achieved by taking advantage of this regularity at run-time.

5.3.2 AMUX

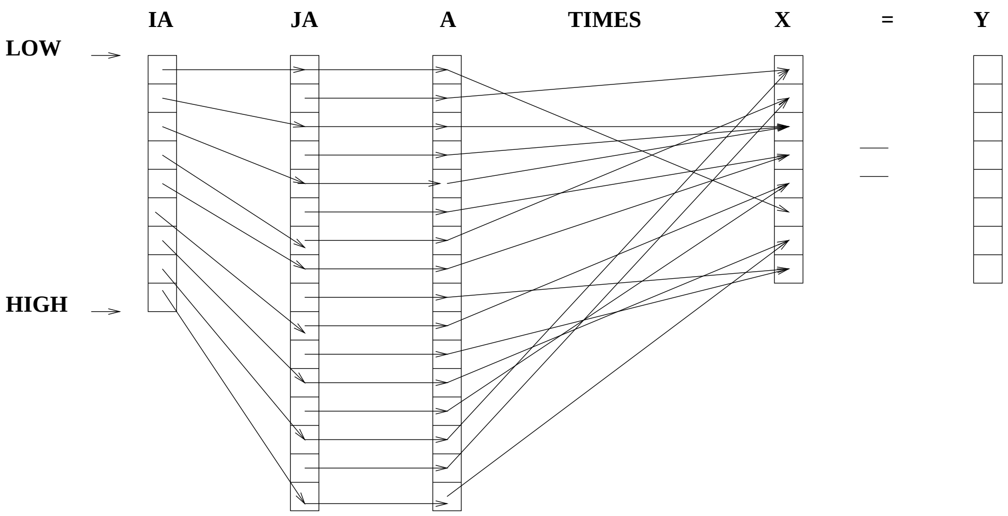

Figure 5.4 AMUX schematic

AMUX is a subroutine taken from the SPARSKIT [61] that performs the product of a sparse matrix in CSR format by a dense vector. This operation is key to many iterative solvers (i.e., Lanczos) and is typically repeated many times using the same matrix and a modified dense vector. This dense vector needs to be partially communicated at each time step according to the sparsity structure of the matrix.

Figure 5.5 shows our input program. Also, Figure 5.5 describes its multiple levels of indirection. Note that array accesses a(k) and ja(k) in the innermost loop have fine grained spatial locality suitable for an interval-based representation and they can share one inspector. Unfortunately,

```
      subroutine amux (net,ttrow,ttrowplus,ttnnz, low,high,
      $ x, y, a,ja,ia)
      double precision x(0:*), y(0:*), a(0:*), t
      integer ja(*0:), ia(0:*),low,high,ttrow,ttrowplus,ttnnz,net
!dir$ irregular a with ttnnz
!dir$ irregular ja with ttnnz
!dir$ irregular x with ttrow
!dir$ irregular y with ttrow
!dir$ irregular ia with ttrowplus
      do i = low,high
        k1 = ia(i)
        k2 = ia(i+1) -1
        t = 0.0d0
        do k=k1, k2
          t = t + a(k)*x(ja(k))
        enddo
        y(i) = t
      enddo
      end
```

Figure 5.5 AMUX

accesses to array x in the innermost loop typically have little locality, requiring an enumeration-based scheme. Clearly, this would limit the potential improvement of using intervals and suggests the need of supporting multiple-representations simultaneously.

The experiments use a large unstructured grid, *rotor*, obtained from NASA.*Rotor* has 99617 nodes with an average degree in each node of 13.299 edges and a maximum degree of 125. The CSR sparse representation is adequate for this grid because the ratio between its maximum degree and its average degree would make very inefficient a two dimensional representation.

Figure 5.6 shows that there is a significant advantage of using an interval-based representation in executor performance even though only two out of three irregular references in the innermost loop may benefit from using intervals. Again, this performance advantage is mainly based on single-node performance improvement. Also, even though the granularity of the computation is small, it is possible to obtain speed-ups up to a reasonable number of processors.

Figure 5.7 shows the inspector performance. Only one out of two main inspectors use intervals so fifty percent would be the best possible improvement. We can see that almost all this improvement is realized, and the remaining enumerated inspector dominates the overall inspection time. This suggests that, in this example only marginal improvements could be achieved by eliminating multiple levels of indirection through alignment restrictions. In practice, real applications perform

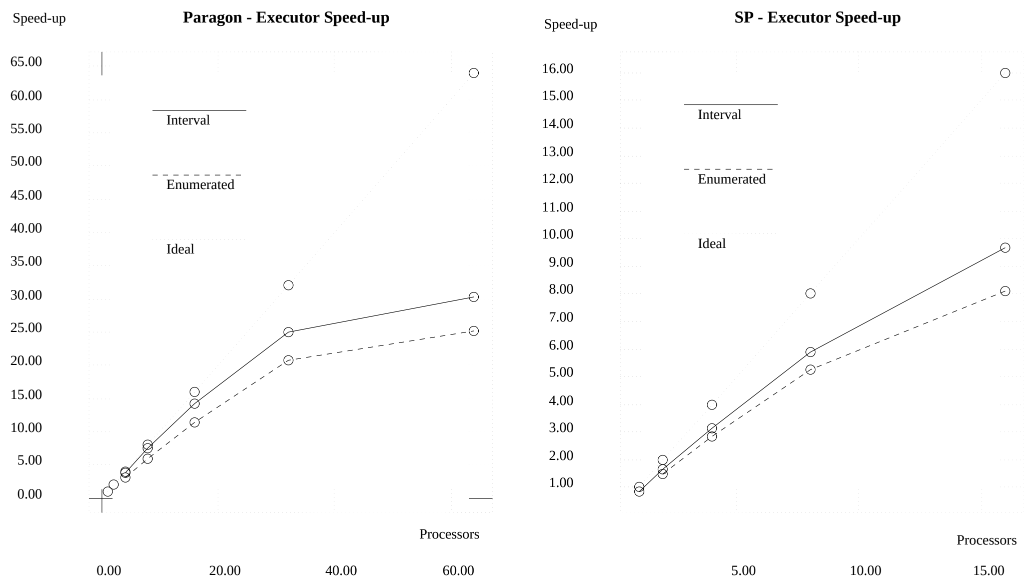

Figure 5.6 Executor speed-up of AMUX

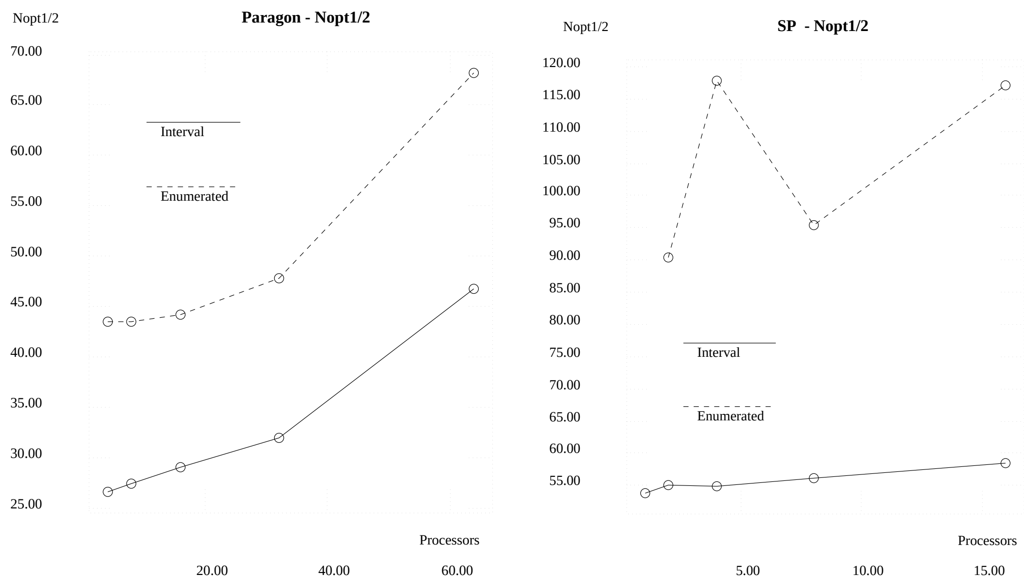

Figure 5.7 Inspector performance: $n_{opt\frac{1}{2}}$ of AMUX

hundreds of matrix vector products, and amortizing the cost of the inspector should not be a problem.

5.3.3 BLOCKAMUX

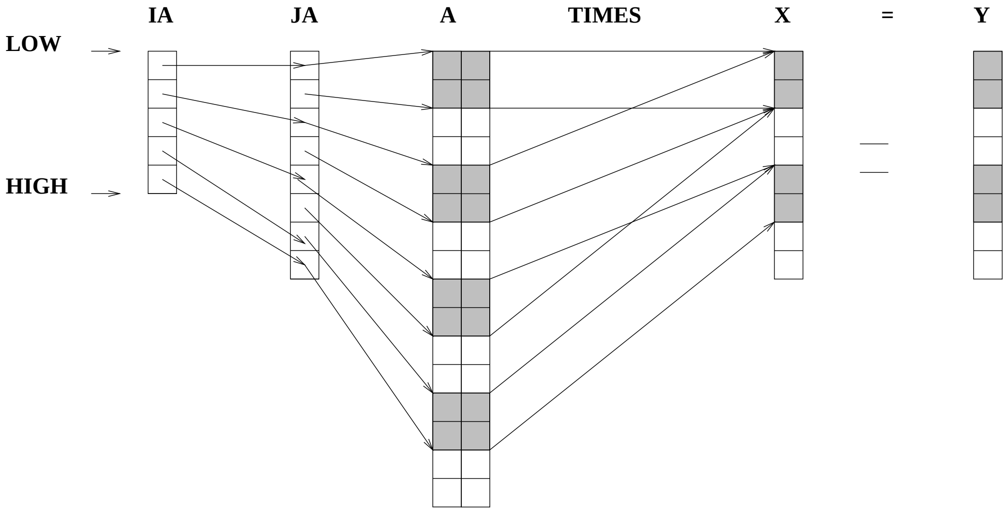

Figure 5.8 BLOCKAMUX schematic

Most real applications that perform sparse matrix- dense vector products have small dense sub-blocks in the sparse matrix. This sub-blocks originate from multiple degrees of freedom associated with each node [36]. We can get advantage of this structure by using a different sparse representation BSR (block-row sparse format) [61], that reduces the overhead of indirect indexing. Figure 5.9 shows a subroutine that performs this product using a BSR sparse representation. Multiple levels of indirection in BSR are explained in Figure 5.8.

We also wanted to show the potential of PILAR through manual parallelization. We modified the code in Figure 5.9 as shown in Figure 5.10, assumed that the small sub-blocks cannot be split-up across processors and performed aggregation of intervals using the two innermost loops[1].

The block size for the dense sub-blocks was 5, a common number in many aerospace applications. We also used a two-dimensional grid from NASA (*4elt*) representing a four element airfoil. This grid has 15606 nodes and an average degree of 5.879. In the Intel Paragon the grid was too large to fit in the memory of one processor and sequential times were approximated using a linear regression

[1]The current compiler can only detect intervals using the innermost loop

```
  subroutine bamux (ttrow,ttrowplus,ttnnz, ttnnzBig,low,high, x, y, a,ja,ia,blockSize)
  double precision x(0:*), y(0:*), a(0:*),t
  integer ja(0:*), ia(0:*),low,high
  integer blockSize,i,k1,k2,k,by,bx,kja,ttrow,ttrowplus,ttnnz,ttnnzBig
!dir$ irregular a with ttnnzBig
!dir$ irregular ja with ttnnz
!dir$ irregular x with ttrow
!dir$ irregular y with ttrow
!dir$ irregular ia with ttrowplus
  do i = low,high
    k1 = ia(i)
    k2 = ia(i+1) -1
    do bx = 0,blockSize -1
      t = 0.0d0
      do k=k1, k2
        kja = ja(k)
        do by = 0,blockSize -1
          t = t + a(by+bx*blockSize+blockSize*
  $         blockSize*k )* x(kja + by)
        enddo
      enddo
      y(i*blockSize+bx) = t
    enddo
  enddo
  end
```

Figure 5.9 BLOCKAMUX

involving 4 and 8 processors runs. Timings measured for our manually parallelized example were used to normalize all the speed-up curves.

Figure 5.11 shows the executor performance of the automatically compiled version, using enumeration and intervals, versus the manual version that also uses intervals. Little improvement, in particular on the SP-1, is obtained by the compiler generated interval-version. Although all the references have spatial locality, the interval size is fixed by the block size (5) and the extra control overhead offsets the benefits of reduced indexing. On the other hand, the manually parallelized version exploits locality across two loops and these larger intervals improve performance significantly.

Inspector performance improves significantly using intervals, as it is shown in Figure 5.12, but the full potential of intervals is only achieved by manual parallelization.

The conclusion is that, for this example, it is possible to obtain significant improvements using an interval based representation in PILAR. Unfortunately, the full potential cannot be achieved with the current compiler implementation. In order to achieve this potential we would need to

```
  subroutine bamux (ttrow,ttrowplus,ttnnz,ttnnzBig,low,high, x, y, a,
       ja,ia,blockSize)
  double precision x(0:*), y(0:*), a(0:*),t(0:10),xtemp
  integer ja(0:*), ia(0:*),low,high
  integer blockSize,i,k1,k2,k,by,bx,kja,ttrow,ttrowplus,ttnnz,ttnnzBig
!dir$ irregular a with ttnnzBig
!dir$ irregular ja with ttnnz
!dir$ irregular x,y with ttrow
!dir$ irregular ia with ttrowplus
  do i = low,high
    k1 = ia(i)
    k2 = ia(i+1) -1
    do by = 0,blockSize -1
      t(by) = 0.0d0
    enddo
    do k=k1, k2
      kja = ja(k)
      do by = 0,blockSize -1
        xtemp = x(kja + by)
        do bx = 0,blockSize -1
          t(bx) = t(bx) + a(by+bx*blockSize+blockSize*
  $       blockSize*k )*xtemp
        enddo
      enddo
    enddo
    do by = 0,blockSize -1
      y(i*blockSize+by) = t(by)
    enddo
  enddo
  end
```

Figure 5.10 BLOCKAMUX2

further extend the array characterization pass, using information from multiple loops and also give more information to the compiler on the array data distribution.

5.3.4 LDSOLL

LDSOLL is a lower triangular sparse solver using a level-scheduling/sparse row format. Figure 5.14 shows a slightly modified version of the original routine from the SPARSKIT [61]. This subroutine uses many levels of indirection, as it can be seen in Figure 5.13. Changes to the original code were needed because the functionality of multiple logical arrays was overloaded into a single array, difficulting array distribution and compile-time analysis. Also, a directive was added to indicate that a loop is parallelizable.

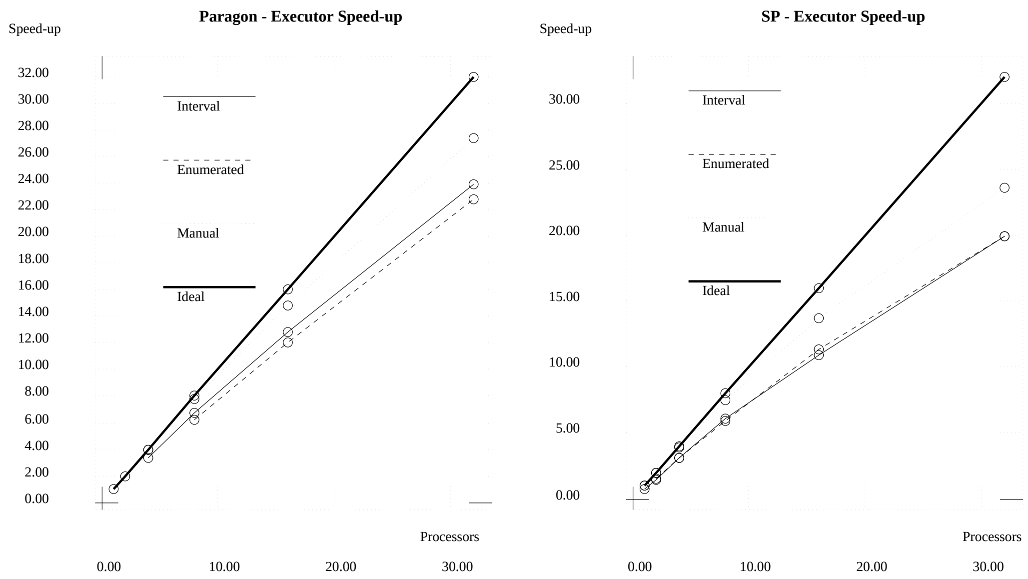

Figure 5.11 Executor speed-up of BLOCK-AMUX

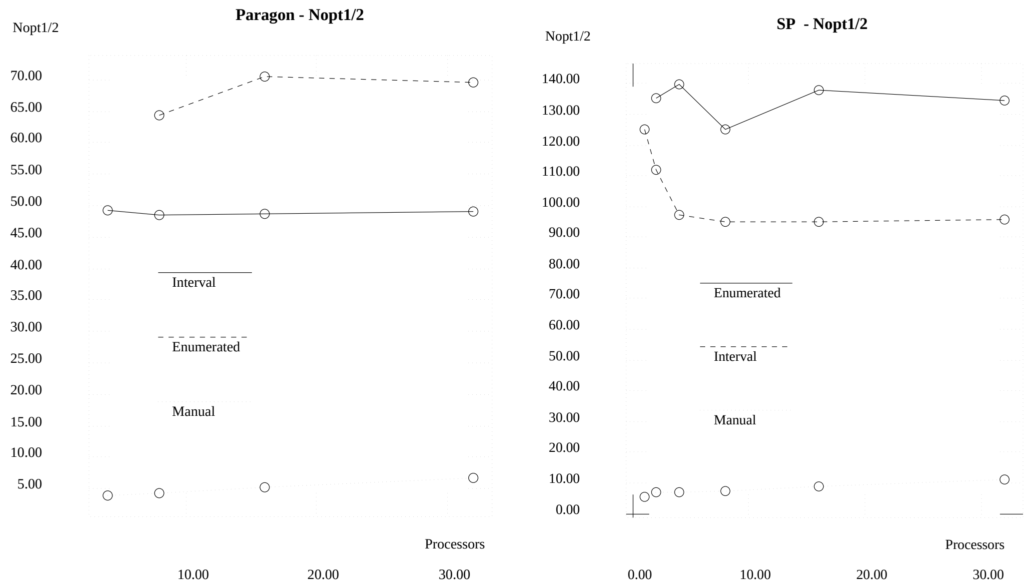

Figure 5.12 Inspector performance: $n_{opt\frac{1}{2}}$ of BLOCK-AMUX

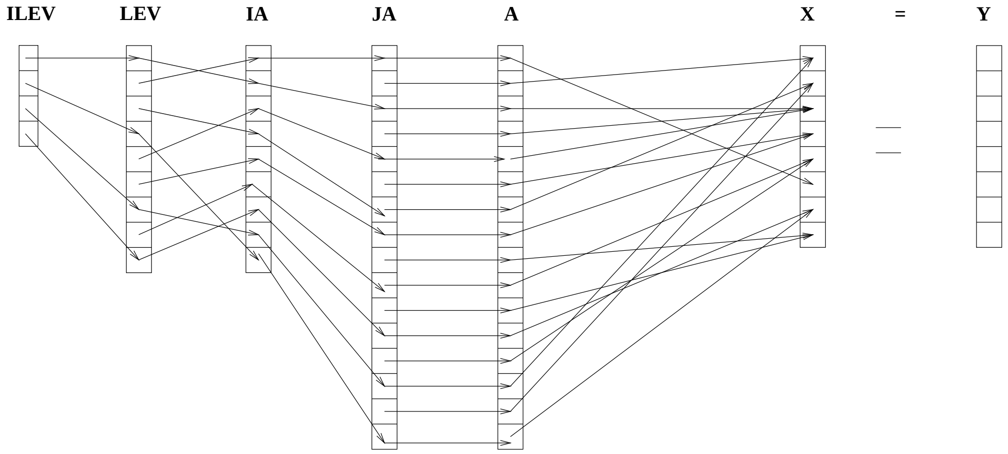

Figure 5.13 LDSOLL schematic

Parallelization of a sparse triangular solver on a multi-computer is a very challenging problem [32]. The fine-grained nature of the parallelism found in many sparse matrices makes it very difficult to exploit. Fortunately, in many cases the solver is part of a preconditioner and we can reorder the entries in the matrix using a coloring technique to increase parallelism[2]. We use a coloring technique described in [42] that uses partitioning information to improve load balance for each color. Applying this technique to the grid *Rotor* described in Section 5.3.2, and forcing a lower triangular matrix (half the edges of the original grid), we were able to color the grid with 13 colors.

LDSOLL is a clear example of multi-phase irregular computation, as described in Section 4.7. Therefore, in order to extract an inspector we need to use parametric schedules. In particular, a parametric schedule with enumeration was used for the accesses to the array x in the innermost loop.

Figure 5.15 shows that no performance benefit in the executor was obtained by using intervals. This was expected because the accesses to the array x in the innermost loop have no spatial locality and we always used enumeration for them, and the effective interval size is greatly reduced by the coloring and by forcing the matrix lower triangular. Speed ups are smaller than in previous experiments and reflect the intrinsic difficulty of the problem. Also, in the Intel Paragon the grid was too large to fit in the memory of one processor and sequential times were approximated using a linear regression involving 2 and 4 processors runs.

On the other hand, we see a significant improvement using intervals in inspector performance in Figure 5.16. This benefit is the result of decreasing the cost of inspection for references a and

[2]This reordering might affect the convergence rate of the iterative algorithm

```
    subroutine ldsoll (net,ttrow,ttrowplus,ttnnz, nlev,lev,ilev,
    $ x, y, a,ja,ia,diag)
    double precision x(*), y(*), a(*), t ,diag(*)
    integer ja(*), ilev(*), lev(*),ia(*),nlev
    integer ttrow,ttrowplus,ttnnz,net
    integer ii,i,k,m1,m2,jrow,i1,i2
!DIR$ irregular ia with TTRowPlus
!DIR$ irregular a,ja with TTNNZ
!DIR$ irregular x,y,diag with TTRow
    do ii=1, nlev
      i1 = ilev(ii)
      i2 = ilev(ii+1)-1
!HPF$ INDEPENDENT
      do i=i1, i2
        jrow = lev(i)
        m1 = ia(jrow)
        m2 = ia(jrow+1)-1
        t = y(jrow)
        do k=m1, m2
          t = t - a(k)*x(ja(k))
        enddo
        t = t*diag(jrow)
        x(jrow) = t
      enddo
    enddo
    end
```

Figure 5.14 LDSOLL

ja in the innermost loop. Also, even though we computed a parametric schedule for array x with 13 sub-schedules, $n_{opt\frac{1}{2}}$ is not unreasonable for most applications.

5.3.5 TINY_INDICO

INDICO is an Indian Ocean circulation simulation code developed by O'Brien's group at Florida State University. An important effort to improve its performance in a shared-memory architecture is described in [73]. It belongs to a class of iterative problems with regular, stencil-based array access patterns but with irregular boundaries. An example of these boundaries is shown in Figure 5.17. For this reason, even though subscripts are simple, communication cannot be analyzed at compile time and run-time preprocessing is required.

Figure 5.18 shows a typical loop in the main routine of INDICO. Computation is only performed at sea points and not at land points. Instead of masking the computation of a point inside the

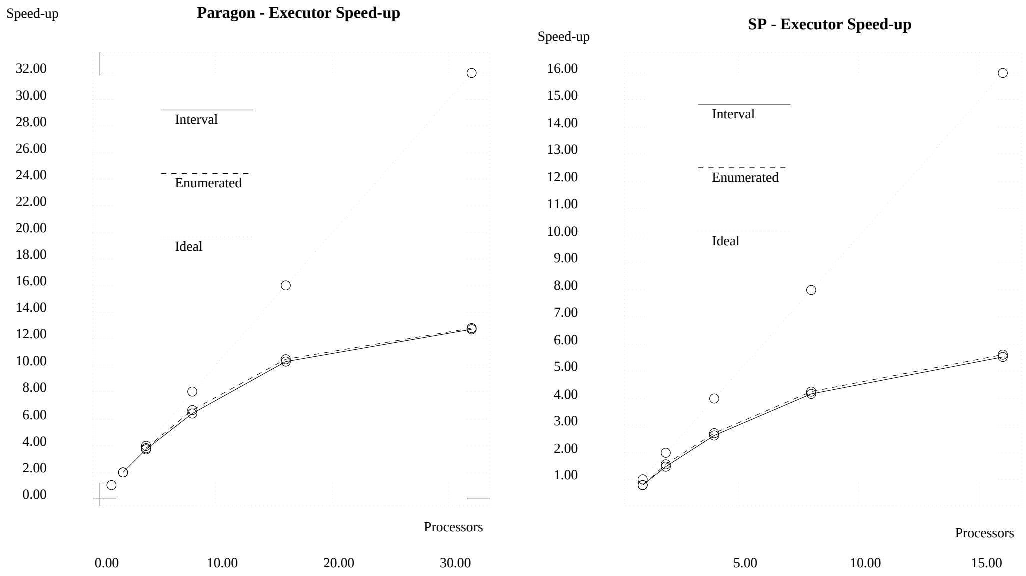

Figure 5.15 Executor speed-up of LDSOLL

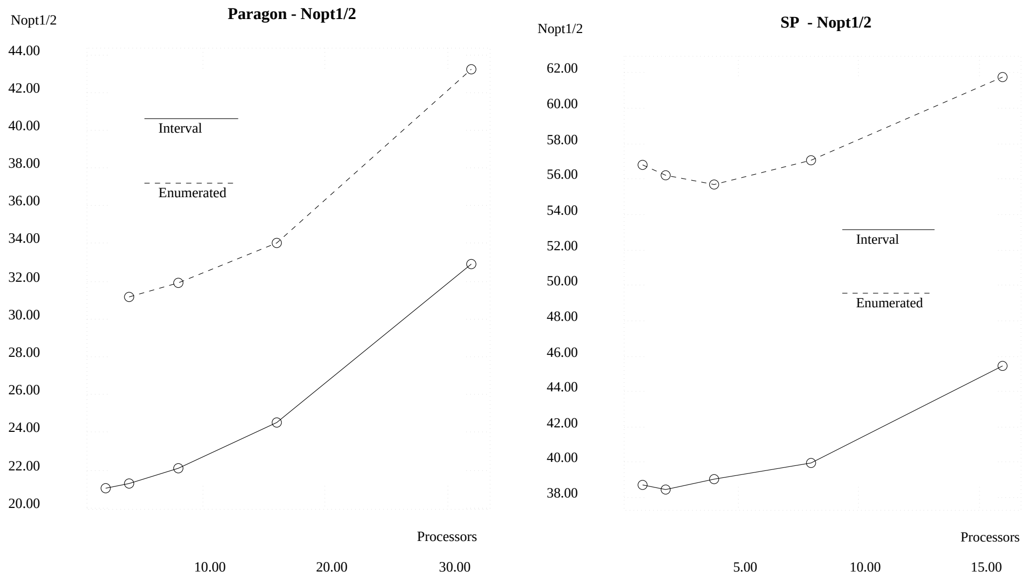

Figure 5.16 Inspector performance: $n_{opt\frac{1}{2}}$ of LDSOLL

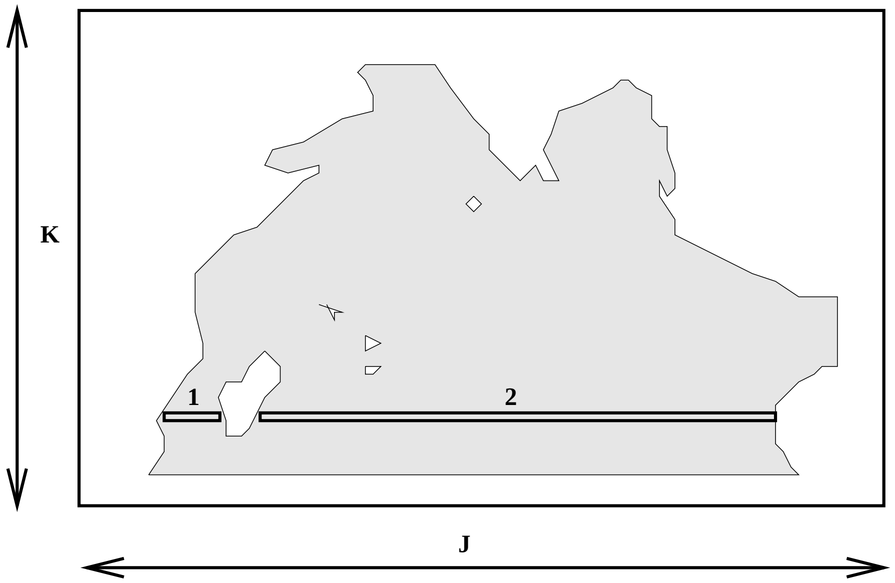

Figure 5.17 TINY_INDICO irregular boundary

```
do k=1,n
   do i=1,nsgh(k)
      do j=jsh(k,i),jeh(k,i)
         h(j,k)=h(j,k)-dphin*(u(j,k)-u(j-1,k)) + ...
      enddo
   enddo
enddo
```

Figure 5.18 TINY_INDICO loop

innermost loop, as it is shown in Figure 5.19, the original program finds, in a preprocessing step, segments of contiguous points at sea for each row. This segments are then described by arrays $NSGH$, JSH and JEH and used during the computation to perform loop bounds reduction.

```
do k=1,n
   do i=1,m
      if (isSea(i,j))
         h(j,k)=h(j,k)-dphin*(u(j,k)-u(j-1,k)) + ...
      endif
   enddo
enddo
```

Figure 5.19 Simplified TINY_INDICO loop

Due to the shape of the Indian Ocean, a partition of the outermost loop in equally sized blocks would lead to heavy load imbalance. Moreover, computations are stencil-based leading to high communication overhead for cyclic distributions. We used a block-based partitioning with varying block sizes, where block sizes where chosen at run-time to equalize the number of sea-points in each partition. This partitioning is performed in parallel with negligible overhead.

The kernel extracted from INDICO, TINY_INDICO has 6 loops of this form containing almost a hundred irregular references. The compiler produced 14 schedules (8 of them aggregated) generating around 2000 lines of Fortran code. All array references may benefit from spatial locality. Clearly, manual parallelization would have required a significant implementation effort.

The input data set was the one provided with the original INDICO program, with a discretization of the Indian Ocean using a 531 by 333 grid, although only one sea layer instead of three was computed.

Figure 5.20 shows a clear improvement in executor performance, in particular on the SP-1, by using an interval based representation. This improvement is mainly due to a better single-node performance. In fact, the single-node performance of the interval based program was very close to sequential performance. Also, in the case of enumeration, the memory overhead caused by trace arrays would make more desirable to use global addresses, replicating the address space in each processor, up to 16 processors. On the other hand, the memory overhead of the interval based representation was less than 10 percent over the sequential program.

Also, Figure 5.21 shows that almost an order of magnitude of improvement in inspector performance can be achieved by using intervals. This ensures that inspector overhead will easily be amortized, because the original program typically perform over a hundred time steps.

The conclusion is that exploiting spatial locality in TINY_INDICO we are able to efficiently execute in parallel, with low memory overhead and quickly amortized inspection. It is unclear how to achieve these goals with other more conventional techniques.

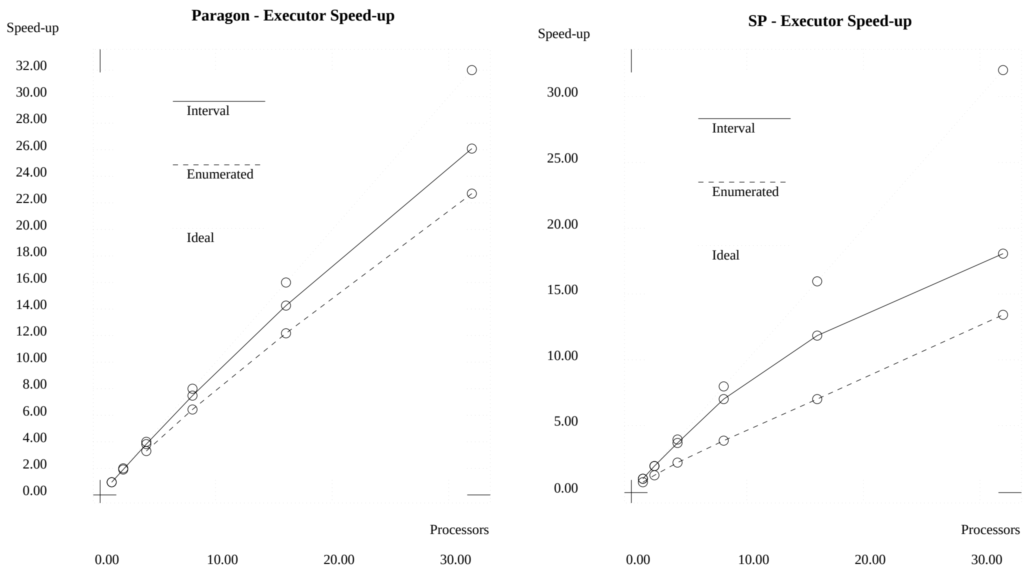

Figure 5.20 Executor speed-up of TINY_INDICO

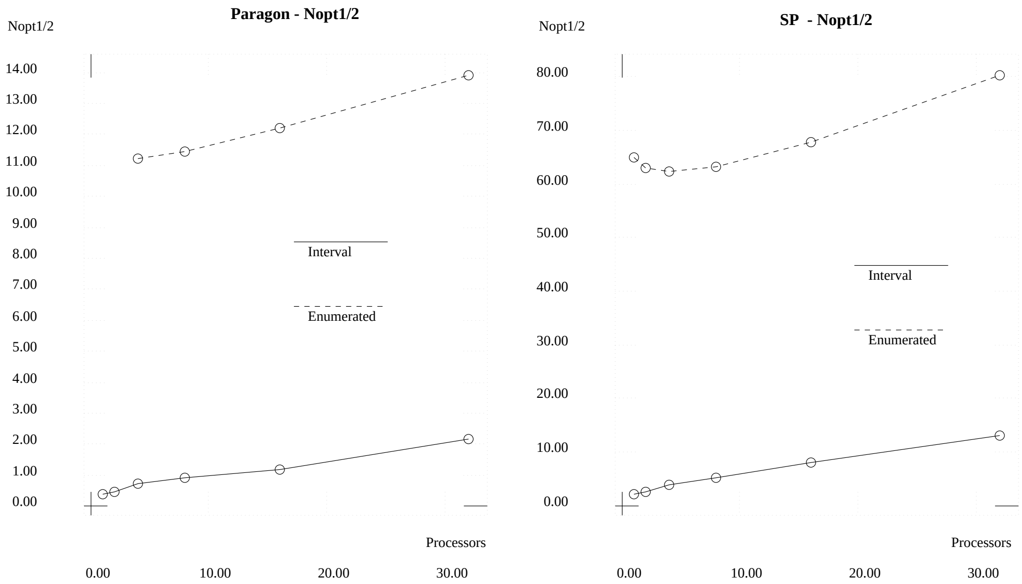

Figure 5.21 Inspector performance: $n_{opt\frac{1}{2}}$ of TINY_INDICO

Chapter 6

CONCLUSION

In this thesis we have shown that it is feasible and profitable to automate the exploitation of regularity in irregular applications to further improve performance after parallelization on a multi-computer. Moreover, we have shown that unless this regularity is exploited, certain applications cannot be successfully parallelized with existing techniques. In order to show this improvement we developed a new RTS for irregular computations called PILAR and extended the PARADIGM compiler to handle irregular computations, inserting calls to PILAR in Fortran programs. Experimental results using a set of applications that cover a wide spectrum of regularity validate our approach.

In particular, our contributions on irregular run-time support are as follows:

- Multiple representations of communication patterns to exploit regularity in irregular accesses.
- Full integration with MPI, taking advantage of communicators and derived data types.
- Non-blocking irregular communication.
- Local combining of Schedules to reduce pre-processing overhead.
- Interval-based Distributed Translation Tables.
- Implementation complexity managed by an object-oriented design.

Also, our contributions on irregular compiler support are the following:

- Implement and in certain cases extend existing algorithms for basic irregular compiler tasks such as communication and inspector placement, inspector extraction and code slicing.
- Automatic characterization of array access patterns used to select PILAR internal representations.
- Support for multi-phased irregular computations in the form of parametric schedules.
- Support for inspectors involving multiple references to the same array using aggregate schedules.

- Local address space conversion in the presence of interval based representations.
- Implementation methodology based on linking existing traditional compiler platforms with off-the-self symbolic packages and object-oriented environments.
- Evaluation of the techniques using several benchmark programs on two real parallel machines.

This work can be extended in multiple directions:

- Exploit regularity in irregular applications to optimize data and computation distributions. The task of a graph partitioner can be greatly simplified if the input graph already reflects the local structure of the computation.
- Extend the basic compiler infrastructure by performing global optimizations, such as global redundant inspection elimination, even across procedures. Also, use data-flow combined with data dependence analysis to place communication and inspection.
- Extend the automatic access characterization analysis to use information from multiple loops and to handle patterns present only in the sparse matrix and not reflected in the array subscript expressions.
- Increase the flexibility of the RTS by letting the compiler generate a new representation, instead of just choosing from existing ones, to further increase performance.
- Unify regular and irregular optimizations such as, communication aggregation, redundant communication elimination or data partitioning, that can improve the performance of programs in which regular and irregular computations are intertwined.
- Adapt existing allocation and scheduling algorithms [56] to exploit functional and data parallelism in irregular applications.
- Complement our approach with a framework that extracts, and not only exploits, parallelism at run-time.

REFERENCES

[1] Anant Agarwal, David Kranz, and Venkat Natarajan. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. In *Proceedings of the 22nd International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[2] Gagan Agarwal, Alan Sussman, and Joel Saltz. Compiler and Runtime Support for Structured and Block Structured Applications. In *the Proceedings of Supercomputing*, Portland, OR, November 1993.

[3] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 126–138, Albuquerque, NM, June 1993.

[4] C. Ancourt and F. Irigoin. Scanning Polyhedra with DO Loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming*, pages 39–50, Williamsburg, VA, April 1991.

[5] Martine Ancourt. *Generation Automatique de Codes de Transfert pour Multiprocesseurs a Memoires Locales*. PhD thesis, Universite Paris VI, March 1991.

[6] P. Banerjee, J. A. 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*, October 1995. To appear.

[7] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In *Proceedings of the First International Workshop on Parallel Processing*, pages 322–330, Bangalore, India, December 1994.

[8] Uptal Banerjee. *Loop Transformations for restructuring compilers: The Foundations*. Kluwer Academic Publishers, Boston, 1993.

[9] Uptal Banerjee. *Loop Transformations for restructuring compilers: Loop parallelization*, volume II. Kluwer Academic Publishers, Boston, 1994.

[10] Stephen T. Barnard and Horst Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Technical Report RNR-92-033, NASA Ames Research Center, November 1992.

[11] Aart J. C. Bik and Harry A.G. Wijshoff. Annotations for a sparse compiler. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[12] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The Next Generation in Parallelizing Compilers. In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*, Ithica, NY, 1994. Springer-Verlag.

[13] William Blume and Rudolf Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[14] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-Linear Expressions. In *Proceedings of Supercomputing '94*, pages 528–537, Washington D.C., 1994.

[15] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results. *Proceedings of the 1993 ACM International Conference on Supercomputing*, July 1993.

[16] Ding-Kai Chen. *Compiler Optimizations for Parallel Loops with Fine-Grained Synchronizarion.* PhD thesis, CSRD, University of Illinois at Urbana-Champaign, 1994.

[17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1989.

[18] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect access to distributed arrays. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 152–168. Springer-Verlag, August 1993.

[19] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving linear systems on vector and shared memory computers.* SIAM, 1991.

[20] Kalluri Eswar, P. Sadayappan, and Chua-Huang Huang. Compile-time characterization of recurrent patterns in irregular computations. In *Proceedings of the 22nd International Conference on Parallel Processing*, volume II, pages 148–155, 1993.

[21] H. Franke, P. Hochschild, P. Pattnaik, and M. Snir. MPI-F: An Efficient Implementation of MPI on IBM-SP1. In *Proceedings of the 23rd International Conference on Parallel Processing*, pages III:197–201, St. Charles, IL, August 1994.

[22] K. Gallivan, W. Jalby, and D. Gannon. On the Problem of Optimizing Data Transfers for Complex Memory Systems. In *Proceedings of the Second ACM International Conference on Supercomputing*, Saint Malo, France, 1988.

[23] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable instrumented communication library, C reference manual. Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.

[24] John R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: implementation and experiments. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 418–427, April 1995.

[25] Milind Girkar and Constantine Polychronopoulos. Partitioning programs for parallel execution. In *Proceedings of the Second ACM International Conference on Supercomputing*, pages 216–229, 1988.

[26] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. Technical Report 996, CSRD, May 1990.

[27] William Gropp and Ewing Lusk. An abstract device definition to support the implementation of a high level point-to-point message passing interface. Technical Report MCS-P342-1193, Argonne National Laboratory, March 1994.

[28] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, September 1992. CRHC-92-19/UILU-ENG-92-2237.

[29] Mohammad Reza Haghighat. *Symbolic Analysis for Parallelizing Compilers*. PhD thesis, University of Illinois, Urbana-Champaign, IL, September 1994.

[30] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[31] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, September 1991.

[32] Michael T. Heath, Esmond Ng, and Barry W. Peyton. *Parallel Algorithms for Matrix Computations*, chapter Parallel Algorithms for Sparse Linear Systems, pages 83–125. SIAM, 1990.

[33] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Labs., 1993.

[34] High Performance Fortran Forum. High Performance Fortran Language Specification, version 1.1. Technical report, Center for Research on Parallel Computation, Rice University, Houston, TX, November 1994.

[35] R. Hockney and C. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms.* Adam Hilger, 1981.

[36] Mark T. Jones and Paul E. Plassmann. The efficient parallel iterative solution of large sparse linear systems. Technical Report MCS-P314-0692, MCS-Argonne National Laboratory, December 1992.

[37] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes. In *Proceedings of the 24th International Conference on Parallel Processing*, pages 113–122, August 1995.

[38] Peter M.W. Knijnenburg and Harry A.G. Wijshoff. On improving data locality in sparse matrix computations. Technical Report 94-15, Department of Computer Science, Leiden University, 1994.

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

[40] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

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

[42] A. Lain and P. Banerjee. Techniques to Overlap Computation and Communication in Irregular Iterative Applications. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 236–245, Manchester, U.K., July 1994.

[43] A. Lain and P. Banerjee. Exploiting Spatial Regularity in Irregular Iterative Applications. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 820–827, Santa Barbara, CA, April 1995.

[44] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and Exact Data Dependence Analysis. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 1–14, Toronto, Ontario, Canada, June 1991.

[45] Message-Passing Interface Forum. *Document for a Standard Message-Passing Interface*, 1994. Version 1.0.

[46] S. Midkiff and D. Padua. Compiler algorithms for synchronization. *IEEE Trans. Comput.*, C-36(12):1485–1495, 1987.

[47] E. Morel and C. Renvoise. Global optimization by supression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[48] Andreas Muller and Roland Ruhl. Extending high performance fortran for the support of unstructured computations. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 127–136, 1995.

[49] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984.

[50] Chao-Wei Ou, Manoj Gunwani, and Sanjay Ranka. Architecture-independent locality-improving transformations of computational graphs embedded in k-dimensions. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 289–297, July 1995.

[51] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten. Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors. In *Proceedings of the 18th International Conference on Parallel Processing*, pages II:39–48, St. Charles, IL, August 1989.

[52] Constantine D. Polychronopoulos. Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Trans. Comput.*, 37(8):991–1004, August 1988.

[53] Raja Ponnusamy and Joel Saltz. *A Manual for the CHAOS Runtime Library.* UMIACS University of Maryland, May 1994.

[54] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Proceedings Supercomputing '93*, pages 361–370. IEEE Computer Society Press, November 1993.

[55] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: an introduction.* Springer-Verlag, 1985.

[56] S. Ramaswamy and P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In *Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 342–349, McLean, VA, February 1995.

[57] Shankar Ramaswamy, Sachin Sapatnekar, and Prithviraj Banerjee. A Convex Programming Approach for Exploiting Data and Functional Parallelism. *Proceedings of the International Conference on Parallel Processing*, August 1994.

[58] Lawrence Rauchwerger. *Run–Time Parallelization: A Framework for Parallel Computation.* PhD thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1995.

[59] Lawrence Rauchwerger and David Padua. The privatizing doall test: A run-time technique for doall loop identification and array privatization. In *ACM International Conference on Supercomputing*, July 1994.

[60] Roland Ruhl. *A parallelizing compiler for distributed memory parallel processors.* PhD thesis, ETH Zurich, 1994.

[61] Youcef Saad. *SPARSKIT: a basic tool kit for sparse matrix computations.* RIACS.

[62] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time Scheduling and Execution of Loops on Message Passing Machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[63] Horst D. Simon, editor. *Parallel Computational Fluid Dynamics.* The MIT Press, 1992.

[64] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Advanced Compilation Techniques in the PARADIGM compiler for Distributed Memory Multicomputers. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 424–433, Barcelona, Spain, July 1995.

[65] E. Su, D. J. Palermo, and P. Banerjee. Processor Tagged Descriptors: A Data Structure for Compiling for Distributed-Memory Multicomputers. In *Proceedings of the 1994 International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Montréal, Canada, August 1994.

[66] Frank Tip. A survey of program slicing techniques. Technical Report CS-R9438, CWI, Amsterdam, 1994.

[67] Manuel Ujaldon and Emilio L. Zapata. Efficient resolution of sparse indirections in data-parallel compilers. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 117–126, July 1995.

[68] Manuel Ujaldon, Emilio L. Zapata, Barbara Chapman, and Hans Zima. New data-parallel language features for sparse matrix computations. In *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[69] Reinhard v. Hanxleden and Ken Kennedy. Give-N-Take: A balanced code placement framework. In *ACM SIGPLAN '94 Program Language Design and Implementation*, June 1994.

[70] Reinhard von Hanxleden. *Compiler Support for Machine-Independent Parallelization of Irregular Problems.* PhD thesis, Rice University, December 1994.

[71] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Computer Systems Laboratory, Stanford University, Stanford, CA, August 1992. CSL-TR-92-538.

[72] Janet Wu, Joel Saltz, Harry Berryman, and Seema Hiranandani. Distributed memory compiler design for sparse problems. Technical Report 91-13, ICASE, January 91.

[73] Yang Zeng and Santosh G. Abraham. Partitioning regular grid applications with irregular boundaries for cache-coherent multiprocessors. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 222–228, April 1995.

[74] C. Zhu and P. C. Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Trans. Softw. Eng.*, 13(6):726–739, June 1987.

APPENDIX A

AMUX

In this appendix we want to show the actual output of our compiler when applied to one of the benchmarks. We also show code generated using a fully enumerated characterization and the one in which the compiler selects appropriate representations.

A.1 Sequential

```
      SUBROUTINE AMUX (LOW,HIGH,
     $     X, Y, A,JA,IA)
      DOUBLE PRECISION  X(0:*), Y(0:*), A(0:*), T
      INTEGER JA(0:*), IA(0:*),LOW,HIGH

      DO  I = LOW,HIGH
         K1 = IA(I)
         K2 = IA(I+1) -1
         T = 0.0D0
         DO  K=K1, K2
            T = T + A(K)*X(JA(K))
         ENDDO
         Y(I) = T
      ENDDO
      END
```

A.2 Enumeration

```
       SUBROUTINE AMUX(NITER,NET,TTROW,TTROWPLUS,TTNNZ,LOW,HIGH,X,Y,A,
     $     JA,IA)
       INTEGER COUNT_22,COUNT_21, COUNT_20, COUNT_19, COUNT_18, COUNT_17
     $      , COUNT_16, COUNT_15, COUNT_14, COUNT_13, COUNT_12, COUNT_11
     $      , COUNT_10, COUNT_9, SCHED_2, COUNT_8, COUNT_7, SIZESLICE_4,
     $      REFLOCOFF_4, REFLOC_4, REFGLOBOFF_4, REFGLOB_4, CB_2,
     $      LOCALSIZE_1, SCHED_1, COUNT_6, COUNT_5, SIZESLICE_3,
     $      REFLOCOFF_3, REFLOC_3, REFGLOBOFF_3, REFGLOB_3, CB_1, CB_0,
     $      LOCALSIZE_0, SCHED_0, COUNT_4, COUNT_3, SIZESLICE_2,
     $      REFLOCOFF_2, REFLOC_2, REFGLOBOFF_2, REFGLOB_2, CB,
     $      LOCALSIZE, SCHED, COUNT_2, REFLOCOFF_1, REFLOC_1,
     $      REFGLOBOFF_1, REFGLOB_1, COUNT_1, REFLOCOFF_0, REFLOC_0,
     $      REFGLOBOFF_0, REFGLOB_0, COUNT_0, SIZESLICE_1, COUNT,
     $      SIZESLICE_0, SIZESLICE, POINTERS(0:2), REFLOCOFF, REFLOC,
```

```
     $      REFGLOBOFF, REFGLOB, NOUT, MYID, NUMNODES, WORKINT(0:1), NET
     $      , TEMP_3, TEMP_2, TEMP_1, TEMP_0, TEMP
      DOUBLE PRECISION X(0:*)
      DOUBLE PRECISION Y(0:*)
      DOUBLE PRECISION A(0:*)
      DOUBLE PRECISION T
      INTEGER JA(0:*)
      INTEGER IA(0:*)
      INTEGER LOW, HIGH, TTROW, TTROWPLUS, TTNNZ, NET
      INTEGER I, K1, K2, K,TIME1,NITER,STEP


      CALL NETWORKINFO(NET,NUMNODES,MYID)
C  ADDED
      CALL NETWORKBARRIER(NET)
      CALL NEWHOWLONG(TIME1,MYID)
C END ADDED
      CALL GETLOCALARRAYSIZE(TTNNZ,MYID,LOCALSIZE_0)
      CALL GETLOCALARRAYSIZE(TTROWPLUS,MYID,LOCALSIZE)
      CALL GETLOCALARRAYSIZE(TTROW,MYID,LOCALSIZE_1)
      POINTERS(0) = 0
      COUNT = 0
      IF (1 .LE. 1 + (HIGH + ( -LOW))) THEN
        COUNT = 1 + (HIGH + ( -LOW)) + COUNT
      END IF
      SIZESLICE_0 = COUNT
      POINTERS(1) = POINTERS(0) + SIZESLICE_0
      COUNT_0 = 0
      IF (1 .LE. 1 + (HIGH + ( -LOW))) THEN
        COUNT_0 = 1 + (HIGH + ( -LOW)) + COUNT_0
      END IF
      SIZESLICE_1 = COUNT_0
      POINTERS(2) = POINTERS(1) + SIZESLICE_1
      CALL NEWCPIRENUALLOC(REFGLOB,POINTERS(2))
      CALL GETCPIRENUVAL(REFGLOBOFF,SIZESLICE,REFGLOB,WORKINT)
      REFGLOBOFF_0 = POINTERS(0) + REFGLOBOFF
      COUNT_1 = 0
      DO I = 1,1 + (HIGH + ( -LOW))
        TEMP = (( -1) + (LOW + I))
        WORKINT(COUNT_1 + REFGLOBOFF_0) = TEMP
        COUNT_1 = 1 + COUNT_1
      END DO
      REFGLOBOFF_1 = POINTERS(1) + REFGLOBOFF
      COUNT_2 = 0
      DO I = 1,1 + (HIGH + ( -LOW))
```

```
    TEMP_0 = (LOW + I)
    WORKINT(COUNT_2 + REFGLOBOFF_1) = TEMP_0
    COUNT_2 = 1 + COUNT_2
  END DO
  CALL BUILDSCHEDLOC(SCHED,NET,REFGLOB,TTROWPLUS,REFLOC,NOUT,LOCALS
1 IZE)
  CALL GETCPIRENUVAL(REFLOCOFF,SIZESLICE,REFLOC,WORKINT)
  CALL NEWCOMMBUFFERINT(CB,NET,SCHED)
  CALL XCHANGEINT(CB,IA,IA)
  COUNT_3 = 0
  COUNT_9 = 0
  COUNT_10 = 0
  DO I = 1,1 + (HIGH + ( -LOW))
    TEMP = (( -1) + (LOW + I))
    K1 = IA(WORKINT(REFLOCOFF + (COUNT_9 + POINTERS(0))))
    COUNT_9 = 1 + COUNT_9
    TEMP_0 = (LOW + I)
    K2 = (( -1) + IA(WORKINT(REFLOCOFF + (COUNT_10 + POINTERS(1))))
1   )
    COUNT_10 = 1 + COUNT_10
    IF (1 .LE. 1 + (( -K1) + K2)) THEN
      COUNT_3 = 1 + (( -K1) + K2) + COUNT_3
    END IF
  END DO
  SIZESLICE_2 = COUNT_3
  CALL NEWCPIRENUALLOC(REFGLOB_2,SIZESLICE_2)
  CALL GETCPIRENUVAL(REFGLOBOFF_2,SIZESLICE_2,REFGLOB_2,WORKINT)
  COUNT_4 = 0
  COUNT_11 = 0
  COUNT_12 = 0
  DO I = 1,1 + (HIGH + ( -LOW))
    TEMP = (( -1) + (LOW + I))
    K1 = IA(WORKINT(REFLOCOFF + (COUNT_11 + POINTERS(0))))
    COUNT_11 = 1 + COUNT_11
    TEMP_0 = (LOW + I)
    K2 = (( -1) + IA(WORKINT(REFLOCOFF + (COUNT_12 + POINTERS(1))))
1   )
    COUNT_12 = 1 + COUNT_12
    DO K = 1,1 + (( -K1) + K2)
      TEMP_1 = (( -1) + (K + K1))
      WORKINT(COUNT_4 + REFGLOBOFF_2) = TEMP_1
      COUNT_4 = 1 + COUNT_4
    END DO
  END DO
  CALL BUILDSCHEDLOC(SCHED_0,NET,REFGLOB_2,TTNNZ,REFLOC_2,NOUT,LOCA
```

```
1 LSIZE_0)
  CALL GETCPIRENUVAL(REFLOCOFF_2,SIZESLICE_2,REFLOC_2,WORKINT)
  CALL NEWCOMMBUFFERINT(CB_0,NET,SCHED_0)
  CALL NEWCOMMBUFFERDOUBLE(CB_1,NET,SCHED_0)
  CALL XCHANGEINT(CB_0,JA,JA)
  COUNT_5 = 0
  COUNT_13 = 0
  COUNT_14 = 0
  DO I = 1,1 + (HIGH + ( -LOW))
    TEMP = (( -1) + (LOW + I))
    K1 = IA(WORKINT(REFLOCOFF + (COUNT_13 + POINTERS(0))))
    COUNT_13 = 1 + COUNT_13
    TEMP_0 = (LOW + I)
    K2 = (( -1) + IA(WORKINT(REFLOCOFF + (COUNT_14 + POINTERS(1))))
1   )
    COUNT_14 = 1 + COUNT_14
    IF (1 .LE. 1 + (( -K1) + K2)) THEN
      COUNT_5 = 1 + (( -K1) + K2) + COUNT_5
    END IF
  END DO
  SIZESLICE_3 = COUNT_5
  CALL NEWCPIRENUALLOC(REFGLOB_3,SIZESLICE_3)
  CALL GETCPIRENUVAL(REFGLOBOFF_3,SIZESLICE_3,REFGLOB_3,WORKINT)
  COUNT_6 = 0
  COUNT_15 = 0
  COUNT_16 = 0
  COUNT_17 = 0
  DO I = 1,1 + (HIGH + ( -LOW))
    TEMP = (( -1) + (LOW + I))
    K1 = IA(WORKINT(REFLOCOFF + (COUNT_15 + POINTERS(0))))
    COUNT_15 = 1 + COUNT_15
    TEMP_0 = (LOW + I)
    K2 = (( -1) + IA(WORKINT(REFLOCOFF + (COUNT_16 + POINTERS(1))))
1   )
    COUNT_16 = 1 + COUNT_16
    DO K = 1,1 + (( -K1) + K2)
      TEMP_1 = (( -1) + (K + K1))
      TEMP_2 = JA(WORKINT(REFLOCOFF_2 + COUNT_17))
      COUNT_17 = 1 + COUNT_17
      WORKINT(COUNT_6 + REFGLOBOFF_3) = TEMP_2
      COUNT_6 = 1 + COUNT_6
    END DO
  END DO
  CALL BUILDSCHEDLOC(SCHED_1,NET,REFGLOB_3,TTROW,REFLOC_3,NOUT,LOCA
1 LSIZE_1)
```

```
      CALL GETCPIRENUVAL(REFLOCOFF_3,SIZESLICE_3,REFLOC_3,WORKINT)
      CALL NEWCOMMBUFFERDOUBLE(CB_2,NET,SCHED_1)

      CALL XCHANGEDOUBLE(CB_1,A,A)
      COUNT_7 = 0
      IF (1 .LE. 1 + (HIGH + ( -LOW))) THEN
        COUNT_7 = 1 + (HIGH + ( -LOW)) + COUNT_7
      END IF
      SIZESLICE_4 = COUNT_7
      CALL NEWCPIRENUALLOC(REFGLOB_4,SIZESLICE_4)
      CALL GETCPIRENUVAL(REFGLOBOFF_4,SIZESLICE_4,REFGLOB_4,WORKINT)
      COUNT_8 = 0
      DO I = 1,1 + (HIGH + ( -LOW))
        TEMP_3 = (( -1) + (LOW + I))
        WORKINT(COUNT_8 + REFGLOBOFF_4) = TEMP_3
        COUNT_8 = 1 + COUNT_8
      END DO
      CALL BUILDSCHEDLOC(SCHED_2,NET,REFGLOB_4,TTROW,REFLOC_4,NOUT,LOCA
     1 LSIZE_1)
      CALL GETCPIRENUVAL(REFLOCOFF_4,SIZESLICE_4,REFLOC_4,WORKINT)
C     ADDED
      CALL NETWORKBARRIER(NET)
      CALL SETHOWLONG(TIME1)
C     END ADDED

      DO STEP  =1, NITER
      CALL XCHANGEDOUBLE(CB_2,X,X)
      COUNT_18 = 0
      COUNT_19 = 0
      COUNT_20 = 0
      COUNT_21 = 0
      COUNT_22 = 0
      DO I = 1,1 + (HIGH + ( -LOW))
        TEMP = (( -1) + (LOW + I))
        K1 = IA(WORKINT(REFLOCOFF + (COUNT_18 + POINTERS(0))))
        COUNT_18 = 1 + COUNT_18
        TEMP_0 = (LOW + I)
        K2 = (( -1) + IA(WORKINT(REFLOCOFF + (COUNT_19 + POINTERS(1))))
     1   )
        COUNT_19 = 1 + COUNT_19
        T = 0.0D0
        DO K = 1,1 + (( -K1) + K2)
          T = (T + (X(WORKINT(REFLOCOFF_3 + COUNT_21)) * A(WORKINT(REFL
     1     OCOFF_2 + COUNT_21))))
          COUNT_21 = 1 + COUNT_21
```

```
    END DO
    Y(WORKINT(REFLOCOFF_4 + COUNT_22)) = T
    COUNT_22 = 1 + COUNT_22
  END DO
 ENDDO
 CALL NETWORKBARRIER(NET)
 CALL DELETEHOWLONG(TIME1)

 END
```

A.3 Intervals

```
      SUBROUTINE AMUX(NITER,NET,TTROW,TTROWPLUS,TTNNZ,LOW,HIGH,
     $     X,Y,A,JA,IA)
      INTEGER COUNT_23,I_6, COUNT_22, COUNT_21, I_5, COUNT_20, COUNT_19
     $      , I_4, COUNT_18, COUNT_17, I_3, COUNT_16, COUNT_15, I_2,
     $      COUNT_14, COUNT_13, I_1, COUNT_12, COUNT_11, I_0, COUNT_10,
     $      COUNT_9, SCHED_2, COUNT_8, COUNT_7, SIZESLICE_4, INTEROFF_3,
     $      REFLOCOFF_4, REFLOC_4, REFGLOBOFF_4, REFGLOB_4, CB_2,
     $      LOCALSIZE_1, SCHED_1, COUNT_6, COUNT_5, SIZESLICE_3,
     $      REFLOCOFF_3, REFLOC_3, REFGLOBOFF_3, REFGLOB_3, CB_1, CB_0,
     $      LOCALSIZE_0, SCHED_0, COUNT_4, COUNT_3, SIZESLICE_2,
     $      INTEROFF_2, REFLOCOFF_2, REFLOC_2, REFGLOBOFF_2, REFGLOB_2,
     $      CB, LOCALSIZE, SCHED, COUNT_2, INTEROFF_1, REFLOCOFF_1,
     $      REFLOC_1, REFGLOBOFF_1, REFGLOB_1, COUNT_1, INTEROFF_0,
     $      REFLOCOFF_0, REFLOC_0, REFGLOBOFF_0, REFGLOB_0, COUNT_0,
     $      SIZESLICE_1, COUNT, SIZESLICE_0, SIZESLICE, INTEROFF,
     $      POINTERS(0:2), REFLOCOFF, REFLOC, REFGLOBOFF, REFGLOB, NOUT,
     $      MYID, NUMNODES, WORKINT(0:1), NET, TEMP_3, TEMP_2, TEMP_1,
     $      TEMP_0, TEMP
      DOUBLE PRECISION X(0:*)
      DOUBLE PRECISION Y(0:*)
      DOUBLE PRECISION A(0:*)
      DOUBLE PRECISION T
      INTEGER JA(0:*)
      INTEGER IA(0:*)
      INTEGER LOW, HIGH, TTROW, TTROWPLUS, TTNNZ, NET
      INTEGER I, K1, K2, K,TIME1,NITER,STEP


      CALL NETWORKINFO(NET,NUMNODES,MYID)
C  ADDED
      CALL NETWORKBARRIER(NET)
      CALL NEWHOWLONG(TIME1,MYID)
C END ADDED
```

```
 CALL GETLOCALARRAYSIZE(TTNNZ,MYID,LOCALSIZE_0)
CALL GETLOCALARRAYSIZE(TTROWPLUS,MYID,LOCALSIZE)
CALL GETLOCALARRAYSIZE(TTROW,MYID,LOCALSIZE_1)
CALL GETLOCALARRAYSIZE(TTROW,MYID,LOCALSIZE_1)
POINTERS(0) = 0
COUNT = 0
IF (1 .LE. 1 + (HIGH + ( -LOW))) THEN
  COUNT = 1 + COUNT
END IF
SIZESLICE_0 = COUNT
POINTERS(1) = POINTERS(0) + SIZESLICE_0
COUNT_0 = 0
IF (1 .LE. 1 + (HIGH + ( -LOW))) THEN
  COUNT_0 = 1 + COUNT_0
END IF
SIZESLICE_1 = COUNT_0
POINTERS(2) = POINTERS(1) + SIZESLICE_1
CALL NEWCPIRINTALLOC(REFGLOB,POINTERS(2))
CALL GETCPIRINTVAL(REFGLOBOFF,SIZESLICE,REFGLOB,WORKINT)
REFGLOBOFF_0 = 2 * POINTERS(0) + REFGLOBOFF
COUNT_1 = 0
IF (1 .LE. 1 + (HIGH + ( -LOW))) THEN
  TEMP = LOW
  WORKINT(COUNT_1 + REFGLOBOFF_0) = TEMP
  TEMP = HIGH
  WORKINT(1 + COUNT_1 + REFGLOBOFF_0) = 1 + TEMP
  COUNT_1 = 2 + COUNT_1
END IF
REFGLOBOFF_1 = 2 * POINTERS(1) + REFGLOBOFF
COUNT_2 = 0
IF (1 .LE. 1 + (HIGH + ( -LOW))) THEN
  TEMP_0 = (1 + LOW)
  WORKINT(COUNT_2 + REFGLOBOFF_1) = TEMP_0
  TEMP_0 = (1 + HIGH)
  WORKINT(1 + COUNT_2 + REFGLOBOFF_1) = 1 + TEMP_0
  COUNT_2 = 2 + COUNT_2
END IF
CALL BUILDSCHEDLOCINTAGG(SCHED,NET,REFGLOB,TTROWPLUS,REFLOC,INTER
1 OFF,NOUT,LOCALSIZE,2,POINTERS,WORKINT)
CALL GETCPIRINTVAL(REFLOCOFF,SIZESLICE,REFLOC,WORKINT)
CALL NEWCOMMBUFFERINT(CB,NET,SCHED)
CALL XCHANGEINT(CB,IA,IA)
COUNT_3 = 0
COUNT_9 = 0
COUNT_10 = 0
```

```
  DO I_0 = 1,WORKINT(INTEROFF + COUNT_10)
    DO I = 0,WORKINT(REFLOCOFF + 1 + COUNT_9) - WORKINT(REFLOCOFF +
1     COUNT_9) - 1
      TEMP = (( -1) + (LOW + I))
      K1 = IA(I + WORKINT(REFLOCOFF + (COUNT_9 + (2 * POINTERS(0)))
1      ))
      TEMP_0 = (LOW + I)
      K2 = (( -1) + IA(I + WORKINT(REFLOCOFF + (COUNT_9 + (2 * POIN
1      TERS(1))))))
      IF (1 .LE. 1 + (( -K1) + K2)) THEN
        COUNT_3 = 1 + COUNT_3
      END IF
    END DO
    COUNT_9 = 2 + COUNT_9
  END DO
  COUNT_10 = 1 + COUNT_10
  SIZESLICE_2 = COUNT_3
  CALL NEWCPIRINTALLOC(REFGLOB_2,SIZESLICE_2)
  CALL GETCPIRINTVAL(REFGLOBOFF_2,SIZESLICE_2,REFGLOB_2,WORKINT)
  COUNT_4 = 0
  COUNT_11 = 0
  COUNT_12 = 0
  DO I_1 = 1,WORKINT(INTEROFF + COUNT_12)
    DO I = 0,WORKINT(REFLOCOFF + 1 + COUNT_11) - WORKINT(REFLOCOFF
1    + COUNT_11) - 1
      TEMP = (( -1) + (LOW + I))
      K1 = IA(I + WORKINT(REFLOCOFF + (COUNT_11 + (2 * POINTERS(0))
1      )))
      TEMP_0 = (LOW + I)
      K2 = (( -1) + IA(I + WORKINT(REFLOCOFF + (COUNT_11 + (2 * POI
1      NTERS(1))))))
      IF (1 .LE. 1 + (( -K1) + K2)) THEN
        TEMP_1 = K1
        WORKINT(COUNT_4 + REFGLOBOFF_2) = TEMP_1
        TEMP_1 = K2
        WORKINT(1 + COUNT_4 + REFGLOBOFF_2) = 1 + TEMP_1
        COUNT_4 = 2 + COUNT_4
      END IF
    END DO
    COUNT_11 = 2 + COUNT_11
  END DO
  COUNT_12 = 1 + COUNT_12
  CALL BUILDSCHEDLOCINT(SCHED_0,NET,REFGLOB_2,TTNNZ,REFLOC_2,INTERO
1 FF_2,NOUT,LOCALSIZE_0,WORKINT)
  CALL GETCPIRINTVAL(REFLOCOFF_2,SIZESLICE_2,REFLOC_2,WORKINT)
```

```
  CALL NEWCOMMBUFFERINT(CB_0,NET,SCHED_0)
  CALL NEWCOMMBUFFERDOUBLE(CB_1,NET,SCHED_0)
  CALL XCHANGEINT(CB_0,JA,JA)
  COUNT_5 = 0
  COUNT_13 = 0
  COUNT_14 = 0
  DO I_2 = 1,WORKINT(INTEROFF + COUNT_14)
    DO I = 0,WORKINT(REFLOCOFF + 1 + COUNT_13) - WORKINT(REFLOCOFF
1   + COUNT_13) - 1
      TEMP = (( -1) + (LOW + I))
      K1 = IA(I + WORKINT(REFLOCOFF + (COUNT_13 + (2 * POINTERS(0))
1     )))
      TEMP_0 = (LOW + I)
      K2 = (( -1) + IA(I + WORKINT(REFLOCOFF + (COUNT_13 + (2 * POI
1     NTERS(1))))))
      IF (1 .LE. 1 + (( -K1) + K2)) THEN
        COUNT_5 = 1 + (( -K1) + K2) + COUNT_5
      END IF
    END DO
    COUNT_13 = 2 + COUNT_13
  END DO
  COUNT_14 = 1 + COUNT_14
  SIZESLICE_3 = COUNT_5
  CALL NEWCPIRENUALLOC(REFGLOB_3,SIZESLICE_3)
  CALL GETCPIRENUVAL(REFGLOBOFF_3,SIZESLICE_3,REFGLOB_3,WORKINT)
  COUNT_6 = 0
  COUNT_15 = 0
  COUNT_17 = 0
  COUNT_18 = 0
  COUNT_16 = 0
  DO I_3 = 1,WORKINT(INTEROFF + COUNT_16)
    DO I = 0,WORKINT(REFLOCOFF + 1 + COUNT_15) - WORKINT(REFLOCOFF
1   + COUNT_15) - 1
      TEMP = (( -1) + (LOW + I))
      K1 = IA(I + WORKINT(REFLOCOFF + (COUNT_15 + (2 * POINTERS(0))
1     )))
      TEMP_0 = (LOW + I)
      K2 = (( -1) + IA(I + WORKINT(REFLOCOFF + (COUNT_15 + (2 * POI
1     NTERS(1))))))
      DO I_4 = 1,WORKINT(INTEROFF_2 + COUNT_18)
        DO K = 0,WORKINT(REFLOCOFF_2 + 1 + COUNT_17) - WORKINT(REFL
1       OCOFF_2 + COUNT_17) - 1
          TEMP_1 = (( -1) + (K + K1))
          TEMP_2 = JA(K + WORKINT(REFLOCOFF_2 + COUNT_17))
          WORKINT(COUNT_6 + REFGLOBOFF_3) = TEMP_2
```

```
            COUNT_6 = 1 + COUNT_6
          END DO
          COUNT_17 = 2 + COUNT_17
        END DO
        COUNT_18 = 1 + COUNT_18
      END DO
      COUNT_15 = 2 + COUNT_15
    END DO
    COUNT_16 = 1 + COUNT_16
    CALL BUILDSCHEDLOC(SCHED_1,NET,REFGLOB_3,TTROW,REFLOC_3,NOUT,LOCA
  1 LSIZE_1)
    CALL GETCPIRENUVAL(REFLOCOFF_3,SIZESLICE_3,REFLOC_3,WORKINT)
    CALL NEWCOMMBUFFERDOUBLE(CB_2,NET,SCHED_1)

    CALL XCHANGEDOUBLE(CB_1,A,A)
    COUNT_7 = 0
    IF (1 .LE. 1 + (HIGH + ( -LOW))) THEN
      COUNT_7 = 1 + COUNT_7
    END IF
    SIZESLICE_4 = COUNT_7
    CALL NEWCPIRINTALLOC(REFGLOB_4,SIZESLICE_4)
    CALL GETCPIRINTVAL(REFGLOBOFF_4,SIZESLICE_4,REFGLOB_4,WORKINT)
    COUNT_8 = 0
    IF (1 .LE. 1 + (HIGH + ( -LOW))) THEN
      TEMP_3 = LOW
      WORKINT(COUNT_8 + REFGLOBOFF_4) = TEMP_3
      TEMP_3 = HIGH
      WORKINT(1 + COUNT_8 + REFGLOBOFF_4) = 1 + TEMP_3
      COUNT_8 = 2 + COUNT_8
    END IF
    CALL BUILDSCHEDLOCINT(SCHED_2,NET,REFGLOB_4,TTROW,REFLOC_4,INTERO
  1 FF_3,NOUT,LOCALSIZE_1,WORKINT)
    CALL GETCPIRINTVAL(REFLOCOFF_4,SIZESLICE_4,REFLOC_4,WORKINT)
    CALL MERGELOCALINTAGG(REFLOC_4,WORKINT(INTEROFF_3),REFLOC_4,REFLO
  1 C,WORKINT(INTEROFF),REFLOC,2,POINTERS,WORKINT)
    CALL GETCPIRINTVAL(REFLOCOFF_4,SIZESLICE_4,REFLOC_4,WORKINT)
    CALL GETCPIRINTVAL(REFLOCOFF,SIZESLICE,REFLOC,WORKINT)
C      ADDED
    CALL NETWORKBARRIER(NET)
    CALL SETHOWLONG(TIME1)
C      END ADDED

    DO STEP  =1, NITER
       CALL XCHANGEDOUBLE(CB_2,X,X)
     COUNT_19 = 0
```

```
  COUNT_21 = 0
  COUNT_23 = 0
  COUNT_22 = 0
  COUNT_20 = 0
  DO I_5 = 1,WORKINT(INTEROFF + COUNT_20)
    DO I = 0,WORKINT(REFLOCOFF + 1 + COUNT_19) - WORKINT(REFLOCOFF
1   + COUNT_19) - 1
      TEMP = (( -1) + (LOW + I))
      K1 = IA(I + WORKINT(REFLOCOFF + (COUNT_19 + (2 * POINTERS(0))
1     )))
      TEMP_0 = (LOW + I)
      K2 = (( -1) + IA(I + WORKINT(REFLOCOFF + (COUNT_19 + (2 * POI
1     NTERS(1))))))
      T = 0.0D0
      DO I_6 = 1,WORKINT(INTEROFF_2 + COUNT_22)
        DO K = 0,WORKINT(REFLOCOFF_2 + 1 + COUNT_21) - WORKINT(REFL
1       OCOFF_2 + COUNT_21) - 1
          T = (T + (X(WORKINT(REFLOCOFF_3 + COUNT_23)) * A(K + WORK
1         INT(REFLOCOFF_2 + COUNT_21))))
          COUNT_23 = 1 + COUNT_23
        END DO
        COUNT_21 = 2 + COUNT_21
      END DO
      COUNT_22 = 1 + COUNT_22
      Y(I + WORKINT(REFLOCOFF_4 + COUNT_19)) = T
    END DO
    COUNT_19 = 2 + COUNT_19
  END DO
  COUNT_20 = 1 + COUNT_20
 ENDDO
 CALL NETWORKBARRIER(NET)
 CALL DELETEHOWLONG(TIME1)
 END
```

VITA

Antonio Lain received the Ingeniero Superior degree in Electrical Engineering from the ETSIT, Universidad Politecnica de Madrid, Spain in 1990. He received the M.Sc. degree in Computer Science from the University of Edinburgh, Scotland, in 1991. He is currently a candidate for the Ph.D. degree in Computer Science at the University of Illinois at Urbana-Champaign.

After completing his doctoral dissertation, Mr. Lain will join Hewlett-Packard, Barcelona, Spain. His research interests include high performance computing and irregular applications.