Exploiting Spatial Regularity in Irregular Iterative Applications

Antonio Lain *

Prithviraj Banerjee

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign

Abstract

The increasing gap between the speed of microprocessors and memory subsystems makes it imperative to exploit locality of reference in sequential irregular applications. The parallelization of such applications requires special considerations. Current RTS (Run-Time Support) for irregular computations fail to exploit the fine grain regularity present in these applications, producing unnecessary time and memory overheads. PILAR (Parallel Irregular Library with Application of Regularity) is a new RTS for irregular computations that provides a variety of internal representations of communication patterns based on their regularity; allowing for the efficient support of a wide spectrum of regularity under a common framework. Experimental results on the IBM SP-1 and Intel Paragon demonstrate the validity of our approach.

1 Introduction

There are many important applications in computational fluid dynamics, circuit simulation and structural analysis that can be accurately modeled using iterations on unstructured grids [1]. Figure 1 shows an example of an unstructured grid. Typical applications associate values to 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 [2]. For this reason, representations that layout contiguously in memory all the edges associated with a particular node are commonly used. Figure 2 shows a contiguous representation of the grid in Figure 1. The advantage of

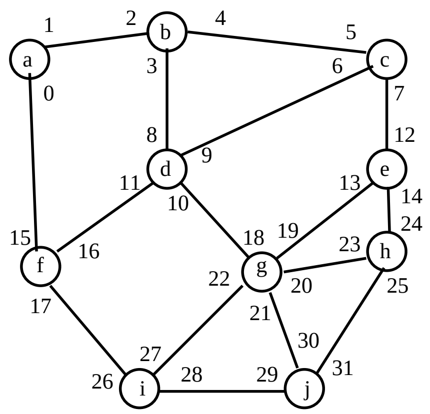

Figure 1: An example of an unstructured grid

this layout comes into play when the iterative algorithm traverses all the edges of a particular node in sequence. In this case spatial locality of accesses improves the performance of the memory hierarchy. A typical algorithm using spatial locality can be seen in Figure 3. Here, references to the array y will first have an irregular access $y[index[i]]$ followed by a sequence of $numEdges[i]$ contiguous accesses.

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 (Massively Parallel Processors) 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 simplifies the parallelization of irregular codes with spatial locality of access.

As it will be shown in Section 5, state of the art run time support for irregular computations on MPP like CHAOS/PARTI [3] fail to exploit this spatial 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 irregu-

*This research was supported in part by the Advanced Research Projects Agency under Contract DAAH04-94-G-0273, administered by the Army Research Office, in part by NASA under contract NASA NAS 1-613, and in part by a Fulbright/MEC fellowship.

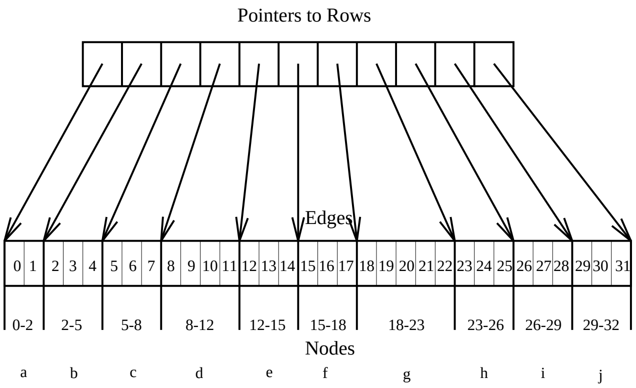

Figure 2: Contiguous layout of an unstructured grid

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

Figure 3: Sweep with spatial locality

lar RTS and to give the basic irregular framework to PARADIGM [4], a HPF/Fortran-D 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. In particular, 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 [5] 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 [6]. These primitives are supported in PILAR in a very efficient way.

The rest of the paper is organized as follows. Section 2 describes related work in more detail. Section 3 discusses MPI support for irregular computations and explains the relationship between PILAR and MPI. Section 4 describes the main components of PILAR. Section 5 shows some experimental results on the IBM SP-1 and Intel Paragon multicomputers evaluating the library performance. Section 6 contains our conclusions and some discussion of our future research directions.

2 Related work

The CHAOS library [3], and in particular, the original PARTI library [7], had a significant impact in the design of PILAR. In particular, 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. Section 4 discusses in more detail the differences and similarities between them.

LPARX [8] 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.

Multiblock PARTI [9] provides support for block-structured applications with regular decompositions. Unknown problem sizes could 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 limited to regular communication patterns.

3 Irregular computations using MPI

MPI [5] 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 what these hooks are and how PILAR uses them.

MPI supports derived data types in point-to-point and collective communications. A derived data type 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 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.

PILAR uses MPI derived data types in the following way: 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 representation 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.

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 tags of messages 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 [10].

4 PILAR

4.1 Overview

In order to parallelize applications that have input-dependent communication patterns on a MPP, run-time compilation techniques [7] 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 [3] or PILAR provide primitives for these operations.

An important subset of irregular applications have spatial locality. Figure 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 reduces the memory overhead and speeds-up these operations dramatically as it will be shown in Section 5.

Although the degree of spatial regularity needed to benefit from an intervals representation in PILAR is very small (see Section 5), patterns with almost no spatial regularity are better represented with an enumerated representation (similar to PARTI). PILAR also provides support for these patterns using a different representation and, as it will be explained in the next section, it allows to optimize together patterns with different representations.

4.2 Communication Pattern Internal Representation (CPIR)

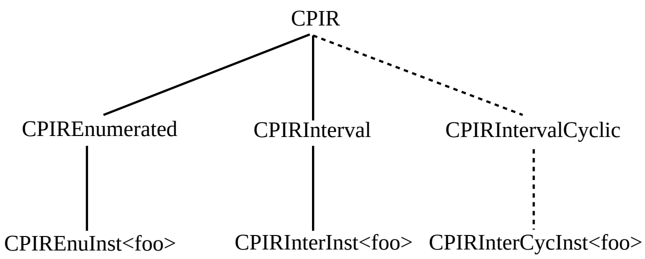

Figure 5: CPIR Hierarchy

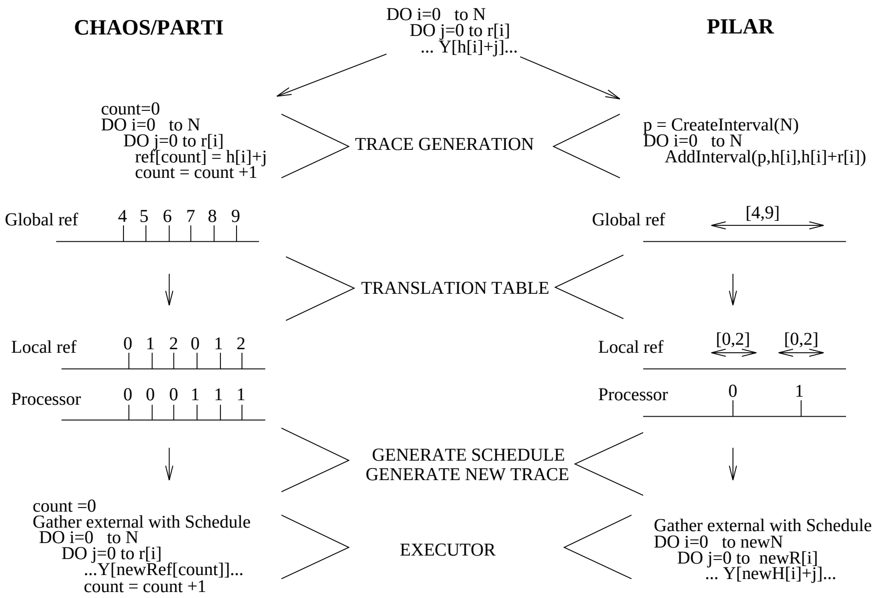

Figure 4: Using spatial regularity

CPIR is a C++ abstract base class that unifies the interface of our representations of communication patterns. Figure 5 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. CPIR provides 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. Also, as it will be shown in Section 5, the granularity of these CPIR operations was carefully chosen to amortize the extra overhead of dynamic dispatching.

A brief description of the derived CPIR classes follows.

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.

CPIRInterval Intervals are used to represent irregular communication patterns with spatial locality in a compact manner. Efficient computational geometry algorithms [11] are used to perform basic operations like union, intersection and difference of array of intervals.

CPIRIntervalCyclic Currently under development, this representation uses a collection of CPIRInterval to represent patterns that have spatial locality with small stride. This representation benefits from recent work on handling block cyclic distributions in PARADIGM [12].

4.3 Schedules

A *Schedule* is an object that represents a communication pattern. 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.

In PILAR, schedules that are compatible can be composed to obtain more complex ones using a set of **local** operations. Two schedules are compatible

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

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. Operations currently supported include **Union** (merge two schedules and remove duplicates), **Intersection** (find duplicates between two schedules), **Difference** (obtain an incremental schedule), **Expansion** (increase the size of the elementary type) and **Replication** (repeat the pattern with an offset).

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.

4.4 Network Interface

A *Network* object encapsulates all the machine dependent code of the library, providing basic access to communication and timing routines. Currently PILAR directly supports Intel Nx, IBM EUIH and a uniprocessor simulator communication libraries. Also, a *NetworkMPI* object provides wrappers to the MPI C interface.

4.5 Global communication primitives

PILAR supports blocking and non-blocking global communication primitives [6] 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.

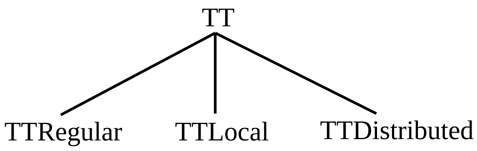

Figure 6: Translation Table Hierarchy

4.6 Translation Tables (TT)

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 6 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.

Regular Translation Tables (RTT) They support regular HPF block, cyclic and block-cyclic mappings with analytical expressions.

Local Translation Tables (LTT) They are used for tables that need to be only locally consistent or they are replicated in every processor.

Distributed Translation Tables (DTT) A 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 [2]. Note the increase in the complexity of implementation when intervals are used in queries, because a single interval query could involve several processors.

5 Experimental results

Two experiments were performed to evaluate the effectiveness of the PILAR approach in exploiting spatial regularity of irregular applications. In both cases we wanted to measure the overhead of redistributing

[2] this index table can be of any type, even distributed.

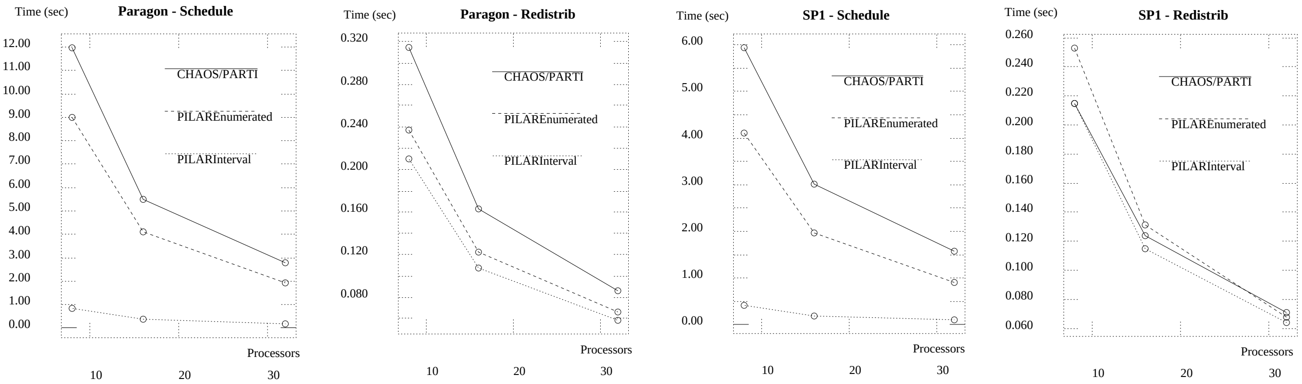

Figure 7: Edge redistribution on the Intel Paragon and IBM SP1

the edges of an unstructured grid after a partitioner has assigned nodes to processors. We assume an initial layout in which edges of a particular node are contiguous in memory. Redistribution is done in two phases: in the first one we compute a *Schedule* that captures the redistribution of the edges, obtaining also a sorted array of the new global indices[3]. In the second phase we redistribute the array with the previously computed schedule using a global data exchange primitive.

Performance limitations of current implementations of MPI derived data types forced us to implement basic MPI primitives directly with EUIH calls on the IBM SP-1 and NX calls on the Intel Paragon. Derived data types were buffered explicitly.

The first experiment uses a large unstructured grids, *rotor*, obtained from NASA[4]. *Rotor* has 99617 nodes with an average degree in each node of 13.299 edges and a maximum degree of 125. The ratio between the maximum degree and the average degree of this grid makes a two dimensional matrix representation of the edges very inefficient. Therefore, we cannot use multi-gather primitives in CHAOS or schedule scaling in PILAR-Enumerated. The initial mapping of nodes to processors is cyclic and the final one is given by the Multi-Level RSB partitioner [13]. In this case we compare the performance of CHAOS/PARTI with PILAR-Enumerated and PILAR-Intervals during both phases of the redistribution. Results for the IBM SP-1 and Intel Paragon up to 32 processors can be seen in Figure 7.

Both machines have a similar trend for the first phase: PILAR-Enumerated performs better than CHAOS mainly because CHAOS uses a Distributed Translation Table during the remapping as a temporary data structure. PILAR-Interval is almost an order of magnitude faster in this phase due to a more compact representation.

During the second phase the same communication takes place in all the libraries but buffering of messages can be done more efficiently with an interval representation[5]. This the reason why PILAR-Intervals performed better than the other libraries in both machines. PILAR-Enumerated performs slightly worse than CHAOS on the IBM SP-1 and slightly better on the Intel Paragon due to buffer alignments but, in any case, the differences are not significant.

In the second experiment we wanted to find the threshold of regularity in PILAR at which it pays to use the interval representation instead of the enumerated one. In order to do that, we use a 32000 node synthetic grid, with constant degree (60), and arbitrarily vary the number of intervals that represents the edges associated with a node. In all cases we perform the same random remapping, so that the communication pattern is constant. Figure 8 shows results for the IBM SP-1 and Intel Paragon with 32 processors. Similar to the first experiment, we also obtain different graphs for building a *Schedule* and remapping. In both machines, with only two edges per node, it pays to use PILAR-Interval during *Schedule* construction. During redistribution the threshold is larger (3 and 4 edges in the SP-1 and Paragon) but still reasonably small for most practical applications.

6 Conclusions

PILAR is a new run time support for irregular computations that efficiently exploits spatial regularity using ad-hoc internal representations of communications

[3]This is the functionality of *remap()* in CHAOS/PARTI

[4]Qualitatively similar results were obtained with other two grids (*wave* and *bcsstk32*). Results not presented due to space limitations.

[5]An extra loop nesting eliminates an array indirection

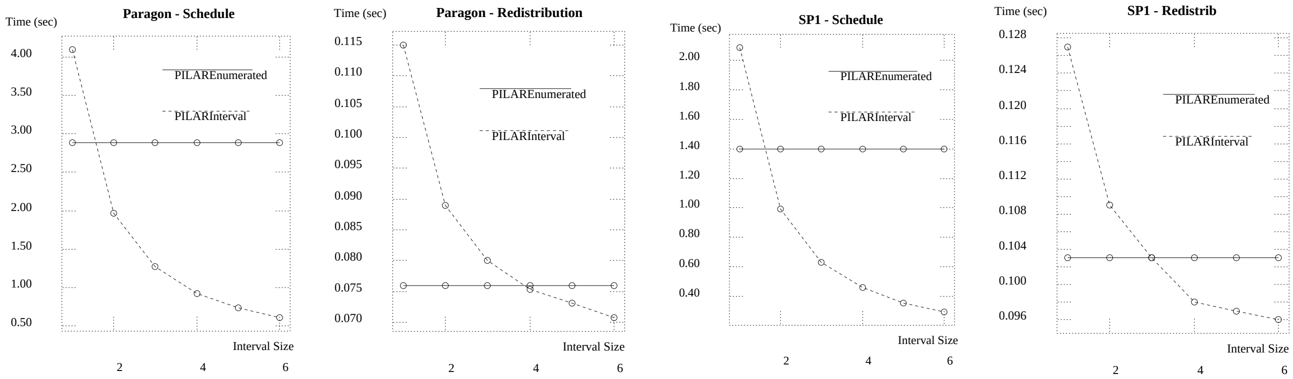

Figure 8: Threshold of regularity on the Intel Paragon and IBM SP1

patterns. Different representations allow the efficient handling of a wide spectrum of regularity. The complexity increase of this flexibility is managed using object oriented techniques. Also, PILAR fully support advanced features of the new MPI message passing standard. Experimental results validate the importance of this flexibility to improve performance.

Our future work is to integrate PILAR with PARADIGM, a HPF/Fortran-D compiler being developed at the University of Illinois [4].

Acknowledgements

The authors would like to thank Horst D. Simon and Steve Barnard for providing us with their ML-RSB code and Joel Saltz for making available to us the CHAOS/PARTI library code. R. Schreiber at RIACS made available to us through anonymous ftp some of the test grids. We are grateful to the San Diego Supercomputing Center and Argonne National Lab for letting us use their MPP.

References

[1] H. D. Simon, ed., *Parallel Computational Fluid Dynamics*. The MIT Press, 1992.

[2] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Solving linear systems on vector and shared memory computers*. SIAM, 1991.

[3] R. Ponnusamy and J. Saltz, *A Manual for the CHAOS Runtime Library*. UMIACS University of Maryland, May 1994.

[4] 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*, (Bangalore, India), pp. 322–330, Dec. 1994.

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

[6] A. Lain and P. Banerjee, "Techniques to Overlap Computation and Communication in Irregular Iterative Applications," *Proceedings of the International Conference on Supercomputing*, July 1994.

[7] R. Ponnusamy, J. Saltz, and A. Choudhary, "Runtime-compilation techniques for data partitioning and communication schedule reuse," in *Proceedings Supercomputing '93*, pp. 361–370, IEEE Computer Society Press, Nov. 1993.

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

[9] G. Agarwal, A. Sussman, and J. Saltz, "Compiler and Runtime Support for Structured and Block Structured Applications," in *the Proceedings of Supercomputing*, (Portland, OR), November 1993.

[10] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A Convex Programming Approach for Exploiting Data and Functional Parallelism," *Proceedings of the International Conference on Parallel Processing*, August 1994.

[11] F. P. Preparata and M. I. Shamos, *Computational Geometry: an introduction*. Springer-Verlag, 1985.

[12] S. Ramaswamy and P. Banerjee, "Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers," in *Proceedings of FRONTIERS 95*, February 1995.

[13] S. T. Barnard and H. Simon, "A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems," Tech. Rep. RNR-92-033, NASA Ames Research Center, Nov. 1992.