A Robust Parallel Programming Model for Dynamic Non-Uniform Scientific Computations

Scott R. Kohn and Scott B. Baden

CSE Technical Report Number CS94-354

March 1994

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California 92093-0114

To appear in the 1994 Scalable High Performance Computing Conference,
May 23-25, 1994, Knoxvilee, TN, U.S.A.

A Robust Parallel Programming Model for Dynamic Non-Uniform Scientific Computations*

Scott R. Kohn Scott B. Baden

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

Abstract

LPARX provides efficient run-time support for dynamic, non-uniform scientific calculations running on MIMD distributed memory architectures. It extends HPF's data decomposition model to provide support for dynamic, block irregular data structures. LPARX represents data decompositions as first-class objects and expresses data dependencies in a manner which is logically independent of data decomposition and problem dimension. LPARX applications are portable across a diversity of MIMD machines. We have implemented a number of applications in LPARX—including a 3d particle calculation and 2d and 3d adaptive multigrid solvers—which could not have been efficiently implemented in HPF.

1 Introduction

The recent development of the High Performance Fortran (HPF) [1] standard has given hope to the scientific computing community that the power of parallel computers may soon be within grasp. The initial work on HPF, based on contributions from a number of Fortran dialects such as Fortran-D [2], Connection Machine Fortran [3], and Vienna Fortran [4], has primarily focused on compile-time optimizations for regular, static applications. An important class of scientific calculations employ dynamic, irregular, block-oriented data structures. Such data structures arise in adaptive multilevel finite difference methods, such as adaptive multigrid [5] and adaptive mesh refinement for computational fluid dynamics [6]. They also occur in applications with non-uniform workload distributions, e.g. particle methods, which require irregular data decompositions on parallel machines to balance workloads. These applications often exhibit rapidly changing data dependencies and communication patterns. Little compile-time information is available for such applications; hence, they require elaborate run-time support. PARTI [7] and Multiblock PARTI [8] provide run-time support for unstructured and block-structured applications; however, neither directly support irregular block decompositions.

LPARX provides run-time support for dynamic, non-uniform scientific calculations. LPARX's abstractions extend HPF's data decomposition model to include dynamic, block irregular data structures. We note that such abstractions do not apply to all irregular scientific computations. For example, LPARX does not address unstructured applications such as sparse linear algebra or finite element methods.

The LPARX system was motivated by a number of design considerations. First, it should be portable across a variety of MIMD architectures and yet provide good performance. LPARX does not rely on architecture-specific facilities, such as fine-grain message passing. Second, all decisions about data decomposition, the assignment of data to processors, and the communication of data dependencies must be made at run-time. LPARX is a run-time system and assumes no compiler support. Finally, LPARX should provide support for arrays of complicated objects in addition to standard built-in types such as floating point numbers and integers. Modern scientific codes often use elaborate data structures which require special treatment when communicated across address spaces.

*This work was supported by NSF contract ASC-9110793 and ONR contract N00014-93-1-0152. Paragon and Cray C-90 time was provided by a UCSD School of Engineering Block Grant. Access to the DEC Alpha workstation farm was provided by SDSC in collaboration with C. Roberto Mechoso of UCLA through a DEC External Research Grant. CM-5 time was provided by the Computer Science Division of the University of California, Berkeley under NSF Infrastructure Grant CDA-8722788 and also by the Advanced Computing Laboratory of Los Alamos National Laboratory.

We have developed a 3d smoothed particle hydrodynamics code [9] and 2d and 3d adaptive multigrid solvers in LPARX; these codes could not have been efficiently implemented using HPF as it does not currently support block irregular data structures. Because LPARX hides architecture-dependent details, applications are portable across a wide range of parallel computers. LPARX, implemented as a `C++` class library, requires only basic message passing support and is currently running on the CM-5, Paragon, iPSC/860, nCUBE/2, KSR-1, single processor workstations, Cray C-90 (single processor), and networks of workstations under PVM [10]. LPARX application development and debugging is generally performed on workstations, which provide a more mature programming environment than most multiprocessors. Applications are then easily moved to the parallel machines for production runs.

This paper is organized as follows. Section 2 surveys conventional approaches to the parallelization of scientific codes and shows how such approaches are inappropriate for dynamic, non-uniform problems. Section 3 describes the LPARX programming model. Implementation details and computational results are given in Section 4. Section 5 discusses LPARX's contributions and future research directions.

2 Conventional Approaches

In this section of the paper, we discuss conventional approaches to data distribution and describe the limitations of these approaches for dynamic, non-uniform applications. We begin with a look at the data decomposition methods typically employed by parallel programming models such as HPF and show why these methods are limited in their ability to balance non-uniform workloads. We then point out limitations in HPF's approach to specifying dynamic data distributions.

2.1 Decomposition Methods

Parallel programming models have typically supported three methods for decomposing arrays across processors: *block*, *cyclic*, and *pointwise* (often called unstructured or irregular) decompositions. HPF and Multiblock PARTI provide block and cyclic decompositions. PARTI/CHAOS implements pointwise decompositions. Fortran-D and Vienna Fortran support all three.

Of the three methods, cyclic and pointwise array decompositions are appropriate for irregular computations. Uniform block decompositions are ineffective in balancing non-uniform workloads. While cyclic decompositions can usually balance non-uniform workloads, they disrupt the locality found in many numerical methods, resulting in increased interprocessor communication costs [11]. Maintaining data locality is a fundamental goal on both message passing and distributed shared memory machines. Communication on message passing architectures is generally expensive relative to computation, and distributed shared memory penalizes non-local (non-cache) memory accesses. Some models support hybrid block-cyclic array decompositions, which give the programmer a limited ability to trade load balance for locality, but such decompositions cannot simultaneously optimize both load balance and locality.

Pointwise decompositions individually assign array elements to processors. While pointwise decompositions are appropriate for unstructured applications such as sweeps over finite element meshes. they are unsuitable for non-uniform applications which possess local structure as they are too fine-grained. Fine-grained decompositions cannot capture the computation's block structure; thus, mapping information must be maintained for each individual array point at substantial cost. Applications incur considerable runtime overhead in the computation of communication schedules, which describe the interprocessor communication necessary to satisfy data dependencies. While the cost of generating a communication schedule may be amortized by reusing the schedule [12], this is often not possible for dynamic applications with rapidly changing data dependencies.

LPARX supports a different style of data decomposition: block-structured, irregular decompositions. Such data distributions provide more freedom in balancing workload than either block or cyclic decompositions. For example, many load balancing algorithms, such as recursive bisection [13], render irregular block decompositions. Unlike cyclic decompositions, irregular block decompositions preserve locality because they are still block structured. Mapping information is only needed for each block, instead of each array point as in pointwise decompositions. Thus, the computation of interprocessor communication schedules is inexpensive.

2.2 HPF Language Support

The HPF language adds layout directives to Fortran 90 which advise the compiler on the decomposition of arrays onto processors. The programmer has some control over the decomposition of arrays via templates, an abstract index space into which arrays are aligned. One limitation of HPF is that templates, as well as all other HPF data decomposition entities, are not first-class language objects. Templates may only appear in directives. They cannot be passed across procedure boundaries, allocated at runtime, or manipulated by the application. The shape of a template is determined at the entry to a program unit and cannot be changed. However, dynamic applications require the dynamic allocation of arrays and the dynamic decomposition and alignment of those arrays across processors. Thus, there is a fundamental mismatch between templates and dynamically allocated arrays which limits the programmer's control over the placement and alignment of data.

In LPARX, decompositions are first-class language objects. They may be allocated at run-time, passed to procedures, and manipulated using LPARX's *domain calculus*, described in Section 3.3. The pC++ language [14] also supports distributions and decompositions as first-class language objects; however, in pC++, the application has limited ability in specifying irregular mappings.

3 LPARX Model Description

The LPARX model is based on the FIDIL programming language [15] and on our previous work with LPAR [16]. LPARX provides a uniform framework for manipulating irregular block-structured data. It separates the expression of data decomposition and communication from numerical computation. The exact method of data decomposition is external to LPARX, which gives LPARX the generality to address a number of different types of applications.

In the following sections, we describe LPARX's model for managing irregular block-structured data. Section 3.1 describes how LPARX represents block irregular data structures. We then discuss LPARX's coarse-grain model of parallelism. Section 3.3 describes the domain calculus, which expresses data dependencies and communication in geometric terms. Finally, we provide a sample of LPARX code for an adaptive mesh refinement application.

3.1 LPARX Data Types

LPARX implements three basic abstract data types—`Region`, `Grid`, and `XArray`. A n-dimensional `Region` represents a subset of Z^n, the space of n-dimensional integer vectors. In the current implementation of LPARX, we restrict `Region`s to be rectangular, although the concepts described here apply to arbitrary subsets of Z^n. (It is an open research question whether other `Region` formulations would be both useful and efficient.) The `Region` is typically used to represent a subset of array index space. Although there is no identical construct in Fortran or C, the `Region` is related to array sections found in Fortran-90. Unlike Fortran-90 array sections, the `Region` is a first-class object and may be assigned and manipulated at runtime. The concept of first-class array section objects was introduced in the FIDIL programming language. As we will see, the `Region` provides the basis for specifying data decompositions.

The `Grid` is an array defined over a `Region`. Each `Grid` remembers its associated `Region`, which can be queried, a convenience that greatly reduces bookkeeping for dynamically defined grids. (Compare this to C, which requires the programmer to keep track of bounds for dynamically allocated array storage.) All `Grid` elements must be of the same type; they may be integers, floating point numbers, or any user-defined type or class. For example, in addition to representing a finite difference mesh of floating point numbers, the `Grid` may also be used to implement the spatial data structures [17] common in particle calculations.

LPARX does not decompose `Grid`s across processors. Instead, LPARX specifies data decomposition using the `XArray`, an array of `Grid`s which represents the high-level structure of a parallel computation. The `Grid`s in an `XArray` may have different origins, sizes, and index sets. Each `Grid` in an `XArray` is assigned to a single processor. As described in the following section, LPARX defines a coarse-grained parallel looping construct—**`forall`**—which loops over the `Grid`s of an `XArray`. An `XArray` is intended to represent coarse-grain irregular decompositions; thus, each processor is typically assigned only a few `Grid`s.

The `XArray` of `Grid` structure provides a common framework for representing various block irregular decompositions. Figure 1 shows decompositions arising in two different applications. In each case, the computational domain has been divided into a relatively small number of `Grid`s, each representing a different portion of the space, which have been assigned to an `XArray`. The following section provides more detail about how `XArray`s are used to organize a parallel computation.

3.2 Coarse-Grain Parallelism

LPARX provides the programmer with a simple model of coarse-grained parallel computation:

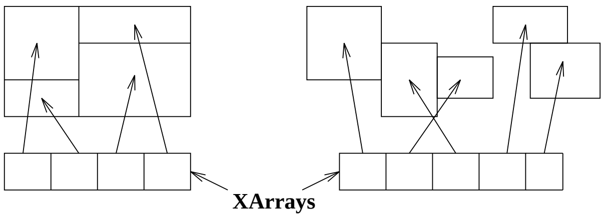


Figure 1: Two examples of an `XArray` of `Grid` data structure. The recursive bisection decomposition on the left is usually employed in calculations with non-uniform workload distributions. The structure on the right is typical of a single level mesh refinement for adaptive mesh methods.

1. Decompose the computational space into an array of `Regions`.
2. Specify an assignment of each region in (1) to a processor.
3. Create an `XArray` of `Grid` which represents the decomposition of space generated in (1) and (2).
4. Satisfy data dependencies between `Grids` in the `XArray` using LPARX's communication facilities.
5. Perform calculations on the `Grids` in the `XArray` in parallel using the `forall` loop.

In step (1), the application represents the decomposition of the space as a first class object—an array of `Regions`. The exact method of data decomposition is external to LPARX and can be heavily application dependent. The decomposition may be managed explicitly by the application, such as in the generation of refinement regions, or through partitioning routines such as recursive bisection and uniform block partitioning which are provided by LPARX's standard library. If LPARX were used in the back-end of a compiler, the data decomposition would be automatically generated by the compiler. Because LPARX provides a uniform interface for representing and parallelizing irregular block decompositions, we have the flexibility and generality to treat a number of different problem classes.

The assignment of `Regions` to processors in (2) provides the application flexibility in delegating work to processors. In general, this information will be returned by the routine which renders the partitions in (1). This step may be omitted, in which case LPARX generates a default assignment.

Using the partitioning information from (1) and the processor assignment information from (2), we instantiate in (3) an `XArray` of `Grid` representing the data decomposition. LPARX creates `Grids` based on the supplied `Region` information and assigns them to the appropriate processors.

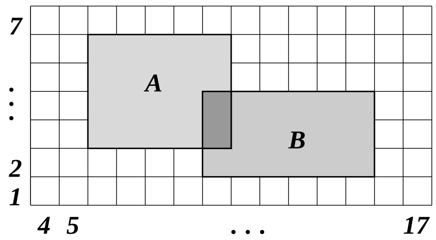


Figure 2: The dark shaded area is intersection of `Regions` `A` and `B`.

After the decomposition and allocation of data, applications typically alternate between steps (4) and (5). In (4), we satisfy data dependencies between the `Grids` in the `XArray` using LPARX's domain calculus. After communication completes, we compute in parallel on the `Grids` in the `XArray` using LPARX's coarse-grain `forall` loop (5). In the execution of the `forall` loop, we assume that the `Grids` are decoupled: `Grids` are processed independently and the computation proceeds asynchronously.

3.3 Domain Calculus

LPARX defines a *domain calculus* [15] which enables the programmer to manipulate index sets (`Regions`) in geometric terms. The domain calculus represents data dependencies in a manner which is logically independent of data decomposition and problem dimension. In this section, we provide a brief overview of the most important domain calculus operations—*intersection*, *grow*, and *injection*.

The intersection of two `Regions` is simply the set of points which the two have in common. The dark shaded area in Figure 2 represents the intersection of `Regions` `A` and `B`. `Regions` are closed under intersection—the intersection of two `Regions` is always another `Region`. If two `Regions` do not overlap, the resulting intersection is said to be *null* or *empty*.

`Grow()` surrounds a `Region` with a boundary layer of a specified width. It takes two arguments—a `Region` and an `integer`—and returns a new `Region` which has been extended in each coordinate direction by the specified amount. Figure 3 shows the `Region` resulting from `Grow(A,1)`.

Our third domain calculus operator, injection, performs a block data copy between two `Grids`. We represent injection with `<<=`. The statement `A <<= B`, where `A` and `B` are `Grids`, copies data from `B` into `A`

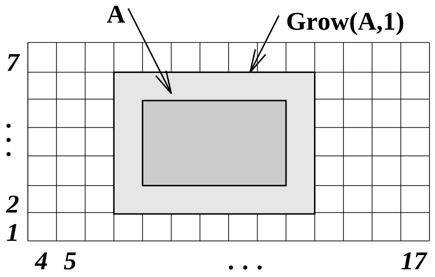


Figure 3: An example of using `Grow()` to add a boundary layer to a `Region`.

where the `Region`s of `A` and `B` intersect. Another form, `A <<= B on C`, where `C` is a `Region`, limits the copy to the index space in which `A`, `B`, and `C` intersect.

3.4 Sample LPARX Code

Figure 4 provides a piece of LPARX pseudo-code for the computation of a single level of refinement in an adaptive mesh refinement (AMR) application [18]. AMR applications place localized refinement patches over areas of high numerical error, such as over the shock wave in Figure 5. While this example is presented using pseudo-code, the equivalent LPARX `C++` code would be similar.

Subroutine `Regrid()`, an application routine, estimates errors, calculates refinement patches, and returns an array of `Region`s describing the areas of refinement. In our example, it also returns the assignment of these `Region`s to processors. Around each refinement region, we add an additional layer of storage `StencilWidth` wide, called *ghost cells*, to hold interior boundary conditions for the patches. Here we overload LPARX's `Grow()` to operate on an array of `Region`s. Using the new `Region` information and the processor assignments, we allocate an `XArray` of `Grid` of `double` for the refinement structure data. Normally, an application would initialize this data using, for example, data from the next coarser level; we do not show this step.

The next piece of code fills ghost cell regions with data from the interior sections (non-ghost cell areas) of adjacent `Grid`s at the same level of refinement. For every pair of `Grid`s `solution[i]` and `solution[j]`, we copy into the ghost cells of `solution[i]` all of the non-ghost cell data from `solution[j]`. Note that we use the `on` clause in the injection operation to select only those data values from `solution[j]` which lie in the interior of the `Grid`.

Finally, we compute on the `Grid`s in parallel using the `forall` loop. The computation in

```
// Get refinement structure for this level

Integer nblocks;
Array of Region refinements, ghosts;
Array of Integer assignments;
Regrid(nblocks, refinements, assignments);
ghosts := Grow(refinements, nblocks,
               StencilWidth);

// Allocate structure to hold refinements

XArray of Grid of double solution :=
   XAlloc(ghosts, assignments, nblocks);

// Application-specific initialization

// Satisfy data dependencies between grids

forall i in 1, nblocks
   for j in 1, nblocks
      solution[i] <<= solution[j] on
                      refinements[j];
   end for
end forall

// Compute solution on grids in parallel

forall i in 1, nblocks
   AdvanceSolution(solution[i]);
end forall
```

Figure 4: LPARX pseudo-code for a single level refinement in an adaptive mesh refinement application. Keywords are underlined. See the text for an explanation.

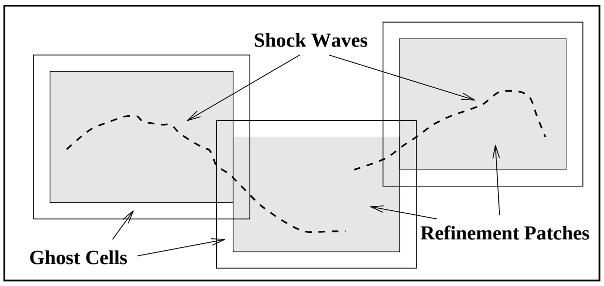


Figure 5: A single level of refinement in an adaptive mesh refinement application. Refinement patches cover the shock wave.

`AdvanceSolution()`, an application-dependent numerical routine, proceeds asynchronously. Some applications may alternate between communication and computation phases until some error criterion is met.

This simple code provides an example of the power of LPARX's domain calculus. The logic of this code will work for any refinement structure, no matter what the shape, and for any dimension problem. An equivalent piece of code written in Fortran or C would be several thousand lines long. We have greatly simplified our work by representing data decompositions and dependencies using the high-level, geometric abstractions of the domain calculus.

4 Computational Results

LPARX has been implemented as a `C++` run-time library with classes for `Region`, `Grid`, and `XArray`. LPARX does not require a special compiler other than a standard `C++` compiler such as GNU `g++`, and LPARX code may be freely mixed with calls to other `C++`, C, or Fortran functions. In fact, most LPARX applications use a programming style which mixes both Fortran and `C++` code: LPARX uses `C++` data structures to manage dynamic memory allocation, and Fortran provides highly optimized and vectorized numerical kernels. This particular style exploits the respective strengths of both `C++` and Fortran. Furthermore, programmers are not required to rewrite highly optimized Fortran kernels when parallelizing applications.

LPARX has been used to parallelize a number of applications, including a genetics algorithm code [19], a 3d particle simulation, 2d and 3d multigrid solvers, and 2d and 3d adaptive multigrid solvers. In the following sections, we provide a brief analysis of LPARX overheads and computational results for a 3d particle simulation and a 3d adaptive multigrid solver. Software versions and compiler options for all benchmarks are reported in Table 1. Further details about the LPARX software structure will be presented in an upcoming paper [20].

4.1 LPARX Overheads

To provide an estimate of LPARX overhead as compared to hand-coding, we implemented a 3d Jacobi iterative solver with a 19 point stencil in LPARX and by hand using message passing. Performance results of the two codes for a $100\times100\times100$ mesh on 32 Paragon nodes is shown in Table 2. The hand-coded application made some simplifying assumptions, namely that each processor was assigned only one subgrid and that the problem was static so that it could precompute communication schedules. Without these simplifying assumptions, the hand-implementation would have been considerably more difficult. While such assumptions may apply to this simple example, they are not valid in general. For example, adaptive mesh calculations may assign several subgrids to each processor; communication dependencies in particle calculations change as particles move.

Table 2 reports four performance numbers: overall execution time, time spent in communication, total number of bytes communicated between processors, and total number of message sends. All numbers are reported per iteration and were averaged over 100 iterations. We note that even the hand-coded application was communication-bound, spending 43% of its time in communication. The performance numbers in the LPARX v1.0 column reflect the current version of the software. For this application, sixty percent of LPARX's overhead can be attributed to overheads in packing and unpacking message buffers. These overheads arise in the buffer management for complicated `Grid` objects employed in calculations such as the particle code in Section 4.2. However, we can reduce this overhead for simple `Grid` object such as floating-point numbers. These optimizations are reflected in the LPAX v1.1 column, which estimates the performance of the next release of the software.

The LPARX code communicated approximately 5% more bytes than the hand-coded version. Most of this overhead is due to the additional information which must be communicated with each LPARX message. Because LPARX cannot assume that only one subgrid is assigned to each processor (as was assumed in the hand-coded version), it must incorporate into each message which subgrid is to receive the message. This overhead adds between 32 and 44 bytes to each LPARX message, depending on the word size of the machine. LPARX's synchronization protocol, which executes a global reduction and a global barrier synchronization, accounts for the additional message sends.

4.2 Smoothed Particle Hydrodynamics

We present performance figures for a 3d smoothed particle hydrodynamics (SPH3D) application[1]. The

[1]The original SPH3D code was provided by John Wallin (Institute for Computational Sciences and Informatics at George Mason University) and Curtis Struck-Marcell (Department of Physics and Astronomy, Iowa State University).

Table 1: Software version numbers and compiler optimization flags for the LPARX benchmarks.

Machine	C++		Fortran		Operating System
	Compiler	Optimization	Compiler	Optimization	
Alphas	g++ v2.4.5	-O2	f77 v3.3-342	-O4	OSF/1 1.2 10 (PVM v3.1.5)
C-90	C++ v1.0.1.0	-O3	cft77 v6.50	-O1	UNICOS 7.C.3 7C3.11
CM-5	g++ v2.5.8	-O2	f77 v1.4	-O4 -libmil	CMOST 7.3 Beta 2.7
Paragon	g++ v2.5.7	-O2 -mnoieee	if77 v4.1.1	-O4 -Knoieee	OSF/1 1.0.4 R1.1.3

Table 2: LPARX overheads for a 3d Jacobi (19 point stencil) relaxation calculation on a 100×100×100 mesh on 32 Paragon nodes. The numbers for LPARX v1.0 reflect the current implementation. The v1.1 numbers are estimates for the performance of the next LPARX release. All measurements are reported per iteration and were averaged over 100 iterations. Message statistics were summed over all nodes.

	LPARX v1.0	*LPARX v1.1*	by hand
Overall time (sec)	0.288	*0.245*	0.221
Communication only (sec)	0.163	*0.120*	0.096
Bytes communicated (megabytes)	2.46	*2.46*	2.35
Message sends (thousands)	1.55	*1.55*	0.91

computational structure of SPH3D is similar to molecular dynamics and other particle calculations which exhibit short-range interactions. Figure 6 presents performance results for the SPH3D code on the Cray C-90, Paragon, CM-5, and a network of Alpha workstations running PVM. Time is reported in seconds per timestep. All floating point arithmetic was performed using 64-bit numbers. The applications code was identical on all machines except that the C-90 version performed gathering and scattering of particles to get longer vector lengths. We use the performance of one processor of the C-90 as a baseline for comparison.

Although the numerical kernels of SPH3D vectorized on the C-90, the kernels are rather complicated and contain a number of conditionals which hinder efficient utilization of the vector units. Furthermore, even though the C-90 code gathers and scatters particles to increase vector lengths, vector lengths are still quite short. (These vectorization limitations are intrinsic to the algorithm and are not artifacts of parallelization.) The C-90 code performs between 150 and 250 64-bit megaflops, depending on problem size. For the 48,000 particle problem size, one processor of the C-90 was roughly equivalent to 8 Alpha processors, 16 Paragon processors, or 64 CM-5 processors. We did not attempt to use the vector units on the CM-5 because of the difficulty of vectorizing the numerical kernels by hand.

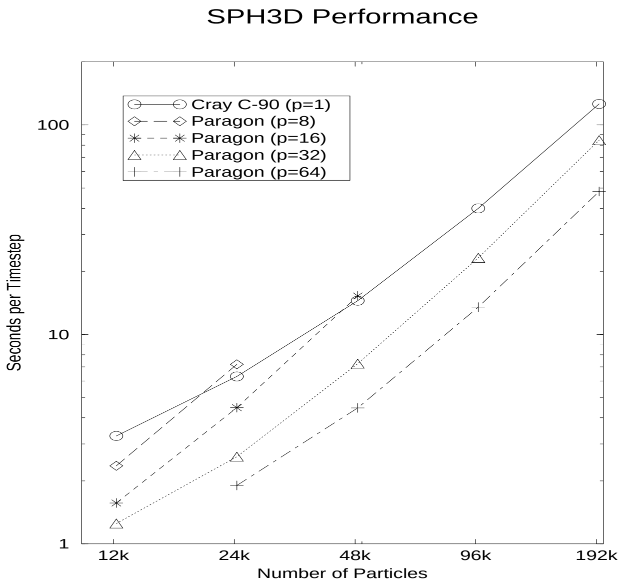
4.3 Adaptive Multigrid Solver

We have implemented 2d and 3d adaptive multigrid solvers based on Brandt's FAS algorithm [5]. Adaptive mesh applications are difficult to parallelize because error estimation, regridding [21], data decomposition, assignment of work to processors, and calculation of data dependencies must be made at run-time.

Figure 7 shows the 2d potential arising from ten smoothed point charges arranged in a ring. Grid refinements were placed over the areas where the solution was changing most rapidly, i.e. surrounding the point charges. In a parallel implementation, the subgrids shown in the Figure would be subdivided and assigned to processors. Table 3 gives lines of code for the 2d and 3d adaptive multigrid solvers. The majority of the LPARX C++ code is independent of the problem dimension; portions which are not consist mostly of calls to the appropriate 2d or 3d Fortran numerical routines. We are currently tuning and optimizing the application on the Paragon and the C-90; performance figures will be available soon.

5 Discussion and Conclusions

We have introduced a new programming model and run-time system which provides support for dynamic, irregular block-structured scientific calculations such



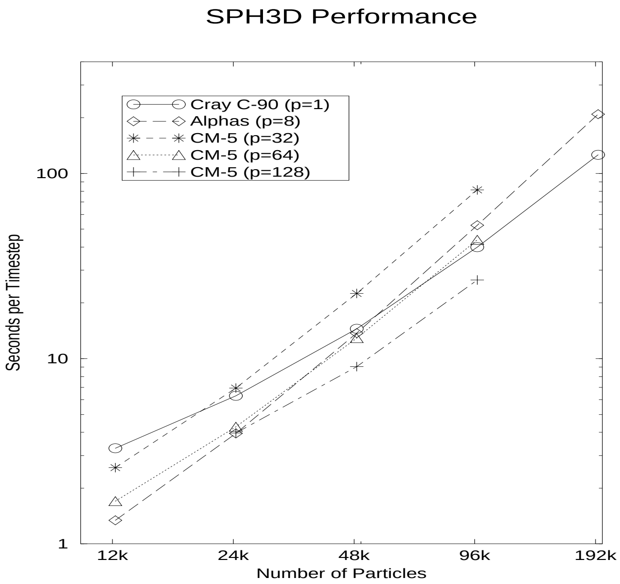


Figure 6: SPH3D performance results on the Cray C-90, Intel Paragon, Thinking Machine's CM-5, and an Alpha workstation farm running PVM. The vector units on the CM-5 were not used. The number of processors is given in parentheses. C-90 times were averaged over five timesteps, CM-5 times over ten timesteps, and Alpha and Paragon times over twenty timesteps.

Table 3: Lines of code for various modules of the 2d and 3d adaptive multigrid solvers.

Module	LPARX C++			Fortran	
	2d or 3d	2d only	3d only	2d only	3d only
Multigrid	620	49	53	263	409
Adaptive multigrid	499	27	32	93	150
Error estimation	262	18	20	71	76
Regridding	1247	15	21	—	—
Main program	385	11	14	51	51

as particle methods and adaptive finite difference methods. LPARX extends HPF's data decomposition model to provide support for dynamic, block irregular data structures. In contrast to HPF, decompositions are first class language objects which can be created and manipulated at run-time. Data dependencies are expressed in geometric terms using the domain calculus and are independent of dimension and data decomposition.

LPARX applications are portable across a wide range of MIMD parallel architectures. LPARX has been used to parallelize a number of dynamic, non-uniform applications which could not have been efficiently implemented in HPF. In the SPH3D application, one processor of the C-90 was roughly equivalent to 16 Paragon nodes or 64 CM-5 nodes.

We are collaborating in the development of LPARX-based adaptive mesh methods for the first principles simulations of real materials. LPARX has been adopted by collaborators at Lawrence Livermore National Laboratory for the parallelization of 2d and 3d gas dynamics calculations [22]. We are investigating LPARX as an intermediate form for compilation and plan to extend the `Region` abstraction to unstructured problems such as sweeps over finite element meshes.

We hope this work will motivate the High Performance Fortran Forum to consider support for irregular block structures such as that provided in LPARX. We have shown that LPARX can efficiently support classes of dynamic, non-uniform applications which are not currently addressed in the HPF standard.

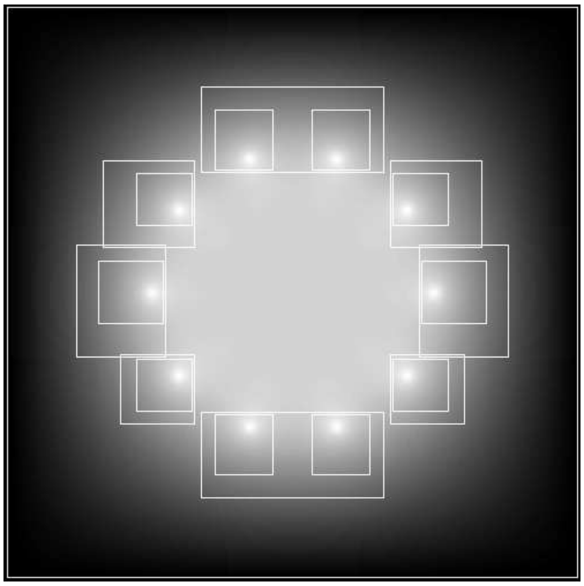


Figure 7: Two levels of mesh refinement for a 2d adaptive FAS code. This image shows the potential from ten smoothed point charges arranged in a ring.

References

[1] High Performance Fortran Forum, Rice University, Houston, Texas, *High Performance Fortran Language Specification*, May 1993.

[2] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "Fortran d language specification," Tech. Rep. TR90-141, Dept. of Computer Science, Rice University, Houston, TX, December 1989.

[3] Thinking Machines Corporation, Inc., Cambridge, MA, *Getting Started in CM Fortran*, January 1991.

[4] B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima, "Dynamic data distribution in vienna fortran," in *Proceedings of Supercomputing '93*, November 1993.

[5] A. Brandt, "Multi-level adaptive solutions to boundary-value problems," *Mathematics of Computation*, vol. 31, pp. 333–390, April 1977.

[6] M. J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *Journal of Computational Physics*, vol. 82, pp. 64–84, May 1989.

[7] H. Berryman, J. Saltz, and J. Scroggs, "Execution time support for adaptive scientific algorithms on distributed memory machines," Tech. Rep. 90-41, ICASE, NASA Langley Research Center, Hampton, Virginia, February 1991.

[8] G. Agrawal, A. Sussman, and J. Saltz, "Compiler and runtime support for structured and block structured applications," in *Proceedings of Supercomputing '93*, November 1993.

[9] J. J. Monaghan, "Smoothed particle hydrodynamics," *Annual Review of Astronomy and Astrophysics*, vol. 30, pp. 543–574, 1992.

[10] V. S. Sunderam, "Pvm: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, pp. 315–339, December 1990.

[11] S. B. Baden and S. R. Kohn, "A comparison of load balancing strategies for particle methods running on mimd multiprocessors," in *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, (Houston, Texas), March 1991.

[12] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis, "Distributed memory compiler methods for irregular problems — data copy reuse and runtime partitioning," Tech. Rep. 91–73, ICASE, Hampton, VA, September 1991.

[13] M. J. Berger and S. H. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Transactions on Computers*, vol. C-36, pp. 570–580, May 1987.

[14] F. B. Irisa, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr, "Implementing a parallel c++ runtime system for scalable parallel systems," in *Proceedings of Supercomputing '93*, November 1993.

[15] P. N. Hilfinger and P. Colella, "Fidil: A language for scientific programming," Tech. Rep. UCRL-98057, Lawrence Livermore National Laboratory, January 1988.

[16] S. R. Kohn and S. B. Baden, "An implementation of the lpar parallel programming model for scientific computations," in *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, March 1993.

[17] R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*. McGraw-Hill, 1981.

[18] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of Computational Physics*, vol. 53, pp. 484–512, March 1984.

[19] W. Hart and R. Belew, "A mimd design for geographically structured genetic algorithms." (in preparation), 1993.

[20] S. R. Kohn and S. B. Baden, "Object-oriented parallel programming with distributed parallel objects." To be submitted to Supercomputing '94, April 1994.

[21] M. J. Berger and I. Rigoutsos, "An algorithm for point clustering and grid generation," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 21, no. 5, pp. 1278–1286, 1991.

[22] W. Y. Crutchfield and M. L. Welcome, "Object oriented implementation of adaptive mesh refinement algorithms," Tech. Rep. UCRL-JC-113502, Lawrence Livermore National Laboratory, April 1993.