Irregular Coarse-Grain Data Parallelism under LPARX

SCOTT R. KOHN[1] AND SCOTT B. BADEN[2]

[1]*Department of Chemistry and Biochemistry, University of California, San Diego, La Jolla, CA 92093-0340*
[2]*Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114*

ABSTRACT

LPARX is a software development tool for implementing dynamic, irregular scientific applications, such as multilevel finite difference and particle methods, on high-performance multiple instruction multiple data (MIMD) parallel architectures. It supports coarse-grain data parallelism and gives the application complete control over specifying arbitrary block decompositions. LPARX provides *structural abstraction,* representing data decompositions as first-class objects that can be manipulated and modified at run-time. LPARX, implemented as a C++ class library, is currently running on diverse MIMD platforms, including the Intel Paragon, Cray C-90, IBM SP2, and networks of workstations running under PVM. Software may be developed and debugged on a single-processor workstation.

1 INTRODUCTION

An outstanding problem in scientific computation is how to manage the complexity of converting mathematical descriptions of dynamic, irregular numerical algorithms into high-performance applications software. Nonuniform applications, such as multilevel adaptive grid methods and particle methods, are particularly challenging. Current parallel computers are much more difficult to use than current vector machines because the programmer must manage computational resources at a very low level. Parallel compiler technology does not yet afford the convenience expected by the user community [13]. While this situation will improve with time, adequate run-time support is essential in applications with dynamic, data-dependent computational structures.

We have developed the LPARX parallel programming system [24, 25] to simplify the development of dynamic, nonuniform scientific computations on high-performance parallel architectures. Such software support is essential to developing high-performance, portable, parallel applications software. LPARX is a domain-specific, coarse-grain data parallel programming model that provides run-time support for dynamic, block-irregular data decompositions. General irregular block decompositions are not currently supported by compiled languages such as High Performance Fortran (HPF) [21], Fortran D [19], Vienna Fortran [12], and Fortran 90D [31]. They arise in two important classes of applications:

Received February 1995
Revised August 1995

Scientific Programming, Vol. 5, pp. 185–201 (1996)
CCC 1058-9244/96/030185-17

1. Adaptive multilevel finite difference methods [8, 9, 27] that represent refinement regions using block-irregular data structures
2. Parallel computations that require an irregular data decomposition [7] to balance nonuniform workloads across parallel processors, such as particle methods [23]

LPARX hides many of the low-level details, such as interprocessor communication, involved in managing complicated dynamic data structures on parallel computers. It provides the programmer with high-level coordination facilities to manage data locality within the memory hierarchy to minimize communication costs.

LPARX should not be thought of as a "language," but rather as a set of data distribution and parallel coordination abstractions which may be implemented in a library (as we have done) or added to a language. The design goals of LPARX are as follows:

1. To express irregular data decompositions, layouts, and data dependencies at run-time using high-level, intuitive abstractions.
2. To require only basic message-passing support and give portable performance across diverse parallel architectures.
3. To separate parallel control and communication from numerical computation and to allow the reuse of highly optimized numerical kernels from existing serial codes with minimal change.
4. To permit the user to develop and debug software on a single-processor workstation.

LPARX has been implemented as a C++ class library and does not require special compiler support. Applications may invoke subroutines written in languages other than C++, such as C or Fortran. The implementation assumes only basic message-passing support and may run on any multiple instruction multiple data (MIMD) machine. LPARX is currently running on the Intel Paragon, IBM SP2, Cray C-90, single processor workstations, and networks of workstations under PVM [30].

This article is organized as follows. Section 2 provides an overview of the LPARX programming abstractions. Section 3 describes in detail the parallelization of a simple application, Jacobi relaxation. The parallelization of a particle calculation is discussed in Section 4. (Structured adaptive mesh methods are beyond the scope of this article and are described elsewhere [24, 26].) Section 5 presents computational results. Finally, Section 6 discusses related work and Section 7 summarizes the contributions of LPARX.

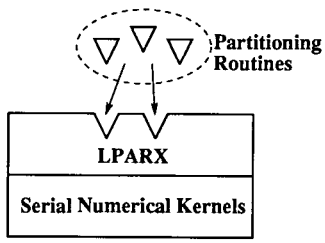

FIGURE 1 The logical organization of an LPARX application consists of three components: partitioning routines, LPARX code, and serial numerical kernels.

2 OVERVIEW

LPARX provides high-level abstractions for representing and manipulating irregular block-structured data on MIMD distributed memory architectures. In the following sections, we give an overview of LPARX's facilities. We begin with a description of the philosophy behind the LPARX model. We introduce LPARX's data types and its representation of irregular block decompositions and present LPARX's underlying coarse-grain data parallel programming model. Finally, we describe the *region calculus*, which expresses data decompositions and dependencies in geometric terms.

2.1 Philosophy

The LPARX parallel programming model separates the expression of data decomposition, communication, and parallel execution from numerical computation. As shown in Figure 1, LPARX applications are logically organized into three separate pieces: partitioners, LPARX code, and serial numerical kernels.

The LPARX layer provides facilities for the coordination and control of parallel execution. LPARX is a coarse-grain data parallel programming model; it gives the illusion of a single global address space and a single logical thread of control. On a MIMD parallel computer, the LPARX run-time system executes in single program multiple data (SPMD) mode.

Computations are divided into a relatively small number of coarse-grain pieces; each work unit represents a substantial computation executing on a

single logical processing node. LPARX does not define what constitutes a single logical node; a node may correspond to a single processor, a processing cluster, or a processor subset. Parallel execution is expressed using a coarse-grain loop; each iteration of the loop iterates as if on its own processor. The computation for each piece is performed by a numerical kernel, and the computations proceed independently of one another. The numerical kernels may be written in a language other than C++, such as C, Fortran, or HPF. The advantage of this approach is that heavily optimized numerical routines need not be reimplemented to parallelize an application. Furthermore, numerical code can be optimized for a processing node without regard to the higher level parallelization. Kernels may be tuned to take advantage of low-level node characteristics, such as vector units, cache sizes, or multiple processors.

An important part of the LPARX philosophy is that data partitioning for dynamic, nonuniform scientific computations is extremely problem dependent and therefore is best left to the application. No specific data decomposition strategies have been built into the LPARX model. Rather, all data decomposition in LPARX is performed at run-time under the direct control of the application. LPARX gives the application a uniform framework for representing and manipulating block-irregular decompositions. Although it provides a standard library of decomposition routines, the programmer is free to write others.

Our approach to data decomposition differs from most parallel languages, such as HPF [21], which require the programmer to choose from a small number of predefined decomposition methods. Vienna Fortran [12] provides some facilities for irregular user-defined data decompositions but limits them to tensor products of irregular 1d decompositions. Block-irregular decompositions may be constructed using the pointwise mapping arrays of Fortran D [19]; however, pointwise decompositions are inappropriate and unnatural for calculations which exhibit block structures. Because pointwise decompositions have no knowledge of the block structure, mapping information must be maintained for each individual array element (instead of for each block) at a substantial cost in memory and communication overheads.

Once a decomposition has been specified, the details of the data partitioning are hidden from the application. The programmer can change partitioning strategies without affecting the correctness of the underlying code. Thus, LPARX views partitioners as interchangeable, and the application may change decomposition strategies by simply invoking a different partitioning routine.

At the core of LPARX is the concept of *structural abstraction*. Structural abstraction enables an application to express the logical structure of data and its decomposition across processors as first-class, language-level objects. The key idea is that the structure of the data—the "floorplan" describing how the data are decomposed and where the data are located—is represented and manipulated separately from the data. LPARX expresses operations on data decompositions and communication using intuitive geometric operations, such as intersection, instead of explicit indexing. Interprocessor communication is hidden by the run-time system, and the application is completely unaware of low-level details. We note that although the LPARX implementation is currently limited to representing irregular, block-structured decompositions, the concept of structural abstraction is general and extends to other classes of applications, such as unstructured finite element meshes [4].

2.2 LPARX Data Types

LPARX provides the following three abstract data types:

1. `Region`: an object representing a subset of array index space
2. `Grid`: a dynamic array instantiated over a `Region`
3. `XArray`: a dynamic array of coarse-grain elements, `Grids`, distributed over processors

The `Region` provides the basis for structural abstraction. An n-dimensional `Region` represents a subset of Z^n, the space of n-dimensional integer vectors. The `Region` does not contain data elements, as an array, but rather represents a portion of index space. In the current implementation of LPARX, we restrict `Regions` to be rectangular; however, the concepts described here apply to arbitrary subsets of Z^n [4]. Although there is no identical construct in Fortran or C, the `Region` is related to array section specifiers found in Fortran-90. Unlike Fortran-90 array section specifiers, the `Region` is a first-class object and may be assigned and manipulated at run-time. The concept of first-class array section objects was introduced in the FIDIL programming language [22].

The Grid is a dynamic array defined over an arbitrary rectangular index set specified by a Region. The Grid is similar to an HPF allocatable array. 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 have 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 [23] common in particle calculations. Grids may be manipulated using high-level block-copy operations, described in Section 2.4.

LPARX is targeted toward applications with irregular, block structures. To support such structures, it provides a special array—the XArray—for organizing a dynamic collection of Grids. Each Grid in an XArray is arbitrarily assigned to a single processor; individual Grids are not subdivided across processors. The XArray can be viewed as a coarse-grain analogue of a Fortran D array decomposed via mapping arrays, except that XArray elements are themselves arrays (Grids).

The Grids in an XArray may have different origins, sizes, and index sets; but all Grids must have the same spatial dimension. When creating an XArray, the user provides an array of Regions representing the structure of the Grids and a corresponding array of processor assignments; LPARX provides a default assignment of Grids to processors if none is given. An XArray is intended to implement coarse-grain irregular decompositions; thus, each processor is typically assigned only a few Grids.

LPARX defines a coarse-grain looping construct—for_all—which iterates concurrently over the Grids of an XArray. The semantics of for_all are similar to HPF's INDEPENDENT forall [21]; each loop iteration is executed as if an atomic operation. In writing a for_all loop, the programmer is unaware of the assignment of Grids to processors—each XArray element is treated as if it were assigned to its own processor—and the LPARX run-time system correctly manages the parallelism.

The XArray of Grid structure provides a common framework for implementing various block-irregular decompositions of data. This framework is used by the load-balancing utilities found in LPARX's standard library and also in application-specific routines, such as a grid generator for an adaptive mesh refinement calculation. Figure 2 shows decompositions arising in two different applications. In each case, the data have been divided into Grids, each representing a different portion of the computational domain, which have been assigned to an XArray. The following section provides more detail about how XArrays are used to organize a parallel computation.

2.3 Coarse-Grain Parallel Computation

Recall that an LPARX application consists of three components: partitioning routines, LPARX code, and serial numerical kernels. Here we show how these pieces work together in an application. LPARX provides the programmer with a simple model of coarse-grain parallel computation by

1. Decomposing the computational structure into an array of Regions.
2. Specifying an assignment of each Region in (1) to a processor.
3. Creating an XArray of Grid representing the decomposition of space generated in (1) and (2).
4. Satisfying data dependencies between Grids in the XArray using LPARX's communication facilities (described in the following section).
5. Perform calculations on the Grids in the XArray in parallel using the coarse-grain for_all loop.

The decomposition in (1) may be managed explicitly by the application, such as in generating refinement regions, or by load balancing utilities that implement partitioning strategies. LPARX has a standard library of partitioners that implement recursive coordinate bisection [7] and uniform block partitioning.

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

Using the partitioning information and the processor assignment information, the application instantiates in (3) an XArray of Grid implementing the data decomposition. LPARX creates Grids based on the supplied Region information and assigns them to the appropriate processors.

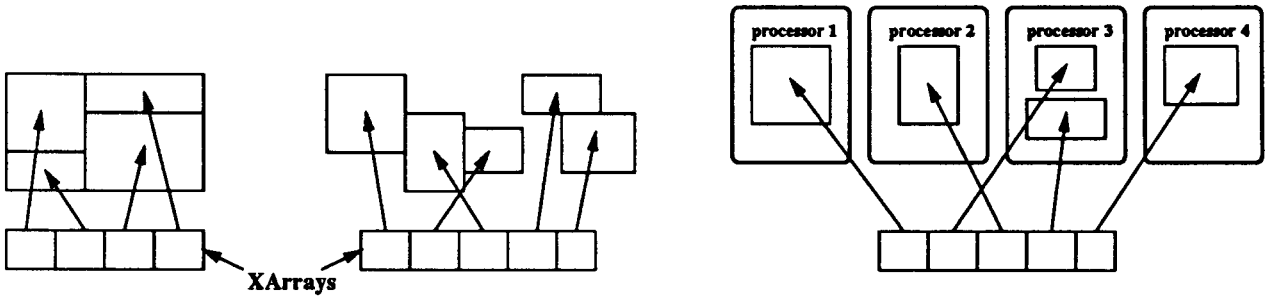

FIGURE 2 Two examples of an `XArray` of `Grid` data structure. The recursive bisection decomposition on the left is usually employed in particle calculations. The structure in the middle is typical of a single-level mesh refinement in adaptive mesh methods. On the right, we show one possible mapping of `XArray` elements to processors. Note that the `XArray` is a container for the `Grids` and its elements are `Grids`, not pointers.

After the decomposition and allocation of data, applications typically alternate between steps (4) and (5). In (4), data dependencies between the `Grids` in the `XArray` are satisfied using LPARX's region calculus and copy operations, described in the following section. After communication completes, the application computes in parallel on the `Grids` in the `XArray` using a `for_all` loop. For each `Grid`, a numerical routine is called to perform the computation; the computation executes on a single logical node which may actually consist of many physical processors. The execution of `for_all` assumes the `Grids` are decoupled; they are processed independently and asynchronously.

2.4 The Region Calculus

LPARX defines a *region calculus* which enables the programmer to manipulate index sets (`Regions`) in high-level geometric terms. In this section, we provide a brief overview of the most important region calculus operations—*intersection* and *grow*—and describe a high-level block copy operation called *copy-on-intersect*.

The intersection of two `Regions` is simply the set of points which the two have in common. The dark shaded area in Figure 3a represents the intersection of `Regions` R and S. `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 *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 3b shows the `Region` resulting from `Grow(R, 1)`.

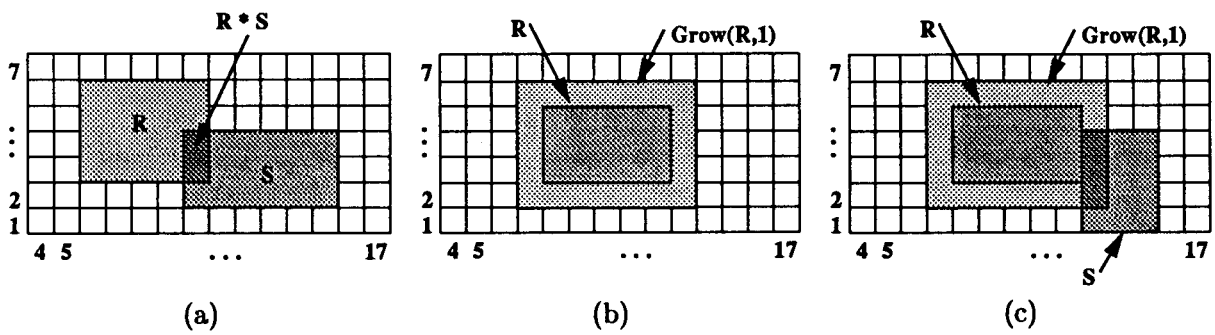

FIGURE 3 (a) The dark shaded area is intersection of `Regions` R and S, denoted by R * S. (b) An example of using `Grow()` to add a boundary layer to a `Region`. (c) Using `Grow()` and intersection to calculate data dependencies for a ghost cell region.

Copy-on-intersect coordinates data motion between two `Grids`. Here we represent copy-on-intersect with $\Leftarrow$. The statement `G` $\Leftarrow$ `H`, where `G` and `H` are `Grids`, copies data from `H` into `G` where the `Regions` of `G` and `H` intersect. Another form, `G` $\Leftarrow$ `H on R`, where `R` is a `Region`, limits the copy to the index space in which `G`, `H`, and `R` intersect.

We now show how these simple but powerful operations are used to calculate data dependencies. One common communication operation in scientific codes is the transmission of data to fill ghost cells, boundary elements added to a processor's local data partition. The region calculus represents the processor's local partition as a `Region`. We `Grow()` the `Region` to define ghost cells and then use intersection to calculate the `Region` of data required from another processor. Finally, a block copy updates the ghost region's data values, as shown in Figure 3c. Recall that copy-on-intersect copies values that lie in the intersection of the ghost region and interacting blocks. The calculation of data dependencies involving no explicit computations involving subscripts. The region calculus is independent of the `Grid` dimension, and the same operations work for any problem dimension. All bookkeeping details and interprocessor communication are managed by the run-time system and are completely hidden from the user.

3 A SIMPLE PROGRAMMING EXAMPLE

In this section, we illustrate how to use LPARX to parallelize a simple application, Jacobi relaxation on a rectangular domain. For concreteness, we will use `C++` notation. Although this particular computation is neither irregular not dynamic, it is easy to explain, and the techniques used to parallelize it under LPARX generalize to far more elaborate irregular computations. We will discuss these in Section 3.5.

3.1 Problem Description

Consider the Laplace equation in two dimensions:

$$\Delta u = 0 \; in \; \Omega$$

subject to Dirichlet boundary conditions on $\partial\Omega$, the boundary of Ω:

$$u = f \; on \; \partial\Omega,$$

where f and u are real-valued functions of two variables and the domain Ω is a rectangle. We discretize the computation using the method of finite differences, solving a set of discrete equations defined on a regularly spaced mesh of $(M + 2) \times (N + 2)$ points in Z^2.

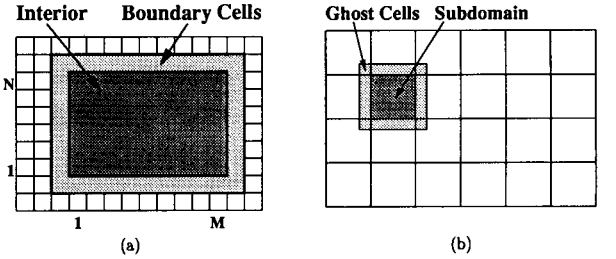

FIGURE 4 (a) A finite difference mesh defined over the 2d `Region [0:M+1, 0:N+1]` with interior `[1:M, 1:N]`. (b) A blockwise decomposition of the computational space into 24 subblocks. The lightly shaded area shows the ghost region for a typical partition.

The boundary points of the mesh, which contain the Dirichlet boundary conditions for the problem, are located along x-coordinates between 0 and $M + 1$ and along y-coordinates between 0 and $N + 1$, as shown in Figure 4a. We number the interior points of the mesh from 1 to M in the x-coordinate and from 1 to N in the y-coordinate. This interior region is defined as follows:

```
Region2 Interior(1,1,M,N)
```

LPARX strongly types all `Regions` by the number of spatial dimensions; thus, `Region2` represents a 2d `Region`, `Region3` a 3d `Region`, and so on.

To parallelize Jacobi relaxation, we decompose the computational domain into subdomains and assign each subdomain to a processor. A standard blockwise decomposition for 24 processors is shown in Figure 4b. Each subdomain is augmented with a ghost cell region, which locally caches either interior data from adjoining subdomains or Dirichlet boundary conditions. We refresh the ghost cell regions before computing on each subdomain. Each processor then updates the solution for the subdomains it owns; this computation proceeds in parallel and each processor performs its calculations independently of the others.

3.2 Decomposing the Problem Domain

LPARX does not predefine specific data-partitioning strategies; rather, data partitioning is under the control of the application. One possible parti-

tioning which is defined in LPARX's partitioning library is the uniform BLOCK decomposition employed by HPF. By convention, LPARX expects the partitioner to return an array of Regions which describe the uniform partitioning:

```
Region2 *Partition = Uniform(Interior, P)
```

The partitioner Uniform takes two arguments: the Region to be partitioned and the desired number of subdomains, usually the number of processors. After determining the partitioning of space, we extend each subdomain with ghost cells. The exact thickness of the ghost cell region depends on the finite difference stencil employed in the finite difference scheme. We will use a five-point finite difference stencil to solve Laplace's equation; thus, the ghost cell region is one cell thick. We apply the grow() function to augment each subdomain of Partition[] with a ghost region*:

```
Region2 *Ghosts = grow(Partition, P, 1)
```

The computational domain is now logically divided into an array of 2d Regions called Ghosts. We will allocate a 1d XArray of 2d Grids of double to implement this data decomposition; the k*th* Grid will be assigned to the k*th* processor. We declare the XArray of Grids structure using:

```
XArray1(Grid2(double)) U
```

and instantiate the storage using XAlloc():

```
XAlloc(U, P, Ghosts)
```

XAlloc() takes three arguments: the XArray to be allocated, the number of elements to be allocated, and an array of Regions, one Region for each element in the XArray. Note that all Grids in an XArray must have the same type (e.g., double) and dimension; of course, the Grids in an XArray may be defined over different Regions.

When defining an XArray with XAlloc(), we may optionally supply a processor assignment for each XArray component. If no such processor assignment is specified, as in the code above, then LPARX chooses a default mapping. To override the default, we provide an array of integer processor identifiers, one for each XArray element, as an optional fourth argument to XAlloc():

```
XAlloc(U, P, Ghosts, Mapping)
```

Such a mapping may be used to better balance workloads or to optimize interprocessor communication traffic for a particular hardware interconnect topology.

After instantiating the XArray, we are ready to compute. The main computational loop is as follows:

```
do {
   dU = relax(U, h);
} while (dU > Epsilon);
```

where relax() is a subroutine which performs communication and the relaxation computation, h is the spacing of the finite difference mesh, dU is the maximum change in magnitude of the solution over all points in the interior of the computational box, and Epsilon is a user-supplied convergence condition. The LPARX code is summarized in Figure 5.

Note that we may change the data-partitioning scheme at any time without affecting the correctness of the code. For example, the "box-like" partitioning may be replaced with a strip decomposi-

```
void main(const int argc, char **argv)
  {
   // Initialize M, N, and h (not shown)

   // Get the number of nodes (number of partitions)
   const int P = mpNodes();

   // Partition the computational space
   Region2 Interior(1,1,M,N);
   Region2 *Partition = Uniform(Interior, P);
   Region2 *Ghosts = grow(Partition, P, 1);

   // Allocate and initialize the data (not shown)
   XArray1(Grid2(double)) U;
   XAlloc(U, P, Ghosts);

   // Relax until the solution has converged
   do {
      dU = relax(U, h);
   } while (dU > Epsilon);

   // Exit program
  }
```

FIGURE 5 LPARX code, which partitions the computational space, allocates the XArray of Grids structure and calls the Jacobi relaxation routine (described in Section 3.3).

* Grow() is overloaded in the obvious way to handle arrays of Regions.

```
double relax(XArray1(Grid2(double))& U, const double h)
  {
   // Refresh the ghost cell regions
   FillPatch(U);

   // Compute in parallel over all subgrids
   double dU = 0.0;
   for_all_1(i, U)
      Region2 inside = grow(U(i).region(), -1);
      const Point2 nl = inside.lower();
      const Point2 nh = inside.upper();
      const double dE = smooth5(U(i).data(), nl, nh, &h);
      dU = MAX(dU, dE);
   end_for_all

   // Find the maximum change over all the processors
   mpReduceMax(&dU);

   return(r0);
  }
```

FIGURE 6 The parallel Jacobi relaxation routine calls `FillPatch()` and `smooth5()`, a computational kernel written in Fortran.

tion or even a recursive bisection decomposition simply by calling a different partitioner. No other changes would need to be made. Furthermore, the computational domain need not be restricted to a rectangle; it may be an "L"-shaped region or, in general, any irregular collection of blocks.

3.3 Relaxation

Function `relax()` performs the major tasks in solving the Laplace equation: It invokes `FillPatch()` to refresh the ghost cell regions and calls the computational kernel, `smooth5()`. The code for `relax()` is shown in Figure 6.

The `for_all` loop in Figure 6 computes in parallel over the `Grids of U`. In LPARX C++ code, looping constructs are typed by the dimension of the arrays over which they iterate. Thus, the `for_all_1` in `relax()` iterates using induction variable `i` over the 1d `XArray U`. (We typically drop the dimension qualifier when discussing a looping construct in general terms.) The `for_all` loop is a parallel loop which executes each iteration on the processor which owns† the corresponding element of the `XArray`. Serial iterations are specified using the `for` loop, whose syntax is identical to the `for_all`. `For` executes every iteration of the loop on each processor. Both LPARX looping constructs implicitly extract from the `XArray` the index space over which to iterate.

Because separately compiled modules may not understand class `Grid` or even the notion of C++ classes (e.g., Fortran), LPARX provides `Grid` member functions to extract data and the bounding box of a `Grid` in a standard form understandable by other languages. LPARX is currently targeted to Fortran users; thus, the implementation stores `Grid` data in column major order. Language interoperability is an important software engineering feature which enables LPARX applications to reuse existing serial numerical kernels with only modest reprogramming, significantly reducing software development time.

The smoother in our example, `smooth5()`, is written in Fortran. By Fortran parameter passing convention, all arguments must be passed by reference (i.e., through a pointer). Member function `data()` returns a pointer to the `Grid`'s data elements, in our case `double`; `upper()` and `lower()` return the lower and upper bounds of the `Region` of `U`. The `Point` class is a low-level LPARX class which represents an integer vector; in the call to `smooth5()`, `Point` data values are type-cast by C++ into a 1d integer array.

Because LPARX runs in SPMD mode, the values of the convergence check `dU` will be different on each processor. The reduction function `mpReduceMax()` takes the maximum over all the local values and returns the global maximum value. Various forms of reduction functions are provided by LPARX's standard library [6].

3.4 `FillPatch()`

The `FillPatch()` code shown in Figure 7 updates the ghost cell regions of each subgrid with data from the interior (nonghost cell) sections of adjacent subgrids. For every pair of `Grids U(i)` and `U(j)`, `FillPatch()` copies into the ghost cells of `U(j)` the overlapping nonghost cell data from `U(i)`. Aggregate data motion between `Grids` is handled through the `copy()` member function:

```
U(j).copy(U(i), inside)
```

Recall that copy-on-intersect copies data from `Grid U(i)` into `Grid U(j)` where the `Regions` of the two intersect with the third argument, `inside`, which is a `Region`. This operation enables efficient data copies between grids, ignoring points which are not shared.

† Ownership, however, is generally hidden from the programmer.

```
void FillPatch(XArray1(Grid2(double))& U)
  {
   // Parallel loop over all subgrids in U
   for_all_1(i, U)
      // Trim away the ghost cell region
      Region2 inside = grow(U(i).region(), -1);
      // Copy interior boundary data from overlapping grids
      for_1(j, U)
         U(j).copy(U(i), inside);
      end_for
   end_for_all
  }
```

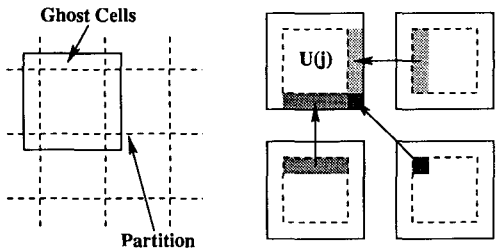

FIGURE 7 Function `FillPatch()` updates ghost cell regions of `Grid U(j)` with nonghost cell data from adjacent `Grids U(i)`.

The outer `for_all` loop iterates in parallel over all `Grids` of U. For each of these `Grids`, the `for` loop checks intersections against all of the other‡ `Grids` of U. LPARX is careful about how it handles communication; in localized computations such as Jacobi, many of these block copies are likely to be empty. In such cases, LPARX avoids unnecessary communication and copying through optimizations built into the run-time system [24].

`FillPatch()` may be converted from 2d to 3d simply by replacing `Grid2` with `Grid3` and `Region2` with `Region3`; the structural abstractions of intersection and copy-on-intersect work independently of the problem dimension. Likewise, we can replace `double` with any other C++ type or class. Note also that `FillPatch()` does not assume a simple uniform partitioning; in fact, this same code will work for any style of data partitioning.

3.5 Dynamic and Irregular Computations

We have used LPARX to develop a straightforward parallel implementation of Jacobi relaxation, a simple application requiring only a uniform static data decomposition. In this section, we briefly show how the LPARX parallelization mechanisms can be used to address dynamic, irregular computations such as structured adaptive mesh methods [9]. Details can be found elsewhere [24, 26, 27].

Structured adaptive mesh methods represent the solution to partial differential equations using a hierarchy of irregular but locally structured meshes. We represent each level of this adaptive mesh hierarchy as an `XArray` of `Grid`. Unlike the Jacobi example, each mesh level typically consists of an irregular collection of blocks. Instead of the uniform block partitioner, the application calls error estimation and regridding routines that perform data decomposition at run-time. `FillPatch()` works without change because LPARX's structural abstractions apply equally well to both uniform decompositions and irregular block structures. Of course, the adaptive application adds other routines to manage the transfer of numerical information between levels of the hierarchy (e.g., interpolation operators). The key observation, however, is that the LPARX abstractions used in the Jacobi code generalize immediately to dynamic, irregular computations.

4 PARTICLE METHODS

In Section 3, we described an application which applied work uniformly over the problem domain. We now address an application for which load balancing becomes an issue, particle dynamics. In particle dynamics, bodies called particles move under mutual interaction, congregating and dispersing unpredictably with time. The computation proceeds over a series of discrete timesteps. At each step, the algorithm evaluates the force on every particle and advances the particles in their mutually induced force field. The force evaluation is typically the most time-consuming portion of the calculation.

A naive force evaluation scheme would, for a system of N particles, calculate all $O(N^2)$ particle-particle interactions directly. Rapid approximation algorithms [2, 3] accelerate the force evaluation by calculating direct interactions only for those particles lying within a specified cut-off distance. The remaining nonlocal interactions, which we will not describe here, are calculated separately. Because the computational work required to calculate the force on a particle depends on the local particle density, a simple uniform partitioning

‡ For computations where the structure of the N blocks is simple and static, as in our Jacobi code, the $O(N^2)$ algorithm is naive. However, the communication structure for dynamic irregular computations is neither static nor regular and thus cannot be easily predicted.

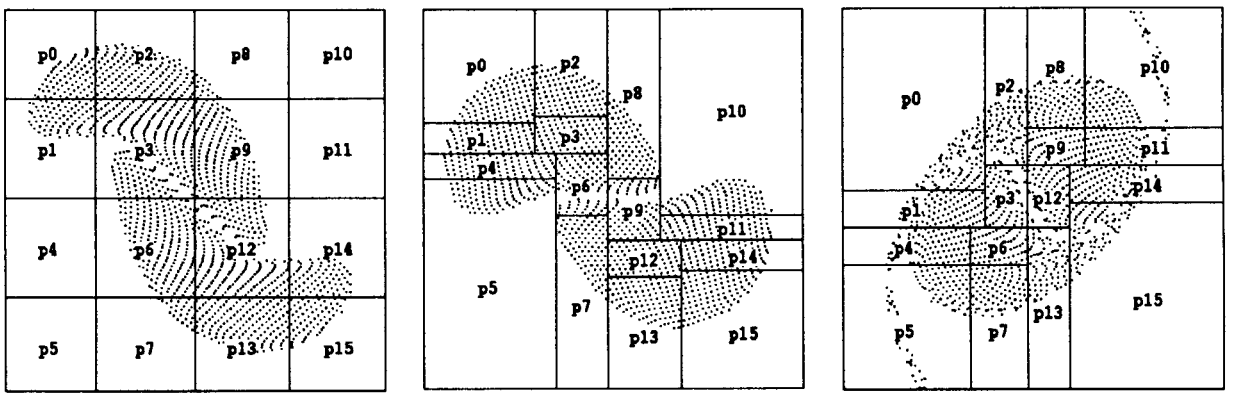

FIGURE 8 A uniform block decomposition (left) of a particle calculation is unable to balance a nonuniform workload distribution; the workload is directly related to particle density. Recursive bisection (center and right) adjusts the assignment of work to processors according to the workload distribution. Partitionings must change dynamically in accordance with the redistribution of the particles. Several repartitioning phases have occurred between the times represented by the two snapshots at the center and the right.

scheme such as that used in Section 3 cannot balance the nonuniform workloads.

To accelerate the search for neighboring particles, we sort the particles into a mesh; each element (or *bin*) of the mesh contains the particles assigned to the corresponding region of space. A 2d application represents this binning structure as a 2d `Grid` of particle lists [23]:

```
Grid2(PList) bins(domain)
```

where `PList` is the user-defined type implementing the particle list and `domain` is a `Region` which describes the computational box. The computational work carried by each bin is a function of the local density of particles, the cut-off distance, and the force law.

Particle methods are difficult to implement on parallel computers because they require dynamic load balancing to maintain an equal distribution of work; the computational cost of the force evaluation is not the same for all particles and varies with time. Furthermore, in partitioning the problem, we would like to take advantage of the spatial locality of the particle-particle interactions. By subdividing the computational space into large, contiguous blocks, we can minimize communication since nearby particles are likely to be assigned the same processor.

We illustrate the need to handle load balancing in Figure 8, which depicts a uniform block decomposition of the computational space. Each of the sixteen blocks has been assigned to a processor numbered from p0 to p15. Such a uniform decomposition does not efficiently distribute workloads; for example, no work has been assigned to processors p4, p5, p10, and p11.

A better method for decomposing non-uniform workloads is shown in Figure 8 (center and right), which illustrates two irregular block assignments rendered using recursive bisection [7]. In these decompositions, each processor receives approximately the same amount of work. Because the distribution of particles changes over time, we must periodically redistribute the workload across processors to maintain load balance.

As in the case of the Jacobi example of Section 3, we represent the parallel partitioning of the space as an array of `Regions` and the computational structure as an `XArray`:

```
XArray1(Grid2(PList)) bins
```

where each 2d `Grid` contains the `PList` data for its corresponding data partition. Note that this representation is independent of whether the partitioning is regular or irregular.

The domains of the subproblems are determined by a partitioning utility which endeavors to evenly divide the work among the processors. The partitioner requires that the application furnish an estimate of the cost of updating the particles in each bin. The binning array must be periodically repartitioned in response to the changing spatial workload distribution.

The partitioning introduces data dependencies

```
void Rebalance(XArray1(Grid2(PList))& Bins,
               XArray1(Grid2(PList))& NewBins,
               Region2 *Part)

{
(1)  Grid2(double) Work = EstimateWork(Bins);
(2)  Region2 *NewPart = RCB(Work, P);
(3)  Region2 *Ghosts = grow(NewPart, P, NGHOST);
(4)  XAlloc(NewBins, P, Ghosts);

(5)  for_all_1 (I, NewBins)
       for_1 (J, Bins)
         NewBins(I).copy(Bins(J), Part[J]);
       end_for
     end_for_all
}
```

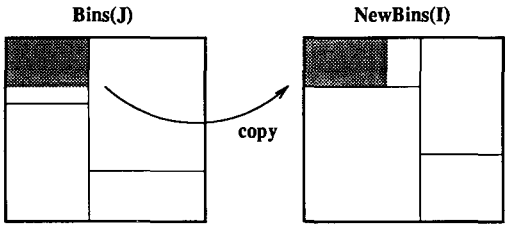

FIGURE 9 LPARX code for partitioning the particle calculation. The programmer supplies the application-specific function `EstimateWork()`. The LPARX library provides a default partitioning function `RCB()`. `P` gives the number of processors and `NGHOST` the width of the ghost cell region.

between the subproblems: particles near the boundary of a partition interact with particles belonging to other processors. One solution is to have each processor locally cache copies of off-processor particles. As in the Jacobi example, each local partition is extended with a ghost cell region. Prior to each force evaluation, these ghost regions are filled in with the most recent off-processor particle data. Unlike Jacobi, the communication structure for ghost cells is no longer regular and varies with time. However, because the region calculus hides details about data decomposition, we can use the same LPARX algorithm as in Section 3.4. In fact, we only need to change `double` to `PList` in the LPARX code of Figure 7.

In summary, there are five steps to dynamic load balancing in LPARX: (1) estimate how much work is required to compute the forces for the particles in each bin, (2) partition the bins according to (1) and assign the partitions to processors, (3) pad each partition with a ghost cell region, (4) allocate a new binning array using the partition information, and (5) copy the particles from the old binning array into the new one.

These activities are handled by `Rebalance()`, shown in Figure 9. The workload estimation (line 1) procedure is application specific and is written by the programmer; typically, timing information from the last timestep may be used. The partitioner (2) takes the workload estimate and returns a description of the balanced subproblems. LPARX provides a standard library to handle the common cases, such as recursive bisection, but because LPARX supports user-level data decompositions, programmers are free to write their own. LPARX provides a default assignment of partitions to processors, but the programmer may override the default if necessary. Finally, LPARX allocates the new binning array (4) using the partitioning. The new binning structure `NewBins` is initially empty; the two nested loops (5) copy the values from the old structure `Bins`. For each `I` and `J`, `NewBins(I)` is assigned the portion of `Grid Bins(J)` that overlaps with `J`'s associated partition.

5 COMPUTATIONAL RESULTS AND PERFORMANCE

LPARX has been implemented as a `C++` run-time library with classes for `Point`, `Region`, `Grid`, and `XArray`. A `Point` is an integer vector and is used to construct `Regions`. `Grid` elements may be standard `C++` types (`double`, `int`, `float`), structures, or other `C++` classes. Further implementation details can be found in the *LPARX User's Guide* [6].

LPARX requires only 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, many 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. Furthermore, programmers may not be required to rewrite highly optimized Fortran kernels when parallelizing an application.

Some current LPARX applications include a 2d geographically structured genetic algorithm application [20], a 3d smoothed particle hydrodynamics method [24], an adaptive eigenvalue solver for the first principles simulation of real materials [11, 24], and a dimension-independent code for 2d, 3d, and 4d connected component labeling for spin models in statistical mechanics [18]. In the following sections, we provide a brief analysis of LPARX overheads and computational results for the smoothed particle hydrodynamics application.

Table 1. Software Version Numbers and Compiler Optimization Flags for the LPARX Computations*

Machine	C++		Fortran		Operating System
	Compiler	Optimization	Compiler	Optimization	
Alphas	g++ v2.6.2	-O2	f77 v3.3	-O4	OSF/1 1.2 (PVM v3.1.5)
C-90	CC v1.0.1.1	-O2	cft77 v6.0	-O1	UNICOS 8.02.2
Paragon	g++ v2.6.2	-O2 -mnoieee	if77 v4.5.2	-O4 -Knoieee	OSF/1 1.0.4 R1.2.4
SP2	x1C v2.1	-O -Q	xlf v3.1	-O3	AIX v3.2

* All benchmarks used LPARX release v2.0.

Software versions and compiler options for all computations are reported in Table 1.

5.1 LPARX Overheads

To provide an estimate of LPARX overhead, we implemented a 3d Jacobi iterative solver with a 19-point stencil in LPARX and by hand using message passing. We chose the Jacobi application because it was simple enough to parallelize by hand. Parallelizing a real application (e.g., the smoothed particle hydrodynamics code described in the following section) without software support such as that provided by LPARX would require a significant investment of programming effort.

Table 2 compares the performance of the two codes for a $100 \times 100 \times 100$ mesh on 32 Paragon nodes. The hand-coded application made a number of 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 for the dynamic irregular applications which are the intended target of LPARX. For example, adaptive mesh calculations may assign several subgrids to each processor, and particle methods change communication dependencies as particles move.

Table 2. LPARX Overheads for a 3d Jacobi (19-point stencil) Relaxation Calculation on a $100 \times 100 \times 100$ Mesh on 32 Paragon Nodes*

	By Hand	LPARX v2.0	No Barrier
Total time (ms)	118.8	126.8	120.3
Computation (ms)	99.9	99.9	99.9
Communication (ms)	18.9	26.9	20.4
Messages (kilobytes)	36.5	38.1	37.4
Message starts	11.5	21.5	11.5

* The "By Hand" application was parallelized using only message passing. The numbers for LPARX v2.0 reflect the current LPARX implementation, and the "No Barrier" numbers estimate the performance of LPARX without the global barrier synchronization. All numbers measure the wall-clock time for one iteration of the algorithm and were averaged over 100 iterations. Message statistics represent single-processor averages for one iteration.

Table 2 reports five performance numbers: total time, computation (i.e., relaxation) time, communication (i.e., `FillPatch()`) time, total number of bytes communicated, and total number of message sends. All numbers are reported per iteration and were averaged over 100 iterations. The performance numbers in the "By Hand" column reflect the message-passing implementation; the "LPARX v2.0" column represents the performance of the current LPARX software release.

The LPARX computation time is identical to that of the message-passing code; LPARX overheads appear only in the communication routines. The LPARX communication time is 42% slower than the message-passing version. This translates into an overall execution time which is 7% longer than the equivalent message-passing code.

The LPARX code communicated approximately 5% more bytes than the message-passing implementation. Part 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 descriptive information into each message identifying the subgrid where data are to be stored. This overhead depends on the problem dimension and adds between 48 (1d) and 108 (4d) bytes to each LPARX message.

Most of LPARX's communication overhead

can be attributed to the extra messages sent as part of its synchronization protocol. At the end of a communications loop, LPARX detects the termination of communication via a global reduction and barrier synchronization, which account for the additional message sends. However, it is possible to eliminate this costly synchronization through some simple run-time schedule analysis techniques [17]. The results of these optimizations are reported in the "No Barrier" column of Table 2. LPARX overheads now drop to approximately 1% to the total execution time of the program.

5.2 Smoothed Particle Hydrodynamics

We present performance figures for a 3d smoothed particle hydrodynamics (SPH3D) application[§] [29]. The computational structure of SPH3D is similar to molecular dynamics and other particle calculations which exhibit short-range interactions. Each particle interaction is expensive, requiring approximately 100 floating point operations. Table 3 and Figure 10 present performance results for the SPH3D code on the Cray C-90, Intel Paragon, IBM SP2,[‖] and a network of Alpha workstations connected by a GIGAswitch running PVM. Time is reported in seconds per timestep. Simulations were run with 12k, 24k, 48k, and 96k particles. All floating point arithmetic was performed using 64-bit numbers. The applications code was identical on all machines except that the C-90 version gathered and scattered particles to obtain longer vector lengths.

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, vectors are still quite short. These vectorization limitations are intrinsic to the algorithm and are not artifacts of parallelization. For 48k particles, hardware performance monitors on the C-90 reported an average performance of 200 megaflops and an average vector length of sixty. For this problem size, one processor of the C-90 is roughly equivalent to 8 Alpha processors, 16 Paragon processors, or 4 SP2 processors.

The C-90 and the Alpha cluster exhibited relatively poor performance on the smallest problem size (12k particles). The Alphas suffered because of the high overheads of message passing through PVM; in larger problems, this overhead was hidden by the increased computational costs of particle interactions. Poor performance on the C-90 was due to short vector lengths. The Cray C-90 times improved relative to the other machines for the largest problem size because of increased vector lengths. Because LPARX applications are portable across a diversity of high-performance machines, the computational scientist may choose the most cost-effective architecture (e.g., Cray C-90 or Alpha cluster) for a particular problem size.

6 RELATED WORK

LPARX's `Region` abstraction and its region calculus are based in part on the domain abstractions explored in the scientific programming language FIDIL [22]. FIDIL's domain calculus provides operations such as union and intersection over arbitrary index sets; LPARX extends FIDIL's calculus operations to provide structural abstractions for data decomposition and interprocessor communication. Whereas FIDIL supports the notion of arbitrary nonrectangular index sets, LPARX restricts index sets to be rectangular. A prototype of LPARX, called LPAR [5], supported FIDIL-style regions, but these incurred a high-performance penalty in applications that did not require the added generality. We believe that the programmer should pay the price of such generality only when necessary and advocate the inclusion of more general FIDIL regions as a separate type. Crutchfield and Welcome [15] independently developed similar abstractions based on FIDIL for single-processor architectures. Based on this framework, they have developed domain-specific libraries for adaptive mesh refinement applications in gas dynamics. The LPARX software is currently being employed to support these libraries on parallel computer architectures. A form of region abstraction is used in the programming language ZPL [28], although ZPL's regions are not first class objects; they are used as an execution mask for data parallel computation whereas LPARX ab-

§ The original sequential 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).

‖ The SP2 results were obtained on a preproduction machine at the Cornell Theory Center; these times should improve as the system is tuned and enters full production use.

Table 3. This Table Presents SPH3D Performance Results on a Cray C-90, Intel Paragon, IBM SP2, and an Alpha Workstation Farm Running PVM*

Particles	Cray C-90 Time P = 1	Alpha Time P = 8
12K	2.90	2.2
24k	5.98	4.98
48k	14.8	15.6
96k	38.4	56.4

Intel Paragon Performance

	P = 8		P = 16		P = 32		P = 64	
Particles	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
12k	2.23	1.0	1.40	1.6	0.998	2.2	0.722	3.1
24k	7.61	1.0	4.73	1.6	2.50	3.0	1.68	4.5
48k	28.5	1.0	17.8	1.6	7.72	3.7	4.73	6.0
96k	114	1.0	70.7	1.6	27.8	4.1	17.6	6.5

IBM SP2 Performance

	P = 4		P = 8		P = 16	
Particles	Time	Speedup	Time	Speedup	Time	Speedup
12k	1.28	1.0	0.803	1.6	0.563	2.3
24k	4.12	1.0	2.44	1.7	1.57	2.6
48k	16.1	1.0	8.85	1.8	5.50	2.9
96k	59.9	1.0	34.6	1.7	21.5	2.8

* All times are in seconds per timestep. Cray times were averaged over 5 timesteps, Alpha times over 50 timesteps, and all other times over 100 timesteps. The C-90 measurements are CPU times on a production system; measurements on the Alpha farm, Paragon, and SP2 are wall-clock times since processor nodes are not time shared. For the Paragon and SP2, speedups are reported relative to the smallest number of processors used to gather data. Some of these numbers are presented in Figure 10.

stractions specify data decomposition and express communication dependencies.

In contrast to LPARX, HPF [21] represents data decompositions using an abstract index space called a template. Arrays are mapped to templates and templates are decomposed across processors through compile-time directives. Because templates are not language-level objects, the programmer has limited control over data decomposition; templates cannot be defined at run-time nor passed across procedure boundaries. To avoid these limitations, Vienna Fortran [12] omits templates but includes more general data decomposition directives. However, data decompositions are restricted to tensor products of 1d irregular decompositions, and it is unclear how Vienna Fortran will support dynamic irregular blocking structures such as those required by adaptive mesh refinement and recursive coordinate bisection decompositions.

The pC++ programming language [10] implements a collection abstraction which includes a coarse-grain data parallel loop over objects within the collection; pC++ employs a data decomposition scheme similar to that of HPF. The Multiblock PARTI [1] and CHAOS [16] libraries provide run-time support for data parallel compilers such as HPF. CHAOS is targeted towards unstructured calculations such as sweeps over finite-element meshes or sparse matrix calculations. Multiblock PARTI has been targeted to applications with a small number of large, static blocks (e.g., irregularly coupled regular meshes [14]).

7 CONCLUSIONS

LPARX is a portable programming model and run-time system which supports coarse-grain data parallelism efficiently over a wide range of MIMD parallel platforms. Its structural abstraction enables the user to manipulate data decompositions as

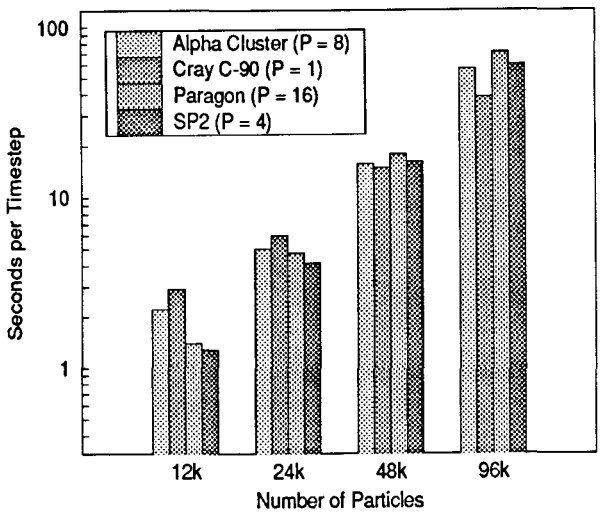

FIGURE 10 This graph compares SPH3D performance results for a Cray C-90, Intel Paragon, IBM SP2, and an Alpha workstation farm running PVM. Measurements were gathered as described in Table 3. The number of processors for a particular machine was chosen to provide performance roughly comparable to a single processor of a Cray C-90; processor numbers are given in parentheses.

first-class objects. Its philosophy is that data partitioning for irregular algorithms are heavily problem dependent and therefore must be under the control of the application. LPARX is intended for applications with changing nonuniform workloads, such as particle methods, and for calculations with dynamic, block-irregular data structures, such as adaptive mesh refinement. LPARX does not apply to unstructured methods, e.g., sweeps over finite element meshes or sparse matrix problems, which are supported by CHAOS.

LPARX provides three new data types: an index set-valued object called a `Region`, a dynamic array called a `Grid`, and an array of distributed coarse-grain elements called an `XArray`. Efficient block copies over `Grids` hide interprocessor communication and low-level bookkeeping details. Coarse-grain data parallelism over `XArrays` is provided via the `for_all` loop. Region calculus operations express data dependencies in geometric terms independently of the spatial dimension and data decomposition. Such abstractions enable the programmer to reason about an algorithm at a high level. While we have emphasized the utility of structural abstraction for multiprocessors, such methods also apply to single-processor systems. Many scientific applications, such as adaptive mesh refinement, exhibit irregular data structures and communication patterns independent of the parallelism. The region calculus provides a powerful methodology for describing and managing such irregularity.

LPARX reduces software development costs by promoting reusability and portability. Applications can be developed and debugged on workstations and then moved to parallel platforms for production runs. Existing optimized serial numerical kernels may be used, often with few changes, in parallelized applications. Region calculus operations are dimension independent; thus, the same data decomposition and data communication code can be used for 2d and 3d versions of an application. The programmer can develop a simpler 2d version of the problem, and, when confident that the code has been debugged, apply the computational resources of a parallel machine to the full 3d application.

LPARX separates parallel execution and data communication from computation. Numerical kernels may be optimized and tuned for a processing node without regard to the higher-level parallelism. Parallel performance is intimately tied to single-node performance; therefore, it is vital that computational scientists can optimize numerical code to reflect specific node characteristics.

The true test of a software development tool is whether it is accepted by the user community. LPARX is currently employed in diverse scientific collaborations, including adaptive mesh refinement for gas dynamics, smoothed particle hydrodynamics, genetics algorithms, and adaptive eigenvalue solvers for the first principles simulations of real materials. The LPARX software is being used by researchers at the University of California, San Diego, George Mason University, Lawrence Livermore National Labs, and the Cornell Theory Center. LPARX has enabled scientists to reduce the development time of challenging applications on high-performance parallel computers.

SOFTWARE AVAILABILITY

The LPARX software libraries, the smoothed particle hydrodynamics application, and an adaptive mesh refinement method for eigenvalue problems in materials design are available via the World Wide Web at address `http://www-cse.ucsd.edu/users/skohn/lparx.html`. The

software is also available through the San Diego Supercomputer Center.

ACKNOWLEDGMENTS

We thank Greg Cook, Stephen Fink, Chris Myers, and Charles Rendleman for their useful suggestions on how to improve LPARX. This work was supported by NSF contract ASC-9110793 and ONR contract N00014-93-1-0152. Intel Paragon and Cray C-90 time were provided by a UCSD School of Engineering Block Grant. Access to the DEC Alpha workstation farm was provided by the San Diego Supercomputer Center, and access to the IBM SP2 was provided by the Cornell Theory Center. Portions of this paper are taken from Kohn's Ph.D. dissertation [24].

REFERENCES

[1] G. Agrawal, A. Sussman, and J. Saltz, "An integrated runtime and compile-time approach for parallelizing structured and block structured applications," *IEEE Trans. Parallel Distrib. Systems* (in press).

[2] A. Almgren, T. Buttke, and P. Colella, "A fast vortex method in three dimensions," in *Proceedings of the 10th AIAA Computational Fluid Dynamics Conference*, Honolulu, Hawaii, June 1991, pp. 446–455.

[3] C. R. Anderson, "A method of local corrections for computing the velocity field due to a distribution of vortex blobs," *J. Computational Phys.*, vol. 62, pp. 111–123, 1986.

[4] S. B. Baden, S. J. Fink, and S. R. Kohn, "Structural abstraction: A unifying parallel programming model for data motion and partitioning in irregular scientific computations," in preparation, 1996.

[5] S. B. Baden and S. R. Kohn, "Portable parallel programming of numerical problems under the LPAR system," *J. Parallel Distrib. Comput.*, vol. 27, pp. 38–55, 1995.

[6] S. B. Baden, S. R. Kohn, S. M. Figueira, and S. J. Fink, "The LPARX user's guide v2.0," University of California—San Diego, La Jolla, CA, Tech. Rep., Nov. 1994.

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

[8] M. J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *J. Computational Phys.*, vol. 82, pp. 64–84, 1989.

[9] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *J. Computational Phys.*, vol. 53, pp. 484–512, 1984.

[10] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang, "Distributed pC++: Basic ideas for an object parallel language," *J. Sci. Programming*, vol. 2, 1993.

[11] E. J. Bylaska, S. R. Kohn, S. B. Baden, A. Edelman, R. Kawai, M. E. G. Ong, and J. H. Weare, "Scalable parallel numerical methods and software tools for material design," in *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, "San Francisco, CA, February 1995.

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

[13] B. Chapman, P. Mehrotra, and H. Zima, "Extending HPF for advanced data parallel applications," ICASE, Tech. Rep. 94-34, May 1994.

[14] C. Chase, K. Crowley, J. Saltz, and A. Reeves, "Parallelization of irregularly coupled regular meshes," ICASE, NASA Langley Research Center, Tech. Rep. 92-1, January 1992.

[15] W. Y. Crutchfield and M. L. Welcome, "Object oriented implementation of adaptive mesh refinement algorithms," *J. Sci. Programming*, vol. 2, pp. 145–156, 1993.

[16] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang, "Communication optimizations for irregular scientific computations on distributed memory architectures," *J. Parallel Distrib. Comput.* (to appear).

[17] S. J. Fink, S. B. Baden, and S. R. Kohn, "Flexible communication schedules for block structured applications," in preparation, 1996.

[18] S. J. Fink, C. Huston, S. B. Baden, and K. Jansen, "Parallel cluster identification for multidimensional lattices" *IEEE Trans. Parallel Distrib. Systems*, 1995 (submitted).

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

[20] W. E. Hart, "Adaptive global optimization with local search," PhD thesis, University of California at San Diego, 1994.

[21] High Performance Fortran Forum, High Performance Fortran Language Specification, November 1994.

[22] P. N. Hilfinger and P. Colella, "FIDIL: A language for scientific programming," Lawrence Livermore National Laboratory, Livermore, CA, Tech. Rep. UCRL-98057, Jan. 1988.

[23] R. W. Hockney and J. W. Eastwood, *Computer*

Simulation Using Particles. New York: McGraw-Hill, 1981.

[24] S. R. Kohn, "A parallel software infrastructure for dynamic block-irregular scientific calculations," PhD thesis, University of California at San Diego, June 1995.

[25] S. R. Kohn and S. B. Baden, "A robust parallel programming model for dynamic non-uniform scientific computations," in *Proceedings of the 1994 Scalable High Performance Computing Conference*, May 1994.

[26] S. R. Kohn and S. B. Baden, "A parallel software infrastructure for structured adaptive mesh methods," in *Proceedings of Supercomputing '95*, December 1995.

[27] S. R. Kohn and S. B. Baden, "The parallelization of an adaptive multigrid eigenvalue solver with LPARX," in *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[28] C. Lin and L. Snyder, "ZPL: An array sublanguage," in U. Banerjee, D. Gelernter, A. Nicolau, and P. Padua, Eds., *Proceedings of the Sixth International Workshop on Languages and Compilers for Parallel Computation.* New York: Springer-Verlag, 1994, pp. 96–114.

[29] J. J. Monaghan, "Smoothed particle hydrodynamics," Annu. Rev. Astronomy Astrophys., vol. 30, pp. 543–574, 1992.

[30] V. S. Sunderam, "PVM: A framework for parallel distributed computing," Concurrency Practice Exp., vol. 2, pp. 315–339, 1990.

[31] M. Wu and G. Fox, "Fortran 90D compiler for distributed memory MIMD parallel computers," Syracuse University, Tech. Rep. SCCS-88B, 1991.