Run-time and Compile-time Support for Adaptive Irregular Problems *

Shamik D. Sharma Ravi Ponnusamy Bongki Moon
Yuan-shin Hwang Raja Das Joel Saltz

UMIACS and Dept. of Computer Science,
University of Maryland,
College Park, MD 20742
dybbuk@cs.umd.edu

Abstract

In adaptive irregular problems, data arrays are accessed via indirection arrays, and data access patterns change during computation. Parallelizing such problems on distributed memory machines requires support for dynamic data partitioning, efficient preprocessing and fast data migration. This paper describes CHAOS, a library of efficient runtime primitives that provides such support. To demonstrate the effectiveness of the runtime support, two adaptive irregular applications have been parallelized using CHAOS primitives: a molecular dynamics code (CHARMM) and a code for simulating gas flows (DSMC). We have also proposed minor extensions to Fortran D which would enable compilers to parallelize irregular forall loops in such adaptive applications by embedding calls to primitives provided by a runtime library. We have implemented our proposed extensions in the Syracuse Fortran 90D/HPF prototype compiler, and have used the compiler to parallelize kernels from two adaptive applications.

1 Introduction

In irregular concurrent problems, patterns of data access cannot be predicted until runtime. In such problems, optimizations that can be carried out at compile-time are limited. At runtime, however, the data access pattern in a loop nest is usually known before entering the loop nest; this makes it possible to utilize various preprocessing strategies to optimize the computation. These preprocessing strategies primarily deal with reducing data movement between processor memories in distributed memory machines. Such preprocessing methods are being developed for a variety of unstructured problems, including explicit multi-grid unstructured computational fluid dynamic solvers, molecular dynamics codes, diagonal or polynomial preconditioned iterative linear solvers, and particle-in-cell (PIC) codes.

Figure 1 illustrates a typical irregular loop. The *indirection arrays*, **ia** and **ib** are read in at runtime. After the indirection arrays are known, a preprocessing step partitions the *data arrays* (i.e. arrays **x**, **y**) and the loop iterations (i.e. indirection arrays **ia**, **ib**) over processors. After partitioning, it is possible to identify the non-local global elements that are accessed by each processor in the loop nest and thereby determine the communication requirements. Various communication optimizations can be applied at this stage. Communication volume can be reduced by eliminating duplicate references. The number of messages can also be reduced by combining prefetches of off-processor data into a single message. These optimizations are called *software caching* and *communication vectorization* respectively.

```
      real    x(max_nodes), y(max_nodes)
      integer ia(max_edges), ib(max_edges)

L1:   do n = 1, n_step
L2:     do i = 1, sizeof_indirection_arrays
          x(ia(i)) = x(ia(i)) + y(ib(i))
        end do
      end do
```

Figure 1: An irregular loop nest

Such optimizations have been successfully used earlier to parallelize *static* irregular problems [7], in which array access patterns do not change during computation. For the static irregular problems considered in Das et al. [7], it is enough to perform preprocessing only once to optimize communication. *Adaptive* irregular problems are more complex. In these problems, the data access patterns may change during computation - each such change results in new communication requirements. This situation usually arises either because the indirection arrays are modified during computation, or because the data arrays are redistributed across processors. In either case, preprocessing must be repeated in order to determine the new communication requirements and to perform optimizations such as software caching and communication vectorization.

This paper presents a library of runtime procedures designed to efficiently implement adaptive irregular programs on distributed memory machines. This runtime library is called CHAOS; it subsumes PARTI, a library aimed at static irregular problems [24]. Specifically, CHAOS introduces two new features — *lightweight schedules* and *hashed schedule generation*, which are useful in certain adaptive problems. We have used CHAOS to parallelize two challenging adaptive applications — CHARMM, a molecular dynamics code [4] and DSMC, a code that simulates gas flows [27]. We also present language support that can enable compilers to generate efficient code for adaptive applications. The Syracuse Fortran 90D/HPF compiler was used as a test-bed for the ideas presented in this paper. The compiler work described here is based on [22] – a seminal paper describing compiler-embedded partitioners.

The paper is structured as follows. In Section 2, we motivate our work by describing two adaptive irregular applications, CHARMM and DSMC. Section 3 describes our runtime support. Section 4 demonstrates the performance of the applications parallelized using CHAOS. Section 5

*This work was supported by NASA under contract No. NAG-11560, by ONR under contract No. SC 292-1-22913 and by ARPA under contract No. NAG-11485. The authors assume all responsibility for the contents of the paper.

Supercomputing, Washington D.C., Novemeber 1994, pages 97-106.

gives an overview of existing language support for irregular data decomposition; it describes language directives that allow users to remap data efficiently. Section 6 discusses related work, and conclusions are presented in Section 7.

2 Adaptive Applications

This section briefly describes the computational structure of two adaptive irregular application programs — Chemistry at HARvard Macromolecular Mechanics (CHARMM), a molecular dynamics code, and Direct Simulation Monte Carlo (DSMC), a code for simulating motion of gas particles. These are large applications; each consists of thousands of lines of code.

2.1 CHARMM

CHARMM is a molecular dynamics simulation program used to derive the structural and dynamic properties of macromolecular systems. The code simulates dynamic interactions among all atoms in the system for a specified period of time. At each time step, the simulation uses the spatial position of the atoms to calculate the forces acting on each atom. It then integrates Newton's equations of motion to determine the new spatial position of each atom for the next time step.

The force calculations in the simulation consist of two types of interactions – bonded forces and non-bonded forces. Bonded forces are due to short-range interactions which remain unchanged during the simulation. Non-bonded forces are long-range interactions existing between each pair of atoms; the the time complexity of calculating them is $O(N^2)$, (N is the total number of atoms). CHARMM approximates the non-bonded calculation by ignoring all interactions which are beyond a certain cutoff distance from each atom. The approximation is achieved by maintaining a *non-bonded list* of interacting partners for each atom, and iterating over this list at each timestep. Thus the non-bonded list is used as an indirection array to identify interacting partners. Since atoms change their spatial positions every iteration, the non-bonded list must be periodically updated. In CHARMM, users can control the frequency at which the non-bonded list is regenerated. Typically, it is regenerated after every 10 to 100 timesteps. Each regeneration results in new communication requirements - therefore, preprocessing needs to be repeated to determine the new communication requirements, and to optimize them.

2.2 Direct Simulation Monte Carlo

The DSMC application is used to study the behavior of a gas by simulating the movement and collisions of a large number of molecules over a period of time. The DSMC simulation method involves laying out a Cartesian grid over the domain, and associating each molecule with the cell it lies in. Molecules are assumed to interact with other molecules only if both molecules are in the same cell at the same timestep. Since the computation in each cell is independent of computations in other cells, substantial parallelism can be extracted by distributing the cells and their constituent molecules across processors.

However, there are three significant impediments to parallelization of DSMC. First, since molecules are in continuous motion, some of them change their cells every timestep. The communication cost of moving molecules between cells every timestep can be substantial on distributed memory computers. Second, indirection arrays used to access

```
do i = 1, num_cells
 do j = 1, size(i)
  new_cells(icell(i,j), new_size(icell(i,j)))= cells(i,j)
  new_size(icell(i,j))=new_size(icell(i,j))+1
 end do
end do
```

Figure 2: Code fragment resembling DSMC's MOVE

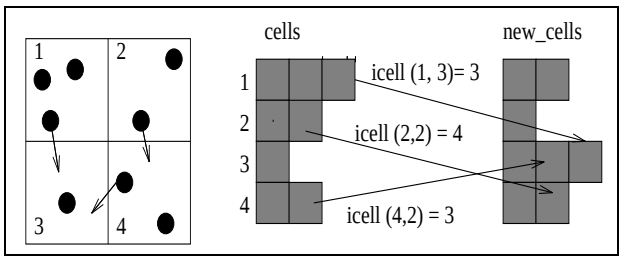

Figure 3: Data movement in DSMC

molecules within each cell must be regenerated every timestep. Third, as molecules move between cells, the workload in each cell keeps changing, affecting the net computational load balance. To maintain reasonable load balance, cells may be periodically redistributed across processors. These characteristics of DSMC are also found in particle-in-cell applications.

Figure 2 shows a code fragment which resembles *MOVE*, the DSMC procedure which moves molecules between cells after each computational time step. Elements migrate across the rows of a 2-D array based on the information provided in indirection array **icell**. The elements of array **cells** are shuffled and stored in array **new_cells**. While the total number of rows remains the same after the shuffle, the size of individual rows may not. Also of crucial importance is that the sequence in which elements are appended to their new rows is not relevant, since the order of computation over these elements does not matter. Knowledge of such application-dependent information enables users to use more efficient CHAOS communication primitives designed for such situations. Section 5.2.1 shows how such information can be conveyed to an optimizing compiler.

The MOVE procedure in DSMC is much more complex than the code fragment shown in Figure 2. Our runtime support has been used to parallelize the real code; however, for testing our compiler implementation, we have used a computational template which is similar to the code fragment shown here.

3 Runtime Support

This section describes the principles and functionality of the CHAOS runtime support library, a superset of the PARTI library [18, 28, 24]. First, a brief overview of the runtime support is presented; the framework is the same as that of PARTI, and has been described in earlier papers [7, 22]. We then focus on the inspector, which is a preprocessing stage that must be repeated frequently in adaptive problems. We introduce light-weight schedules which are used for fast data migration and describe how we have optimized the inspector to generate schedules efficiently.

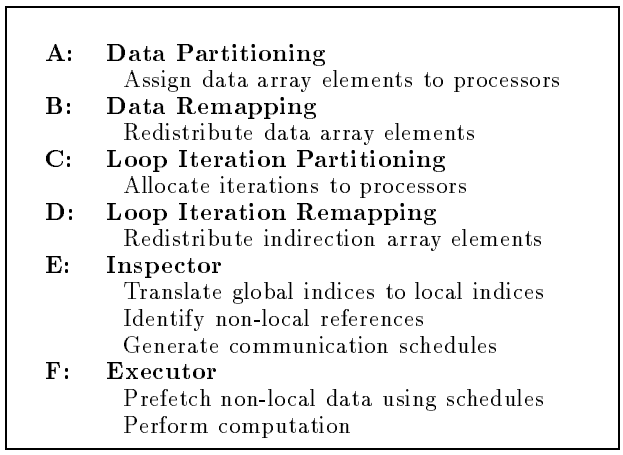

Figure 4: Solving irregular problems with CHAOS

3.1 Overview of CHAOS

The CHAOS runtime library has been developed to efficiently handle irregular problems that consist of a sequence of clearly demarcated concurrent loop-nests. Parallelizing such irregular problems on distributed memory machines using the CHAOS runtime support involves six phases (Figure 4). The first four phases concern mapping data and computations onto processors. The next two steps involve analyzing data access patterns in a loop nest and generating optimized communication calls. A brief description of these phases follows.

A: Data Partitioning : Phase A determines how data arrays are to be partitioned across processors. CHAOS supports a number of parallel partitioners that partition data arrays using heuristics based on spatial positions, computational load, connectivity, etc. [8]. These partitioners return an irregular assignment of array elements to processors, which is stored as a CHAOS construct called the *translation table*. A translation table is a globally accessible data structure which lists the home processor and offset address of each data array element. The translation table may be replicated, distributed regularly, or stored in a paged fashion, depending on storage requirements. (In the section on compile-time support, Section 5.1, we have followed the Fortran D convention of representing irregular distributions as *maparrays*, which are equivalent to translation tables.)

B: Data Remapping : Phase B remaps data arrays from the initial distribution to the newly calculated irregular distribution. Note that phase A, does not actually move any data; it only determines a good mapping. The same mapping may be used to remap several arrays.

C: Loop Iteration Partitioning : Phase C determines how loop iterations should be partitioned across processors. CHAOS uses the *almost-owner-computes* rule to assign loop iterations to processors. Each iteration is assigned to the processor which owns a majority of data array elements accessed in that iteration. This heuristic is biased towards reducing communication costs. The owner-computes rule can also be used with CHAOS.

D: Loop Iteration Remapping : In Phase D, indirection array elements are remapped to conform with the loop iteration partitioning.

E: Inspector : Phase E carries out the preprocessing needed to identify the communication requirements, generating optimized communication schedules to prefetch non-local data, and translating the indirection array's global indices to local indices. This phase is described in more detail in Section 3.2.

F: Executor : Phase F uses information from the earlier phases to prefetch off-processor data, to execute the loop nest and to move elements back to their home locations after the computation. Communication is carried out by CHAOS data movement primitives (e.g. `gather`, `scatter`) which use communication schedules constructed in Phase E.

In static irregular problems, Phase F is typically executed many times, while the preprocessing phases, A through E, are executed only once. In adaptive problems some of the preprocessing steps need to be repeated. For instance, in some adaptive problems data access patterns change periodically but reasonable load balance is still maintained. In such applications, phase E must be repeated whenever the data access pattern changes. In even more highly adaptive problems, the data arrays may need to be repartitioned in order to maintain load balance. In such applications, all the phases described above may be repeated.

3.2 Inspector

The inspector phase has two goals — index translation and communication schedule generation. Index translation involves converting the global array indices in indirection arrays into local indices. The purpose of index translation has been discussed in greater detail elsewhere [7]. Communication schedule generation involves analyzing data access patterns and performing optimizations such as software caching and communication vectorization. In adaptive problems, data access patterns are modified frequently; hence index translation and schedule regeneration are repeated many times.

3.2.1 Communication Schedules

After data arrays and loop iterations have been partitioned across processors, the communication requirements of each loop nest can be determined. This information can be used for communication optimizations, such as removing duplicates and aggregating messages. The result of these optimizations is a communication schedule, which is used by CHAOS data movement primitives (`gather`, `scatter` and `scatter_append`) to move data efficiently. A schedule for processor p stores the following information:

1. send list — a list of local array elements that must be sent to other processors,
2. permutation list — an array that specifies the data placement order of incoming off-processor elements, (in a local buffer area which is designated to receive incoming data),
3. send size — an array that specifies the sizes of outgoing messages from processor p to other processors,

```
L1:  do n = 1, nsteps
L2:    do i = 1, sizeof_indirection_arrays
         x(ia(i)) = x(ia(i)) + y(ia(i)) * y(ib(i))
       end do

L3:    do i = 1, sizeof_ic
         x(ic(i)) = x(ic(i)) + y(ic(i))
       end do
     end do
```

Figure 5: A code fragment with two irregular loops

4. fetch size – an array that specifies the sizes of incoming messages to processor p from other processors.

While a communication schedule can be generated for the data access pattern of each irregular loop nest, there are significant advantages in considering many such loops simultaneously. For instance, consider the loops L2 and L3 in Figure 5. In loop L2, data arrays **x** and **y** are accessed through data access patterns specified by indirection arrays **ia** and **ib**. In loop L3, the same data arrays are accessed through indirection array **ic**. Instead of building two separate communication schedules, loop L3 can reuse many of the off-processor elements of array **y** brought in by the schedule for loop L2. Thus, only an *incremental schedule* for loop L2 needs to be built [7]. The incremental schedule gathers only those elements of **y** which were not brought in by earlier schedules. Another optimization that can be applied in this example is *schedule merging*. Instead of building separate schedules for gathering off-processor elements of **y**, one could build a single schedule that gathers all elements of **y** required by both loops. While PARTI provided support for building incremental and merged schedules, these primitives were not designed for adaptive applications, where such optimizations must be performed repeatedly. CHAOS allows such schedule optimizations to be performed frequently (Section 3.2.2).

CHAOS also supports specialized communication schedules. For some adaptive applications, particularly those from the particle-in-cell domain, there is no significance attached to the placement order of incoming array elements. Such application-specific information can be used to build much cheaper *lightweight* communication schedules. In inspectors for such applications, index translation is not required, and the permutation list need not be generated for the schedule data structure. Besides being faster to construct, lightweight schedules also speed up data movement by eliminating the need for rearranging the order of incoming off-processor elements. Lightweight schedules have been used in the parallelization of DSMC. In Section 5.2.1 we show how lightweight schedules can be used by a compiler.

3.2.2 Optimizing the Inspector

CHAOS provides efficient runtime primitives for analyzing data access patterns and generating optimized schedules. The inspector phase is carried out in two steps — these steps are called index analysis and schedule generation. In the index analysis stage, the data access pattern is analyzed to determine which references are off-processor; duplicate off-processor references are eliminated from the list of elements to be fetched and global indices are translated to local indices. To remove duplicates, we use a hash table. This hash table is also used for storing all results of index analysis for later reuse. In the schedule generation stage, the hash table entries are read and the communication schedule data structure is constructed. The principal advantage of using such a two-step process is that some of the index analysis can be reused in adaptive applications.

A hash table stores global index references obtained from indirection arrays. For each global index hashed, the hash table stores the following information :

1. global index — the global index accessed.
2. translated address — the processor and offset where the element is stored. This information is accessed from the translation table.
3. local index — the local buffer address assigned to cache a copy of the element, if it is off-processor.
4. stamp — this is an integer, used to identify which indirection array entered the element into the hash table. The same global index entry might be hashed in by many different indirection arrays; a bit in the stamp is marked for each such entry.

Index analysis is expensive. However, building a hash table for duplicate removal is usually worth the effort. In CHARMM, for example, the non-bonded partners of nearby atoms are almost identical, and duplicate removal greatly reduces communication volume. Translation table lookup is another costly part of index analysis especially if a non-replicated translation table is used, in which case, communication is required to find the translated address. However, much of the costs of index analysis can be amortized by building the hash table and maintaining it throughout the computation. In most adaptive applications, indirection arrays keep changing; however, the adaptation is slow enough that most of the indirection array elements remain unchanged. Index analysis for these unchanged indices involves only a lookup in the hash table.

Indirection arrays are hashed in by the CHAOS procedure `CHAOS_hash`. This primitive enters all the indices into a designated hash table and returns an identifying stamp. The stamp identifies all entries in the hash table that correspond to that indirection array. The entries in the indirection arrays have their global indices changed to local indices in the course of hashing. A procedure called `CHAOS_schedule` is then used to construct communication schedules from the entries in the hash table. For a given stamp, this primitive extracts all entries in the hash table with that stamp and constructs a communication schedule. By specifying different logical combinations of stamps, we can build merged or incremental schedules.

In CHARMM, the indirection array specifying the non-bonded partners of each atom is changed periodically. Whenever such an change occurs, the new indirection array must be rehashed into the indirection array. However, most of the entries in the new indirection array can be found in the old hash table; thus, the cost of index analysis is greatly reduced. Moreover, if the modified indirection array participates in a merged or incremental schedule, the cost of regenerating a new schedule of this type is minimized; we can reuse information about the unchanged indirection arrays maintained in the hash table.

4 Experimental Results

This section describes how CHAOS was applied to the targeted applications, CHARMM and DSMC. Experimental

results were obtained on the Intel iPSC/860.

4.1 CHARMM

Recall that the critical part of CHARMM is its non-bonded force calculation phase. This is an adaptive computational phase since the non-bonded interaction list for each atom is modified periodically, as atoms change their spatial positions.

4.2 Parallelization Approach

The computation associated with an atom primarily depends on the number of entries in the non-bonded list that atom. Therefore, data partitioners which use spatial information and computational load perform significantly better than simple BLOCK or CYCLIC distributions. We have used a recursive coordinate bisection (RCB) partitioner and a recursive inertial bisection (RIB) partitioner on CHARMM with almost identical results. All data arrays that are associated with atoms are distributed in an identical fashion, so we do not have to separately partition each array. Information about the distribution is stored in a replicated translation table.

Since the non-bonded loop consumes 90% of the execution time, balancing the computational load due to these calculations is of primary concern. Currently, the entire non-bonded force computation of each atom is assigned to the processor that owns the atom (owner-computes). The load balance achieved depends on the quality of the data distribution returned by the data partitioner. The bonded force calculation loop is partitioned using the *almost-owner-computes* rule described in Section 3.1.

Indirection arrays used in bonded force computation remain unchanged while the non-bonded list adapts during computation. Hence, preprocessing for bonded force calculation loops need not be repeated. Preprocessing must be repeated for the non-bonded force calculation loops whenever the non-bonded list changes. An array containing the non-bonded lists of all on-processor atoms is hashed as described in Section 3.2.2. The hash table removes all duplicate references among the non-bonded lists, and assigns local memory locations where data from off-processor atoms can be stored.

4.2.1 Performance

The performance of CHARMM, parallelized using CHAOS, was studied for one benchmark case (MbCO + 3830 water molecules) on the Intel iPSC/860. The program was run for 1000 timesteps with the cutoff distance for non-bonded interaction set to 14 Å. The non-bonded list was updated 40 times during the simulation. The results are presented in Table 1. These results were obtained with a recursive coordinate bisection (RCB) partitioning of atoms. The execution time in the table specifies the maximum of the net execution time over all processors. The computation and communication times were averaged over processors. The load balance index was calculated as

$$LB = \frac{(\max_{i=1}^{n} comp.\ time\ of\ proc.\ i) \times (\#\ of\ procs.,\ n)}{\sum_{i=1}^{n} comp.\ time\ of\ proc.\ i}$$

As can be seen from Table 1, CHARMM scaled well and good load balance was maintained up to 128 processors[1].

[1] Brooks and Hodoscek estimate that it would take 74595.5 seconds to execute the same benchmark on a single node of the iPSC/860 [5].

Table 1: Performance of Parallel CHARMM on Intel iPSC/860 (time in sec.)

Procs.	16	32	64	128
Total time	4356.0	2293.8	1261.4	781.8
Compute time	4099.4	2026.8	1011.2	507.6
Comm. time	147.1	159.8	181.1	219.2
Load Bal.	1.03	1.05	1.06	1.08

Table 2: Preprocessing Overheads of CHARMM (in sec.)

Processors	16	32	64	128
Data Partitioning	0.27	0.47	0.83	1.63
N.B. list Generation	7.18	3.85	2.16	1.22
Remapping	0.03	0.03	0.02	0.02
Sched. Gen.	1.31	0.80	0.64	0.42
Sched. Regen.(×40)	43.51	23.36	13.18	8.92

The preprocessing overheads of the simulation are shown in Table 2. The data partitioning time shown in the table reflects the execution time of the RCB partitioner. The non-bonded list is generated after the atoms are partitioned. The remapping time reflects the cost of remapping all the arrays associated with the atoms to the new distribution determined by the partitioner. The table shows the time to generate the optimized communication schedules for the first time as well as the total time spent in repeating the preprocessing for the 40 non-bonded list updates during the simulation. By comparing these times to those in Table 1, it can be observed that the preprocessing overhead is relatively small when compared to the total execution time.

Several indirection arrays are used in CHARMM's bonded and non-bonded force calculations to reference the same data arrays in different loop nests. One possible approach to prefetching is to compute separate schedules for gathering and scattering off-processor data for each loop nest. A second approach is to form merged schedules whenever possible, as discussed in Section 3.2.1. Table 3 compares the performance of these two approaches and demonstrates the usefulness of schedule merging.

4.3 DSMC

The computational characteristics of the DSMC code were described in Section 2.2. Recall that the interesting feature of DSMC was the motion of gas molecules between cells of a 2-D or 3-D Cartesian grid every timestep.

Table 3: Communication Time (in sec.)

Procs.	Merging sched.		no Merging	
	Comm.	Total	Comm.	Total.
16	147.1	4356.0	182.1	4427.5
32	159.8	2293.8	201.0	2364.2
64	181.1	1261.4	223.2	1291.9
128	219.2	781.8	253.1	815.2

Table 4: Regular Schedules vs. Lightweight Schedules (2-D DSMC)

(Time in secs)	48x48 Cells				96x96 Cells			
	Processors				Processors			
	16	32	64	128	16	32	64	128
Regular Schedules	63.74	50.50	79.58	95.50	226.89	131.99	125.64	118.89
Light-Weight Schedules	20.14	11.54	7.60	6.77	79.89	40.46	21.77	14.23

Table 5: Performance effects of remapping (every 40 time steps) (3-D DSMC)

(Time in secs)	Number of processors					Sequential Code
	8	16	32	64	128	
Static partition	1161.69	675.75	417.17	285.56	215.06	4857.69
RIB partition	850.75	462.15	278.23	209.75	267.24	
Chain partition	807.19	423.50	237.12	154.39	127.26	

4.3.1 Parallelization Approach

The key component of a typical DSMC computation is the *MOVE* phase which calculates new positions of molecules and moves them to appropriate cells. This calls for data exchange between processors every timestep. Also, the motion of molecules creates load imbalances which can be corrected by periodically remapping cells to processors.

As noted while introducing DSMC in Section 2.2, the order in which molecules are appended to their new cells during the MOVE phase does not matter. This allows use of lightweight schedules, which can be generated efficiently, and allow faster data migration. The lightweight schedules used by the data transportation primitive `scatter_append` perform much better than the `gather` and `scatter` primitives used with regular schedules.

Static partitioning of cells across processors does not work well for DSMC. As molecules move across cells, the computational load balance deteriorates over time. Performance can be substantially improved by periodically redistributing the cells with the help of parallel partitioners such as recursive coordinate bisection (RCB) [1] and recursive inertial bisection (RIB) [21]. While these partitioners are parallelized, they are still expensive and are affordable only when the load imbalance becomes too severe. We have also used a fast one dimensional partitioner, called the *chain partitioner* [20], which takes advantage of the highly directional nature of particle flow that is observed in most DSMC simulations. For instance, in the experiments reported here, more than 70 percent of the molecules were found moving along the positive x-axis. Partitioning cells along the direction of flow gives good load balance in such a case. For most DSMC simulations, experiments show that the chain partitioner reduced partitioning cost dramatically, while achieving nearly the same quality of load balance as RCB and RIB [19]. Information about the mapping of cells to processors was maintained in a replicated translation table.

4.3.2 Performance Results

Table 4 compares the execution time of the 2-dimensional DSMC code using lightweight schedules with the execution time obtained using regular communication schedules. The code used in these measurements utilized a static block-partitioning scheme to distribute cells across processors. This benchmark evenly distributed the workload over the entire domain - therefore load balance was not a major concern. The times shown in the table represent the time for executing 200 timesteps. These numbers demonstrate that the cost of generating and using lightweight schedules is much lower than that of regular schedules in DSMC.

We also show that periodic data remapping provides better performance than static partitioning for problems where the workload changes over time. Table 5 compares the performance of periodic domain partitioning methods with that of static partitioning (i.e. no redistribution) for a 3-dimensional DSMC code. The table compares execution time for 1000 time steps, with cells being remapped every 40 time steps based on the workload information collected for each cell. The results show that periodic remapping outperforms static partitioning significantly on a small number of processors. However, using a recursive inertial bisection (RIB) partitioner leads to performance degradation on a large number of processors. This performance degradation is a result of the large communication overhead incurred during partitioning, which increases with the number of processors used. At high levels of parallelism the cost of performing the partitioning dominates the gain in load balance. The chain partitioner, which has much smaller communication requirement than the recursive inertial bisection partitioner, did not show such slowdowns.

5 Compiling Adaptive Irregular Problems

There are a wide range of languages such as Vienna Fortran [6], pC++ [10], Fortran-D [9] and High Performance Fortran (HPF) [12], which provide a rich set of directives allowing users to specify desired data decompositions. With these decomposition directives, compilers can partition loop iterations and generate communication required to parallelize programs. We show how a similar framework can be used to parallelize irregular forall loops with Fortran D array extensions in adaptive irregular problems. Our work has been based on existing Fortran D language support; however the same methods could be extended for other languages.

5.1 Language Support for Data Distribution

On distributed memory machines, large data arrays need to be partitioned over the local memories of processors. Since most irregular problems do not run efficiently with standard BLOCK and CYCLIC data distributions [25],

```
S1   REAL*8 x(N),y(N)
S2   INTEGER map(N)
S3   DECOMPOSITION reg(N),irreg(N)
S4   DISTRIBUTE reg(block)
S5   ALIGN map with reg
S6   ... set map array using some partitioner
S7   DISTRIBUTE irreg(map)
S8   ALIGN x,y with irreg
```

Figure 6: Irregular distribution using Fortran D

```
L2:   do i = 1, n_step
L2:     forall i = 1, sizeof_indirection_arrays
S1        REDUCE(SUM, x(ia(i)), y(ib(i)))
        end forall
      end do
```

Figure 7: Reduction loop in Fortran D

researchers have developed a variety of heuristic methods to obtain irregular data distributions [25, 1]. The distributions produced by these methods typically result in a data structure called a *maparray* or *translation table*, that lists the processor assignment for each array element.

Fortran D provides an interface that allows users to define irregular distributions and align arrays with such distributions. This is done by declaring a *distribution* template that characterizes the size, dimension and way in which an array is to be partitioned between processors. A distribution is produced using two declarations, namely **DECOMPOSITION** and **DISTRIBUTE**. DECOMPOSITION binds a name to the dimensionality and size of a distributed array template. DISTRIBUTE is an executable statement and specifies how a template is to be mapped onto processors. Arrays are associated with distributions using the Fortran D statement **ALIGN**. Figure 6 illustrates the use of these statements. We have used this interface to make CHAOS partitioners visible to the user. A detailed description of this interface can be found in [22].

5.2 Computational Loop Structures

The implementation of the Forall construct in Fortran D follows copy-in-copy-out semantics – Forall constructs with general loop-carried dependencies are not defined. However, a limited class of loop-carried dependencies can be specified using the intrinsic **REDUCE** function, inside a Forall construct. Figure 7 shows how the reduction in Figure 1 would be written within this framework. In a loop which performs a reduction, the output dependencies between different iterations are ignored, thus enabling parallelization. Reduction inside a Forall construct is important for representing computations such as those found in sparse and unstructured problems.

5.2.1 Reduce Append

In highly adaptive codes, as described in Section 2, the data access patterns change frequently. Figure 2, the code fragment resembling DSMC's MOVE procedure is an example of such a code. Elements of the 2-D array **cells** are moved across rows based on the indirection array **ia**. When such a program is executed on distributed memory machines, array elements will be moved across processors, based on the distribution of rows of array **cells**.

```
 forall i = 1, num_cells
  forall j = 1, size(i)
   REDUCE(APPEND, new_cells(ia(i,j), :), cells(i,j))
  end forall
 end forall
```

Figure 8: Reduce-Append Loop in Fortran D

In DSMC, the computational results do not depend on the ordering of elements in each row of array **cells**. The computation only depends on the number of elements in each row, and the values of those elements. Therefore, the data movement operation can be considered equivalent to appending each element onto an unordered list. An operation which appends elements to unordered lists is associative and commutative. Therefore, the data movement in DSMC can be viewed as a **reduction** operation. Recognizing that a particular data movement is a reduction operation can lead to significant optimizations, since preprocessing is no longer needed to determine data placement order.

Generally, it is possible with existing compiler techniques to compile irregular loops where data access patterns are known only at runtime due to indirections [7, 22]. The compiler generates preprocessing code for such a loop that, at runtime, carries out the appropriate communication and places off-processor data in a pre-determined order. However, this technique does not detect reductions. In order to allow the compiler to detect reductions in data-movement, we propose a new operator, **APPEND**, for the Fortran D intrinsic function – **REDUCE**. Thus, a call to **REDUCE(APPEND,..)** directs the compiler to adopt the appropriately efficient data moves. For instance, while parallelizing the loop in Figure 2, a user with application-specific knowledge can recognize that the loop is a reduction and can convey this information to the compiler using the proposed intrinsic. Figure 8 shows how such an intrinsic would be used for the loop shown in Figure 2.

5.3 Compiler Implementation

This section presents an outline of the compiler transformations used to handle irregular templates that appear in CHARMM and DSMC. The runtime support has been incorporated in the Fortran 90D compiler that is being developed at Syracuse University [2]. The Fortran 90D compiler transformations generate translated codes which embed calls to CHAOS procedures. The performance of the compiler generated code is compared with that of the hand parallelized versions. All measurements were done on the Intel iPSC/860 machine.

5.3.1 CHARMM

The non-bonded force calculation loop is computationally intensive and it also adapts every few time steps. A simplified Fortran D version of the non-bonded force calculation loop is shown in Figure 9. The non-bonded list *jnb* is used to address the coordinate arrays (**x** and **y**) and the displacement arrays (**dx** and **dy**) of atoms. The size of the non-bonded list of atom i is $(inblo(i+1) - inblo(ia))$.

In Figure 9, data arrays are initially distributed using a BLOCK mapping. A maparray *map* is used to distribute data arrays irregularly. The values of *map* are set using a partitioner. The compiler embeds CHAOS remap procedures to redistribute data irregularly. The compiler trans-

```
C     Initially arrays are distributed in blocks
C$    DECOMPOSITION reg(num_atoms)
C$    ALIGN x, y, dx, dy WITH reg
      ...
S1    Get new distribution (map) from a partitioner
C$    DISTRIBUTE reg (map)
      ...
C     Calculate DX and DY
L1:   forall i = 1, natom
       forall j = inblo(i), inblo(i+1) - 1
        REDUCE(SUM, dx(jnb(j)), x(jnb(j)) - x(i))
        REDUCE(SUM, dy(jnb(j)), y(jnb(j)) - y(i))
        REDUCE(SUM, dx(i), x(i) - x(jnb(j)))
        REDUCE(SUM, dy(i), y(i) - y(jnb(j)))
       end forall
      end forall
```

Figure 9: CHARMM's non-bonded force calculation in Fortran D

forms the irregular loop L1 into an *inspector* and an *executor* by embedding appropriate CHAOS runtime procedures.

Carrying out preprocessing for irregular loops can be an expensive process. However, if data access patterns do not change, the results from preprocessing can be reused. Therefore, it is important that the compiler-generated code be able to detect when preprocessing can be reused. An implementation of reusing results of preprocessing in compiler-generated code is described in [22]. In this approach, the compiler-generated code maintains a record of when statements or array intrinsics in loops may have modified indirection arrays. Before executing an irregular loop, the *inspector* checks this record to see whether any indirection array used in the loop has been modified since the last time the inspector was invoked. If an indirection array is found to be modified, the inspector removes the current schedule, generates a new schedule and updates the loop bound information. Otherwise, the same schedule can be reused.

Table 6 shows the performance of the compiler-generated code. As a reference point, we have also given the performance of the corresponding code with manually embedded CHAOS primitives. The principal difference between the two versions is that the compiler-generated version must frequently check if the indirection arrays have been modified while the hand coded version uses knowledge of the code structure to eliminate these checks. We have invested substantial effort in hand optimizing the full application codes from which these kernels were extracted - therefore we believe that the hand parallelized kernels are reasonably optimized.

For these experiments we used a smaller version of the program with computational characteristics resembling the actual applications. Both the hand-coded and compiler-generated versions of the program ran the calculations of the case described in Section 2.1 (MbCO + 3830 water molecules) for 100 iterations. In order to simulate adaptivity of the non-bonded force calculation loop, data arrays were redistributed every 25 iterations by alternately using the RCB and RIB partitioners (i.e RCB was used before the 1st and 50th iteration, RIB was used in the 25th and 75 iteration). Thus, data arrays and iterations were redistributed four times during the execution. The timings in Table 6 include the costs of data partitioning, data and indirection array remapping, and preprocessing and execution.

```
C$    DECOMPOSITION celltemp(num_cells)
C$    DISTRIBUTE celltemp(BLOCK)
C$    ALIGN icell(*,:),vel(*,:) WITH celltemp
C$    ALIGN size(:),new_size(:) WITH celltemp

C     Reduce-append the particle information into
C     new cells according to icell array

L1:   forall j = 1, num_cells
      forall i=1, size(j)
       REDUCE(APPEND,vel(icell(j,i),:),vel(j,i))
      end forall
      end forall

C     Compute the number of particles in each cell
L2:   forall j = 1, num_cells
        new_size(j) = 0
      end forall
L3:   forall j = 1, num_cells
       forall i=1, size(j)
        new_size(icell(j,i))= new_size(icell(j,i))+1
       end forall
      end forall
```

Figure 10: DSMC particle movement code in Fortran D

5.3.2 DSMC

Recall that the key component of the DSMC computation is the *MOVE* procedure which computes new positions of particles and moves them to proper locations in global address space. Particles move from one cell to another when their spatial locations change, consequently data associated with the particles must be redistributed as well.

Figure 10 shows a simplified version of MOVE procedure of the 2-dimensional DSMC code in Fortran D. Cells are distributed across processors using a regular BLOCK distribution. An indirection array **icell(i,j)** is used to represent the new index of a cell to which particle j in cell i must be assigned. An array **size** identically aligned with the second dimension of **icell** stores the number of particles in each cell. Loop L1 redistributes velocity components **vel** associated with individual particles using the **reduce append** intrinsic introduced in Section 5.2. Loops L2 and L3 are responsible for recomputing the number of particles in each cell.

The compiler generates a sequence of calls to CHAOS data migration primitives, which carry out the data movement. Having recognized that the movement is a *reduction*, the compiler generates calls to the CHAOS primitives which construct and use lightweight schedules. As shown in Section 4.3, lightweight schedules yield much better performance than regular schedules for such movements. The loop bounds of loops L2 and L3 are determined by the compiler. The compiler parallelizes loop L3 (which involves indirection) by embedding appropriate CHAOS runtime procedures.

Performance results for both the compiler-generated and

Table 6: Performance of Hand-Coded and Compiler-Generated CHARMM Loop

(Time in sec.)	Processors	Partition	Remap	Inspector	Executor	Total
Hand Coded	32	3.2	8.2	2.8	84.6	98.8
	64	4.2	6.7	2.0	62.9	75.8
Compiler	32	3.3	8.7	3.1	85.0	100.1
	64	4.3	7.1	2.2	63.6	77.2

Table 7: Performance of compiler generated DSMC code

(Time in secs)	Compiler generated				Manually parallelized			
	Processors				Processors			
	4	8	16	32	4	8	16	32
Reduce append	2.75	1.89	1.79	2.39	1.83	1.41	1.49	2.05
Total time	15.47	8.99	6.71	5.30	8.51	4.90	4.05	3.75

the manually parallelized 2-dimensional DSMC code with 32x32 cells and 5K molecules are presented in Table 7. These performance numbers include computation of the velocity and position of each molecule which is changed in the molecule collision phase, and also includes *reduce-append* operations for molecule movement. The table presents the time for executing the DSMC loop 50 times on the Intel iPSC/860.

6 Related Work

Several researchers have developed programming environments that target particular classes of irregular or adaptive problems. Williams [26] describes a programming environment (DIME) for parallelizing calculations with unstructured triangular meshes. Kohn and Baden [14] have developed a programming environment that is targeted towards particle computations. This programming environment supports irregular block distributions and provides facilities for dynamic load balancing.

There are several compiler projects targeting distributed memory multiprocessors. These include the Fortran D compiler projects at Rice and Syracuse [9, 2] and the Vienna Fortran compiler project [6, 3] at the University of Vienna, among others. The Jade project at Stanford [16], the DINO project at Colorado [23], the CODE project at UT, Austin, and the Split-C project [15] at Berkeley provide parallel programming environments coupled with varying degrees of compiler support. Many compiler projects that have used runtime compilation techniques — the Fortran D project [11], the Kali project [13], Marina Chen's work at Yale [17], the PARTI project [18, 24], the PYRROS project at Rutgers [29], the Split-C project at Berkeley [15], the Jade project at Stanford [16] and the DINO project at Colorado [23].

7 Conclusions

The CHAOS procedures described in this paper can be viewed as forming part of a portable, compiler independent, runtime support library. The CHAOS runtime support library contains procedures that support

1. static and dynamic distributed array partitioning,
2. partitioning loop iterations and indirection arrays,
3. remapping of arrays from one distribution to another, and
4. index translation, buffer allocation and communication schedule generation.

In this paper, we described features of CHAOS that enable parallelization of a class of adaptive irregular problems. These features include efficient schedule generation and support for lightweight communication schedules. Two adaptive applications, CHARMM and DSMC, were successfully parallelized using CHAOS.

We have also discussed how irregular forall loops in such adaptive codes can be automatically parallelized by compilers. Computational templates extracted from a molecular dynamics code and a particle simulation code were parallelized using a prototype compiler implementation.

Acknowledgments

We are grateful to Richard Wilmoth for his helpful advice about the parallelization and the use of the production DSMC codes. We would also like to thank Bernard Brooks and Milan Hodoscek for teaching us about CHARMM; and Robert Martino and DCRT use of the NIH iPSC/860.

The authors would like to thank Geoffrey Fox, Alok Choudhary, Sanjay Ranka and Zeki Bozkus for many helpful suggestions and for allowing us access to the Syracuse Fortran 90D compiler. We would also like to thank Ken Kennedy, Reinhard von Hanxleden and Chuck Koelbel for many useful discussions.

We thank Donna Meisel for carefully proofreading the paper and Alan Sussman for providing many helpful comments.

References

[1] M.J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. on Computers*, C-36(5):570–580, May 1987.

[2] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, Sanjay Ranka, and Min-You Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. To appear in the Journal of Parallel and Distributed Computing, 1994.

[3] P. Brezany, M. Gerndt, V. Sipkova, and H.P. Zima. SUPERB support for irregular scientific computations. In *Proceedings of the Scalable High Performance Com-*

puting Conference (SHPCC-92), pages 314–321. IEEE Computer Society Press, April 1992.

[4] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4:187, 1983.

[5] B. R. Brooks and M. Hodoscek. Parallelization of charmm for mimd machines. *Chemical Design Automation News*, 7:16, 1992.

[6] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[7] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. Technical Report CS-TR-3163 and UMIACS-TR-93-109, University of Maryland, Department of Computer Science and UMIACS, October 1993. To appear in Journal of Parallel and Distributed Computing.

[8] J. Saltz et. al. A manual for the CHAOS runtime library. Technical report, University of Maryland, 1993.

[9] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Department of Computer Science Rice COMP TR90-141, Rice University, December 1990.

[10] Dennis Gannon, Shelby Yang, and Peter Beckman. *User Guide for a Portable Parallel C++ Programming System, pC++*. Department of Computer Science and CICA, Indiana University, January 1994.

[11] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In *Compilers and Runtime Software for Scalable Multiprocessors, J. Saltz and P. Mehrotra Editors*, Amsterdam, The Netherlands, To appear 1991. Elsevier.

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

[13] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 177–186. ACM, March 1990.

[14] S.R. Kohn and S.B. Baden. A robust parallel programming model for dynamic non-uniform scientific computations. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 509–517. May 1994.

[15] A. Krishnamurthy, D.E. Culler, A. Dusseau, S.C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proc. of Supercomputing '93*, pages 262–273. November 1993.

[16] Monica Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of block algorithms. In *Proc. of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 63–74. ACM Press, April 1991.

[17] L. C. Lu and M.C. Chen. Parallelizing loops with indirect array references or pointers. In *Proc. of the 4th Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[18] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley. Principles of runtime support for parallel processors. In *Proc. of the 1988 ACM International Conference on Supercomputing*, pages 140–152, July 1988.

[19] B. Moon and J. Saltz. Adaptive runtime support for direct simulation Monte Carlo methods on distributed memory architectures. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 176–183. IEEE Computer Society Press, May 1994.

[20] David M. Nicol and David R. O'Hallaron. Improved algorithms for mapping pipelined and parallel computations. *IEEE Transactions on Computers*, 40(3):295–306, March 1991.

[21] B. Nour-Omid, A. Raefsky, and G. Lyzenga. Solving finite element equations on concurrent computers. In *Proc. of Symposium on Parallel Computations and theis Impact on Mechanics*, Boston, December 1987.

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

[23] Matthew Rosing, Robert B. Schnabel, and Robert P. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, September 1991.

[24] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, 3(6):573–592, December 1991.

[25] H. Simon. Partitioning of unstructured mesh problems for parallel processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*, 1991.

[26] R. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency, Practice and Experience*, 3(5):457–482, February 1991.

[27] Richard G. Wilmoth. Direct simulation Monte Carlo analysis of rarefied flows on paralle processors. *AIAA Journal of Thermophysics and Heat Transfer*, 5(3):292–300, July-Sept. 1991.

[28] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 ICPP*, volume 2, pages 26–30, 1991.

[29] Tao Yang and Apostolos Gerasoulis. Pyrros: Static scheduling and code generation for message passing multiprocessors. In *Proc. of 6th ACM International conference on Supercomputing*, pages 428–437, Washington, D.C., July 1992.