# 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 Computer Science Dept.*
*University of Maryland*
*College Park, MD 20742*
*(dybbuk@cs.umd.edu)*

‡*Northeast Parallel Architectures Center*
*Syracuse University*
*Syracuse, NY 13244*
*(ravi@npac.syr.edu)*

**Abstract**

In adaptive irregular problems the data arrays are accessed via indirection arrays, and data access patterns change during computation. Implementing such problems on distributed memory machines requires support for dynamic data partitioning, efficient preprocessing and fast data migration. This research presents efficient runtime primitives for such problems. This new set of primitives is part of the CHAOS library. It subsumes the previous PARTI library which targeted only static irregular problems. To demonstrate the efficacy of the runtime support, two real adaptive irregular applications have been parallelized using CHAOS primitives: a molecular dynamics code (CHARMM) and a particle-in-cell code (DSMC). The paper also proposes extensions to Fortran D which can allow compilers to generate more efficient code for adaptive problems. These language extensions have been implemented in the Syracuse Fortran 90D/HPF prototype compiler. The performance of the compiler parallelized codes is compared with the hand parallelized versions.

*This work was sponsored in part by ARPA (NAG-1-1485), NSF (ASC 9213821), ONR (SC292-1-22913) and EPRI (RP3103-6).

***CS-TR-3266 and UMIACS-TR-94-55. Also appears in Supercomputing'94.***
***Available at URL ftp://hpsl.cs.umd.edu/pub/papers/sc94-tr.ps.Z.***

# 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 patterns of a loop-nest are 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. Preprocessing methods are being developed for a variety of unstructured problems including explicit multi-grid unstructured computational fluid dynamic solvers [18, 11], molecular dynamics codes (CHARMM, AMBER, GROMOS, etc.) [5], diagonal or polynomial preconditioned iterative linear solvers [26], and particle-in-cell (PIC) codes [3]. These problems share the characteristics of (1) arrays accessed through one or more levels of indirection, and (2) formulation of the problem as a sequence of loop nests each of which prove to be parallelizable.

Figure 1 illustrates a typical irregular loop. The data access pattern is determined by arrays, **ia** and **ib**, which are known only at runtime. These arrays are called *indirection arrays.* Once the data access pattern is known, preprocessing makes it possible to partition *data arrays* (i.e. arrays **x**, **y**) and to partition loop iterations (i.e. indirection arrays **ia**, **ib**) over processors. The goal of such partitioning is to balance the computational load and to reduce the net communication volume. Once data and work have been partitioned between processors, prior knowledge of loop data access patterns (values of **ia**, **ib**) makes it possible to predict which data elements need to be communicated between processors. This communication pattern remains unchanged as long as the data access patterns do not change. The ability to predict communication requirements allows various communication optimizations. For instance, communication volume can be reduced by pre-fetching a single copy of each off-processor datum, even if it is referenced several times. The number of messages can also be reduced by pre-fetching large quantities of off-processor data in a single message. These optimizations are called *software caching* and *communication vectorization* respectively.

Such optimizations have been successfully used to parallelize *static* irregular problems [8], in which array access patterns do not change during computation. For the static irregular problems considered in Das et al. [8], 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, resulting in complex preprocessing requirements.

Consider, for instance, adaptive fluid dynamics and molecular dynamics codes. In these applications, interactions between entities (mesh points, molecules etc.) are specified by indirection arrays. If the interaction pattern is modified, then the data access pattern changes. Consequently, it may

```
        real      x(max_nodes), y(max_nodes)     ! data arrays
        integer ia(max_edges), ib(max_edges)     ! indirection arrays

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

Figure 1: An Example with an Irregular Loop

become necessary to fetch off-processor data elements which were not needed before. Thus preprocessing needed for software caching and communication vectorization optimizations will have to be repeated. In other applications, such as the Direct Simulation Monte Carlo (DSMC) code and other PIC codes, data access patterns and computational load change frequently. Thus, data arrays may need to be redistributed frequently to relocate moving particles and to maintain load balance. For such applications, efficient runtime support to perform particle migration and data redistribution is necessary.

This paper presents a new set of runtime procedures designed to efficiently implement adaptive programs on distributed memory machines. This runtime library is called CHAOS; it subsumes PARTI, a library aimed at static irregular problems [24]. CHAOS introduces two new features — *light-weight schedules* and *efficient schedule generation* , which are useful in certain types of adaptive problems. We describe these features in Section 3. CHAOS has been used to parallelize two challenging real-life adaptive applications — CHARMM, a molecular dynamics code and DSMC, a particle-in-cell code. 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.

This paper is structured as follows. In Section 2, we introduce two adaptive applications, CHARMM and DSMC Section 3 describes our runtime support. Section 4 demonstrates the performance of the runtime support library when used on the targeted applications. Section 5 gives an overview of existing language support for irregular data decomposition; it describes the language directives that allow users to remap data and introduces a list-append intrinsic function. The performance of the compiler-generated codes is also compared to that of the hand-written codes. Section 6 discusses related work. Finally, conclusions are presented in Section 7.

```
L1:  do n = 1, number_of_time_steps                            ! outer loop

S:     if (required) then                                      ! under certain criteria
          regenerate jnb(:)                                    ! non-bond list changes

L2:    do i = 1, number_of_bonds
          BF(ib(i)) = BF(ib(i)) + f (Atom(ib(i)), Atom(jb(i))  ! bonded forces
          BF(jb(i)) = BF(jb(i)) + g (Atom(ib(i)), Atom(jb(i))
       end do

L3:    do i = 1, number_of_atoms                               ! non-bondeded forces
         do j = inblo(i), inblo(i+1)-1                         ! partners of atom i
            NBF(i) = NBF(i) + h (Atom(i),Atom(jnb(j)))
         end do
       end do

        ......Calculate new positions based on BF and NBF
     end do
```

Figure 2: A code fragment resembling CHARMM

# 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 particle-in-cell code for simulating motion of gas particles. These are real-life applications; each consists of thousands of lines of code.

## 2.1 CHARMM

CHARMM is a program which models macromolecular systems in order to derive their structural and dynamic properties. The computationally intensive part of CHARMM is the molecular dynamics simulation section. This calculation simulates dynamic interactions among all atoms in the system for a period of time. For each time step, the simulation calculates the forces between atoms, the energy of the whole structure, and the movements of atoms by integrating Newton's equations of motion. It then updates the spatial positions of all atoms based on these calculations.

The energy calculations in the simulation comprise of two types of interactions – bonded and non-bonded. Bonded forces are short-range forces that exist between atoms connected by chemical bonds. They remain unchanged during the entire simulation. Non-bonded forces, due to Van der Waals interactions and electrostatic potential, exist between all pairs of atoms. The time complexity of calculating non-bonded forces is $O(N^2)$ because each atom interacts with all other atoms in the system. CHARMM approximates this calculation by ignoring all interactions beyond a certain cutoff distance from each atom. This approximation is achieved by maintaining a *non-bonded list* of interacting partners for each atom. Since spatial positions of atoms may change after each time step, the non-bonded list must be periodically updated. In CHARMM, users have control over the frequency at which the non-bonded list is regenerated. Typically it is regenerated after every 10 to 100 time-steps.

The code fragment in Figure 2 resembles the computational structure of CHARMM. Here, multiple loops access the same data arrays but with different access patterns. In loop L2, the data array **Atom** is indirectly accessed using arrays **ib** and **jb**. In loop L3, the same data arrays are indirectly accessed using array **jnb**. The data access pattern in loop L3 changes whenever the indirection array **jnb** is modified by the conditional statement S. Whenever indirection array **jnb** changes, pre-processing for loop L3 must be repeated so that communication optimizations such as software caching and communication vectorization can continue.

## 2.2 Direct Simulation Monte Carlo

The DSMC code is used to study the behavior of a gas by simulating the motion of a large number of molecules. The simulation tracks the motion, collisions and boundary interactions of molecules for a specified period of time. The DSMC simulation method involves laying out a cartesian grid over the domain, which may be either 2-dimensional or 3-dimensional, and associating each molecule with its cartesian cell. Molecules are assumed to interact only with other molecules in the same cell at each time step. By distributing cartesian cells and their constituent molecules across processors, substantial parallelism can be extracted.

However, there are three significant impediments to parallelization of DSMC. First, since molecules are in continuous motion, many molecules change their cells every time-step. The cost of transmitting molecules between cells every time step can be substantial on distributed memory computers. Second, indirection arrays used to access molecules within each cell must be regenerated every time step. Third, as molecules move between cells, the workload in each cell keeps changing, affecting the net computational load balance. Periodically, cells must be remapped to processors in order to maintain load balance. These characteristics of DSMC are found in many PIC applications.

Figure 3 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

```
do i = 1, num_cells
  do j = 1, size(i)                    ! size(i) is the number of elements in the ith cell
     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
```

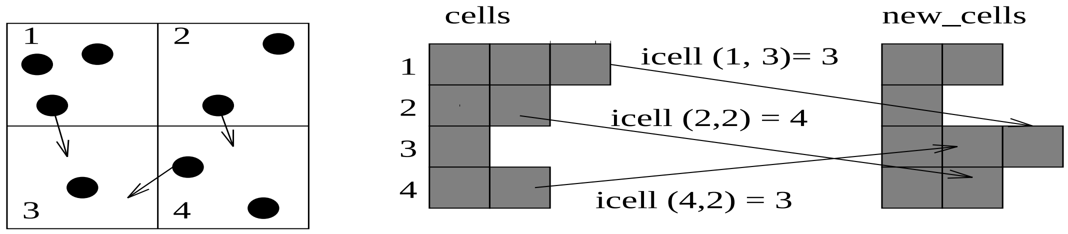


Figure 3: Data movement in DSMC

the shuffle, the size of individual rows may not. Usually, the order in which elements are appended to new rows is not relevant, since the order of computation over these elements does not matter. The runtime support takes advantage of such application-specific information. 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 3. Our runtime support has been used to parallelize the real code; however, for testing our compiler implementation, we have used a computational template similar to the one shown here.

# 3 Runtime Support

This section describes the principles and functionality of the CHAOS runtime support library, a superset of the PARTI library [19, 28, 24]. First, a brief overview of the runtime support is presented; the framework is same as that of PARTI, and has been described in earlier papers [8, 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 for supporting fast data migration and describe how we have optimized the inspector to generate schedules efficiently.

## 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. Solving such irregular problems on distributed memory machines using the CHAOS runtime support involves six major phases (Figure 4). The first four phases concern mapping data and computations onto processors. The next

| | | |
|---|---|---|
| Phase A : | Data Partitioning | Assign elements of data arrays to processors |
| Phase B : | Data Remapping | Redistribute data array elements |
| Phase C : | Iteration Partitioning | Allocate iterations to processors |
| Phase D : | Iteration Remapping | Redistribute indirection array elements |
| Phase E : | Inspector | Translate indices; Generate schedules |
| Phase F : | Executor | Use Schedules for Data Transportation; Perform computation |

Figure 4: Solving Irregular Problems

two steps concern analyzing data access patterns in a loop and generating optimized communication calls. A brief description of these phases follows.

1. **Data Distribution :** Phase A calculates how data arrays are to be partitioned by making use of partitioners provided by CHAOS or by the user. CHAOS supports a number of parallel partitioners that partition data arrays using heuristics based on spatial positions, computational load, connectivity, etc. The 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.)

2. **Data Remapping :** Phase B remaps data arrays from the current distribution to the newly calculated irregular distribution. A CHAOS procedure `remap` is used to generate an optimized *communication schedule* for moving data array elements from their original distribution to the new distribution. Other CHAOS procedures, `gather`, `scatter`, and `scatter_append`, use the communication schedule to perform data movement.

3. **Loop Iteration Partitioning :** Phase C determines how loop iterations should be partitioned across processors. There are a large number of possible schemes for assigning loop iterations to processors based on optimizing load balance and communication volume. 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. CHAOS also allows the owner-computes rule.

4. **Remapping Loop Iterations :** Phase D is similar to phase B. Indirection array elements are remapped to conform with the loop iteration partitioning. For example, in Figure 1, once loop L2 is partitioned, indirection array elements **ia(i)** and **ib(i)** used in iteration $i$ are moved to the processor which executes that iteration.

5. **Inspector :** Phase E carries out the preprocessing needed for communication optimizations and *index translation*. This phase is described in Section 3.2.

6. **Executor :** Phase F uses information from the earlier phases to carry out the computation and communication. Communication is carried out by CHAOS data transportation primitives which use communication schedules constructed in Phase E.

In static irregular problems, Phase F is executed many times, while phases A through E are executed only once. In some adaptive problems data access patterns change periodically but reasonable load balance is 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 are 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 [8]. 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. Special attention has been devoted towards optimizing the inspector for adaptive applications.

### 3.2.1 Communication Schedules

After data and work have been partitioned across processors, the communication requirements of each computational phase 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 transportation primitives

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

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

Figure 5: A code with two computational phases

`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 out-going messages from processor $p$ to other processors.

4. fetch size – an array that specifies the sizes of in-coming messages to processor $p$ from other processors.

While a communication schedule can be generated for the data access pattern of each irregular computational phase, there are significant advantages in considering many such phases simultaneously. For instance, consider the sample code in Figure 5. There are two computational phases L2 and L3. 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. 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 is used to build much cheaper *light-weight* 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, light-weight schedules also speed up data movement by eliminating the need of rearranging the order of incoming off-processor elements. Light-weight schedules have been used in the parallelization of DSMC. In section 5.2.1 we show how light-weight 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 from these. 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 indexes are translated to local indexes. 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 in, the hash-table stores the following information :

1. global index — the global index hashed in.

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 hold 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. The costs of dynamically allocating memory to store new elements in the hash-table are significant, even though CHAOS uses customized memory allocators. However,

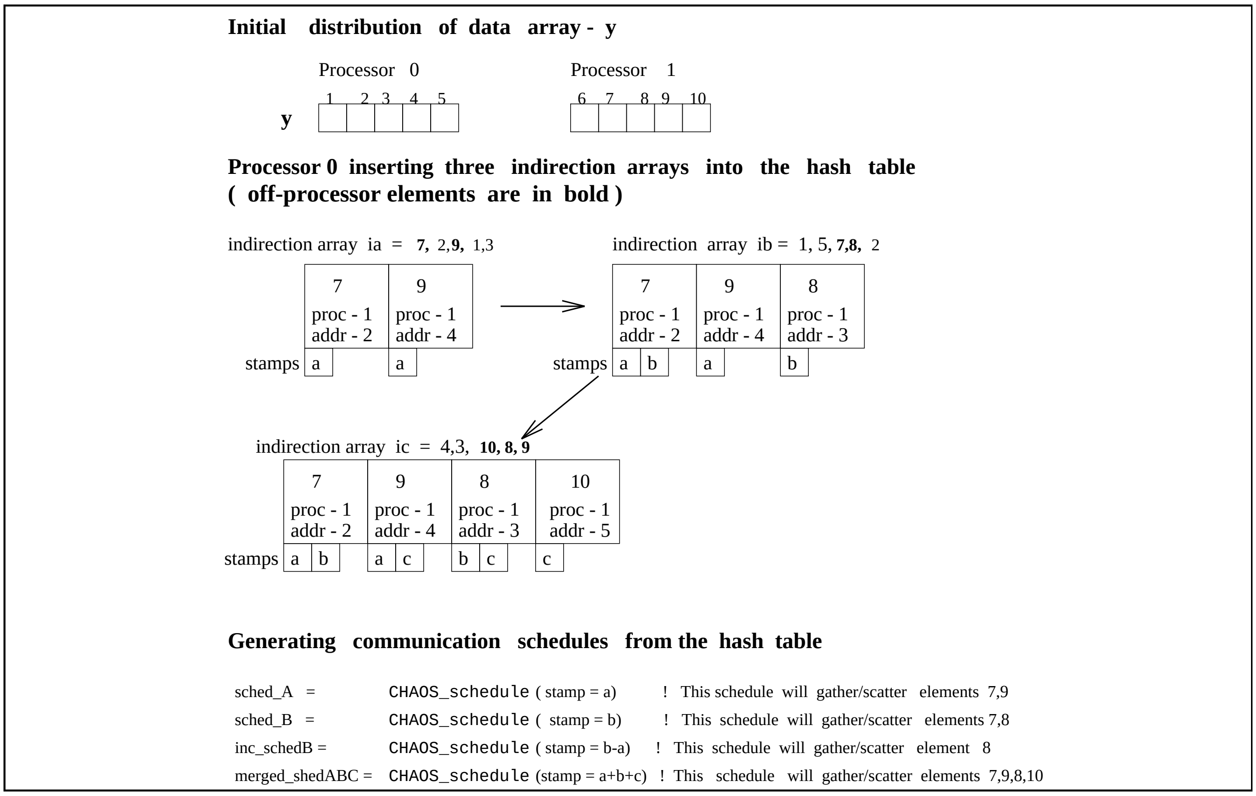


Figure 6: Schedule generation with hash table

building a hash-table for duplicate removal is 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. However, much of the costs of index analysis can be amortized by retaining the hash-table. In adaptive applications, indirection arrays keep changing; however, most of the indirection array elements remain unchanged. Index analysis for these unchanged indexes involves only a lookup in the hash table.

Indirection arrays are hashed in by the CHAOS procedure `CHAOS_hash`. This primitive enters all the indexes 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 indexes changed to local indexes in the course of hashing.

A procedure called `CHAOS_schedule` is then used to construct communication schedules from the entries in a 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. Figure 6 demonstrates (in pseudo-code) how this is done.

In CHARMM, the indirection array (**jnb**) 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 spent in non-bonded force calculation. The non-bonded interaction list for each atom is modified periodically, as atoms change their spatial positions.

**Data Partitioning**

Bonded interactions occur between atoms in close proximity to each other, while non-bonded interactions are excluded beyond a certain cutoff range. Additionally, the amount of computation associated with an atom depends on the number of atoms with which it interacts – the number of non-bonded list entries for that atom. This implies that data partitioners which use spatial information as well as computational load will perform significantly better than naive BLOCK or CYCLIC distributions. We have tried a recursive coordinate bisection (RCB) partitioner as well as a recursive inertial bisection (RIB) partitioner on CHARMM with almost identical results. Both the RCB and RIB partitioners use spatial positions to guide partitioning, and also consider computational weights while allocating partitions to processors. All data arrays that are associated with atoms are distributed in an identical fashion, so we do not have to repeat partitioning for each array separately. Information about the distribution is stored in a replicated translation table.

**Iteration Partitioning**

Once atoms are partitioned, the data distribution is used to decide how loop iterations are partitioned among processors. Since the non-bonded loop consumes 90% of the execution time, balancing the computational load due to these calculations is of primary concern. The non-bonded force calculation loop nest iterates over each atom's non-bonded list. Currently, each iteration of the outer loop is assigned to the processor that owns the atom being iterated over. The load balance achieved by the owner-computes rule depends on 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. The non-bonded force calculation loop nest iterates over each atom's non-bonded list. Each iteration of the outer loop is assigned to the processor that owns the atom being iterated over.

**Remapping and Loop Pre-processing**

Once new distributions of data and loop iterations are known, CHAOS primitives can be used to remap the data and indirection arrays from the current distributions to new distributions. After remapping, loop preprocessing is carried out for the bonded and non-bonded force calculation loops.

Indirection arrays used in bonded force calculation loops remain unchanged while the non-bonded list adapts during computation. Hence, pre-processing for bonded force calculation loops need not be repeated, whereas it must be repeated for non-bonded force calculation loops whenever the non-bonded list changes. In this case, the hash table and stamps (see Section 3.2.2) are very useful for loop pre-processing. While building schedules, indirection arrays are hashed with unique time stamps. The hash table is used to remove any duplicate off-processor references. When the non-bonded list is regenerated, non-bonded list entries in the hash table are cleared with the corresponding stamp. Then the same stemp can be reused and the new non-bonded list entries are hashed with the reused stamp.

### 4.1.1 Performance

The performance of CHARMM, parallelized using CHAOS, was studied with a benchmark case (MbCO + 3830 water molecules) on the Intel iPSC/860. The program was run for 1000 time-steps 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

$$LB = \frac{(\max_{i=1}^{n} computation\ time\ of\ processor\ i) \times (number\ of\ processors\ n)}{\sum_{i=1}^{n} computation\ time\ of\ processor\ i}$$

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

**Overheads of Preprocessing**

Data and iteration partitioning, remapping, and loop preprocessing must be done at runtime. Preprocessing overheads of the simulation are shown in Table 2. The data partition time is the

[1]Estimation done by Brooks and Hodoscek[6]

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

| Number of Processors | 1 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|
| Execution Time | 74595.5[1] | 4356.0 | 2293.8 | 1261.4 | 781.8 |
| Computation Time | 74595.5 | 4099.4 | 2026.8 | 1011.2 | 507.6 |
| Communication Time | 0.0 | 147.1 | 159.8 | 181.1 | 219.2 |
| Load Balance Index | 1.00 | 1.03 | 1.05 | 1.06 | 1.08 |

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

| Number of Processors | 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Data Partition | 0.27 | 0.47 | 0.83 | 1.63 |
| Non-bonded List Update | 7.18 | 3.85 | 2.16 | 1.22 |
| Remapping and Preprocessing | 0.03 | 0.03 | 0.02 | 0.02 |
| Schedule Generation | 1.31 | 0.80 | 0.64 | 0.42 |
| Schedule Regeneration ($\times 40$) | 43.51 | 23.36 | 13.18 | 8.92 |

execution time of the RCB partitioner. After partitioning atoms, the non-bonded list is regenerated. This non-bonded list regeneration was performed because atoms were redistributed over processors and it was done before simulation occurred. In Table 2, this regeneration time is denoted as non-bonded list generation time. During simulation, non-bonded list was regenerate periodically. When the non-bonded list was updated, the schedule must be regenerated. The schedule regeneration time in Table 2 gives the total schedule regeneration time spent 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.

**Schedule Merging vs. Multiple Schedules**

There are several indirection arrays used in bonded and non-bonded force calculations to reference data arrays that are distributed in identical fashion. One possible approach is to compute separate schedules to gather and scatter off-processor data for each irregular loop. A second approach is to compute a single schedule using the schedule merging technique, discussed in Section 3.2.1. Table 3 compares the performance of these techniques and demonstrates the usefulness of schedule merging.

## 4.2 DSMC

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

Table 3: Communication Time (in sec.)

| Number of Processors | Schedule Merging | | Multiple Schedules | |
|---|---|---|---|---|
| | Comm. Time | Exec. Time | Comm. Time | Exec. Time |
| 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 |

grid every time-step.

### 4.2.1 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 time-step. Also, the motion of molecules creates load imbalances which must be periodically corrected by remapping cells to processors.

**Efficient Data Migration**

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 light-weight schedules, described in Section 3.2.1, which can be generated efficiently, and allow faster data migration. Light-weight schedules are used by the data transportation primitive `scatter_append` which performs much better than the `gather` and `scatter` primitives used with regular schedules.

**Remapping for Load Balancing**

Static partitioning of cells across partitioners 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 CHAOS's parallel partitioners such as recursive coordinate bisection (RCB) [2] 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. CHAOS also provides a fast one dimensional partitioner, called the *chain partitioner* [20], which takes advantage of the highly directional nature of particle flow that characterizes many DSMC communication patterns. For instance, in the experiments reported here, more than 70 percent of the molecules were found moving along the positive $x$-axis. Partioning along the direction of flow gives good load balance in such a case. Experiments show that the chain partitioner reduces partitioning cost dramatically to a scale conformable to adaptive data migration primitives. It also achieves nearly the same quality of load balance as RCB and RIB. Information about the distribution of cells to processors was maintained in a replicated translation

Table 4: Regular Schedules vs. Light-weight Schedules

| (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 (remapped every 40 time steps)

| (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 |
| Recursive bisection | 850.75 | 462.15 | 278.23 | 209.75 | 267.24 | |
| Chain partition | 807.19 | 423.50 | 237.12 | 154.39 | 127.26 | |

table.

### 4.2.2 Performance Results

Table 4 compares the execution time of 2-dimensional DSMC code using the light-weight schedules with the time obtained using regular communication schedules (Section 3.2.1). The computational load was deliberately evenly distributed over the whole domain, so load balance is not an issue. The times shown in the table represent the time for the entire execution. These numbers demonstrate that the cost of generating and using light-weight schedules is much lower than that of regular schedules in DSMC.

We also show that periodic data remapping provides better performance than static partitioning. Table 5 compares the performances of periodic domain partitioning methods with that of static partitioning (i.e. no remapping) for 3-dimensional DSMC codes. The table presents execution time for 1000 time steps. Cells were remapped every 40 time steps based on the workload information collected for each cartesian mesh cell. The results show that periodic remapping outperformed static partitioning significantly on a small number of processors. However, using a recursive bisection 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 as the number of processors increases. At high levels of parallelism the costs of performing the partitioning dominate over the gains in load balance. The chain partitioner, however, provided the better results for this problem.

# 5 Compiling Adaptive Irregular Problems

There are a wide range of languages such as Vienna Fortran [7], pC++ [10], Fortran-D [9] and High Performance Fortran (HPF) [13], 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. This paper presents language features required to support adaptive problems within the Fortran D framework. However, the same could be extended for other languages. In the following sections, the existing Fortran D language support and the proposed language extensions for adaptive problems are discussed.

## 5.1 Language Support

On distributed memory machines, large data arrays need to be partitioned over the local memory of processors. These partitioned data arrays are called *distributed arrays.* Long term storage of distributed array data is assigned to specific processor and memory locations in the machine. Many applications can be efficiently implemented by using simple schemes for mapping distributed arrays. One example of such a scheme would be the division of an array into equal sized contiguous subarrays and assignment of each subarray to a different processor. Another example would be to assign consecutively indexed array elements to processors in a round-robin fashion. These two data distribution schemes are often called BLOCK and CYCLIC data distributions [13], respectively.

### 5.1.1 Irregular Distribution

On distributed memory machines, irregular concurrent problems may not run efficiently with standard data distributions such as BLOCK and CYCLIC [25]. Researchers have developed a variety of heuristic methods to obtain data mappings that are designed to optimize irregular problem communication requirements [25, 27, 2]. The distribution produced by these methods typically results in a table that lists a processor assignment for each array element. This kind of distribution is often called an *irregular distribution.*

Fortran D provides the user with a choice of several standard distributions. In addition, a user can define non-standard distributions, or irregular distribution as well. Figure 7 presents an example of such a Fortran D declaration. In Fortran D, one declares a template called a *distribution* that is used to characterize the significant attributes of a distributed array. The distribution fixes the size, dimension and way in which the array is to be partitioned between processors. A distribution is produced using two declarations. The first declaration is **DECOMPOSITION**. Decomposition binds a name to the dimensionality and size of a distributed array template. The second declaration is **DISTRIBUTE**. Distribute is an executable statement and specifies how a template is to be

```
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 values of map array using some mapping method ..
S7   DISTRIBUTE irreg(map)
S8   ALIGN x,y with irreg
```

Figure 7: Fortran D Irregular Distribution

```
L2:  DO i = 1, n_step                            ! outer loop
L2:    FORALL i = 1, sizeof_indirection_arrays   ! inner loop
S1       REDUCE(SUM, x(ia(i)), y(ib(i)))
       END DO
     END DO
```

Figure 8: Example Reduction Loop in Fortran D

mapped onto the processors.

A specific array is associated with a distribution using the Fortran D statement **ALIGN**. In statement S3 of Figure 7, two 1-D decompositions, each of size `N`, are defined. In statement S4, decomposition `reg` is partitioned into equal sized blocks, with one block assigned to each processor. In statement S5, array `map` is aligned with distribution `reg`. Array `map` is used to specify (in statement S7) how distribution `irreg` is to be partitioned between processors. An irregular distribution is specified using an integer permutation array *map*; when $map(i)$ is set equal to $p$, element $i$ of the distribution `irreg` is assigned to processor $p$. A data partitioner can be invoked to set the values of the permutation array. The partitioner may not always be available in Fortran D. In such cases, it can be called as an extrinsic procedure.

### 5.2 Computational Loop Structures

The implementation of the Forall construct in Fortran D follows copy-in-copy-out semantics – 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 8 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 can be ignored, thus enabling parallelization. Reduction inside a Forall construct is important for representing computations such as those found in sparse and unstructured problems.

```
FORALL i = 1, num_cells
  FORALL j = 1, size(i)          ! size(i) is the number of elements in the ith cell
    REDUCE(APPEND, new_cells(ia(i,j), :), cells(i,j))
  END DO
END DO
```

Figure 9: Example Reduce Append Loop in Fortran D

### 5.2.1 Reduce Append

In highly adaptive codes, as described in Section 2, the data access patterns change frequently. Figure 3 shows an example of such codes. 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**.

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 into an unordered list. An operation which adds elements to unordered lists is associative and commutative, therefore, the data movement 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. Data movements occur frequently in adaptive problems, hence it is important to optimize them.

Generally, it is possible with existing compiler techniques to compile irregular loops where data access patterns are known only at runtime due to indirections [8, 22]. The compiler generates a pre-processing code for such a loop that, at runtime, generates appropriate communication calls 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 an intrinsic function called **reduce**(**append**, ..) This intrinsic function will direct the compiler to adopt the appropriately efficient data moves. Thus, while parallelizing the loop in Figure 3, 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 9 shows how such an intrinsic would be used for the loop shown in Figure 3.

```
C     Initially arrays are distributed in blocks
C$    DECOMPOSITION reg(14026)
C$    DISTRIBUTE reg(BLOCK)
C$    ALIGN x, y, dx, dy WITH reg
      ...
S1    Obtain new distribution format (map) from the extrinsic 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 DO
      END DO
```

Figure 10: Non-bonded Force Calculation Loop of CHARMM in Fortran D

## 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 [4]. 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 10. 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 $inblo(i+1) - inblo(i)$.

In Figure 10, data arrays are initially distributed by BLOCK. 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 transforms the irregular loop L1 into an *inspector* and an *executor* by embedding appropriate CHAOS runtime procedures.

Carrying out pre-processing for irregular loops can be an expensive process. However, if data

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

| (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 |

access patterns do not change, the results from pre-processing 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 pre-processing in compiler-generated code is described in Ponnusamy et al. [22]. In this approach, the compiler-generated code maintains a record of when statements or array intrinsics of 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 presents experimental results that compare the performance of compiler-generated code with that of hand-coded version. For these experiments we used a smaller version of the program with computational characteristics resembling the real-life 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 applying RCB and RIB alternately. Thus, data arrays and iterations were redistributed four times during the execution. Table 6 lists the cost of data partitioning, data and indirection arrays remapping, and pre-processing and execution. The performance of the compiler-generated code is almost matches that of the hand parallelized code.

### 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 11 shows a simplified version of MOVE procedure of 2-dimensional DSMC code in Fortran D. Cells are distributed across processors using a regular BLOCK distribution. An indirection array

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

C     Reduce-append the particle information into new cells according to icell array
L1:   FORALL j = 1, num_cells
       FORALL i=1, size(j)
        REDUCE(APPEND, vel(i,icell(i,j)), vel(i,j))
       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(i,j)) = new_size(icell(i,j))+1
       END FORALL
      END FORALL
```

Figure 11: DSMC particle movement code in Fortran D

**icell(i,j)** is used to represent a new index of a cell in 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 which was proposed in Section 5.2. Loops L2 and L3 are responsible for recomputing the number of particles in each cell.

When a **reduce-append** statement is encountered, 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 those CHAOS primitives which construct and use light-weight schedules. As shown in Section 4.2, light-weight 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 the manually parallelized 2-dimensional DSMC code with 32x32 cells and 5K molecules are presented in Table 7. These performance numbers include computation of velocity and position of each molecule which are changed by the molecule collision phase, and also include *reduce-append* operations for molecule movement. The table presents the time for executing the DSMC loop 50 times on the Intel iPSC/860. While the manually parallelized version utilizes the functionality of CHAOS data migration primitives

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 |

which return the new number of particles in each cell, the compiler generated code needs to carry out an additional computation to obtain this. Hence, the compiler-generated code performs extra communication. (This communication is done by invoking CHAOS procedures.)

## 6 Related Work

Several researchers have developed programming environments that target particular classes of irregular or adaptive problems. Williams [27] describes a programming environment (DIME) for calculations with unstructured triangular meshes using distributed memory machines. Baden [1] has developed a programming environment targeting particle computations. This programming environment provides facilities that support dynamic load balancing.

There are a variety of compiler projects targeting at distributed memory multiprocessors: the Fortran D compiler projects at Rice and Syracuse [9, 4] and the Vienna Fortran compiler project [7] at the University of Vienna, among others. The Jade project at Stanford [16], the DINO project at Colorado [23], and the CODE project at UT, Austin, provide parallel programming environments. The Split-C project [15] at Berkeley is targeted towards providing a parallel programming environment on distributed memory machines. Runtime compilation methods have been employed in four compiler projects: the Fortran D project [12], the Kali project [14], Marina Chen's work at Yale [17] and the PARTI project [19, 24]. The Kali compiler was the first compiler to implement inspector/executor type runtime preprocessing [14] and the ARF compiler was the first compiler to support irregularly distributed arrays [28].

## 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. partitions loop iterations and indirection arrays,
3. remap arrays from one distribution to another, and
4. carry out index translation, buffer allocation and communication schedule generation.

In this paper, we introduced new features of CHAOS that enable parallelization of certain types of adaptive irregular problems. These include light-weight communication schedules and efficient schedule generation. We have described how two real-life adaptive applications, CHARMM and DSMC, were parallelized using the runtime support.

We have also discussed how such adaptive codes can be automatically parallelized by compilers. Computational templates extracted from a molecular dynamics code and a PIC code were tested using a prototype compiler implementation. The performance of the compiler-generated codes was compared to that of the hand-parallelized codes.

## Acknowledgments

The authors thank Richard Wilmoth at NASA Langley for his helpful advice about the parallelization and the use of the production DSMC codes. The authors also thank Bernard Brooks and Milan Hodoscek for teaching us about CHARMM; and Robert Martino and DCRT for the general support and the use of NIH iPSC/860.

The authors thank Geoffrey Fox, Alok Choudhary, and Sanjay Ranka for many enlightening discussions about universally applicable partitioners and how to embed such partitioners in compilers; also Chuck Koelbel, Ken Kennedy and Seema Hiranandani for many useful discussions about integrating Fortran-D runtime support for irregular problems.

The authors gratefully acknowledge Zeki Bozkus and Tom Haupt for the time they spent explaining the internals of the Fortran 90D compiler.

The authors would also like to thank Donna Meisel for carefully proof-reading this paper.

## References

[1] S. Baden. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. *SIAM J. Sci. and Stat. Computation.*, 12(1), January 1991.

[2] 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.

[3] Graeme A. Bird. *Molecular Gas Dynamics and the Direct Simulation of Gas Flows.* Clarendon Press, Oxford, 1994.

[4] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, April 1994.

[5] 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.

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

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

[8] 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. Submitted to Journal of Parallel and Distributed Computing.

[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. Hammond and T. Barth. An optimal massively parallel Euler solver for unstructured grids. *AIAA Journal, AIAA Paper 91-0441*, January 1991.

[12] 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.

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

[14] 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.

[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 *Proceedings Supercomputing '93*, pages 262–273. IEEE Computer Society Press, November 1993.

[16] Monica Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of block algorithms. In *Proceedings 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 *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[18] D. J. Mavriplis. Three dimensional unstructured multigrid for the Euler equations, paper 91-1549cp. In *AIAA 10th Computational Fluid Dynamics Conference*, June 1991.

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

[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, Alok Choudhary, Yuan-Shin Hwang, and Geoffrey Fox. Runtime support and compilation methods for user-specified irregular data distributions. Technical Report CS-TR-3194 and UMIACS-TR-93-135, University of Maryland, Department of Computer Science and UMIACS, November 1993. Appears in IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 8.

[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*. Pergamon Press, 1991.

[26] P. Venkatkrishnan, J. Saltz, and D. Mavriplis. Parallel preconditioned iterative methods for the compressible navier stokes equations. In *12th International Conference on Numerical Methods in Fluid Dynamics, Oxford, England*, July 1990.

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

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