SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(6), 597–621(JUNE 1995)

Runtime and Language Support for Compiling Adaptive Irregular Programs on Distributed Memory Machines *

YUAN-SHIN HWANG AND BONGKI MOON AND SHAMIK D. SHARMA
Department of Computer Science, University of Maryland, College Park, MD 20742

AND

RAVI PONNUSAMY
Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY 13244

AND

RAJA DAS AND JOEL H. SALTZ
UMIACS and Department of Computer Science, University of Maryland, College Park, MD 20742

SUMMARY

In many scientific applications, arrays containing data are indirectly indexed through indirection arrays. Such scientific applications are called irregular programs and are a distinct class of applications that require special techniques for parallelization.

This paper presents a library called CHAOS, which helps users implement irregular programs on distributed memory message-passing machines, such as the Paragon, Delta, CM-5 and SP-1. The CHAOS library provides efficient runtime primitives for distributing data and computation over processors; it supports efficient index translation mechanisms and provides users high-level mechanisms for optimizing communication. CHAOS subsumes the previous PARTI library and supports a larger class of applications. In particular, it provides efficient support for parallelization of adaptive irregular programs where indirection arrays are modified during the course of computation. To demonstrate the efficacy of CHAOS, two challenging real-life adaptive applications were parallelized using CHAOS primitives: a molecular dynamics code, CHARMM, and a particle-in-cell code, DSMC.

Besides providing runtime support to users, CHAOS can also be used by compilers to automatically parallelize irregular applications. This paper demonstrates how CHAOS can be effectively used in such a framework. By embedding CHAOS primitives in the Syracuse Fortran 90D/HPF compiler, kernels taken from the CHARMM and DSMC codes have been parallelized automatically.

KEY WORDS Distributed memory multiprocessors Runtime compilation Adaptive irregular programs Parallelizing compilers

INTRODUCTION

In recent years, distributed memory parallel machines have been the popular target for development of scientific applications. Distributed memory machines are popular because their architectures are scalable, and they promise the most likely means of achieving teraflops performance. However, programming current distributed memory machines to get good speedups and efficiency proves to be cumbersome, since users must explicitly coordinate data movements through low-level message passing calls. To ease programming, there have been major efforts in developing programming languages, compiler support and runtime support for distributed memory machines.

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

0038–0644/95/060597–25$17.50

Received 29 August 1994
Revised 5 December 1994

Programming languages such as Fortran D [1], High Performance Fortran (HPF) [2], and Vienna Fortran [3] allow users to program in the *data-parallel* framework, where the computation is single threaded and structured in loosely synchronous phases. The user views memory through a global name space. Compilers for these languages automatically partition data across processors, distribute loop iterations, perform global to local translations, and produce Single Program Multiple Data (SPMD) codes with appropriate embedded message passing calls. Runtime support provides optimized libraries to the compiler for performing common high-level tasks involved in parallelization, such as partitioning data, distributing loop iterations, translating from global name space to local name space and communicating data. Thus the focus of much research has been defining languages that ease the task of the programmer [1,2,3], and developing compiler [4,5,6] and runtime support [7] that allow efficient parallelization of codes written in these languages.

This paper addresses a particular class of scientific application programs, called *irregular programs*, which require special runtime and compiler support for parallelization. Irregular applications are characterized by the use of indirect indexing of data arrays. This means that the data arrays are indexed through the values in other arrays, which are called *indirection arrays*. The use of indirect indexing causes the data access patterns, i.e the indices of the data arrays being accessed, to be highly irregular, leading to difficulties in determining communication requirements. Examples of irregular applications are found in unstructured computational fluid dynamic solvers [8], molecular dynamics codes (CHARMM [9], AMBER [10], GROMOS [11], etc), diagonal or polynomial preconditioned iterative linear solvers [12], and particle-in-cell (PIC) codes [13].

```
      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

Figure 1 illustrates a typical irregular loop. The data access pattern is determined by indirection arrays, `ia` and `ib`. At runtime, once the data access pattern is known, a two-phase *inspector/executor* strategy can be used to parallelize such a loop-nest [7]. The loop in Figure 1 is sometime called a *static* irregular loop, since the data access pattern of the inner loop (L2) remains unchanged throughout all iterations of the outer loop (L1). However, some irregular applications are *adaptive*, in the sense that the data access patterns may change during computation.

This paper presents a new set of runtime procedures designed to efficiently implement adaptive irregular programs on distributed memory machines. This runtime library is called CHAOS [14]; it subsumes PARTI, a library aimed at static irregular programs [7,15]. CHAOS introduces two new features — *light-weight schedules* and *efficient schedule generation*, which are useful in certain types of adaptive programs. CHAOS has been used to parallelize two challenging real-life adaptive applications — CHARMM, a molecular dynamics code and DSMC [16], a particle-in-cell code. Also presented is language support to enable compilers to generate efficient codes for adaptive programs. The Syracuse Fortran 90D/HPF compiler [4] was used as a test-bed for the ideas presented in this paper.

ADAPTIVE IRREGULAR LOOP STRUCTURES

In adaptive irregular programs, such as adaptive fluid dynamics and molecular dynamics codes, interactions between entities (mesh points, molecules, etc.) change during computation (due to mesh refinement or movement of molecules). Since interactions are specified by indirection arrays, the adaptivity of irregular programs is represented by the frequency of modification on indirection arrays.

```
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

S:   if (required) then                             ! under certain conditions
       regenerate ic(:)                             ! indirection array may change

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

Figure 2. A Code That Adapts Occasionally

Figure 2 illustrates properties of loops found in molecular dynamics codes and unstructured fluid dynamics codes. Here, multiple loops access the same data arrays but with different access patterns. In loop L2 the data arrays `x` and `y` are indirectly accessed using arrays `ia` and `ib`. In loop L3 the same data arrays are indirectly accessed using indirection array `ic`. The data access pattern in loop L2 remains static, whereas the data access pattern in loop L3 changes whenever the indirection array `ic` is modified. The adaptivity of the loop is controlled by the conditional statement S.

```
do i = 1, rows
  do j = 1, size(i)                ! size(i) is the number of elements in the ith row
    new_data(icell(i,j), next(icell(i,j))) = data(i,j)
    next(icell(i,j)) = next(icell(i,j)) + 1
  end do
end do
```

Figure 3. Example of data movement in a Particle-in-cell code

In other applications, such as DSMC and PIC codes, data access patterns and computational load change frequently. Thus, data arrays may need to be reshuffled or 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. A typical loop for performing such data movement is shown in Figure 3. Here, elements are moved across the rows of a 2-D array based on the information provided in indirection array `icell`. The elements of array `data` are shuffled and stored in array `new_data`. While the total number of rows remains the same after the shuffle, the size of individual rows may not. At the programming language level this process can be viewed as the first dimension being statically distributed while the second dimension is compressed. Here, elements migrate across the first dimension. Figure 4 shows an example of such data movement. Usually, the order in which elements append to new rows is not strictly maintained since the order of computation over these elements does not matter. By taking advantage of this property efficient runtime procedures for fast data migration have been developed.

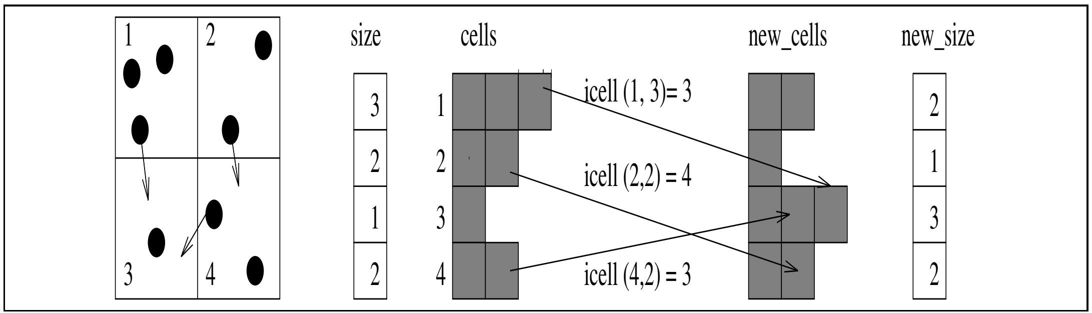

Figure 4. Data movement in a Particle-in-cell code

RUNTIME SUPPORT

This section presents a brief overview of the CHAOS runtime support library, a superset of the PARTI library, and describes the new features, light-weight schedules and two-phase schedule generation, which are designed to handle adaptive irregular programs.

Overview of CHAOS

The CHAOS runtime library has been developed to efficiently handle irregular programs. The library is designed to ease the implementation of computational problems on parallel architecture machines by relieving users of low-level machine specific issues. Solving such irregular problems on distributed memory machines using CHAOS runtime support involves six major phases (Figure 5). The first four phases concern mapping data and computations onto processors. The next two steps concern analyzing data access patterns in loops and generating optimized communication calls. Detailed description of these phases can be found in Reference 17.

In static irregular programs, Phase F is typically executed many times, while phases A through E are executed only once. In some adaptive programs where data access patterns change periodically but reasonable load balance is maintained, phase E must be repeated whenever the data access patterns change. In highly adaptive programs, the data arrays may

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 5. Solving Irregular Problems

need to be repartitioned in order to maintain load balance. In such applications, all the phases are repeated.

Two-Phase Schedule Generation

A *communication schedule* is used to fetch off-processor elements into a local buffer before computation phase, and to scatter these elements back to their home processors after the computational phase is completed. Communication schedules determine the number of communication startups and the volume of communication. Therefore, it is important to optimize the schedule generation.

The basic idea of the inspector/executor concept is to hoist preprocessing outside the loop as much as possible so that it need not be repeated unnecessarily. In adaptive codes where the data access pattern occasionally changes, the inspector is not a one-time preprocessing cost. Every time an indirection array changes, the schedules associated with it must be regenerated. For example, in Figure 6, if the indirection array `ic` is modified, the schedules inc_sched_c and $sched\_c$ must be regenerated. Generating inc_sched_c involves inspecting $sched\_ab$ to determine which off-processor elements are duplicated in that schedule. Thus, it must be certain that communication schedule generators are efficient while maintaining the necessary flexibility.

In CHAOS, the schedule-generation process is carried out in two distinct phases.

- **Index analysis** phase examines the data access patterns to determine which references are off-processor, removes duplicate off-processor references by only keeping distinct references in hash tables, assigns local buffer for off-processor references, and translation global indices to local indices.
- **Schedule generation** phase generates communication schedules based on the information stored in hash tables.

The communication schedule for processor p stores the following information:

1. send list – a list of arrays that specifies the local elements of a processor p required by all processors,

```
L1:  do n = 1, nsteps                                         ! outer loop
       call gather(y(begin_buff), y, sched_ab)                ! fetch off-proc data
       call zero_out_buffer(x(begin_buff), offp_x)            ! initialize buffer
L2:    do i = 1, local_sizeof_indir_arrays                    ! inner loop
         x(local_ia(i)) = x(local_ia(i))
                    + y(local_ia(i)) * y(local_ib(i))
       end do

S:     if (required) then
         modify part_ic(:)                                    ! ic is modified
         CHAOS_clear_mask(hashtable, stamp_c)                 ! clear ic
         local_ic(:) = part_ic(:)
         stamp_c = CHAOS_enter_hash(local_ic)                 ! enter new ic
         inc_sched_c = CHAOS_incremental_schedule(stamp_c)    ! incremental sched
         sched_ac = CHAOS_schedule(stamp_a, stamp_c)          ! sched for ia, ic
       endif

       call gather(y(begin_buff2), y, inc_sched_c)            ! incremental gather
       call zero_out_buffer(x(begin_buff2), offp_x2)          ! initialize buffer
L3:    do i = 1, local_sizeof_ic                              ! inner loop
         x(local_ic(i)) = x(local_ic(i)) + y(local_ic(i))
       end do
       call scatter_add(x(begin_buff), x, sched_ac)           ! scatter addition
     end do
```

Figure 6. Schedule Generation for an Adaptive Program

2. permutation list – an array that specifies the data placement order of off-processor elements in the local buffer of processor p,
3. send size – an array that specifies the sizes of out-going messages of processor p to all processors, and
4. fetch size – an array that specifies the sizes of in-coming messages to processor p from all processors.

The principal advantage of such a two-step process is that some of the index analysis can be reused in adaptive applications. In the index analysis phase, hash tables are used to store global to local translation and to remove duplicate off-processor references. Each entry keeps 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, and
4. stamp – 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.

Stamps are very useful when implementing adaptive irregular programs, especially for those programs with several index arrays in which most of them are static. In the index analysis phase, each index array hashed into the hash table is assigned a unique stamp that marks all its entries in the table. Communication schedules are generated based on the combination of stamps. If any one of the index arrays changes, only the entries pertaining to the index array, i.e. those entries with the stamp assigned for the index array, have to be removed from hash table. Once the new index array is hashed into the hash table, a new schedule can be generated without rehashing other index arrays.

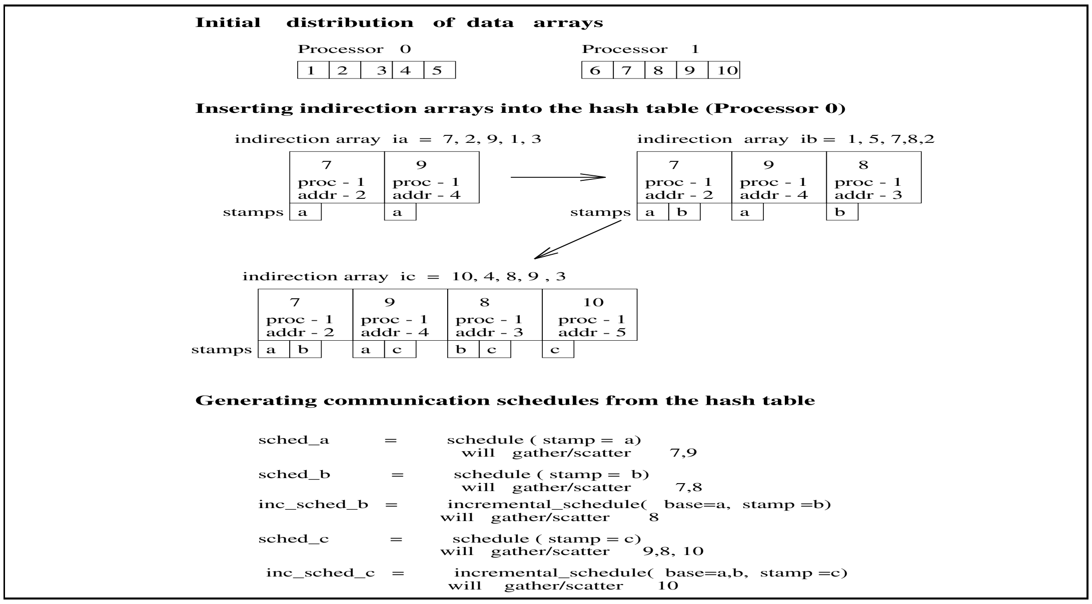

Figure 7. Schedule Generation with Hash Table

Figure 6 illustrates how CHAOS primitives (in pseudo-code) are used to parallelize the adaptive problem. The conditional statement S may modify the indirection array `ic`. Whenever this occurs, the communication schedules that involve prefetching references of `ic` must be modified. Since the values of `ic` in the hash table are no longer valid, the entries with stamp *stamp_c* are cleared by calling $CHAOS\_clear\_mask()$. New values of `ic` are then entered into the hash table by $CHAOS\_enter\_hash()$. After all indirection arrays have been hashed in, communication schedules can be built for any combination of indirection arrays by calling $CHAOS\_schedule()$ or $CHAOS\_incremental\_schedule()$ with an appropriate combination of stamps.

An example of schedule generation for two processors with sample values of indirection arrays `ia`, `ib`, and `ic` is shown in Figure 7. The global references due to indirection array `ia` are stored in hash table H with stamp a, `ib` with stamp b and `ic` with stamp c. The indirection arrays might have some common references. Hence, a hashed global reference might have more than one stamp. The gather schedule $sched\_ab$ for the loop L2 in Figure 6 is built using the union of references with time stamps a or b. The scatter operation for loop L2 can be combined with the scatter operation for the loop L3. The gather schedule *inc_sched_c* for loop L3 is built with those references that have time stamp c alone because references with time stamps a or b as well as with c can be fetched by using the schedule *sched_ab*. The scatter schedule for loops L2 and L3 is built using the union of references with time stamps a and c.

PARTI, the runtime library that preceeded CHAOS, also had support for building incremental and merged schedules [18]. However, in PARTI, such schedules were built using specialized functions for these purposes. The CHAOS library restructures the schedule generation process and by the using a global hash table provides a uniform interface for building all types of schedules. Such a uniform interface is easier to use for both users and compilers that automatically embed CHAOS schedule generation calls.

Light-Weight Schedules

In certain highly adaptive problems, such as those using particle-in-cell methods, data elements are frequently moved from one set to another during the course of the computation. The implication of such adaptivity is that preprocessing for a loop must be repeated whenever the data access pattern of the loop changes. In other words, previously built communication schedules cannot be reused and must be rebuilt frequently.

In such applications a significant optimization in schedule generation can be achieved by recognizing that the semantics of set operations imply that elements can be stored in sets in any order. This information can be used to build much cheaper *light-weight* communication schedules. During schedule-generation, processors do not have to exchange the addresses of all the elements they will be accessing with other processors , they only need to exchange information about the number of elements they will be appending to each set. This greatly reduces the communication costs in schedule generation. A light-weight schedule for processor p stores the following information:

1. send list – a list of arrays that specifies the local elements of processor p required by all processors,
2. send size – an array that specifies the out-going message size of processor p to all processors, and
3. fetch size – an array that specifies the in-coming message size of processor p from all processors.

Thus, light-weight schedules are similar to the previously described schedules except that they do not carry information of data placement order in the receiving processor. While the cost of building a light-weight schedule is less than that of regular schedules, a light-weight schedule still provide the same communication optimizations of aggregating and vectorizing messages [18].

EXPERIMENTAL RESULTS

This section presents computational structures and performance of two adaptive irregular application programs: 1) a molecular dynamics code – Chemistry at HARvard Macromolecular Mechanics (CHARMM), and 2) a particle-in-cell code – direct simulation Monte Carlo (DSMC). These two application programs are ported to distributed memory machines using CHAOS primitives. The application code CHARMM adapts occasionally, whereas the application code DSMC adapts frequently. Experimental results are presented for these two parallelized programs on the Intel iPSC/860 machine.

CHARMM

Overview

CHARMM is a program that calculates empirical energy functions to model macromolecular systems. The purpose of CHARMM is to derive structural and dynamic properties of molecules using the first and second order derivative techniques [9].

The computationally intensive part of CHARMM is the molecular dynamics simulation. It simulates the 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 new spatial positions of the atoms. The positions of the atoms are fixed during the energy calculation; however, they are updated when spatial displacement due to force are calculated.

The loop structures of molecular dynamics simulations is shown in Figure 8. The physical values associated with atoms, such as velocity, force and displacement are accessed using indirection arrays (IB, JB, etc.). The energy calculations in the molecular dynamics simulations comprises two types of interactions – bonded and non-bonded.

Bonded forces exist between atoms connected by chemical bonds. CHARMM calculates four types of bonded forces – bond potential, bond angle potential, dihedral angle (torsion) potential, and improper torsion. These forces are short-range, i.e. forces existing between atoms that lie close to each other in space. Bonded interactions remain unchanged during the entire simulation process because the chemical bonds of structures do not change. The complexity of bonded forces calculations is nearly linear to the number of atoms because each atom has a finite number of bonds with other atoms.

Non-bonded forces are the van der Waals interactions and electrostatic potential between all pairs of atoms. The time complexity of non-bonded forces calculations is $O(N^2)$ because each atom interacts with all other atoms in the system. When simulating large molecular structures, CHARMM approximates this calculation by ignoring all interactions beyond a certain cutoff point. This approximation is done by generating a non-bonded list, array JNB, which contains all pairs of interactions within the cutoff point. The spatial positions of the atoms change after a time step, consequently, the same set of atoms may not be included in subsequent time steps. Hence, a new non-bonded list must be generated. However, in CHARMM, the user has control over non-bonded list regeneration frequency.

Parallelization Approach

Data Partition

Spatial information is associated with each atom. Bonded interactions occur between atoms in close proximity to each other. Non-bonded interactions are excluded beyond a certain cutoff point. Additionally, the amount of computation associated with an atom depends on the number of atoms with which it interacts – the number of JNB entries for that atom. The way in which the atoms are numbered frequently does not have a useful correspondence to the interaction pattern of the molecule. A naive data distribution such as BLOCK or CYCLIC may result in a high volume of communication and poor load balance. Hence, data partitioners such as recursive coordinate bisection (RCB) [19] and recursive inertial bisection (RIB) [20], which use spatial information as well as computational load, are good candidates to partition atoms

```
L1:   DO N = 1, nsteps
        Regenerate non-bonded list if required
        ...
C       Bonded Force Calculations
L2:     DO I = 1, NBONDS
          Calculate force between atoms IB(I) and JB(I)
        END DO
L3:     DO I = 1, NANGLES
          Calculate angle potential of atoms IT(I), JT(I), and KT(I)
        END DO
        ...
C       Non-Bonded Force Calculation
L4:     DO I = 1, NATOMS
          DO J = INBLO(I)+1, INBLO(I+1)
            Calculate force between atoms I and JNB(J)
          END DO
        END DO
        Integrate Newton's Equations and Update Atom Coordinates
      END DO
```

Figure 8. Molecular Dynamics Simulation Code from CHARMM

over processors. Note that all data arrays that are associated with atoms are distributed in an identical manner.

Iteration Partition

Once the atoms are partitioned, the distribution is used to decide how loop iterations are partitioned among processors. Each iteration of bonded force calculations is assigned to the processor that has the maximum number of local distributed array elements. If the choice of processor is not unique, the processor with the lowest computational load is chosen primarily to reduce off-processor accesses. Bonded force calculations consume about 1% of the total execution time for energy calculation. Non-bonded calculations consume 90% of the execution time. Hence, balancing the computational load due to non-bonded calculations is of primary concern. To balance load, the non-bonded force calculation pertaining to an atom is assigned to the processor that owns the particular atom since the atoms are distributed using both geometrical and computational load information. Hence, each iteration of the outer loop is assigned to the processor that owns the atom.

Remapping and Loop Preprocessing

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 to translate global references to local ones and to generate communication schedules for exchanging data among processors.

Indirection arrays used in bonded force calculation loops remain unchanged while the non-bonded list adapts during computation. Hence, preprocessing 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 are very useful for loop preprocessing. 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 stamp can be reused and the new non-bonded list entries are hashed with the reused stamp.

Performance

The performance of molecular dynamics simulations was studied with a benchmark case (MbCO + 3830 water molecules) on the Intel iPSC/860. It ran for 1000 steps with 40 non-bonded list updates. The cutoff for non-bonded list generation was 14 Å. The non-bonded list was regenerated 40 times during the simulation. The results are presented in Table I. The RCB partitioner was used to partition atoms. The execution time included the energy calculation time and communication time of each processor. The computation time was the average of the computation time of dynamics simulations over processors; the communication time was the average communication time. The *load balance index* was calculated as

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

The results showed that CHARMM scaled well and that good load balance was maintained up to 128 processors.

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

Number of Processors	1	16	32	64	128
Execution Time	74595.5*	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

* Estimation done by Brooks and Hodošček[21]

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 II. The data partition time is the execution time of RCB. 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 II, the regeneration time is shown as non-bonded list generation time.

During simulation, the non-bonded list was regenerate periodically. When the non-bonded list was updated, the schedule must be regenerated. The schedule regeneration time in Table II gives the total schedules regeneration time for 40 non-bonded list updates. By comparing these numbers to those in Table I, it can be observed that the preprocessing overhead is relatively small 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 a separate schedule to gather and scatter off-processor data for each irregular loop.

Table II. 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

A second approach is to compute a single schedule using the schedule merging technique. Table III compares the performance of these techniques and demonstrates the usefulness of schedule merging.

Table III. Communication Time (in sec.)

	Schedule Merging		Multiple Schedules	
Number of Processors	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

Direct Simulation Monte Carlo

This section presents an example of a highly adaptive code, DSMC code. It is similar to PIC codes in that it tries to simulate the physics of flows directly through Lagrangian movement and particle interaction. The fundamental structure of the computational requirements of DSMC methods is described along with the parallelization strategy using CHAOS runtime support. The performance results of 2-dimensional and 3-dimensional DSMC codes on the Intel iPSC/860 hypercube are also presented.

Overview

The DSMC method is a technique for computer modeling a gas by a large number of simulated molecules. It features highly efficient movement and collision handling of simulated molecules on a spatial flow field domain overlaid by a Cartesian mesh [22]. The spatial location of each molecule is associated with a Cartesian mesh cell. Physical quantities such as the velocity components, rotational energy and position coordinates are associated with each molecule and are modified with time as the molecules are concurrently followed through representative collisions and boundary interactions in simulated physical space.

Changes in position coordinates may cause molecules to move between cells The cost of transmitting molecules among cells can be significant on distributed memory parallel computers since a substantial number of molecules migrate in each time step, and each molecule is usually associated with several bytes of data. Moreover, since data associated with the molecules need to be reshuffled in order to accommodate the moved molecules within each processor's local memory, new indirect access patterns are regenerated every time step. Applications that have characteristics like the DSMC computation require runtime support that provides

- efficient data transportation mechanisms for particle movement, and
- fast dereferencing mechanisms to identify the destination processors of the moved particles.

Furthermore, such molecule movement may lead to fluctuation in work load distribution among processors. The problem domain may need to be repartitioned frequently in order to balance the work load. This characteristic raises an issue of load balance and requires

- effective domain partitioning methods, and
- good policy for domain repartitioning decisions.

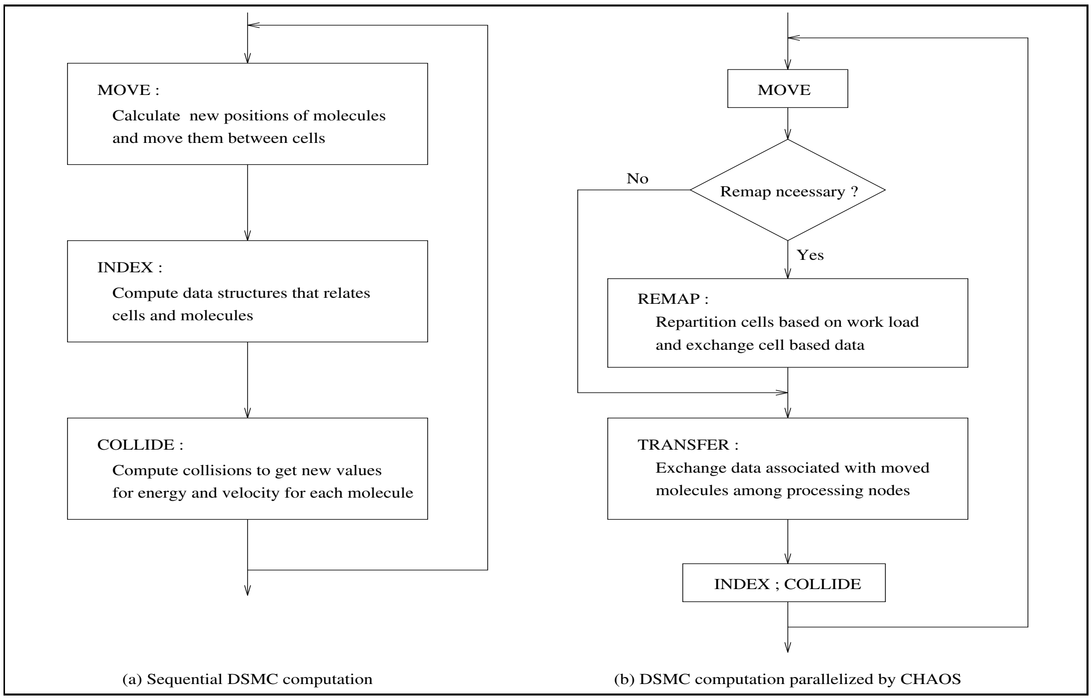

Figure 9. Structures of 3-dimensional DSMC Computation

Parallelization Approach

Figure 9 depicts the fundamental structure of a typical DSMC computation along with its parallelized version. In the COLLIDE phase, molecules within each cell collide with each other and update their positions. If the arrays associated with molecules are distributed using a regular data distribution scheme then there can be a significant amount of communication in this phase in bringing the molecule-data from the home processor to the cell that owns it and moving it back after the collision phase. Instead, the molecule arrays can be redistributed every time step such that the molecule data are owned by the same processor as the cell in which the molecule lies. The key component of this redistribution is the *MOVE* procedure which calculates new positions of molecules and moves them to the proper cells in the global address space. When a DSMC code runs on distributed memory parallel machines, the *MOVE* procedure requires information exchange among participating processing nodes. Two new procedures are appended to deal with molecule movement and cell partitioning for load balance. Procedure *TRANSFER* carries out actual communication to exchange data associated with moved molecules using CHAOS adaptive data migration primitives. When it is needed to repartition cells to achieve better load balance, procedure *REMAP* invokes a domain partitioner that may be chosen by the user, and redistributes cell-based distributed arrays according to a new partition of cells.

Efficient Data Migration
Communication optimization is crucial for optimal performance of the *TRANSFER* and *REMAP* procedures. The regular communication schedules, such as those used in CHARMM offer highly communication optimizing strategies to exchange off-processor data. However, in applications such as DSMC, the data movement is quite different from that in CHARMM. Firstly, the data movement can be viewed as a redistribution of the molecule arrays, rather than a data-copying. Secondly, because this is a redistribution of molecules, the order in which the molecules are placed locally in each processor is not important. In such cases, the light-weight schedules can be used to transport data, at much lower overheads than regular communication schedules, which have to ensure proper ordering communicated data.

Remapping for Load Balancing
DSMC codes can be characterized by statistical calculations carried out involving particles associated with each cell. Particles move to new cells as a result of calculations, and cells are partitioned over the processing nodes. The combination of these characteristics may lead to system performance deterioration over time when the problem domain is partitioned statically. The performance of the 3-dimensional DSMC code can be improved by periodic redistribution of computational load with the help of partitioners such as RCB and RIB. A *chain partitioner* [23,24] is also used to take advantage of the highly directional nature of particle flow that characterizes some DSMC communication patterns. In the experiments reported here, more than 70 percent of the molecules move along the positive x-axis. This property allows us to use a very fast one dimensional partitioner. Experiments show that the chain partitioner reduces partitioning cost dramatically to a scale that is conformable to adaptive data migration primitives. It also achieves nearly the same quality of load balance when compared with recursive bisections.

Performance Results

Table IV. Regular Schedules vs. Light-Weight Schedules

(Time in secs)	48x48 Cells				96x96 Cells			
Number of 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 IV compares the execution time of 2-dimensional DSMC code using the light-weight schedules with the time obtained using the regular communication schedules. In this experiment, molecules were evenly distributed over the whole domain, so load balance was not an issue. Regular communication schedules performed much worse than light-weight schedules because of two reasons. Firstly, the overhead of building regular schedules is much higher than that of constructing light-weight schedules. This is because regular schedules need to keep track of the ordering of incoming data, whereas light-weight schedules only need to keep track of the amount of incoming data, not their local ordering. This extra overhead must be borne every iteration. Secondly, the version of DSMC using regular schedules assumes a static distribution of the arrays containing molecule information. As molecules move across cells, maintaining the original distribution causes a large amount of communication in the COLLIDE phase where off-processor molecules lying in the cell must be gathered, collided and scattered back to their home processors. In contrast, the light-weight schedule version of the code redistributes the molecule arrays so that molecules are always aligned with the cells they fall in, ensuring the COLLIDE computation is local.

Table V. Performance Effects of Remapping (Remapped Every 40 Time Steps)

	Number of processors					Sequential
(Time in secs)	8	16	32	64	128	Code
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	

Periodic data remapping provides better performance compared with that of static partitioning. Experiments done with the 3-dimensional DSMC code on 128 processors reveal that the degree of load imbalance does not exceed 30 percent of perfect load balance, whereas static partitioning exceeds 400 percent. Table V compares the performances of periodic domain partitioning methods with that of static partitioning (i.e. no remapping), for 3-dimensional DSMC codes. The domain is repartitioned every 40 time steps based on the workload information collected for each Cartesian mesh cell. The table presents execution time for 1000 time steps. The results show that the repartitioning method outperforms static partitioning significantly on a small number of processors. However, using a recursive inertial bisection partitioner leads to performance degradation on a large number of processors. The net result of this method is a poorer performance than that of the static partitioning. This performance degradation is a result of the large communication overhead that increases as the number of processors increases. The chain partitioner, on the contrary, provides better results for this problem.

COMPILING ADAPTIVE IRREGULAR PROGRAMS

There are a wide range of languages, such as Fortran D [1], HPF [2], pC++ [25], and Vienna Fortran [3], 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.

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*. 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 distributions [2], respectively.

Irregular Distribution

On distributed memory machines, irregular concurrent problems may not run efficiently with standard data distributions such as BLOCK and CYCLIC [26,27,28]. Researchers have developed a variety of heuristic methods to obtain data mappings that are designed to optimize irregular problem communication requirements [26,29,30,31,32]. 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 distributions as well. In Figure 10 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 mapped onto the processors.

```
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 10. Fortran D Irregular Distribution

A specific array is associated with a distribution using the Fortran D statement **ALIGN**. In statement S3 of Figure 10, 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` will be 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. However, the partitioner may not always be available in Fortran D. In such cases, it can be called as an extrinsic procedure.

Computational Loop Structures

The implementation of the FORALL construct in Fortran D follows copy-in-copy-out semantics – loop carried dependencies are not defined. In this implementation, loop carried dependencies that arise due to reduction operations are defined. Reduction operations are specified in a FORALL construct using the Fortran D **REDUCE** construct. Reduction inside a FORALL construct is important for representing computations such as those found in sparse and unstructured problems. This representation also preserves explicit parallelism available in the underlying computations. Figure 11 shows how the reduction in Figure 1 would be written within this framework.

```
L1:  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 11. Example Reduction Loop in Fortran D

Reduce Append

In highly adaptive codes, the data access patterns change frequently. Figure 3 shows an example of such a code. Elements of the 2-D array `data` are moved across rows based on the indirection array `icell`. When these programs are executed on distributed memory machines, array elements will be moved across processors based on the distribution of rows of the 2-D array `data`.

In some programs, such as DSMC, the computational results after such inter-row data movement do not depend on the ordering of elements in each row, and hence the data placement order needs not be strictly maintained. 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 implemented by appending each element into an array that is being logically used as an unordered list. Since an operation that adds elements to unordered lists is associative and commutative, this process of appending elements can be viewed as a *reduction* operation. A reduction is defined as an accumulation of elements of a vector, where the accumulating operation performed on each element is associative and commutative (e.g. summing the elements of a vector). In a loop that performs a reduction, the output dependencies between different iterations can be ignored, thus enabling parallelization.

Recognizing that a particular data movement is a reduction operation can lead to significant optimizations. It is possible with the existing compiler techniques to compile irregular loops where data access patterns are known only at runtime due to indirections [6], provided the computation inside the loop is known to be a reduction. For such loops, the compiler generates a preprocessing code that at runtime generates appropriate communication calls and places off-processor data in a pre-determined order. However, current techniques do not automatically detect reductions in data movement, such as that in Figure 3.

In order to allow compilers to detect reduction in data movement, an intrinsic function called *reduce*(*append*, ...) is proposed, which is much like the Fortran D *reduce* intrinsics. This intrinsic function will direct the compiler to adopt the appropriate data moves (i.e to use light-weight schedules). 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 compilers using the proposed intrinsic. Figure 12 shows how such an intrinsic would be used for the loop shown in Figure 3.

```
FORALL i = 1, rows
 FORALL j = 1, size(i)        ! size(i) is the number of elements in the ith row
  REDUCE(APPEND, new_data(ia(i,j), :), data(i,j))
 END DO
END DO
```

Figure 12. Example Reduce Append Loop in Fortran D

```
!HPF$   TEMPLATE rowtemp(rows)
!HPF$   DISTRIBUTE rowtemp(BLOCK)
!HPF$   ALIGN with rowtemp :: data(:,*), new_data(:,*), ia(:,*),
        .  mask(:,*), bit(:,*), size(:), new_size(:)
        real, allocatable(rows,:) :: new_data

C       Mask off array elements which do not store data and
           compute the number of elements which will be stored in new rows
L1:     forall(i=0:rows,j=1:max_size) mask(i,j) = (j ≤ size(i))
L2:     forall(i=0:rows,j=1:max_size, mask) bit(i,j) = 1
L3:     forall(i=0:rows) new_size(i) = 0
S1:     new_size = sum_scatter(bit,new_size,ia,mask)

C       Allocate memory for new rows and perform data movement
S2:     max_size = maxval(new_size)
S3:     allocate(new_data(rows,max_size))
S4:     new_data = list_scatter(data,new_data,ia,mask)
```

Figure 13. Example Reduce Append Operation in HPF

Relationship to HPF

While the Fortran D system has been used to demonstrate the runtime techniques, the same techniques can be used by other compilers as well. This section presents how the techniques could be extended to HPF.

The current version of HPF (HPF-1) does not support distributions other than standard BLOCK and CYCLIC distributions. However, HPF can indirectly support such distributions using the data reordering technique, which has been successfully implemented recently by researchers to parallelize their applications [33,34]. In their approach, the effect of a non-standard distribution is achieved while maintaining BLOCK distribution by first obtaining the non-standard distribution using a domain partitioner and then renumbering the array elements. Those functions that partition domains and renumber array elements can be incorporated into HPF programs as extrinsic procedures.

For each reduction operation, HPF introduces a combining-scatter function [2]. For instance, the **sum_scatter** function in the statement S1 in Figure 13 accumulates values of unmasked elements of a 2-dimensional array `bit` to an 1-dimensional array `new_size` whose particular offsets are determined by a 2-dimensional indirection array `ia`. In this particular example, since the values of all the unmasked elements of `bit` are one, each element of the `new_size` is assigned the number of unmasked elements in each row of the `bit` array.

The reduce append operation can also be extended to HPF in the same way to the **sum_scatter** function. Since the reduce append operation is associative and commutative, it can be considered as a new extrinsic reduction operation. The **list_scatter** function in statement S4 in Figure 13 is the a combining-scatter function for the reduce append operation. The **list_scatter** function adds elements of its operand to an unordered list in random order. In this example, the **list_scatter** function takes the elements of a source array `data` appends them into rows of a destination array `new_data` as determined by an indirection array `ia`. It should be noted that since the sizes of the unordered lists (i.e., the sizes of the rows of the destination array `new_data`) are determined by the indirection array `ia` they are known only at runtime. Therefore the destination operand needs to be an array that has ALLOCATABLE attribute or POINTER attribute in order to store varying numbers of elements by dynamically allocating memory. The statements S2 and S3 are responsible for the dynamic memory allocation.

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 generates codes that the CHAOS procedures are embedded in the translated codes. The performance of the compiler generated codes is compared with that of the hand parallelized versions. All measurements were done on the Intel iPSC/860 machine.

CHARMM

The non-bonded force calculation loop is computationally intensive and also adapts every few time steps. A simplified Fortran D version of the non-bonded force calculation loop is shown in Figure 14. 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 14, 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. The transformed loop is shown in Figure 15.

Carrying out preprocessing for irregular loops is an expensive process. However, if data access patterns do not change, the results from preprocessing could be reused. A detailed

```
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 14. Non-bonded Force Calculation Loop of CHARMM in Fortran D

implementation of reusing the results of preprocessing is discussed in Reference 6. In this approach, compiler generated code maintains a record of when the 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 was 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 VI. Performance of Hand-Coded and Compiler-Generated CHARMM Loops

(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

The experimental results that compare the performance of compiler-generated code with that of a hand-coded version are presented in this section. Both versions of the program ran the calculations of the benchmark case (MbCO + 3830 water molecules) for 100 iterations. In order to simulate adaptivity of the non-bonded force calculation loop, data arrays are redistributed every 25 iterations by applying RCB and RIB alternately. Therefore, data arrays and iterations are redistributed four times during the execution. Table VI lists the cost of data partitioning, data and indirection arrays remapping, and preprocessing and execution. The performance of the compiler generated code is almost as good as that of the hand parallelized code.

```
C    Start with block distribution of arrays
S1   Obtain new distribution format (map) from an extrinsic partitioner procedure
S2   Remap arrays (x and y) aligned with distribution reg to distribution map
     ...
C$   Parallelized irregular loop - inspector and executor with schedule saving
S1   Check if new inspector needed
     If needed
S3       call CHAOS procedure to Compute new schedule
S4       Store new schedule and loop bound information
     Else
S5       Retrieve previous schedule and loop information
     End if
S6   call CHAOS procedure to Gather off-processor data
S7   Execute loop
S8   call CHAOS procedure to Scatter off-processor data
```

Figure 15. Compiler Transformations

DSMC

The key component of the DSMC computation is a *MOVE* procedure that 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 16 shows a simplified version of particle movement of 2-dimensional DSMC code in Fortran D. Cells are distributed across processors. An indirection array *icell(i,j)* is used to represent a 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. A loop L1 in the figure redistributes velocity components *vel* associated with individual particles using the *reduce append* intrinsic that is proposed in this paper. Loops L2 and L3 are responsible for recomputing the number of particles in each cell.

An abstract version of the compiler generated particle movement code is shown in Figure 17. When a *reduce append* statement is encountered, the compiler generates a sequence of calls to CHAOS data migration primitives (statements S1 and S2) in order to carry out the data movement as a *reduction*. 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.

Table VII. Performance of Compiler Generated and Manually Parallelized DSMC Codes

(Time in secs)	Compiler Generated				Manually Parallelized			
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

Performance results for both the compiler generated and the manually parallelized 2-

```
C$     DECOMPOSITION celltemp(num_cells)
C$     DISTRIBUTE celltemp(BLOCK)
C$     ALIGN icell(*,:),vel(*,:),size(:),newsize(:) 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
        newsize(j) = 0
       END FORALL
L3:    FORALL j = 1, num_cells
        FORALL i=1, size(j)
         newsize(icell(i,j)) = newsize(icell(i,j))+1
        END FORALL
       END FORALL
```

Figure 16. DSMC Particle Movement Code in Fortran D

dimensional DSMC code with 32x32 cells and 5K molecules are presented in Table VII. 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 that automatically return the new number of particles updated by *reduce append* operation, the compiler generated code explicitly computes it. Hence, the compiler generated code performs additional communication by invoking CHAOS procedures.

RELATED WORK

Several researchers have developed programming environments that target particular classes of irregular or adaptive problems. Williams [29] describes a programming environment (DIME) for calculations with unstructured triangular meshes using distributed memory machines. Baden [35] 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 distributed memory multiprocessors: the Fortran D compiler projects at Rice and Syracuse [1,4] and the Vienna Fortran compiler project [3] at the University of Vienna are two examples. The Jade project at Stanford [36], the DINO project at Colorado [37], Kathy Yelick's work [38] at Berkeley, and the CODE project at University of Texas, Austin provide parallel programming environments. Runtime compilation methods have been employed in four compiler projects: the Fortran D project [39], the Kali project [40], Marina Chen's work at Yale [41] and the PARTI project [7,15,42]. The Kali compiler was the first compiler to implement inspector/executor type runtime preprocessing [40] and the ARF compiler

```
C     Reduce-append the particle information into new cells according to icell array
S1:   call CHAOS schedule generator for data migration
S2:   call CHAOS data migration primitives to relocate moved particles

C     Compute the number of particles in each cell
L2:   do j = 1, local_num_cells
        newsize(j) = 0
      enddo
S3:   call CHAOS procedure to compute schedule for off-processor elements of newsize
L3:   do j = 1, local_num_cells
        do i = 1, size(j)
          newsize(icell(i,j)) = newsize(icell(i,j)) + 1
        enddo
      enddo
S4:   call CHAOS procedure to scatter-add off-processor elements of newsize
```

Figure 17. Compiler Generated DSMC Particle Movement Code

was the first compiler to support irregularly distributed arrays [42].

CONCLUSIONS

The CHAOS procedures described in this paper forms a portion of a portable, compiler independent, runtime support library. The CHAOS runtime support library contains procedures that

1. support static and dynamic distributed array partitioning,
2. partition 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.

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

Also discussed were 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.

ACKNOWLEDGEMENTS

The authors would like to thank Richard Wilmoth at NASA Langley for his help about the parallelization of direct simulation Monte Carlo methods.

The authors would like to thank Geoffrey Fox, Alok Choudhary, and Sanjay Ranka for many enlightening discussions, and thank Chuck Koelbel, Ken Kennedy and Seema Hiranandani for many useful discussions about integrating Fortran D runtime support for irregular problems.

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

The authors wish to thank Bernard Brooks and Milan Hodoščeк for their help about CHARMM, and thank Robert Martino and DCRT for the general support and the use of NIH iPSC/860.

REFERENCES

1. 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).
2. D. Loveman (Ed.), 'Draft High Performance Fortran language specification, version 1.0', Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, (January 1993).
3. B. Chapman, P. Mehrotra, and H. Zima, 'Programming in Vienna Fortran', *Scientific Programming*, **1**, (1), 31–50, (Fall 1992).
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. R. v. Hanxleden, K. Kennedy, and J. Saltz, 'Value-based distributions in fortran d — a preliminary report', Technical Report CRPC-TR93365-S, Center for Research on Parallel Computation, Rice University, (December 1993). submitted to Journal of Programming Languages - Special Issue on Compiling and Run-Time Issues for Distributed Address Space Machines.
6. 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). To appear in IEEE Transactions on Parallel and Distributed Systems.
7. R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley, 'Principles of runtime support for parallel processors', *Proceedings of the 1988 ACM International Conference on Supercomputing*, July 1988, pp. 140–152.
8. D. J. Mavriplis, 'Three dimensional unstructured multigrid for the Euler equations, paper 91-1549cp', *AIAA 10th Computational Fluid Dynamics Conference*, June 1991.
9. 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).
10. P. K. Weiner and P. A. Kollman, 'Amber:assisted model building with energy refinement. a general program for modeling molecules and their interactions', *Journal of Computational Chemistry*, **2**, 287, (1981).
11. W. F. van Gunsteren and H. J. C. Berendsen, 'Gromos: Groningen molecular simulation software.', Technical report, Laboratory of Physical Chemistry, University of Groningen, Nijenborgh, The Netherlands, (1988).
12. P. Venkatkrishnan, J. Saltz, and D. Mavriplis, 'Parallel preconditioned iterative methods for the compressible navier stokes equations', *12th International Conference on Numerical Methods in Fluid Dynamics, Oxford, England*, July 1990.
13. Graeme A. Bird, *Molecular Gas Dynamics and the Direct Simulation of Gas Flows*, Clarendon Press, Oxford, 1994.
14. J. Saltz and et. al., 'A manual for the CHAOS runtime library', Technical report, University of Maryland, (1993).
15. Joel Saltz, Harry Berryman, and Janet Wu, 'Multiprocessors and run-time compilation', *Concurrency: Practice and Experience*, **3**, (6), 573–592, (December 1991).
16. 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).
17. Ravi Ponnusamy, Yuan-Shin Hwang, Joel Saltz, Alok Choudhary, and Geoffrey Fox, 'Supporting irregular distributions in FORTRAN 90D/HPF compilers', Technical Report CS-TR-3268 and UMIACS-TR-94-57, University of Maryland, Department of Computer Science and UMIACS, (May 1994). To appear in IEEE Parallel and Distributed Technology, Spring 1995.
18. Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang, 'Communication optimizations for irregular scientific computations on distributed memory architectures', *Journal of Parallel and Distributed Computing*, **22**, (3), 462–479, (September 1994). Also available as University of Maryland Technical Report CS-TR-3163 and UMIACS-TR-93-109.

19. 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).
20. B. Nour-Omid, A. Raefsky, and G. Lyzenga, 'Solving finite element equations on concurrent computers', *Proc. of Symposium on Parallel Computations and theis Impact on Mechanics*, Boston, December 1987.
21. B. R. Brooks and M. Hodoscek, 'Parallelization of charmm for mimd machines', *Chemical Design Automation News*, **7**, 16, (1992).
22. D. F. G. Rault and M. S. Woronowicz, 'Spacecraft contamination investigation by direct simulation Monte Carlo - contamination on UARS/HALOE', *Proceedings AIAA 31th Aerospace Sciences Meeting and Exhibit, Reno, Nevada*, January 1993.
23. Shahid H. Bokhari, 'Partitioning problems in parallel, pipelined, and distributed computing', *IEEE Transactions on Computers*, **37**, (1), 48–57, (January 1988).
24. 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).
25. F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang, 'Distributed pC++: Basic ideas for an object parallel languages', Report, Indiana University, (Jan 1993).
26. H. Simon, 'Partitioning of unstructured mesh problems for parallel processing', *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Pergamon Press, 1991.
27. R. D. Williams and R. Glowinski, 'Distributed irregular finite elements', Technical Report C3P 715, Caltech Concurrent Computation Program, (February 1989).
28. R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy, 'The design and implementation of a parallel unstructured Euler solver using software primitives', *AIAA Journal*, **32**, (3), 489–496, (March 1994).
29. R. Williams, 'Performance of dynamic load balancing algorithms for unstructured mesh calculations', *Concurrency, Practice and Experience*, **3**, (5), 457–482, (February 1991).
30. N. Mansour, 'Physical optimization algorithms for mapping data to distributed-memory multiprocessors', Technical report, Ph.D. Dissertation, School of Computer Science,Syracuse University, (1992).
31. W. E. Leland, 'Load-balancing heuristics and process behavior', *Proceedings of Performance 86 and ACM SIGMETRICS 86*, 1986, pp. 54–69.
32. R. v. Hanxleden and L. R. Scott, 'Load balancing on message passing architectures', *Journal of Parallel and Distributed Computing*, **13**, 312–324, (1991).
33. V. Venkatakrishnan, H. D. Simon, and T. J. Barth, 'A MIMD implementation of a parallel Euler solver for unstructured grids, submitted to Journal of Supercomputing', Report RNR-91-024, NAS Systems Division, NASA Ames Research Center, (Sept 1991).
34. A. Vidwans, Y. Kallinderis, and V. Venkatakrishnan, 'A new parallel dynamic load balancing algorithm for 3d adaptive unstructured grids', *Proceedings of the 11th AIAA CFD Conference*, Orlando FL, July 1993.
35. 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).
36. M. Lam and M. C. Rinard, 'Coarse grain parallel programming in Jade', *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Williamsburg VA*. ACM Press, 1991.
37. 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).
38. Soumen Chakrabarti and Katherine Yelick, 'Implementing an irregular application on a distributed memory multiprocessor', *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, May 1993. ACM SIGPLAN Notices, Vol. 28, No. 7.
39. S. Hiranandani, K. Kennedy, and C. Tseng, 'Compiler support for machine-independent parallel programming in Fortran D', *Compilers and Runtime Software for Scalable Multiprocessors, J. Saltz and P. Mehrotra Editors*, Amsterdam, The Netherlands, To appear 1991. Elsevier.
40. C. Koelbel, P. Mehrotra, and J. Van Rosendale, 'Supporting shared data structures on distributed memory architectures', *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, March 1990, pp. 177–186.
41. L. C. Lu and M.C. Chen, 'Parallelizing loops with indirect array references or pointers', *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.
42. J. Wu, J. Saltz, S. Hiranandani, and H. Berryman, 'Runtime compilation methods for multicomputers', *Proceedings of the 1991 International Conference on Parallel Processing*, volume 2, 1991, pp. 26–30.