Runtime Support for Dynamic Space-Based Applications on Distributed Memory Multiprocessors

Immaneni Ashok

Technical Report # 94-12-03

December 1994

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, University of Washington, 1994.

The research described in this dissertation was supported in part by the National Science Foundation (Grants CCR-8619663, CCR-9123308, and CCR-9200832), the Washington Technology Center, and Digital Equipment Corporation (Systems Research Center and External Research Program).

University of Washington

Abstract

Runtime Support for Dynamic Space-Based Applications on Distributed Memory Multiprocessors

by Immaneni Ashok

Chairperson of Supervisory Committee: Professor John Zahorjan
Department of
Computer Science and Engineering

Dynamic space-based applications are simulations of objects moving through a closed k-dimensional space subject to mutual forces. There are a wide variety of such applications, differing in the kinds of objects and forces being simulated. These applications exhibit strong data locality patterns, but these patterns change during the computation as objects change position in the simulated space.

To achieve good performance when run on distributed memory multiprocessors, two conflicting goals must be addressed: the strong spatial data locality must be exploited, and the computational load must be balanced across the processors. These optimizations can be done statically, by either the programmer or the compiler, or dynamically, by handwritten application code or a runtime system. This dissertation investigates the issues involved in designing a specialized runtime system that provides convenience of programming as well as efficient execution.

First, we address the issues in designing a programming interface for dynamic space-based applications. We propose a new programming model that enables the development of parallel codes that involve very few additional lines of code and very few additional concepts beyond those required to construct a sequential version of the application.

Second, we address the issues in data partitioning and dynamic load balancing. We propose heuristics that can be effectively used to determine a good partitioning scheme based on the application execution characteristics and the machine characteristics. We propose a novel dynamic load balancing scheme that uses a *non-uniform,* *adaptive load discretization* method to estimate the load distribution, a *hierarchical* scheme to balance the load, and a *predictive* method to determine how often to re-balance. We show that the schemes that we propose are very effective in reducing the execution overheads due to communication, load imbalance and load balancing.

Third, we study the issues in adapting to processor reallocations performed by the operating system on a multiprogrammed system. We show that it is advantageous for the runtime system to remap the data, unless the processor allocation changes too often or there is very little time left over to finish the job.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I'd like to thank:

- The Almighty for everything
- My advisor John Zahorjan for his wonderful guidance, enormous patience and understanding
- Ed Lazowska for his support and valuable comments
- My parents for their encouragement to pursue my goal

The research described in this dissertation was supported in part by the National Science Foundation (Grants CCR-8619663, CCR-9123308, and CCR-9200832), the Washington Technology Center, and Digital Equipment Corporation (Systems Research Center and External Research Program).

Part I

Introduction

Chapter 1
DYNAMIC SPACE-BASED APPLICATIONS

Dynamic space-based applications are simulations of objects moving through bounded k-dimensional space. The objects are subject to mutual forces, and change their spatial positions dynamically. There are a wide variety of such applications, differing in the kinds of objects and forces being simulated.

Many important scientific applications in plasma physics, chemistry, materials science and aeronautics, among others, fall into the category of dynamic space-based applications. For example, an electro-magnetic particle-in-cell application simulates the movement of charged particles subject to forces induced by electric and magnetic fields. A molecular dynamics application, on the other hand, simulates molecules subject to inter-atomic forces.

1.1 Characteristics of Dynamic Space-Based Applications

Dynamic space-based applications exhibit the following common characteristics:

1. They simulate the trajectory of objects moving in a bounded k-dimensional space.

2. The simulation proceeds in a series of time steps. In each time step, the spatial positions of all the objects are updated.

3. Each time step consists of one or more data-parallel phases in which the same computation is performed on all the objects.

4. They exhibit strong spatial locality, i.e., computation on an object depends most strongly on the objects that are spatially close to itself [1].

5. Spatial relationships change dynamically, as the objects move around in space.

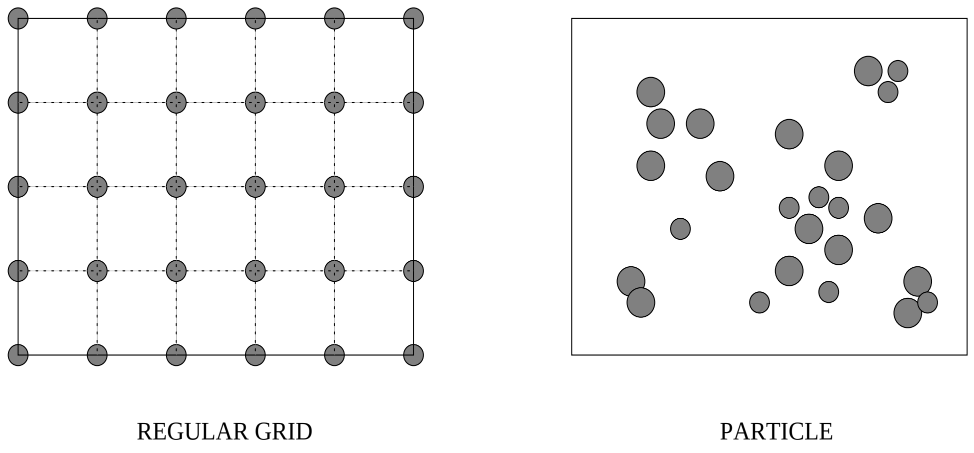


Figure 1.1: Examples of Two-Dimensional Spatial Data Structures

The components of the simulation can be grouped into one or more spatial data structures. Examples of spatial data structures are *regular-grid* and *particle* (Figure 1.1). The objects in a regular-grid (grid points) are uniformly and statically distributed in the space, whereas the objects in a particle data structure are non-uniformly distributed in the space and the distribution can change dynamically.

1.2 Examples of Dynamic Space-Based Applications

1.2.1 Electro-Magnetic Particle-In-Cell

The Electro-Magnetic Particle-In-Cell (EMPIC) application simulates movement of charged particles that interact by exerting electric and magnetic field forces on each other [Birdsall & Langdon 85, Hockney & Eastwood 88, Walker 90]. The force experienced by a particle depends on the current position and velocity of all the particles, and this changes continuously with time. The goal of the simulation is to understand the behavior of the particles.

The standard solution uses a particle-mesh method, which discretizes space by a grid and time by updating the position and velocity of the particles only at the boundaries of small time intervals. Each time step consists of three phases. In the *scatter phase*, current-density is assigned to the grid points using the position and velocity of all the particles (Figure 1.2(a)). In the *solve phase*, new values of electric

[1] In this thesis, we address primarily simulations that use short-range forces.

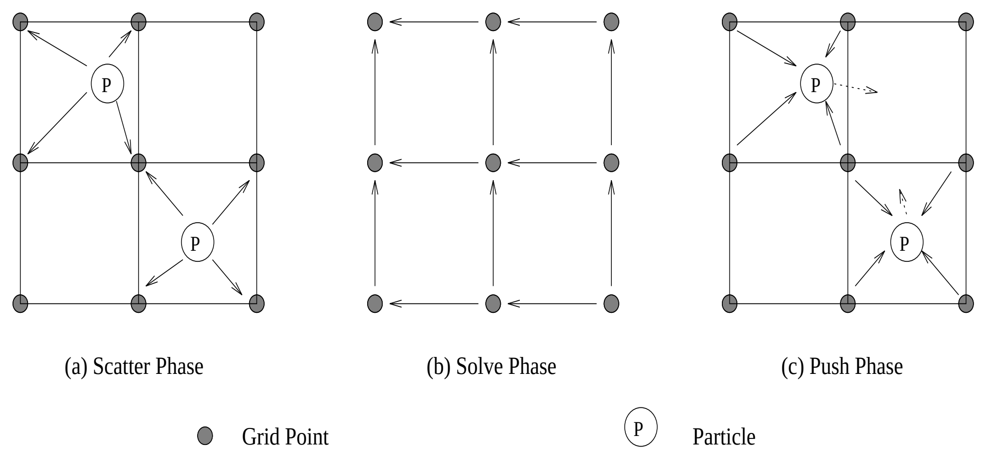


Figure 1.2: Data Dependencies in various phases in one time step of the EMPIC simulation
(Only two dimensions are used for simplicity of illustration. The grid points refer to the Electric Field, Magnetic Field or Current Density data, depending on the context. An arrow between two objects indicates data dependency, and a dotted arrow represents movement of the object.)

and magnetic fields are computed at each grid point, using the old field values and the current density assigned in the scatter phase (Figure 1.2(b)). In the *push phase*, using the field values at the grid points, the force on each particle is computed, and the position and velocity of all the particles are updated (Figure 1.2(c)).

1.2.2 Rarefied Fluid Flow

The rarefied fluid flow application, referred to as MP3D in the SPLASH benchmarks [Singh et al 90], simulates trajectories of gaseous molecules in low density regions. This application is used by aerospace researchers to study the forces exerted on space vehicles as they pass through the upper atmosphere at hypersonic speeds.

In low density conditions, Monte Carlo methods that rely on the discrete particle nature are used [Fallavollita et al. 92, McDonald 89]. Computation is performed on random pairs of molecules. For the purposes of efficient collision pairing, the active space is represented as a three-dimensional *space array* of unit-sized cells. Molecules can move among cells, but are only eligible for collision with other molecules occupying the same cell at that time. Each time step consists of a single phase in

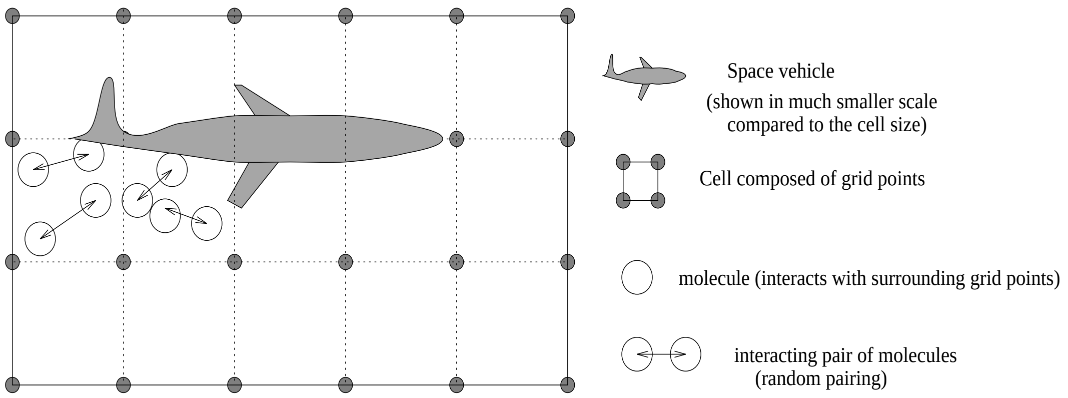

Space vehicle
(shown in much smaller scale compared to the cell size)
Cell composed of grid points
molecule (interacts with surrounding grid points)
interacting pair of molecules (random pairing)

Figure 1.3: Data dependencies in one time step of the MP3D simulation
(Only eight atoms in two dimensional space are used for simplicity of illustration.)

which random pairs of molecules within each cell are collided and the positions and velocities of the molecules are updated.

1.2.3 Molecular Dynamics

The molecular dynamics application can be used to simulate several types of material systems. Examples of simulations are study of metallic interfaces and film deposition (from vapor and by sputtering process).

Here we describe a film deposition simulation that studies the formation of crystals of metals such as platinum and copper [Allen & Tildesley 87]. Atoms are dropped, one by one, onto the existing crystal. When a new atom collides with the stable crystal, the existing atoms are disturbed, settling down after a while. For each new atom, the system is simulated for several time steps, until the crystal comes back to a stable state. Each time step has two phases: In the *potential phase*, $potential_{i,j}$ and $electronDensity_{i,j}$ are computed for each atom pair (i, j) such that distance between i and j is less than some cut-off distance. (Figure 1.4 shows the interactions between the atoms for a small two-dimensional problem.) In the *push phase*, using the values computed in the previous phase, the force on each atom is computed, and the position and velocity of all the particles are updated.

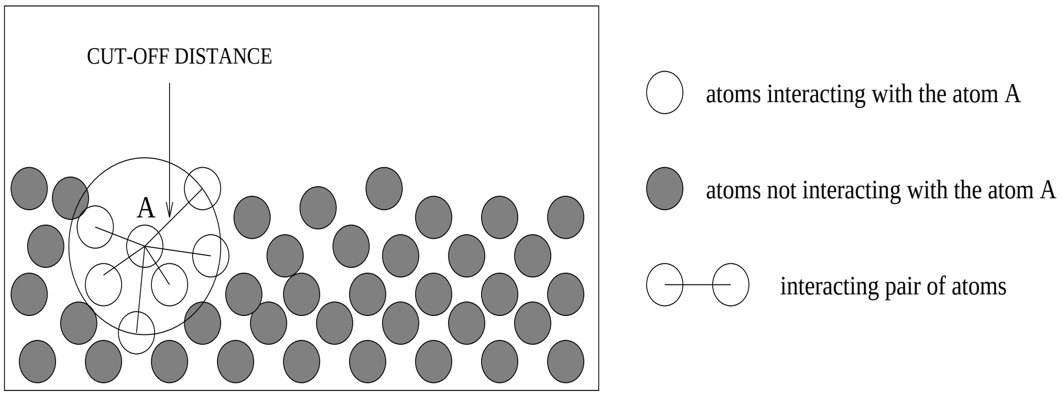

CUT-OFF DISTANCE
A
atoms interacting with the atom A
atoms not interacting with the atom A
interacting pair of atoms

Figure 1.4: Data Dependencies in one time step of the CGV simulation
(The figure shows the dependencies only between the atom A and its neighboring atoms that are within the cutoff distance from A.)

1.3 Parallelizing Dynamic Space-Based Applications on Distributed Memory Multiprocessors

1.3.1 Distributed Memory Multiprocessors

In this work, we address the class of multiple-instruction-multiple-data (MIMD) multiprocessors where the memory is physically distributed across the machine (Figure 1.5). Current examples of such machines are the Intel Paragon, CM-5, nCUBE and transputer based systems. In the future, networks of workstations might be used in this fashion. Each node consists of a general purpose processor and a memory unit. The nodes are connected via high-speed interconnection network. A processor can access the memory situated in another node (termed as the *non-local* memory) either directly through hardware (as in the KSR-1 and DASH [Singh et al. 93]) or indirectly through messages (as in the Intel Paragon, CM-5 and nCUBE). In either case, the time required to access non-local memory is at least an order of magnitude greater than the time required to access local memory. We focus on the non-shared memory (or message passing) machines in this work because it is harder to develop efficient parallel programs for message passing machines and since the techniques used for achieving efficiency on non-shared memory machines apply well for shared memory machines [Lin & Snyder 90, Ngo & Snyder 92].

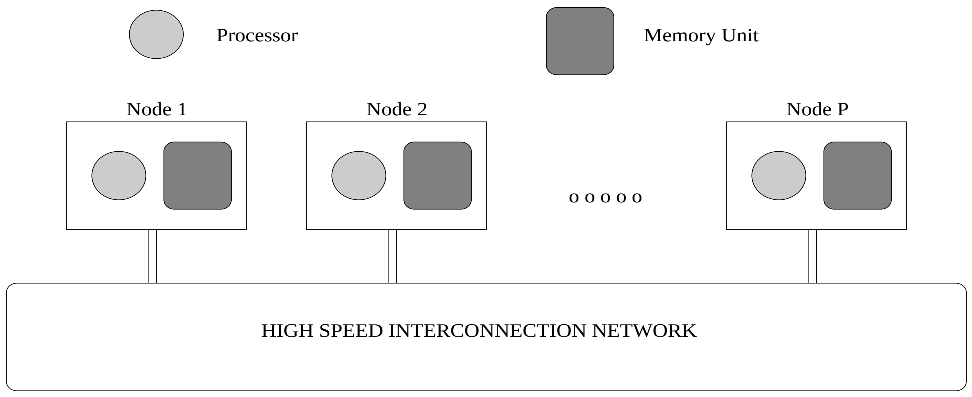


Figure 1.5: Distributed Memory Machine Model

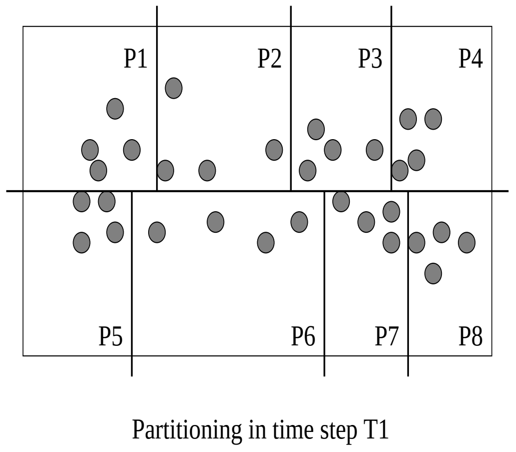
1.3.2 Programming Distributed Memory Multiprocessors

It is hard and time consuming to develop efficient parallel programs for distributed memory multiprocessors for two reasons: it is hard to write correct parallel programs and it is hard to tune them for efficiency. Writing parallel programs is difficult because the programmers need to think about several concepts not necessary for expressing the computation, such as communication, synchronization and load distribution.

To achieve good efficiency, the overheads due to communication and load imbalance need to be minimized. Since the data is distributed across the machine, the processors need to communicate to share the data, and this introduces communication overhead. If the computational load is not well balanced across the processors, some processors waste time while waiting for the other processors to complete their work. This situation introduces load imbalance overhead. It is hard to tune for efficiency because communication and load imbalance overheads are antithetical, that is, minimizing one of them in isolation typically increases the other. These two overheads must be minimized simultaneously, and it is hard to develop code to perform the necessary optimizations.

1.3.3 Parallelizing Dynamic Space-Based Applications

Dynamic space-based applications exhibit strong spatial locality because communication is spatially local. This characteristic can be used effectively to minimize communication overhead. Spatial locality is exploited by partitioning the space into contiguous regions and assigning one region per processor. Each processor is respon-



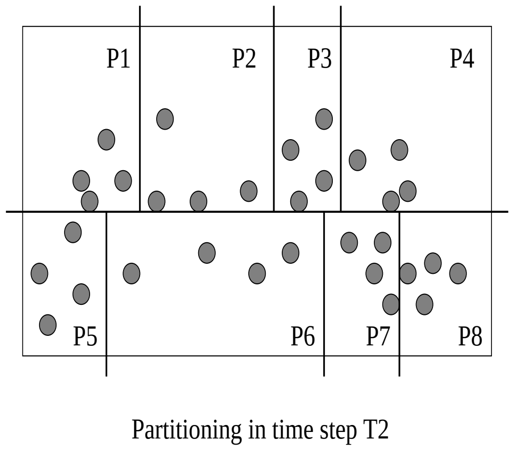


Figure 1.6: A Spatial Partitioning Scheme for a Two-Dimensional Application

sible for managing the values of those objects located in its region. The overhead due to load imbalance is minimized by partitioning the space in such a way that the regions have (nearly) equal computational load. Since the objects change their locations in space dynamically, to maintain load balance the space needs to be repartitioned dynamically.

Figure 1.6 shows a partitioning scheme for reducing the total communication and load imbalance overhead for a sample application. This figure shows the distribution of particles in a two-dimensional space at two different time steps of the simulation. A spatial partitioning scheme brings down communication overhead by reducing the number of pairs of processors that need to communicate. For example, in Figure 1.6, processor P2 needs to communicate only with processors P1, P3 and P6 which own regions adjacent to the region owned by P2.

1.4 Specialized Runtime Support

A programmer can parallelize dynamic space-based applications in three ways: (1) by using a parallelizing compiler, (2) by programming using message passing primitives, or (3) by employing a specialized runtime system. Here we compare these approaches based on the following features that a programming environment must support for simplifying the process of developing efficient parallel code for dynamic space-based applications:

1. *Mechanisms for expressing and operating on space-based data objects*: Examples of such mechanisms are creating a particle data structure and iterating over pairs of closely located particles. Without these mechanisms, the user needs to develop complicated code for maintaining distributed spatial data structures. If the system does not support dynamic space-based data, such as particles, the burden of determining the granularity of the space partitioning and load balancing falls on the user.

2. *Mechanisms for sharing data along partition boundaries*: These mechanisms relieve the user from managing communication, and increase portability.

3. *Automatic data partitioning*: The performance impact of a partitioning scheme depends not only on the application characteristics (such as communication patterns and load distribution and movement), but also on the number of processors and machine architecture (in particular, the speed of inter-processor communication). It is difficult for a programmer to choose a good partitioning scheme without using the knowledge about application and machine characteristics. To achieve high performance and portability, the system must support automatic data partitioning.

4. *Automatic dynamic load balancing*: The optimal frequency of load balancing depends on the overheads due to load imbalance and load balancing. The load balancing cost depends on the algorithm as well as the machine architecture. Clearly, implementing this feature in each application would be a significant burden on application developers.

Message passing primitives are the "assembly language" approach to parallel programming: best efficiency is possible, but only with substantial effort. The programmer needs to spend enormous time developing code that manages spatial data structures, data partitioning, communication and dynamic load balancing.

High-level languages, such as HPF, try to provide a more convenient environment for the general class of applications, but at the cost of reduced efficiency. This type of environment reduces program development time, but may not offer good performance, since not all characteristics of the application can be exploited by a general purpose compiler. There is no convenient way of expressing dynamic space-based data objects and their spatial relationships. The user is responsible for deciding on the partitioning scheme, and when and how to perform dynamic load balancing.

More specialized support can offer even more convenient environments without loss of efficiency, but at the cost of reduced applicability. A specialized system can provide support for managing spatial data structures and take care of data partitioning, communication and dynamic load balancing without requiring much effort from the programmer. A specialized programming environment can offer convenience of programming as well as good performance.

Chapter 2

THESIS

In this chapter, we list the contributions of the thesis and describe the organization of this report.

2.1 Thesis Contributions

The thesis addresses the following fundamental issues for parallelizing dynamic space-based applications on distributed memory multiprocessors:

- How to provide support for efficient programming?

- How to provide support for efficient execution?

The thesis makes the following contributions:

Support for Efficient Programming

- *Design of a parallel programming model*: We propose a new programming model, which we call *Ādhāra*, to conveniently express dynamic space-based computations [Ashok & Zahorjan 94]. The model requires the programmer to specify information using only concepts that are natural to the application. The programmer need not be concerned about the artifacts of a parallel execution, e.g., partitioning, communication and load balancing, which are automatically managed by the associated runtime system.

- *Evaluation of the model using real applications*: We evaluate the effectiveness of the *Ādhāra* programming model by using three real applications from different scientific fields: physics, aeronautics and materials science. We evaluate the model by converting the existing sequential programs into parallel *Ādhāra* programs and measuring the programming overhead (in terms of the extra lines of code) introduced by the conversion process.

Support for Efficient Execution

- *Design of an automatic partitioning scheme*: We propose a new scheme that determines a good method of partitioning a three-dimensional space into rectangular regions, based on the following information: execution specific characteristics such as communication patterns and the movement and distribution of spatial objects; number of processors allocated to the application; and machine specific characteristics such as the speed of inter-processor communication.

- *Design of an efficient, scalable dynamic load balancing scheme*: We propose a novel scheme that estimates the load distribution by a non-uniform, adaptive discretization of the problem space. Compared to the traditional schemes that uniformly and statically discretize the space, our scheme is faster, uses less memory, and scales better. We also propose a predictive scheme that adjusts the frequency of load balancing so as to minimize the sum of load balancing and load imbalance overheads.

- *Implementation of the portable runtime system*: We developed an efficient implementation of the prototype *Ādhāra* runtime system. The system is portable across a variety of distributed memory machines, such as the Intel Paragon, CM-5, and KSR.

- *Performance evaluation using real applications*: We evaluated the novel partitioning and load balancing schemes, and the overall performance of the runtime system, using three real applications: electro-magnetic particle-in-cell (plasma physics), rarefied fluid flow (aeronautics) and molecular dynamics (materials science).

- *Study of issues in adapting to dynamic processor reallocations*: We studied the issues in adapting to processor reallocations performed by the kernel scheduler in a multi-programmed environment.

2.2 Organization of the Thesis

The thesis is organized in five parts. The first part (chapters 1-3) describes the class of dynamic space-based applications, contributions of the thesis and related work.

The second part (chapters 4-6) focuses on the programming support for dynamic space-based applications. Chapter 4 describes the *Ādhāra* programming model. Chapter 5 gives programming examples, and Chapter 6 evaluates the effectiveness of the programming model.

The third part (chapters 7-12) deals with aspects of application induced load balancing and implementation of the runtime system. The issues in space-based partitioning and load balancing are discussed in Chapter 7. Chapters 8, 9 and 10 describe the novel schemes for automatic data partitioning and dynamic load balancing. Chapter 11 describes the implementation of the prototype runtime system. Performance results are discussed in Chapter 12.

The fourth part (chapter 13) focuses on the issues in adapting to dynamic processor reallocations performed by the operating system scheduler on a multi-programmed machine. The fifth part (chapter 14) summarizes and concludes the thesis.

Chapter 3
RELATED WORK

In this chapter we provide an overview of other research work related to our thesis. Details of the relevant work and comparison with our work are given in Chapters 6 and 9.

3.1 Parallel Programming Environments

Research work on programming environments falls into three categories: parallelizing compilers for general-purpose high-level languages, runtime systems specialized for specific classes of applications, and parallel programs for specific applications.

3.1.1 Parallelizing Compilers

A substantial amount of research has been done on automatic parallelizing compilers for distributed memory machines. Fortran-D [Hiranandani et al. 91], Vienna Fortran [Chapman et al. 93a] and HPF [Bozkus et al. 94, Chapman et al. 93b] extend the Fortran language with annotations for controlling data partitioning. Various techniques for extracting parallelism from the sequential source code and for minimizing execution overheads are discussed in the literature [Bozkus et al. 94, Gupta & Banerjee 93, Hiranandani et al. 94, Rogers & Pingali 94, Saltz et al. 91].

3.1.2 Specialized Runtime Systems

The Phase Abstractions model [Snyder 89, Griswold et al. 90] provides scalable abstractions for decomposing parallel computations into phases of different data access, computation and communication characteristics. The ZPL language [Lin & Snyder 93] is specialized for array based computations.

DINO [Rosing et al 91] is a language for expressing data-parallel numerical computations, and its extension DYNO [Weaver & Schnabel 92] is specialized for unstructured applications.

The LPAR programming environment [Baden & Kohn 91, Baden & Kohn 94, Kohn & Baden 93] supports dynamic non-uniform scientific applications. The PARTI primitives [Berryman et al 91, Agrawal et al. 93] provide low-level support for block structured and irregular mesh applications. The DIME system [Williams 91a] and the Voxel Database system [Williams 92] provide high-level support for irregular mesh applications.

3.1.3 Parallel Scientific Applications

Several researchers studied issues in parallelizing specific scientific applications in different fields of science and engineering. Here we mention the work done on dynamic space-based applications: particle-in-cell (plasma physics) [Campbell et al. 90, Ferraro et al. 93, Liewer & Decyk 89, Walker 90], rarefied fluid flow (aeronautics) [Fallavollita et al. 92, McDonald 89, Singh et al 90], and molecular dynamics (materials science and chemistry) [Brugè & Fornili 90, Fincham 87, Pinches et al. 91, Raine et al 89, Rapaport 91, Smith 91].

3.2 Application Induced Load Balancing

3.2.1 Automatic Data Partitioning

Much research has been done on parallel schemes for partitioning spatial data. One part of this work deals with the schemes for partitioning the space into rectangular regions [Belkhale & Banerjee 90, Berger & Bokhari 87, Bokhari et al. 93, Cybenko & Allen, Nicol 91], while the other looks at non-rectangular regions [Hinz 90, Pilkington & Baden 94, Reed et al. 87, Weaver & Schnabel 92, Williams 92].

Specialized runtime systems such as the DIME [Williams 91a], Voxel Database [Williams 92] and DYNO [Weaver & Schnabel 92] perform automatic data partitioning based on the characteristics of the specific classes of applications being targeted. Programming environments such as the LPAR [Baden & Kohn 94] leave the task of data partitioning to the programmer (the programmer can choose from the generic schemes available in the application libraries).

Parallelizing compilers for languages such as Fortran-D [Hiranandani et al. 91], Vienna Fortran [Chapman et al. 93a] and HPF [Bozkus et al. 94] make use of the annotations provided by the programmer to partition the data. Recently, researchers started looking at automatic data partitioning techniques for parallelizing compilers [Gupta & Banerjee 92].

3.2.2 Dynamic Load Balancing

Researchers have analyzed the performance of different load balancing techniques for specific classes of applications, such as particle-based simulations [Baden & Kohn 91, Campbell et al. 90, Hanxleden & Scott 91, Hinz 90], molecular dynamics [Brugè & Fornili 90] and unstructured mesh computations [Weaver & Schnabel 92, Williams 91b].

3.3 Operating System Induced Load Balancing

Researchers have studied the issues in scheduling on multiprogrammed shared-memory machines, such as process control [Tucker & Gupta 89], dynamic process allocation policies [McCann et al. 92], and adapting to processor reallocations [Anderson et al. 92].

On distributed memory multiprocessors, researchers have studied the issues in multiprogramming [Leuze et al. 89, Park & Dowdy 89], process scheduling [Setia et al. 93] and dynamic reallocation policies [McCann & Zahorjan 93, McCann 94].

3.4 Summary

In the first part (chapters 1-3) of this report, we discussed the need for specialized programming support for dynamic space-based applications, the contributions of this thesis, and an overview of relevant research work. In the next part (chapters 4-6), we describe *Ādhāra*, a specialized programming environment that we propose for developing parallel dynamic space-based applications.

Part II

Parallel Programming Model for Dynamic Space-Based Applications

Chapter 4

THE ĀDHĀRA PROGRAMMING MODEL

We propose a new programming model, which we call *Ādhāra*, to conveniently express dynamic space-based computations. In this chapter, we describe the design of *Ādhāra*.

In order to program with *Ādhāra*, the user needs have an understanding of the following concepts:

- Data Parallelism
- Data partitioning on distributed memory machines
- Data dependencies, local and non-local data, and need for sharing data

These are the basic concepts that are necessary to understand how a data-parallel program is parallelized on a distributed memory machine. Fortunately, these concepts are very easily understood in the context of dynamic space-based applications.

Ādhāra supports all the four features that are required to simplify the process of developing efficient parallel code for dynamic space-based applications (details can found in Section 1.4):

1. Mechanisms for expressing and operating on space-based data objects
2. Mechanisms for sharing data along partition boundaries
3. Automatic data partitioning
4. Automatic dynamic load balancing

Ādhāra presents a non-shared memory, data-parallel (SPMD - Single Program Multiple Data) model. The same program executes on all the processors, but each processor operates on different data. *Ādhāra* extends the C-language with primitives

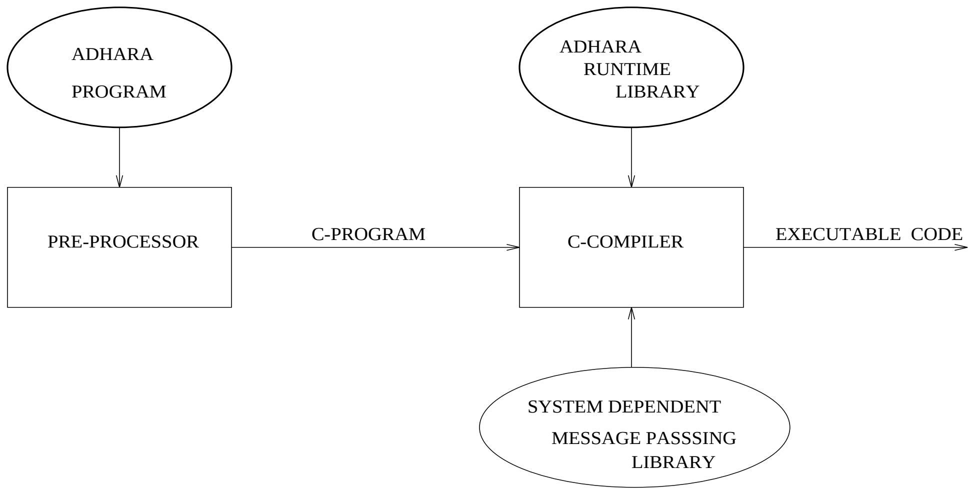


Figure 4.1: Compiling an *Ādhāra* Program

for expressing space-based computations. An *Ādhāra* program is converted into a C-program by a pre-processor. The resulting C-code is compiled and linked with the *Ādhāra* runtime library and the system dependent library that supports message passing primitives (Figure 4.1). Implementation of the *Ādhāra* runtime library is discussed in Chapter 11.

Ādhāra enables the programmer to specify space-based computations using four concepts that are natural to dynamic space-based applications:

1. *Computation space*: This corresponds to the bounded physical space that is modelled by the application. *Ādhāra* uses this space as a basis to exploit spatial locality and to balance the load.

2. *Spatial data structures*: *Ādhāra* provides mechanisms to declare and operate on distributed space-based data structures such as grids and particles.

3. *Data sharing*: The user can specify spatial data dependencies in a natural way. *Ādhāra* uses this information to maintain consistency of the distributed data.

4. *Phases*: *Ādhāra* enables the user to decompose the parallel computation into *phases* exhibiting different data access and communication characteristics. The phases are used as a basis to balance the load dynamically.

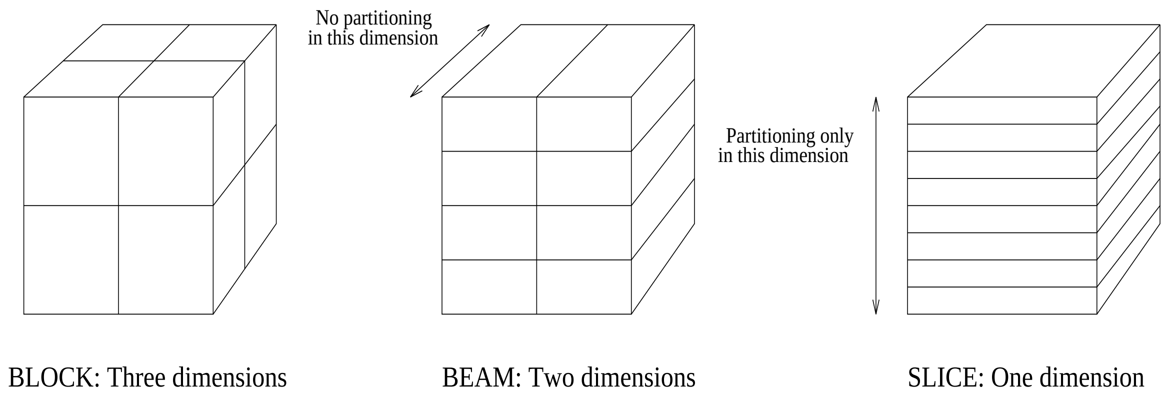

No partitioning in this dimension
Partitioning only in this dimension
BLOCK: Three dimensions
BEAM: Two dimensions
SLICE: One dimension

Figure 4.2: Methods of Partitioning a Three Dimensional Space into Eight Rectangular Regions

These concepts are described in detail in the following sections. For simplicity of illustration, we assume a three-dimensional space, but the ideas can be extended to k dimensions.

4.1 Computation Space

The *computation space* refers to the bounded physical space that is modelled by the application. The statement

```
        COMPUTATION-SPACE ( 2.0, 1.5, 1.0 )
```

creates a bounded three dimensional space of size $2.0 \times 1.5 \times 1.0$ units. Each spatial object must lie within this space. To exploit spatial locality, *Ādhāra* partitions the data using domain decomposition [Campbell et al. 90, Walker 90], where the *computation space* is partitioned into contiguous regions and each processor is assigned one region. Each region is a rectangular block and forms the basis for exploiting spatial locality and for balancing the load. (The motivation for choosing domain decomposition into rectangular blocks is discussed in Section 7.1.2.) Load balance is maintained by balancing the computational load across the regions. Each node *owns* all the data elements that map into its regions, and is responsible for maintaining consistent values of the data that it owns. (Note: Properties of a data element are not necessarily computed exclusively by its owner.)

There are many ways of partitioning a k-dimensional space, depending on how many of the dimensions are partitioned. For example, a three dimensional space can be partitioned in seven different ways (Figure 4.2): partitioning all the three dimensions (one choice: BLOCK), two dimensions (three choices: BEAM-X, BEAM-Y and BEAM-Z) and one dimension (three choices: SLICE-X, SLICE-Y and SLICE-Z). The programmer can optionally specify how to partition the space:

```
PARTITIONING-SCHEME BEAM-Z /* optional */
```

If the user does not specify a partitioning scheme, *Ādhāra* uses heuristics to determine a good choice. The heuristics are based on the application specific characteristics such as the communication patterns and movement and density of the spatial objects, number of processors allocated to the application, and machine specific characteristics such as the speed of the inter-connection network. These heuristics are described in Chapter 8.

4.2 Spatial Data Structures

Ādhāra distinguishes between two types of data structures: *particle* and *regular-grid* (Figure 4.3). The data objects of a particle data structure can lie anywhere in the computation space, and can change their positions dynamically. A *regular-grid* is a restricted form of a particle data structure that is regular and static. It is perfectly aligned with the computation space. The mapping of a *regular-grid* onto the *computation space* is implicitly specified by giving the dimensions of the grid, whereas the mapping of a particle data structure is explicitly specified by giving the coordinate of each object of the data structure, and is allowed to change dynamically.

Ādhāra supports the most common operation on the spatial data structures, which is to iterate over the sets of their objects. A *regular-grid* is declared and accessed as follows:

```
typedef struct vectornode { double X, Y, Z; } vectorType;
REGULAR-GRID vectorType MagneticField ( GridSizeX, GridSizeY, GridSizeZ );

FORALL-GRIDPOINTS (I,J,K) IN MagneticField DO {
    MagneticField[I,J,K] = function_of (
            ElectricField[I,J,K], ElectricField[I,J,K+1], .. )
}
```

When a regular-grid is declared, each node allocates enough memory to store its portion of the distributed grid. The "FORALL-GRIDPOINTS" loop accesses only those

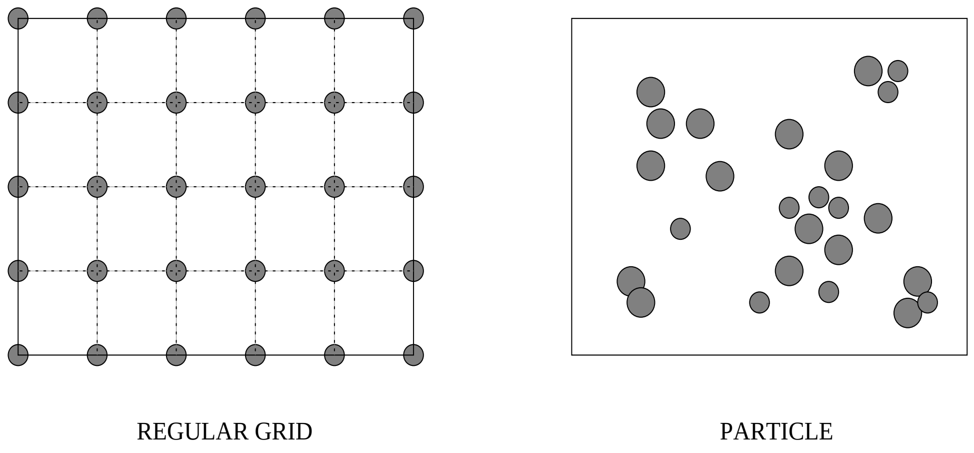


Figure 4.3: Examples of Two-Dimensional Spatial Data Structures

grid points that are within the region owned by the processor. If the computation needs to iterate over the grid points that are located just outside the boundary of the region, the "INCLUDING" primitive can be used:

```
FORALL-GRIDPOINTS (I,J,K) IN MagneticField INCLUDING
                          BOUNDARY { ALL-DIRS = 1 } DO
    CurrentDensity[I,J,K] = 0;
```

The "BOUNDARY { ... }" specifies the directions of interest. More details about specifying the boundaries are given in the following section.

A *particle* data structure is declared by specifying the data-type (structure) of the particle. The first field of the C-struct must denote the coordinate of the particle. (The runtime system uses this information to maintain spatial relationships.)

```
typedef struct vectornode { double X, Y, Z; } vectorType;
typedef struct atomnode {
    vectorType coordinate;  /* coordinate must be the first field */
    int        atom_type;
    vectorType force;
    vectorType velocity;
    ......
} atomType;

3D-PARTICLE atomType Atoms;
```

Initially the "Atoms" data structure is empty. Atoms are inserted using the ADD-PARTICLE primitive, as shown below.

```
for( i = 0; i < NumAtoms; i++ ) {
    atomType aa;
    aa.coordX = ...; aa.coordY = ..; ...
    ADD-PARTICLE aa TO Atoms;
          /* data is copied into the memory allocated by Adhara */
}
```

This particle is stored in the local memory only if it is located in the region owned by this processor. *Ādhāra* assumes that each particle is inserted by every processor, so the ADD-PARTICLE primitive guarantees that exactly one processor stores this particle in its local memory. If this assumption is not valid (i.e., this particle is inserted only by one processor), then the "ADD-PARTICLE-LOCAL" primitive can be used:

```
.. generate a new atom 'aa' locally ..
ADD-PARTICLE-LOCAL aa TO Atoms;
```

This primitive guarantees that the particle is stored in the local memory of the processor executing this statement, even if the particle is located outside of the region owned by this processor. If the particle is not within the owned region, it is sent to the owner when the "UPDATE-COORDINATES" primitive is executed (see the end of this section).

The user can access the objects in a *particle* data structure either one at a time or in groups of arbitrary size such that all members in each group are within some fixed distance of each other. (Molecular dynamics applications, for example, iterate over pairs of atoms that are within a cut-off distance of each other.)

The particles can be accessed one at a time using a "FORALL-PARTICLES" loop as shown below.

```
FORALL-PARTICLES ( Atom ) IN Atoms DO {
    Atom->coordinate = function_of ( Atom->force, ..., Atom->velocity );
}
```

If the particles interact with surrounding grid points (from a *regular-grid*), the "SURROUNDING" primitive can be used to extract the necessary data:

```
FORALL-PARTICLES ( Particle ) IN ChargedParticles DO {
    field_on_particle = 0;
    FORALL-GRIDPOINTS (I,J,K) IN ElectricField SURROUNDING ( Particle ) DO
        field_on_particle += interpolate_field( ElectricField[I,J,K] );
    Particle->force = function_of( field_on_particle );
}
```

If the particle interacts only with one corner of the cell (formed by the surrounding grid points) in which it is located, the "CELL-INDEX" primitive can used to extract the necessary index:

```
FORALL-PARTICLES ( Particle ) IN Atoms DO {
   (I,J,K) = CELL-INDEX OF ( Particle );
   NumAtomsInTheCell[I,J,K]++;  /* NumAtomsInTheCell is a regular-grid */
}
```

If the computation uses pairs of particles that are within a cut-off distance of each other (particle-particle interaction), the programmer needs to specify additional information while declaring the *particle* data structure. To generate the pairs efficiently, *Ādhāra* partially sorts the particles into three-dimensional cells. The user can specify the size of the cell while declaring the *particle* data structure, as given below[1].

```
3D-PARTICLE atomType Atoms SORT-INTO-CELLS( cutoffX, cutoffY, cutoffZ );

FORALL-PARTICLES ( Atom1, Atom2 ) IN Atoms CELL-DISTANCE 1 DO {
    pair_force = function_of ( Atom1->coordinate, Atom1->type,
                               Atom2->coordinate, Atom2->type, ... );
    update Atom1->force and Atom2->force using pair_force;
}
```

The "SORT-INTO-CELLS" primitive enables the system to keep the particles partially sorted into cells of specified size. Pairs of particles are generated by specifying the cutoff distance in terms of cells. If the user specifies a "CELL-DISTANCE" of N, then *Ādhāra* gives all those pairs of particles that are within N cells of each other. This operation is efficient, since *Ādhāra* does not need to compute distances between

[1] The current implementation supports only pairs, and those that can be formed using the particles within one PARTICLE data structure. The model can be extended to support sets of three or more particles among multiple PARTICLE data structures, and this presents no conceptual difficulty.

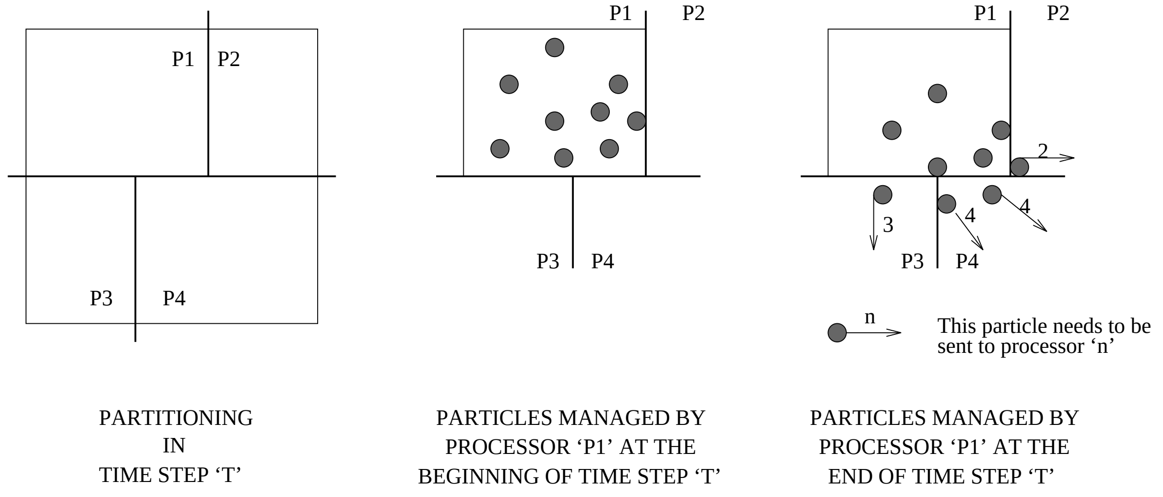

P1
P2
P3
P4
PARTITIONING IN TIME STEP 'T'
P1
P2
P3
P4
PARTICLES MANAGED BY PROCESSOR 'P1' AT THE BEGINNING OF TIME STEP 'T'
P1
P2
P3
P4
2
3
4
4
n
This particle needs to be sent to processor 'n'
PARTICLES MANAGED BY PROCESSOR 'P1' AT THE END OF TIME STEP 'T'

Figure 4.4: Communicating Out-Of-Bound Particles

the particles. The user can screen the pairs depending on the computational requirements. *Ādhāra* guarantees that each pair of particles is processed by exactly one processor (i.e., there is no duplication of pairs), and that no pairs are left unprocessed.

The size of the cells can be changed dynamically using the following statement:

```
SORT-INTO-CELLS( newCutoffX, newCutoffY, newCutoffZ ) Atoms;
```

When the coordinates of the particles are updated, the user must inform the system using the "UPDATE-COORDINATES" primitive.

```
FORALL-PARTICLES ( Atom ) IN Atoms DO
    Atom->coordinate = function_of( Atom->velocity, .., Atom->force )
UPDATE-COORDINATES OF Atoms;
```

When this primitive is executed, *Ādhāra* checks if the particles that are currently managed by this processor are still within the region assinged to this processor. If any particles have moved out of the assigned region, they are sent to appropriate processors. This operation is done for maintaining spatial locality (Figure 4.4)

4.3 Data Sharing

In dynamic space-based applications, since communication is spatially local, processors need to share data only along partition boundaries. Here we describe how

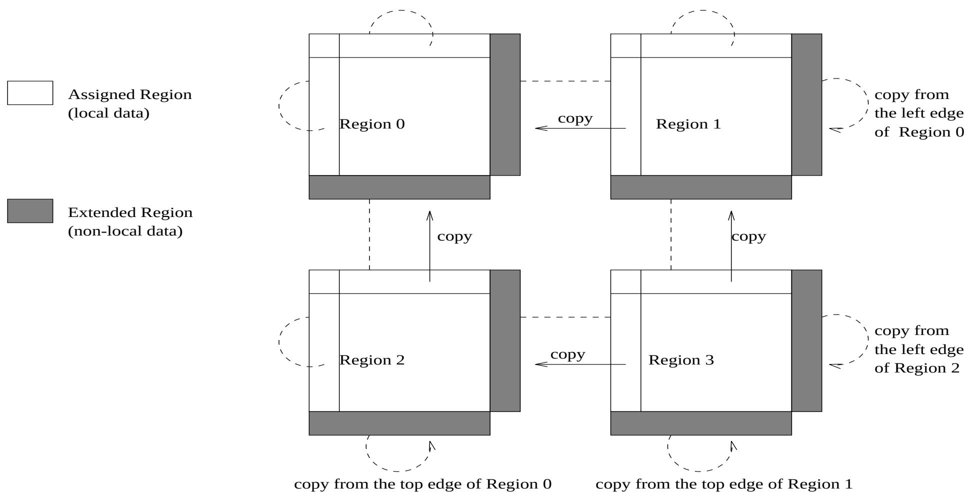

Assigned Region
(local data)
Extended Region
(non-local data)
Region 0
Region 1
Region 2
Region 3
copy
copy
copy
copy
copy from the left edge of Region 0
copy from the left edge of Region 2
copy from the top edge of Region 0
copy from the top edge of Region 1

Figure 4.5: Example of Read Overlap
(The space is partitioned into four regions. The figure gives the read overlap for all the regions. Only two dimensions are used for simplicity of illustration.)

spatial dependencies can be expressed in a natural way. *Ādhāra* uses this information to prepare efficient communication schedules for data sharing.

A processor can access that portion of the data that it *owns* (local data), and some overlapped data owned by the other nodes (non-local data).

4.3.1 Sharing Regular-Grid Data

The sharing of grid data along the boundaries of the regions is described by specifying how each region must be overlapped with the neighboring regions. The user can specify, for each direction, the overlap distance in terms of the number of grid cells, and the boundary condition:

```
READ-BOUNDARY ElectricField OVERLAP(
                              { X-DIR = (0,1),Y-DIR = (0,1),Z-DIR = (0,1) },
                              PERIODIC-BOUNDARY );
```

In the READ-BOUNDARY statement, "X-DIR = (a,b)" means that there is an overlap of 'a' cells in the negative X direction and an overlap of 'b' cells in the positive X direction. The statement given above specifies that the partition of the *regular-grid* ElectricField must be overlapped by one cell along the positive X, Y, and

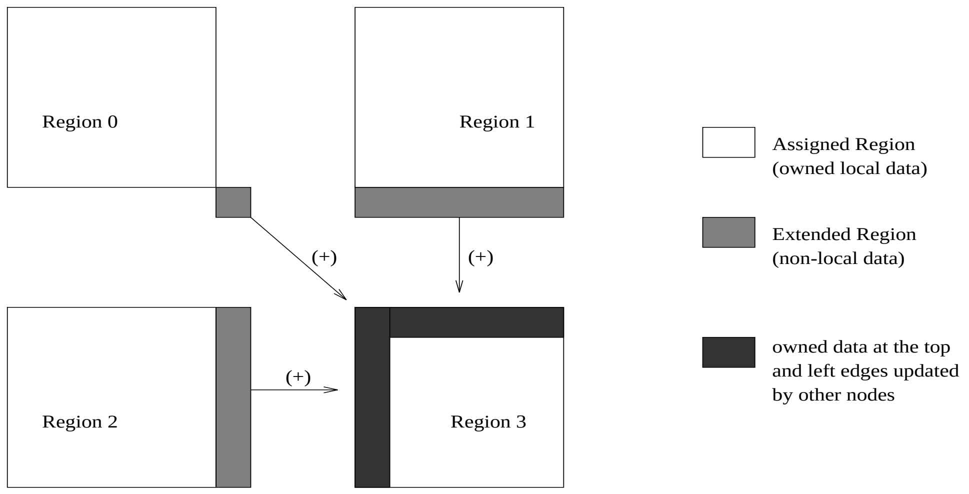

Region 0
Region 1
Region 2
Region 3
(+)
(+)
(+)
Assigned Region (owned local data)
Extended Region (non-local data)
owned data at the top and left edges updated by other nodes

Figure 4.6: Example of Write Overlap
(The figure gives the write overlap for the data assigned to Region-3 only. Only two dimensions are used for simplicity of illustration.)

Z directions, and that the data must be *read* into the extended region using a *periodic* boundary condition [2] (Figure 4.5).

In some cases, a processor may need to update *non-local* data. For example, in the *scatter phase* of the EMPIC application (Section 1.2.1), for each particle *p*, current density is assigned to the grid points enclosing *p*. If the particle lies near the boundary of its region, then some of the *non-local* values of current density may need to be updated. At the end of the phase, the updated *non-local* data must be sent to its *owner*. This task is accomplished by the following statement:

```
WRITE-BOUNDARY CurrentDensity OVERLAP( { ALL-DIRS=1 }, PERIODIC-BOUNDARY )
                                            OPERATION ( ADD-DOUBLETYPE );
```

More than one node can update the current density at the same grid point. The OPERATION specifies the commutative-associative operation that must be used to combine the values of *CurrentDensity* at the same grid point that are updated by more than one node (Figure 4.6).

[2] A *periodic boundary condition* is a tool that is implemented by people who do computer simulations to allow one to mimic an infinitely sized system with a "smaller" image. The configuration of the particles in *computation space* represents the central image and then one pictures that central image as being surrounded on all sides by similar images. For example, in two dimensions, the central image is surrounded by eight images - one on each axis and one at each corner.

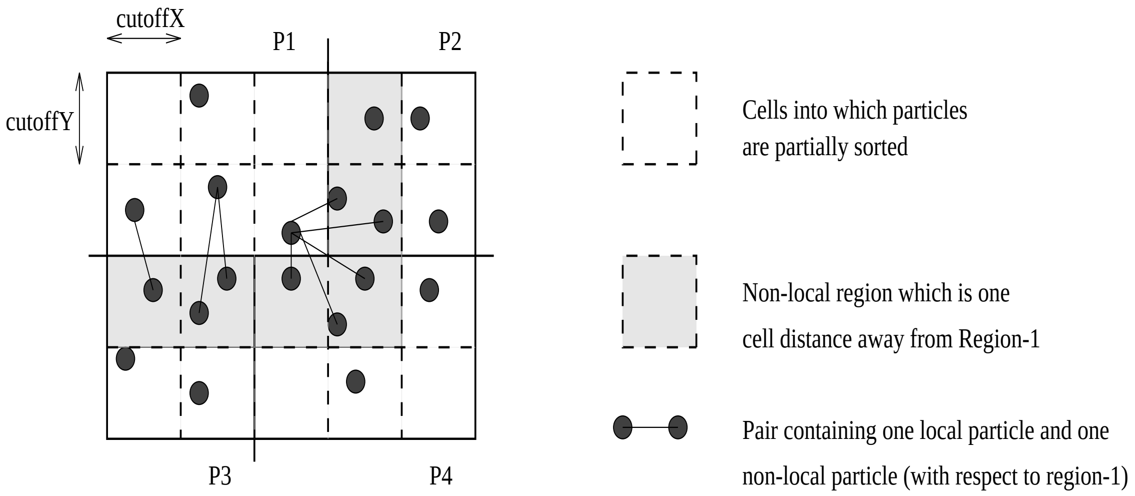

cutoffX
cutoffY
P1
P2
P3
P4
Cells into which particles are partially sorted
Non-local region which is one cell distance away from Region-1
Pair containing one local particle and one non-local particle (with respect to region-1)

Figure 4.7: Replication of Particle Data for Extracting Pairs of Particles
(The space is partitioned into four regions. Only two dimensions are used for simplicity of illustration. The figure gives the read overlap for Region-1 only. The particles in the shared region are copied to Node-1.)

4.3.2 Sharing Particle Data

The *Ādhāra* runtime system will automatically replicate particles along the boundaries of the regions when the "FORALL-PARTICLES" loop is executed for extracting pairs of particles:

```
FORALL-PARTICLES ( Atom1, Atom2 ) IN Atoms CELL-DISTANCE 1 DO { ... }
```

Ādhāra copies the entire structure when replicating the particle data. If only a small portion of the structure is needed for the pair computation, the user can specify which portion of the structure needs to be copied:

```
typedef struct atomnode {
    vectorType coordinate;
    int        atom_type;
    vectorType force;
    vectorType velocity;
    ......
} atomType;

3D-PARTICLE atomType Atoms;
........
```

```
READ-BOUNDARY Atoms USING-FIELDS FROM coordinate TO atom_type;
FORALL-PARTICLES ( Atom1, Atom2 ) IN Atoms CELL-DISTANCE 1 DO { ... }
```

If the computation updates *non-local* particle data, the updated data must be sent to its *owner* using the following statement:

```
WRITE-BOUNDARY Atoms USING-FIELDS FROM force TO velocity
                                         OPERATION ( ADD-DOUBLETYPE );
```

The fields specify which portion of the data was updated, and the OPERATION specifies the commutative-associative operation that must be used to combine the values of the same particle that are updated by more than one node.

4.3.3 Global Operations

Ādhāra provides primitives for combining data from different processors. The system supports standard commutative, associative global operations such as *add*, *min* and *max* for different data-types. Examples of global operations are given below.

```
GLOBAL-SUM num_exiting_atoms DATATYPE-INT;
GLOBAL-MAX energy DATATYPE-DOUBLE;
```

4.4 Phases

The computation is divided into *phases*, where each phase computes on a particular data structure called its *primary* data structure. (It often corresponds to the computation in a *do-loop* of a sequential algorithm.) This *phase*, in concept, is similar to the phase defined in the Phase Abstractions Model [Griswold et al. 90]. The compute load in a *phase* is assumed to be proportional to the sum of the computation (measured in terms of iterations) on the objects in the corresponding primary data structure. This information is used by *Ādhāra* to balance the load in each phase. Phases, which are natural to the application, are specified by the user. For example, the phases in the electro-magnetic particle-in-cell application (Section 1.2.1) can be declared as follows:

```
PHASE ScatterPhase {
        ( ChargedParticle, PRIMARY, USAGE READ-ONLY )
        ( CurrentDensity, USAGE WRITE-ONLY )
```

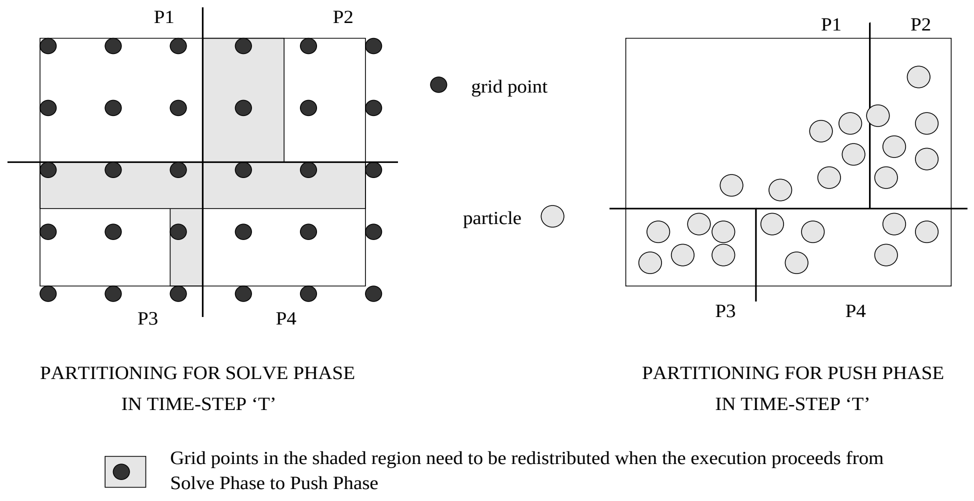

P1
P2
P3
P4
grid point
particle
PARTITIONING FOR SOLVE PHASE IN TIME-STEP 'T'
PARTITIONING FOR PUSH PHASE IN TIME-STEP 'T'
Grid points in the shaded region need to be redistributed when the execution proceeds from Solve Phase to Push Phase

Figure 4.8: Automatic Data Redistribution in between Phases

```
}
PHASE Solvephase {
        ( MagneticField, USAGE READ-WRITE )
        ( CurrentDensity, USAGE READ-ONLY )
}
PHASE PushPhase {
        ( ChargedParticle, PRIMARY, USAGE READ-WRITE )
        ( ElectricField, USAGE READ-ONLY )
        ( MagneticField, USAGE READ-ONLY )
}
```

In the *Solvephase*, ElectricField is the *primary* data structure, so the computation is proportional to the number of grid points. The data structures MagneticField and CurrentDensity are also accessed in the *Solvephase*. The USAGE specifies how the data is used in this computation, and enables *Ādhāra* to determine whether to redistribute the data or not when the execution proceeds from one phase to another. The computation is performed by executing a procedure within a phase:

```
void ScatterRoutine() { ..... }
void SolveRoutine() { ..... }
void PushRoutine() { ..... }
main() {
    for T time-steps do:
        EXECUTE ScatterRoutine IN ScatterPhase;
        EXECUTE SolveRoutine IN Solvephase;
```

```
            EXECUTE PushRoutine IN PushPhase;
    }
```

In the *Solvephase*, the space is partitioned into equal sized regions, since the computation is proportional to the number of elements in ElectricField, a *regular-grid*. In the *PushPhase*, the computation is proportional to the number of particles, which are non-uniformly distributed in space, so the space is partitioned in such a way that all regions contain approximately equal numbers of particles. Since the distribution of particles in space changes dynamically, the space partitioning in *PushPhase* must also change dynamically. In the above program fragment, in between the computation of *SolveRoutine* and *PushRoutine*, the data elements of ElectricField and MagneticField are automatically redistributed, if the space partitioning in *PushPhase* is different from that in *Solvephase* (Figure 4.8).

4.5 Summary

In this chapter, we described the design of the *Ādhāra* programming model. In order to use *Ādhāra*, the programmer needs to know basic concepts about parallelizing data-parallel programs on distributed memory machines. In the following chapter, we describe how to parallelize applications using *Ādhāra*.

Chapter 5

PARALLELIZING DYNAMIC SPACE BASED APPLICATIONS USING ĀDHĀRA

In this chapter, we describe how to develop parallel programs using *Ādhāra*. We illustrate using three dynamic space-based applications from different scientific fields: electro-magnetic particle-in-cell (plasma physics), rarefied fluid flow (aeronautics) and molecular dynamics (materials science). We chose these three applications because they exhibit different computation and communication characteristics. The important differences are listed in Table 5.1.

Table 5.1: Comparison of the Characteristics of the Example Applications

	ELECTRO-MAGNETIC PARTICLE-IN-CELL	RAREFIED FLUID FLOW	MOLECULAR DYNAMICS
Type of simulation	movement of charged particles in electric and magnetic fields	movement of gaseous atoms in low density regions	movement of atoms interacting via Lennard-Jones potential
Spatial data structures	regular-grid and particle	regular-grid and particle	particle
Types of interactions	particle-grid grid-grid	particle-grid particle-particle (random pairing within a grid-cell)	particle-particle (pairing based on cut-off distance)
Granularity (computation per particle)	medium	small	large

For each application, we first describe the computation, give the sequential program, discuss the changes that are needed to convert the sequential program into an *Ādhāra* program, and then present the *Ādhāra* program.

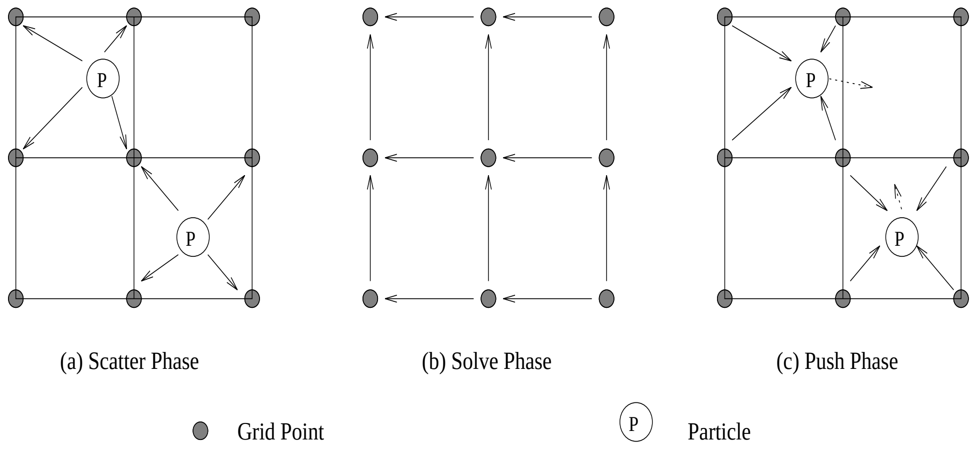


Figure 5.1: Data Dependencies in various phases in one time step of the EMPIC simulation
(Only two dimensions are used for simplicity of illustration. The grid points refer to the Electric Field, Magnetic Field or Current Density data, depending on the context.)

5.1 Electro-Magnetic Particle-In-Cell

The electro-magnetic particle-in-cell (EMPIC) application simulates movement of charged particles that interact by exerting electric and magnetic field forces on each other [Birdsall & Langdon 85, Hockney & Eastwood 88, Walker 90]. The force experienced by a particle depends on the current position and velocity of all the particles, and this changes continuously with time. The standard solution uses a particle-mesh method which discretizes space by a grid, and time by updating the position and velocity of the particles only at the boundaries of small time intervals.

Each time step consists of three phases. In the *scatter phase*, current-density is assigned to the grid points using the position and velocity of all the particles (Figure 5.1(a)). In the *solve phase*, new values of electric and magnetic fields are computed at each grid point, using the old field values and the current density assigned in the scatter phase (Figure 5.1(b)). A *leap-frog* time integration scheme is used for solving Maxwell's differential equations [Birdsall & Langdon 85, Hockney & Eastwood 88]. In the *push phase*, using the field values at the grid points, the force on each particle is computed, and the position and velocity of all the particles are updated (Figure 5.1(c)).

5.1.1 Sequential Program for the EMPIC Application

```
typedef struct vectornode { double X, Y, Z; } vectorType;

typedef struct particleNode {
    vectorType coordinate;
    vectorType velocity;
    double     charge;
    ....
} particleType;

particleType Particle[ MAX_NUM_PARTICLES ];
vectorType   ElectricField[ MAXNUM_X, MAXNUM_Y, MAXNUM_Z ],
             MagneticField[...], CurrentDensity[...];
                   /* three dimensional grids */

main() {
    /* read simulation parameters */
    ......

    /* read initial particle configuration */
    for ( i = 0; i < NumParticles; i++ ) {
        .. initialize fields of Particle[i] ..
    }

    /* initialize grid points */
    /* LOOP 1 */
    for ( i = 0; i < NumGridX; i++ )
     for ( j = 0; j < NumGridY; j++ )
      for ( k = 0; k < NumGridZ; k++ )
          ElectricField[i,j,k] = MagneticField[i,j,k] = 0;

    for ( step = 0; step < num_time_steps; step++ ) {
        ScatterRoutine();
        SolveRoutine();
        PushRoutine();
    }
}

ScatterRoutine() {
    /* initialize current density at all the grid points */
    /* LOOP 2 */
    for ( i = 0; i < NumGridX; i++ )
```

```
     for ( j = 0; j < NumGridY; j++ )
      for ( k = 0; k < NumGridZ; k++ )
          CurrentDensity[i,j,k] = 0;

    /* for each particle P, assign P's contribution of the
       current density to the surrounding grid points */
    /* LOOP 3 */
    for ( n = 0; n < NumParticles; n++ ) {
        P = &Particle[i];
        ... for each grid point (i,j,k) surrounding particle P do ...
            /* periodic boundary condition is used for
               determining neighboring grid points */
            CurrentDensity[i,j,k] += function_of( distance between
                                       grid-point (i,j,k) and P, .. );
    }
}

SolveRoutine() {
    /* advance magnetic fields from step K to K+1/2 */
    /* LOOP 4 */
    for ( i = 0; i < NumGridX; i++ )
     for ( j = 0; j < NumGridY; j++ )
      for ( k = 0; k < NumGridZ; k++ )
          /* periodic boundary condition is used for
             determining neighboring grid points */
          MagneticField[i,j,k] = function_of( ElectricField[i,j,k],
                                              ElectricField[i,j,k+1],
                                              ElectricField[i,j+1,k],
                                              ElectricField[i+1,j,k],
                                             .... );
    /* advance electric fields from step K to K+1 */
    /* LOOP 5 */
    for ( i = 0; i < NumGridX; i++ )
     for ( j = 0; j < NumGridY; j++ )
      for ( k = 0; k < NumGridZ; k++ )
          /* periodic boundary condition is used for
             determining neighboring grid points */
          ElectricField[i,j,k] = function_of( MagneticField[i,j,k],
                                              MagneticField[i,j,k-1],
                                              MagneticField[i,j-1,k],
                                              MagneticField[i-1,j,k],
                                              CurrentDensity[i,j,k],
                                              .... );
    /* advance magnetic fields from step K+1/2 to K+1 */
    /* LOOP 6 */
    .........................
}

PushRoutine() {
    /* for each particle P, compute the effective field on P by
       interpolating electric and magnetic fields from the
       surrounding grid points, and then update the velocity
       and position of P */
    /* LOOP 7 */
    for ( n = 0; n < NumParticles; n++ ) {
        P = &Particle[i];
        field_on_P = 0;
        ... for each grid point (i,j,k) surrounding particle P do ...
            /* periodic boundary condition is used for
               determining neighboring grid points */
            field_on_P += function_of( ElectricField[i,j,k],
                                       MagneticField[i,j,k],
                                       distance between (i,j,k) and P );
        P->velocity = function_of( field_on_P, P->velocity );
        P->coordinate = function_of( P->velocity, timestep );
    }
}
```

5.1.2 Ādhāra *Program for the EMPIC Application*

The following changes were made to the sequential program to convert it into an *Ādhāra* program:

- Declare and define the *computation space*, spatial data structures and *phases*. A new routine called *InitSpatialData()* is added for this purpose.
- Change the "for" loops 1, 2, 4, 5 and 6 to "FORALL-GRIDPOINTS" loops, and the "for" loops 3 and 7 to "FORALL-PARTICLES" loops.
- Use the EXECUTE primitive for organizing the computation into *phases* (see main() below).
- Use the READ-BOUNDARY and WRITE-BOUNDARY primitives for reading and updating *non-local* data when necessary.

- Use the UPDATE-COORDINATES primitive to inform the runtime system when the coordinates of the particles are updated (see PushRoutine() below).

The *Ādhāra* program for the EMPIC application is given below.

```
/* declare variables for spatial data structures */
3D-PARTICLE-TYPE Particle;
REGULAR-GRID-TYPE ElectricField, MagneticField, CurrentDensity;
PHASE-TYPE ScatterPhase, GridPhase, PushPhase;

/* this rountine defines the computation space, spatial data
   structures and phases */
InitSpatialData() {
    COMPUTATION-SPACE( BoxSizeX, BoxSizeY, BoxSizeZ );
                   /* size of the simulation box */
    3D-PARTICLE particleType Particle;
    REGULAR-GRID vectorType
                  ElectricField( NumGridX, NumGridY, NumGridZ );
    REGULAR-GRID vectorType MagneticField( ... );
    REGULAR-GRID vectorType CurrentDensity( ... );

    PHASE ScatterPhase {
                 ( ChargedParticle, PRIMARY, USAGE READ-ONLY ),
                 ( CurrentDensity, USAGE WRITE-ONLY ) };
    PHASE GridPhase {
                 ( ElectricField, PRIMARY, USAGE READ-WRITE ),
                 ( MagneticField, USAGE READ-WRITE ),
                 ( CurrentDensity, USAGE READ-ONLY ) };
    PHASE PushPhase {
                 ( ChargedParticle, PRIMARY, USAGE READ-WRITE ),
                 ( ElectricField, USAGE READ-ONLY ),
                 ( MagneticField, USAGE READ-ONLY ) };
}

main() {
    ADHARA-INITIALIZE-NODE;
       /* initializes the runtime system data structures */

    /* read simulation parameters */
    ........

    /* the following routine must be called before accessing any
       spatial data */
    InitSpatialData();

    /* read initial particle configuration */
    for ( i = 0; i < NumParticles; i++ ) {
        particleType tempP;
        .. initialize fields of tempP ..
        ADD-PARTICLE tempP TO ChargedParticle;
    }

    /* initialize grid points */
    /* LOOP 1 */
    FORALL-GRIDPOINTS (I,J,K) IN ElectricField DO
        ElectricField[I,J,K] = 0;
    FORALL-GRIDPOINTS (I,J,K) IN MagneticField DO
        MagneticField[I,J,K] = 0;

    for ( step = 0; step < num_time_steps; step++ ) {
        EXECUTE ScatterRoutine IN ScatterPhase;
        EXECUTE SolveRoutine IN GridPhase;
        EXECUTE PushRoutine IN PushPhase;
    }
}

ScatterRoutine() {
    /* some non-local grid-points are going to be updated, so
       initialize boundary data also */
    /* LOOP 2 */
    FORALL-GRIDPOINTS (I,J,K) IN CurrentDensity
                    INCLUDING BOUNDARY { ALL-DIRS = 1 } DO
        CurrentDensity[I,J,K] = 0;

    /* LOOP 3 */
    FORALL-PARTICLES (P) IN ChargedParticle DO {
        FORALL-GRIDPOINTS (I,J,K) IN CurrentDensity
                                SURROUNDING (P) DO {
            CurrentDensity[I,J,K] += function_of( distance between
                                  grid-point (I,J,K) and P, .. );
    }
    /* communicate updated non-local data */
    WRITE-BOUNDARY CurrentDensity
                 OVERLAP( { ALL-DIRS=1 }, PERIODIC-BOUNDARY )
                 OPERATION ( ADD-DOUBLETYPE );
}
```

```
SolveRoutine() {
    /* advance magnetic fields from step K to K+1/2 */
    /* non-local data along the positive boundary needed */
    READ-BOUNDARY ElectricField OVERLAP(
                 { X-DIR = (0,1),Y-DIR = (0,1),Z-DIR = (0,1) }
                     PERIODIC-BOUNDARY );
    /* LOOP 4 */
    FORALL-GRIDPOINTS (I,J,K) IN MagneticField DO
          MagneticField[I,J,K] = function_of( ElectricField[I,J,K],
                                              ElectricField[I,J,K+1],
                                              .... );
    /* advance electric fields from step K to K+1 */
    /* non-local data along the negative boundary needed */
    READ-BOUNDARY MagneticField OVERLAP(
                 { X-DIR = (1,0),Y-DIR = (1,0),Z-DIR = (1,0) }
                     PERIODIC-BOUNDARY );
    /* LOOP 5 */
    FORALL-GRIDPOINTS (I,J,K) IN MagneticField DO
          ElectricField[I,J,K] = function_of( MagneticField[I,J,K],
                                              MagneticField[I,J,K-1],
                                              ....
                                              CurrentDensity[I,J,K] );
    /* advance magnetic fields from step K+1/2 to K+1 */
    /* LOOP 6 */
    ..........................
}


PushRoutine() {
    /* non-local data along all the boundaries needed */
    READ-BOUNDARY ElectricField
                 OVERLAP( { ALL-DIRS=1 }, PERIODIC-BOUNDARY )
    READ-BOUNDARY MagneticField
                 OVERLAP( { ALL-DIRS=1 }, PERIODIC-BOUNDARY )
    /* LOOP 7 */
    FORALL-PARTICLES (P) IN ChargedParticle DO {
        field_on_P = 0;
        FORALL-GRIDPOINTS (I,J,K) IN ElectricField
                                 SURROUNDING (P) DO {
            field_on_P += function_of( ElectricField[I,J,K],
                                       MagneticField[I,J,K],
                                       distance between (I,J,K) and P );
        P->velocity = function( field_on_P, P->velocity );
        P->coordinate = function( P->velocity, timeStep );
    }
    UPDATE-COORDINATES OF ChargedParticle;
}
```

5.2 Rarefied Fluid Flow

The rarefied fluid flow application, referred to as MP3D in the SPLASH benchmarks [Singh et al 90], simulates trajectories of the gaseous molecules using Monte Carlo method [Fallavollita et al. 92, McDonald 89].

The active space is a rectangular tunnel with openings at each end and reflecting walls on the remaining sides. The object being studied (e.g., a space vehicle) is represented as a set of additional boundaries in the active space. Atoms generally flow through the tunnel in the positive x direction. Exiting atoms are reused after being thermalized to the free stream temperature and randomly distributed near the entrance to the tunnel. A reservoir of atoms is maintained to keep track of a set of random coordinates and velocities. These values are used for randomly distributing new atoms near the entrance of the tunnel.

Computation is performed on random pairs of atoms, and atomic collisions are statistically determined. For the purposes of efficient collision pairing, the active space is represented as a three-dimensional *space array* of unit-sized cells. Atoms can move among cells, but are only eligible for collision with other atoms occupying the same cell at that time. Each time step consists of a single phase in which random pairs of atoms within each cell are collided and the positions and velocities of the atoms are updated.

5.2.1 Sequential Program for Rarefied Fluid Flow Application

```
typedef struct vectornode { double X, Y, Z; } vectorType;
typedef struct atomnode {
    vectorType coordinate;
    vectorType velocity;
    double     r, s; /* rotational velocities */
} atomType;

typedef struct cellnode {
```

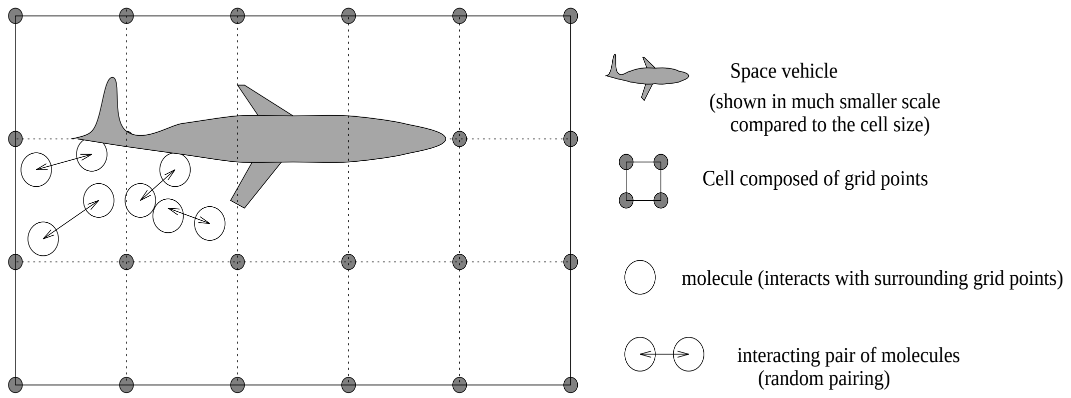

Space vehicle
(shown in much smaller scale compared to the cell size)
Cell composed of grid points
molecule (interacts with surrounding grid points)
interacting pair of molecules (random pairing)

Figure 5.2: Data dependencies in one time step of the rarefied fluid flow simulation *(Only eight atoms in two dimensional space are used for simplicity of illustration.)*

```
    int       boundary_tag; /* if greater than zero, this cell
                                 falls in the boundary region;
                                 the number gives boundary type */
    int       cell_population; /* no.of particles in the cell */
    double    avg_probability; /* this value is used for random
                                    collisions within the cell */
    atomType *atom_in_cell; /* used for pairing up atoms within
                            a cell for collision -- see move_atoms() */
    .....
} cellType;

atomType  Atoms[ MAX_NUM_ATOMS ];
atomType  ReservoirAtoms[ MAX_NUM_RESERVOIR_ATOMS ];
cellType  Cell[ MAXNUM_X ][ MAXNUM_Y ][ MAXNUM_Z ];
               /* active space of three-dimensional cells */

main() {
    /* read simulation parameters */
    ......

    /* read initial atom configuration */
    for ( i = 0; i < NumAtoms; i++ ) {
        .. initialize fields of Atoms[i] ..
    }

    /* initialize cells */
    /* LOOP 1 */
    for ( i = 0; i < NumCellX; i++ )
```

```
     for ( j = 0; j < NumCellY; j++ )
      for ( k = 0; k < NumCellZ; k++ )
          .. initialize Cell[i,j,k] ..

    for ( time_step = 0; time_step < num_time_steps; time_step++ ) {
        advance();
    }
}

advance() {
    ....
    reset();
    move_atoms();
    add_atoms_to_free_stream();
    move_and_collide_reservoir_atoms();
}

reset() {
    /* reset cells */
    /* LOOP 2 */
    for ( i = 0; i < NumCellX; i++ )
     for ( j = 0; j < NumCellY; j++ )
      for ( k = 0; k < NumCellZ; k++ ) {
          C = &Cell[i,j,k];
          C->avg_probability = function( C->avg_probability,
                                         C->cell_population, ... );
          C->atom_in_cell = NULL;
          C->cell_population = 0;
      }
}

boundary_case( boundary_tag, A /* atom */) {
    switch (boundary_tag) {
     case SOLID_WALL: .. bounce back A .. ; break;
     case AFTER_EXIT_OF_TUNNEL: exit_num_flow++;
                 /* keep track of the number of atoms exiting
                    from the tunnel; this value is used in
                    add_atoms_to_free_stream() to determine
                    how many extra atoms to add to the system */
                 /* reuse this atom -- place it near the
                    entrance of the tunnel; the position and
                    velocity are randomized using the coordinate
                    and velocity of the next reservoir atom */
     case BEFORE_ENTRANCE_OF_TUNNEL:
                 R = &ReservoirAtoms[ next_R_atom++ ];
                 A->velocity = R->velocity;
                 /* randomly distribute at the entrance */
                 A->coordinate = function( R->coordinate, .. );
    }
}

move_atoms() {
    /* For each atom, move using the current velocity. If the new
       position falls in a boundary region, apply appropriate
       boundary condition. Then pair it up with another atom from
       the same cell and randomly collide to determine new
       set of velocities */
    /* LOOP 3 */
    for ( n = 0; n < total_num_atoms; n++ ) {
        A = &Atoms[i];
        A->coordinate = function( A->velocity );
        C = .. ptr to cell in which A is located ..;

        /* if A is in a boundary cell, apply boundary condition */
        if ( C->boundary_tag > 0 ) {
            boundary_case( C->boundary_tag, A );
            C = .. ptr to the new cell in which A is located ..;
        }
        C->cell_population++;

        /* If A is in an open cell, collide with another atom
           within the same cell. Pairing is done as follows:
           For each cell, the odd-numbered atom is placed in the
           "atom_in_cell" slot. When the next (even-numbered) atom
           from the same cell is encountered, it is paired up with
           the stored odd-numbered atom, and the slot is cleared. */

        if ( C->boundary_tag == 0 ) { /* open cell */
            if ( C->atom_in_cell == NULL ) /* odd-numbered atom */
                C->atom_in_cell = A; /* store it in the slot */
            else { /* even-numbered atom, so pair up */
                if ( RANDOM_NUMBER <= C->avg_probability )
                   .. collide atom "A" with atom "C->atom_in_cell"
                   to compute new velocities for both the atoms ..
                C->atom_in_cell = NULL; /* clear the slot */
            }
            }
            else  ...
        }
    }

    add_atoms_to_free_stream() {
        /* add new atoms to the system */
        total_num_atoms_to_be_added = function( total_num_atoms,
                                                exit_num_flow, ..);
        /* LOOP 4 */
        for ( n = 0; n < total_num_atoms_to_be_added; n++ ) {
            A = &Atoms[total_num_atoms];
            R = &ReservoirAtoms[ next_R_atom++ ];
            /* use the coordinates and velocities of the
               reservoir atoms to randomly distribute new atoms
               at the entrance of tunnel */
            A->velocity = R->velocity;
            A->coordinate = function( R->coordinate, .. );
        }
        total_num_atoms += total_num_atoms_to_be_added;
    }

    move_and_collide_reservoir_atoms() {
        .. move each reservoir atom to update coordinates ..
        .. collide pairs of reservoir atoms to update velocities ..
    }
```

5.2.2 Ādhāra *Program for Rarefied Fluid Flow Application*

We made the following changes to the sequential program to convert it into an *Ādhāra* program:

- We split the reservoir of atoms into small reservoirs, and assigned one small reservoir to each processor. This enables the processors to maintain smaller reservoirs independently of each other (Loop 4).

- We declared and defined the *computation space*, spatial data structures and *phases* (see InitSpatialData() below). We organized the computation into one *phase* (see main() below).

- We changed the "for" loops 1 and 2 to "FORALL-GRIDPOINTS" loops, and the "for" loop 3 to a "FORALL-PARTICLES" loop.

- The only *non-local* data that is read and updated is the cell population, so to minimize the communication overhead, we separated the "cell_population" field from the "cellType", and used another regular grid called "CellPopulation" to keep track of the number of atoms in each cell.

- We used the READ-BOUNDARY and WRITE-BOUNDARY primitives for reading and updating *non-local* data when necessary. (See move_atoms() and reset() below.)

- We used the UPDATE-COORDINATES primitive to inform the runtime system when the coordinates of the particles are updated (see move_atoms() below). In add_atoms_to_free_stream() routine, we used the GLOBAL-SUM primitive for summing up the value "num_exit_flow" which was updated by all the processors.

The *Ādhāra* program for the rarefied fluid flow application is given below.

```
/* declare variables for spatial data structures */
3D-PARTICLE Atoms;
REGULAR-GRID Cell, CellPopulation;
PHASE-TYPE AdvancePhase;

InitSpatialData() {
    COMPUTATION-SPACE( BoxSizeX, BoxSizeY, BoxSizeZ );
                       /* size of the simulation box */
    3D-PARTICLE atomType Atoms;
    REGULAR-GRID cellType Cell( NumCellX, NumCellY, NumCellZ );
    REGULAR-GRID int CellPopulation( ... );

    PHASE AdvancePhase {
                 ( Atoms, PRIMARY, USAGE READ-WRITE ),
                 ( Cell, USAGE READ-WRITE ),
                 ( CellPopulation, USAGE READ-WRITE ) };
}

main() {
    ADHARA-INITIALIZE-NODE;
    /* read simulation parameters */
    ......

    /* the following routine must be called before accessing any
       spatial data */
    InitSpatialData();

    /* read initial atom configuration */
    for ( i = 0; i < NumAtoms; i++ ) {
        atomType tempA;
        .. initialize fields of tempA ..
        ADD-PARTICLE tempA TO Atoms;
    }

    /* initialize cells */
    /* LOOP 1 */
    FORALL-GRIDPOINTS (I,J,K) IN Cell DO
          .. initialize Cell[I,J,K] ..
    FORALL-GRIDPOINTS (I,J,K) IN CellPopulation DO
          CellPopulation[I,J,K] = 0;

    for ( time_step = 0; time_step < num_time_steps; time_step++ ) {
        EXECUTE advance IN AdvancePhase;
    }
}

advance() {
    ....
    reset();
    move_atoms();
    add_atoms_to_free_stream();
    move_and_collide_reservoir_atoms();
}

reset() {
    /* need non-local CellPopulation data along the boundary */
    READ-BOUNDARY CellPopulation
                 OVERLAP( { ALL-DIRS=1 }, RIGID-BOUNDARY )

    /* reset cells */
    /* LOOP 2 */
    FORALL-GRIDPOINTS (I,J,K) IN Cell
```

```
                       INCLUDING BOUNDARY { ALL-DIRS = 1 } DO {
            C = &Cell[I,J,K];
            C->avg_probability = function( C->avg_probability,
                                      CellPopulation[I,J,K], ... );
            C->atom_in_cell = NULL;
            CellPopulation[I,J,K] = 0;
      }
}

boundary_case( boundary_tag, A /* atom */) {
    switch (boundary_tag) {
     case SOLID_WALL: .. bounce back A .. ; break;
     case AFTER_EXIT_OF_TUNNEL: exit_num_flow++;
     case BEFORE_ENTRANCE_OF_TUNNEL:
                R = &ReservoirAtom[ next_R_atom++ ];
                A->velocity = R->velocity;
                /* randomly distribute at the entrance */
                A->coordinate = function( R->coordinate, .. );
    }
}

move_atoms() {
    /* LOOP 3 */
    FORALL-PARTICLES (A) IN Atoms DO {
        A->coordinate = function( A->velocity );
        (I,J,K) = CELL-INDEX OF (A);
        C = Cell[I,J,K];

        /* if A is in a boundary cell, apply boundary condition */
        if ( C->boundary_tag > 0 ) {
            boundary_case( C->boundary_tag, A );
            (I,J,K) = CELL-INDEX OF (A);
            C = Cell[I,J,K];
        }
        CellPopulation[I,J,K]++;

        if ( C->boundary_tag == 0 ) { /* open cell */
            if ( C->atom_in_cell == NULL ) /* odd-numbered atom */
                C->atom_in_cell = A; /* store it in the slot */
            else { /* even-numbered atom, so pair up */
                if ( RANDOM_NUMBER <= C->avg_probability )
                   .. collide atom "A" with atom "C->atom_in_cell"
                   to compute new velocities for both the atoms ..
```

```
                    C->atom_in_cell = NULL; /* clear the slot */
                }
            }
            else  ...
        }
        UPDATE-COORDINATES OF Atoms;

/* communicate updated non-local data */
        WRITE-BOUNDARY CellPopulation
                      OVERLAP( { ALL-DIRS=1 }, RIGID-BOUNDARY )
                      OPERATION ( ADD-INTTYPE );
    }

    add_atoms_to_free_stream() {
        /* sum up the number of exit atoms from all the nodes */
        GLOBAL-SUM exit_num_flow DATATYPE-INT;

        /* add new atoms to the system */
        total_num_atoms_to_be_added = function( total_num_atoms,
                                                exit_num_flow, ..);
        /* LOOP 4 */
/* split the reservoir into small reservoirs, and maintain small
   reservoirs at each processor */
        num_per_processor = total_num_atoms_to_be_added / NumProcessors;
        for ( n = 0; n < num_per_processor; n++ ) {
            atomType tempA;
            R = &ReservoirAtoms[ next_R_atom++ ];
            tempA.velocity = R->velocity;
            tempA.coordinate = function( R->coordinate, .. );
            ADD_PARTICLE-LOCAL tempA TO Atoms;
        }
        total_num_atoms += total_num_atoms_to_be_added;
    }

    move_and_collide_reservoir_atoms() {
        .. move each reservoir atom to update coordinates ..
        .. collide pairs of reservoir atoms to update velocities ..
    }
```

5.3 *Molecular Dynamics*

Molecular Dynamics simulation, herein after referred to as MD, is a very powerful and popular technique used to study a number of interesting problems related to film deposition, materials interfaces, formation and properties of minerals in extreme atmospheric conditions, etc. [Allen & Tildesley 87, Fincham 87, Pinches et al. 91]. Conceptually, MD involves numerical integration (over time) of the classical Newton's equations of motion for a system of interacting particles. These particles move about inside a simulation box with periodic boundaries. From the computed particle trajectories, many of the material properties can be derived.

Particles in the system can interact via a number of potentials. In the example shown here, we use the classic Lennard-Jones potential, a function of inter-particle separation. It is a short-range potential, i.e. two particles separated by a distance greater than *rcut*, the cutoff distance for the potential, will not interact with each other. Identifying particle pairs separated by distances less than *rcut* is the first step in force computation. Next, gradient of the potential for each interacting pair is computed, and this gives the force due to pair interaction. Net force on each particle is the vector sum of these pair forces.

Once the forces are evaluated, the atom position is updated to the next time step by a second order numerical scheme. For this new set of positions, forces are once again computed and their values used to update the particle velocities. This iterative scheme is continued over a large number of time steps.

The code given below uses a *multiple time step* scheme for simulating a system containing light and heavy atoms. Since light atoms tend to move faster than heavy atoms, in every time step, the properties of the light atoms are updated several times before updating the properties of the heavy atoms.

5.3.1 *Sequential Program for the Molecular Dynamics Application*

```
typedef struct vectornode { double X, Y, Z; } vectorType;
typedef struct atomNode {
    vectorType coordinate;
    int        atom_type;
    vectorType force;
    vectorType velocity;
    vectorType acceleration;
```

```
      vectorType displacement;
  } atomType;

atomType Atoms[ MAX_NUM_ATOMS ];

  main() {
      /* read simulation parameters */
      ......

      /* read initial atom configuration */
      for ( i = 0; i < NumAtoms; i++ ) {
          .. initialize fields of Atoms[i] ..
      }

      /* partially sort the atoms into cells */
      sort_atoms_into_cells( cutoffX, cutoffY, cutoffZ );
               /* code for this is quite involved, and not given here */

      for ( big_time_step = 0; big_time_step < num_big_time_steps;
                                          big_time_step++ ) {
          update_atoms();
      }
  }

  update_coordinate( A, timestep ) {
      new_coordinate = function_of( A->coordinate, A->velocity, timestep );
      A->displacement = A->displacement + (new_coordinate - A->coordinate);
      if ( A->displacement > threshold ) need_to_re-sort_into_cells = 1;
      A->coordinate = new_coordinate;
  }

  update_velocity( A, timestep ) {
      A->velocity = function_of( A->velocity, A->acceleration, A->force,
                                 A->atom_type, timestep );
  }

  correct_velocity( A, timestep ) {
      correction = function_of( A->acceleration, A->force,
                                A->atom_type, timestep );
      A->velocity += correction;
  }

  update_atoms() {
```

```
/* fix heavy atoms and integrate light atoms for
                            num_small_time_steps */
for ( step = 0; step < num_small_time_steps; step++ ) {
    /* LOOP 1 */
    for ( n = 0; n < NumAtoms; n++ ) {
        A = &Atoms[i];
        if ( A->atom_type == LIGHT_ATOM )
            update_coordinate( A );
    }
    /* compute force on each light atom */

    /* LOOP 2 */
    .. for each atom pair (A1,A2) within cutoff distance do .. {
    /* the code for extracting pairs is complicated, not given here */
        if ( A1->atom_type == LIGHT_ATOM ||
                       A2->atom_type == LIGHT_ATOM ) {
            pair_force12 = function_of(A1->coordinate,A1->atom_type,
                                       A2->coordinate,A2->atom_type );
            A1->force += pair_force12;
            A2->force -= pair_force12;
        }
    }

    /* LOOP 3 */
    for ( n = 0; n < NumAtoms; n++ ) {
        A = &Atoms[i];
        if ( A->atom_type == LIGHT_ATOM ) {
            update_velocity( A, small_timestep );
            A->force = 0;
        }
    }

    if ( need_to_re-sort_into_cells ) {
        re-sort_atoms_into_cells(); /* code not given here */
        need_to_re-sort_into_cells = 0;
    }
}

/* update coordinates of heavy atoms by big_timestep */
/* LOOP 4 */
for ( n = 0; n < NumAtoms; n++ ) {
    A = &Atoms[i];
    if ( A->atom_type == HEAVY_ATOM ) {
```

```
                update_coordinate( A, big_timestep );
                A->force = 0;
            }
        }

        /* compute forces on all atoms */
        /* LOOP 5 */
        for each atom pair (A1,A2) within cutoff distance do {
            pair_force12 = function_of( A1->coordinate, A1->atom_type,
                                        A2->coordinate, A2->atom_type );
            A1->force += pair_force12;
            A2->force -= pair_force12;
        }

        /* compute velocities of heavy atoms and
           correct velocities of light atoms */
        /* LOOP 6 */
        for ( n = 0; n < NumAtoms; n++ ) {
            A = &Atoms[i];
            if ( A->atom_type == HEAVY_ATOM )
                update_velocity( A, big_timestep );
            else
                correct_velocity( A, big_timestep );
            A->displacement = 0;
        }
    }
```

5.3.2 Ādhāra *Program for the Molecular Dynamics Application*

We made the following changes to the sequential program to convert it into an *Ādhāra* program:

- We declared and defined the *computation space*, spatial data structures and *phases* (see InitSpatialData() below). We organized the computation into one *phase* (see main() below).

- We used the SORT-INTO-CELLS primitive to partially sort the atoms into cells (see InitSpatialData()), since we need to form pairs of atoms for the force computation (see loops 2 and 5).

- We changed the "for" loops 1, 3, 4 and 6 to "FORALL-PARTICLES (A)" loops, and the loops 2 and 5 to "FORALL-PARTICLES (A1,A2) .." loops.
- We used the UPDATE-COORDINATES primitive to re-sort the atoms into cells.

The *Ādhāra* program for the molecular dynamics application is given below.

```
3D-PARTCLE-TYPE Atoms; PHASE-TYPE UpdatePhase;

InitSpatialData() {
    COMPUTATION-SPACE( BoxSizeX, BoxSizeY, BoxSizeZ );
    3D-PARTICLE atomType Atoms SORT-INTO-CELLS (cutoffX, cutoffY, cutoffZ);
    PHASE UpdatePhase { (Atoms, PRIMARY, USAGE READ-WRITE) };
}
main() {
    ADHARA-INITIALIZE-NODE;
    /* read simulation parameters */
    ........
    InitSpatialData();

    /* read initial atom configuration */
    for ( i = 0; i < NumAtoms; i++ ) {
        atomType tempA;
        .. initialize fields of tempA ..
        ADD-PARTICLE tempA TO Atoms;
               /* atoms will be automatically sorted into cells */
    }
    for ( big_time_step = 0; big_time_step < NumBigTimeSteps;
                                       big_time_step++ ) {
        EXECUTE update_atoms IN UpdatePhase;
    }
}

update_atoms() {
    /* fix heavy atoms and integrate light atoms for num_small_time_steps */
    for ( step = 0; step < num_small_time_steps; step++ ) {
        /* LOOP 1 */
        FORALL-PARTICLES (A) IN Atoms DO {
            if ( A->atom_type == LIGHT_ATOM )
                update_coordinate( A );
        }
    /* compute force on each light atom */
    /* LOOP 2 */
    FORALL-PARTICLES (A1,A2) IN Atoms CELL-DISTANCE 1 DO {
        if ( A1->atom_type == LIGHT_ATOM ||
                       A2->atom_type == LIGHT_ATOM ) {
            pair_force12 = function_of(A1->coordinate,A1->atom_type,
                                       A2->coordinate,A2->atom_type );
            A1->force += pair_force12;
            A2->force -= pair_force12;
        }
    }
    /* LOOP 3 */
    FORALL-PARTICLES (A) IN Atoms DO {
        if ( A->atom_type == LIGHT_ATOM ) {
            update_velocity( A, small_timestep );
            A->force = 0;
        }
    }
    if ( need_to_re-sort_into_cells ) {
        UPDATE-COORDINATES OF Atoms;
           /* atoms will be automatically re-sorted into cells */
        need_to_re-sort_into_cells = 0;
    }
}
/* update coordinates of heavy atoms by big_timestep */
/* LOOP 4 */
FORALL-PARTICLES (A) IN Atoms DO {
    if ( A->atom_type == HEAVY_ATOM ) {
        update_coordinate( A, big_timestep );
        A->force = 0;
    }
}
/* compute forces on all atoms */
/* LOOP 5 */
FORALL-PARTICLES (A1,A2) IN Atoms CELL-DISTANCE 1 DO {
    pair_force12 = function_of( A1->coordinate, A1->atom_type,
                                A2->coordinate, A2->atom_type );
    A1->force += pair_force12;
    A2->force -= pair_force12;
}
/* compute velocities of heavy atoms and
   correct velocities of light atoms */
/* LOOP 6 */
```

```
        FORALL-PARTICLES (A) IN Atoms DO {
            if ( A->atom_type == HEAVY_ATOM )
                update_velocity( A, big_timestep );
            else
                correct_velocity( A, big_timestep );
        }
    }
```

5.4 *Summary*

In this chapter, we described how to develop parallel programs using *Ādhāra*. Using three dynamic space-based applications as examples, we illustrated the process of converting the sequential programs into *Ādhāra* programs. We showed that the conversion process uses simple concepts about data-parallel programming and very few lines of additional code.

Chapter 6

EVALUATION OF THE ĀDHĀRA PROGRAMMING MODEL AND COMPARISON WITH EXISTING PROGRAMMING ENVIRONMENTS

We evaluate the effectiveness of the *Ādhāra* programming model using the following metrics:

- *Programming effort*: We measure the programming effort in terms of the amount of extra code that need to be written, in order to convert a sequential program into a parallel program. Let L_{par} be the number of lines in the parallel source code and L_{seq} be the number of lines in the sequential source code. Then

$$\text{programming effort} = \frac{L_{par} - L_{seq}}{L_{seq}}$$

- *Parallel efficiency*: Let T_{seq} be the execution time of the sequential program for a given input. Let $T_{par,P}$ be the execution time of the parallel program for the same input on P processors. Then

$$\text{parallel efficiency (on } P \text{ processors)} = \frac{T_{seq}}{P T_{par,P}}$$

6.1 *Evaluation of the* Ādhāra *Programming Model*

In order to evaluate the effectiveness of the *Ādhāra* model, we compare *Ādhāra* programs with two versions of hand-coded parallel programs (that use message passing primitives): one that uses static data partitioning and other that employs dynamic load balancing. Both versions exploit spatial locality by using domain decomposition. Actual hand-coded parallel programs that perform all optimizations to minimize communication, load balance and load imbalance overheads are not available, since it is very hard and time consuming to hand-code the necessary optimizations. (This is the motivation for developing *Ādhāra* programming environment.) Hence we estimate the programming effort for the hand-coded versions based on the amount of *Ādhāra*

Table 6.1: Evaluation of the *Ādhāra* programming model

(For the first two applications, inputs with 32K particles are used, and for the third application, an input with 16K particles is used. An Intel Paragon with 16 nodes is used for all the measurements.)

	Electro-Magnetic Particle-In-Cell	Rarefied Fluid Flow	Molecular Dynamics
Sequential code	1500 lines	1700 lines	4500 lines
HAND-CODED (with static partitioning)			
Parallel code	2500 lines	2700 lines	6500 lines
Programming effort	67%	59%	44%
Parallel efficiency	70%	49%	59%
HAND-CODED (with dynamic load balancing)			
Parallel code	6500 lines	6700 lines	10500 lines
Programming effort	333%	294%	133%
Parallel efficiency	88%	80%	79%
ADHARA			
Parallel code	1600 lines	1800 lines	3800 lines
Programming effort	7%	6%	-15%
Parallel efficiency	84%	76%	71%

runtime system code that is exercised by each application. We estimate the parallel efficiency for the hand-coded versions that employ dynamic load balancing based on how much the performance of the *Ādhāra* program can be improved by employing an idealized load balancing scheme that balances the load perfectly by incurring absolutely no overhead.

Table 6.1 evaluates the *Ādhāra* model using the three dynamic space-based applications that were discussed in the previous chapter. The hand-coded versions that use static partitioning take reasonable amount of programming effort, but do not perform well. The hand-coded versions that use dynamic load balancing perform very well, but at the cost of substantial programming effort. The *Ādhāra* programs perform reasonably close to the optimized hand-coded versions that employ idealized load balancing scheme, but require very little programming effort. In the case of molecular dynamics application, the programming effort is negative, that is, it is quicker to develop an *Ādhāra* program than to develop a sequential program. This is

Table 6.2: Comparison of *Ādhāra* with existing programming environments
(*In the row that compares the support for data sharing,* explicit *means explicitly specified by the user, and* implicit *means automatically performed by the system.*)

	HPF like language	PARTI	LPAR	ADHARA
Environment	general purpose language / parallel compiler	general purpose runtime system for parallel compilers	specialized for dynamic non-uniform applications	specialized for dynamic space-based applications
Spatial data structures	regular-grid	regular and irregular grids	composition of regular domains	regular-grid and particle
Operations on spatial data structures	iteration over grid points	iteration over grid points	iteration over set of regular domains	iteration over grid points, particles and pairs of particles;
Data Sharing	implicit for grids	explicit for grids	explicit for domains	explicit for grids, implicit for particles
Data Partitioning	specified by the user	managed by the user	managed by the user	automatic
Load Balancing	managed by the user	managed by the user	managed by the user	automatic

because, *Ādhāra* supports high-level primitives for operating on space-based objects, such as iterating on pairs of closely located particles. In the sequential program, the code for implementing such primitives must be developed, where as, in the *Ādhāra* program such code is not needed.

6.2 Comparison with Existing Programming Environments

In Table 6.2, we compare *Ādhāra* with the existing programming environments that can be used for parallelizing dynamic space-based applications. Compared to these environments, it is more convenient to program using *Ādhāra* because *Ādhāra* supports high-level primitives to create and operate on spatial data structures in a natural way, and data partitioning and load balancing are automatically performed by the system.

6.3 Summary

In the second part of this report (chapters 4-6), we presented the *Ādhāra* programming model, described how to develop programs using three sample applications, evaluated the model using the programming effort and parallel efficiency metrics, and compared *Ādhāra* with the existing programming systems. In the next part of the thesis, we discuss the issues in application induced load balancing.

Part III

Application Induced Load Balancing

Chapter 7

ISSUES IN APPLICATION INDUCED LOAD BALANCING

There are three issues in dynamic load balancing in an application domain: (1) how to partition the space, (2) how to load balance, and (3) when to load balance. These three issues are discussed in the rest of this chapter.

7.1 How to Partition the Space

There are several ways of decomposing a three-dimensional space into small regions and assigning them to processors. The regions could be rectangular or non-rectangular, a processor could be assigned one or more regions, and the space could be partitioned either non-hierarchically or hierarchically. In this section, we describe various alternatives and discuss the tradeoffs.

7.1.1 Rectangular vs. Non-Rectangular Regions

The regions can be either rectangular or non-rectangular. The motivation of using non-rectangular regions (Figure 7.1) is to exploit the characteristics of the application for achieving a fine load balance with a small communication overhead [Hinz 90, Weaver & Schnabel 92, Williams 91b]. Non-rectangular regions are typically used for partitioning irregular meshes [Williams 91b]. Computing a good non-rectangular partition is expensive. A non-rectangular region also introduces significant overhead of maintaining the data partitions. This approach might be practical for the problems where the load distribution is static and the partitioning is performed just once at the beginning of the execution. For dynamic problems it is probably not worth spending a lot of time finding a perfect partitioning each time the load is rebalanced, since the load is going to change in the very next problem step.

In this work we consider only rectangular regions for the following reasons: (1) the overhead of maintaining and repartitioning the data is small, (2) the load balancing

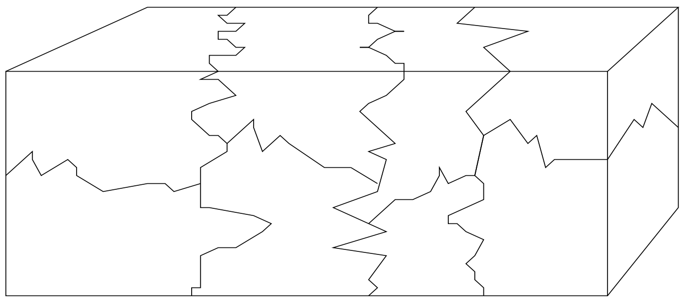


Figure 7.1: Partitioning a Three Dimensional Space into Eight Non-rectangular Regions

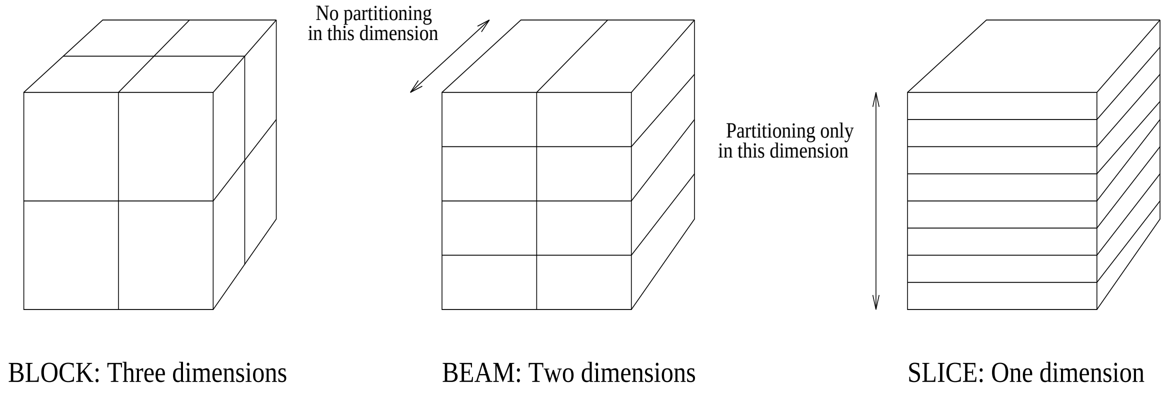

No partitioning in this dimension
Partitioning only in this dimension
BLOCK: Three dimensions
BEAM: Two dimensions
SLICE: One dimension

Figure 7.2: Methods of Partitioning a Three Dimensional Space into Eight Rectangular Regions

overhead can be adjusted in a simple way depending on the desired quality of the load balance (Section 9.4), and (3) certain issues in the rectangular partitioning have not received enough attention by researchers, and these need to be studied in detail before moving on to more complicated schemes.

There are many ways of decomposing a k-dimensional space into rectangular regions, depending on how many of the dimensions are partitioned. Each dimension can either be partitioned or not partitioned, so there are $2^k - 1$ choices in total. (The case where none of the dimensions is partitioned is not allowed.) A three dimensional space can be decomposed in seven different ways (Figure 7.2): partitioning all three dimensions (one choice), two dimensions (three choices) or one dimension (three choices).

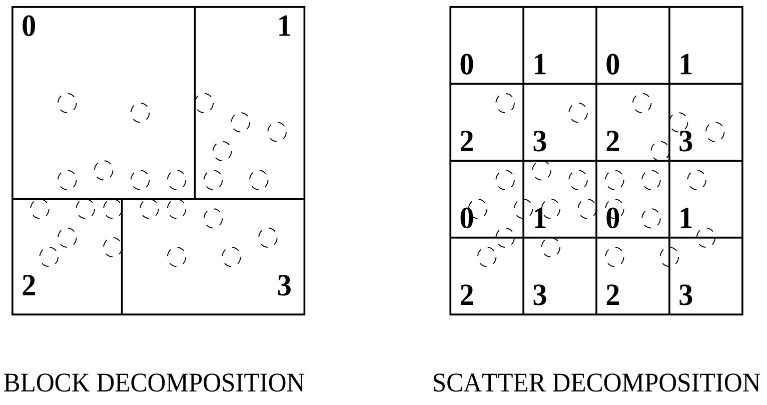


Figure 7.3: Block vs. Scatter Decompositions
(The figure shows how a two dimensional space can be partitioned onto four processors using block and scatter decompositions. The dotted circles represent the particles in space. The numbers represent the processors to which the corresponding regions are assigned.)

7.1.2 Block vs. Scatter Decompositions

Let us suppose there are P processors. The method of decomposing the space into P contiguous regions of equal load (but not necessarily of equal volume) is called a *block* decomposition. If the load distribution changes dynamically, the space needs to be repartitioned. Dynamic load balancing can be avoided by a *scatter* decomposition scheme [Nicol & Saltz 90]. This scheme decomposes the space into sP sub-regions of equal volume and assigns s sub-regions to each processor in a regular manner (Figure 7.3). This method gives good load balance if an appropriate s is chosen, but results in a higher communication cost, since the region (collection of sub-regions) assigned to a processor is not contiguous. Moreover, it is hard to choose an optimal parameter s: if s is too small, the load imbalance overhead might be significant, and if s is too large, the communication overhead might be significant. In this work, we propose efficient dynamic load balancing schemes, hence we focus only on the block decomposition.

7.1.3 Non-Hierarchical vs. Hierarchical Partitioning

In a *non-hierarchical* scheme, no attention is paid to the order in which dimensions are partitioned (Figure 7.4). In a *hierarchical* scheme, the space is partitioned at

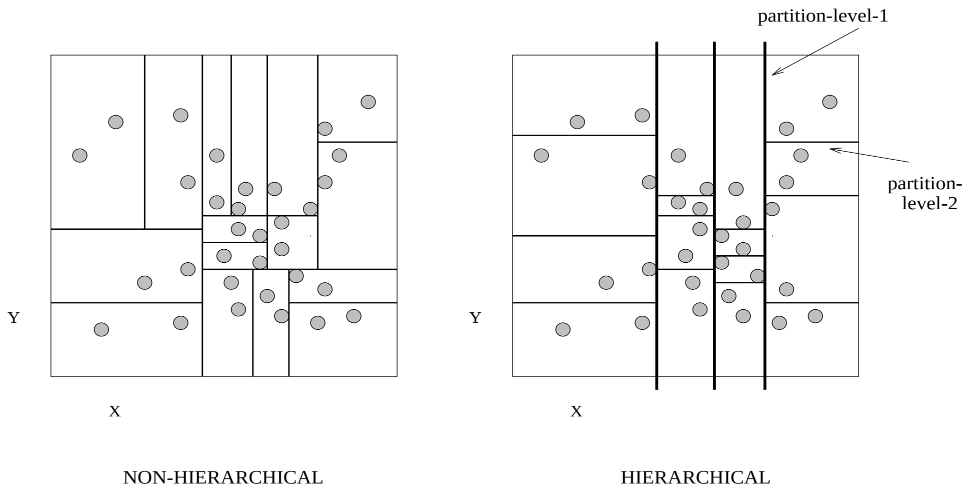


Figure 7.4: Non-hierarchical vs. Hierarchical Partitioning
(The figure shows how a two dimensional space can be partitioned into sixteen regions using non-hierarchical and hierarchical schemes.)

different levels. In each level L, the region is split into a specified number of parts N_L along the specified dimension D_L, such that each part has approximately the same amount of computational load.

There are two types of hierarchical partitioning, one in which the number of levels are related to the number of processors, and the other in which the number of levels are related to the number of dimensions of the space. Figure 7.5 gives specific examples in which the space is partitioned into sixteen regions. The TYPE-1 scheme has 4 levels (number of levels = log P, where P is the number of processors), and in each level, the region is split into 2 parts along alternate dimensions: $D_1 = X, D_2 = Y, D_3 = X$ and $D_4 = Y$. The TYPE-2 scheme has 2 levels (number of levels = number of dimensions). At level-1, the space is split into 4 parts along X, and at level-2, each of the resulting parts is split into 4 regions along Y.

A hierarchical scheme is preferable to a non-hierarchical scheme for two reasons: it is easy to specify, and it can take advantage of preferential movement [1] of spatial objects when the computational load is balanced dynamically. If the movement of

[1] We say that there is preferential movement along dimension X, if the average displacement of the particles per time step along X is much greater than the displacement along the other dimensions.

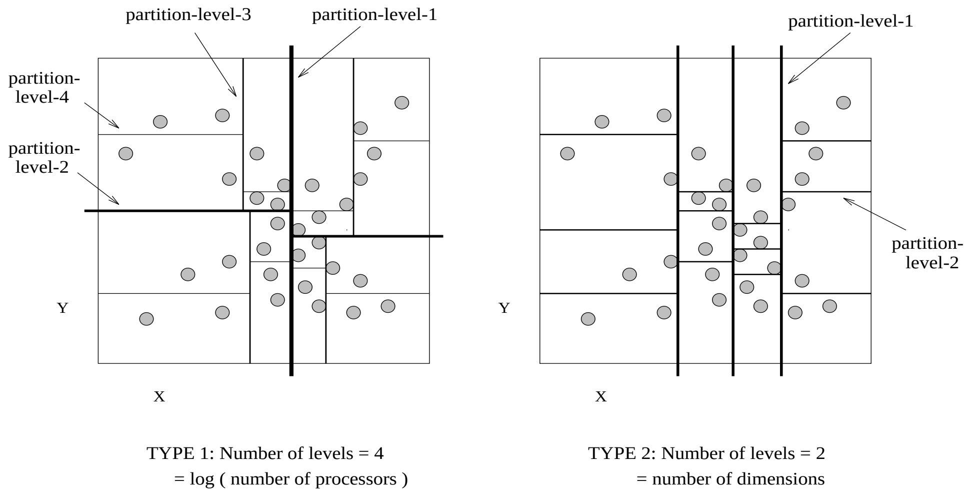


Figure 7.5: Types of Hierarchical Partitions
(The figure shows how a two dimensional space can be partitioned into sixteen regions using different types of hierarchical schemes.)

spatial objects is localized (say, at the corners), a TYPE-1 scheme is advantageous, since the load can be balanced just by adjusting partitions at levels 3 and 4. If the movement is global, and the objects are preferentially moving along one particular dimension, say along Y, a TYPE-2 scheme is advantageous, since only the partitions at level-2 need to be adjusted.

7.2 How to Load Balance

This section describes different methods of estimating the load distribution and partitioning the space into regions of approximately equal load.

7.2.1 Sorting vs. Discretization

There are two general schemes for partitioning the space into regions of equal load: one uses the list of objects sorted by their spatial coordinates [Belkhale & Banerjee 90] and the other discretizes the space into rectangular slots and uses the load estimate in each slot.

We describe the sorting method for a simple case where the space is partitioned along just one dimension, say along the X-axis. Let the processors $P_1, ..., P_N$ manage consecutive regions of space. Let us suppose that for balancing the load P_i needs to

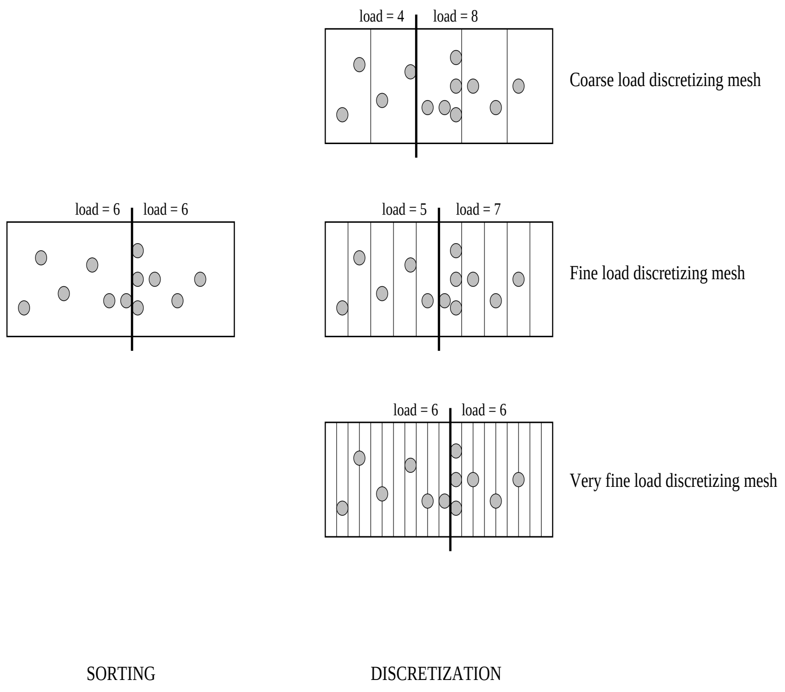

load = 4
load = 8
Coarse load discretizing mesh
load = 6
load = 6
load = 5
load = 7
Fine load discretizing mesh
load = 6
load = 6
Very fine load discretizing mesh
SORTING
DISCRETIZATION

Figure 7.6: Sorting vs. Discretization Methods for Load Balancing
(The figure illustrates how a one-dimensional space can be partitioned into two regions using sorting and discretization methods.)

send $n_{i,i-1}$ data objects to P_{i-1} and $n_{i,i+1}$ to P_{i+1}. P_i sorts the local data according to the x-coordinate, sends the first $n_{i,i-1}$ objects to P_{i-1} and the last $n_{i,i+1}$ objects to P_{i+1}. The sorting method uses continuous load distribution (the boundaries of the partitions can fall anywhere in space), and hence results in a fine load balance (Figure 7.6). However, this method incurs significant overhead to sort the data.

The overhead of sorting can be reduced by partially sorting the data using a *load discretizing mesh*. The space is discretized into a number of slots and the load estimate in each slot is used for partitioning the space. This approach imposes a restriction that the boundaries of the partitions must fall along the boundaries of the slots. In general, the finer the mesh the better the load balance and higher the load balancing cost (Figure 7.6). If the load distribution changes rapidly and the computation time per data item is small relative to the communication time, it is better to load balance quickly using a coarse mesh than to balance with a fine mesh. This approach is flexible, since we can control the fineness of the *load discretizing mesh* depending on how good a load balance is required. Hence we focus on this approach in our work.

7.2.2 Static vs. Adaptive Discretization

The fineness of the *load discretizing mesh* can be either fixed or adapted dynamically based on the distribution of the load. A static scheme cannot adjust to changes in the load density. It results in high load imbalance overhead if the mesh is too coarse and in high load balancing cost if the mesh is too fine. It is difficult, if not impossible, to choose the optimal fineness statically. An adaptive scheme can use a coarse mesh when the load density is low, and a finer mesh when the density becomes high, thereby minimizing the load balancing and load imbalance overheads. However, the adaptive scheme is harder to implement, since the load distribution needs to be monitored dynamically.

7.2.3 Uniform vs. Non-Uniform Discretization

In a *uniform discretization* scheme, the whole space is discretized into equal sized slots, and the load in each slot is estimated. This scheme has the following disadvantages: In many dynamic space-based applications, the load distribution changes slowly with time. If the load is balanced frequently, the change in the partitions from one load balancing step to another is small. In order to find a new partition, load distribution along the boundaries of the current regions is sufficient. The *uniform* scheme estimates load throughout the space, thereby generating a lot of unnecessary data. This scheme uses considerable time to produce this data, and a large amount of memory to store it. It also requires larger messages to communicate this data, thus incurring high communication cost.

A *non-uniform discretization* scheme estimates the load only along the boundaries of the regions (Figure 7.7). It uses a non-uniform load discretizing mesh to discretize the space. It is hard to choose an appropriate non-uniform mesh, since it's effectiveness is very sensitive to the load distribution and movement. If the load estimates are not sufficient, i.e., if the boundary region that is used to estimate the load is not sufficiently wide, this scheme incurs high load imbalance overhead. If the estimates are much more than necessary, this scheme suffers from the same disadvantages as those of the uniform scheme. In order to benefit from the non-uniform scheme, the load discretizing mesh must be adapted dynamically based on the load distribution

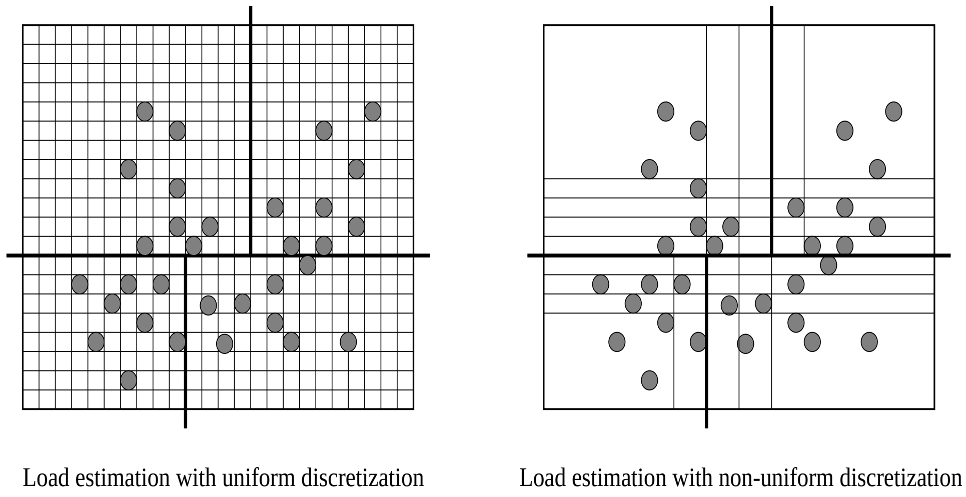

Load estimation with uniform discretization
Load estimation with non-uniform discretization

Figure 7.7: Uniform vs. Non-Uniform Discretization
(The figure shows how a two-dimensional space can be discretized using uniform and non-uniform meshes. The space is partitioned into four regions.)

and movement.

Although a *non-uniform, adaptive discretization* scheme is hard to implement, it has several advantages: It estimates the load only along the boundaries of the regions, thereby reducing the time required for load estimation. It requires only a small amount of memory and small messages to communicate the load information. Thus, it incurs less communication cost and is more scalable than the uniform scheme.

7.2.4 Centralized vs. Distributed Schemes

Once the load is estimated, the next step is to communicate the local load information to compute a new global partitioning of the space. In a *centralized* scheme, a designated central node collects information from all the nodes, computes the new partition and broadcasts the result. Although easy to implement, this scheme is not scalable because of two reasons: a large amount of memory is needed at the central node to store the global load information, and the central node becomes a bottleneck.

A *distributed* scheme removes this bottleneck by distributing the work across the processors. A hierarchical partitioning scheme induces a natural implementation of the distributed scheme for computing a new partition. We explain this using a simple example illustrated in Figure 7.8. The two dimensional space is partitioned into four

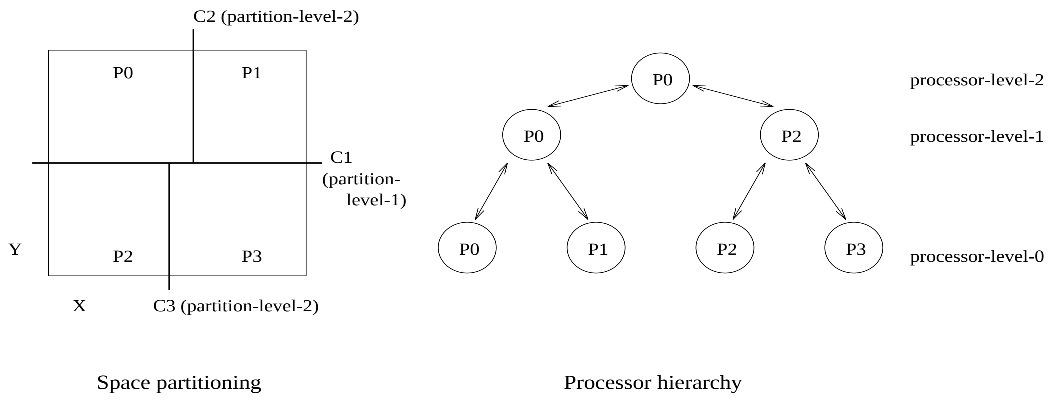


Figure 7.8: Distributed Scheme for Computing New Partition
(The figure shows an example of a two dimensional space partitioned onto four processors, and how the processors are organized into a hierarchy for communicating load estimates and for computing a new partition.)

regions using a hierarchical scheme. At level-1, the space is partitioned into two regions along Y dimension (using line C1). The resulting regions are called level-1 regions. At level-2, each level-1 region is partitioned into two level-2 regions along X (using lines C2 and C3). For collecting the load estimates and for computing a new partition, the processors are organized into a hierarchy, as shown in Figure 7.8. Processors P0 and P1 communicate to estimate the load in the region above line C1, and processors P2 and P3 estimate the load in the region below C1. Processor P0 is finally responsible for computing level-1 regions (i.e., for computing a new C1), and processors P0 and P2 are responsible for computing level-2 regions (lines C2 and C3).

Although the distributed scheme requires more communication steps than the centralized scheme, in practice, it is faster and scalable, since the amount of data communicated is much smaller, the computation is distributed across the processors, and the communication is performed in parallel.

7.2.5 Hierarchical Load Balancing

Hierarchical load balancing can be used to exploit preferential movement of the spatial objects. Consider the example given in Figure 7.8. If the objects are preferentially moving along X, the level-1 regions (partitioned by the line C1) need not be updated as often as the level-2 regions (partitioned by the lines C2 and C3), since very few objects move across the line C1. That is, most of the time, it is sufficient to balance the load globally by balancing just along X, thereby reducing the load balancing overhead.

7.3 When to Load Balance

The final issue in dynamic load balancing is when to load balance. In this section, we look at different schemes for determining the frequency of load balancing.

7.3.1 Fixed vs. Adaptive Frequency

In a *fixed frequency* scheme, the load is balanced every f time steps, for a fixed f. If f is too small, this scheme balances the load too often, thus incurring high load balancing overhead. On the other hand, if f is too large, it incurs high load imbalance overhead. The fixed frequency scheme has two disadvantages: it performs poorly if the load distribution changes unpredictably, and it is hard to choose statically an f that minimizes the sum of load balancing and load imbalance overheads.

An *adaptive frequency* scheme overcomes these disadvantages by dynamically adjusting f based on the execution characteristics. Different methods for adapting load balancing frequency are discussed below.

7.3.2 Methods for Adapting Frequency

There are three approaches to deciding when to load balance: *event driven*, *monitoring*, and *predictive*. All these methods attempt to minimize the sum of load balancing and load imbalance overheads per time step of the simulation, which we call the LILB overhead. Note that the load balancing cost is amortized over the interval in which load balancing is not done.

The *event driven* method [Walker 90] initiates load balancing when the load imbalance crosses a threshold. The threshold value is chosen to minimize LILB overhead. This method assumes that the load imbalance increases linearly, an assumption which does not hold most of time in real applications. Moreover, global communication is required for computing the load imbalance at every time step.

The *monitoring* method [Nicol & Saltz 88] monitors the load imbalance after each time step of the simulation, and initiates load balancing when it detects a minimal value of the LILB overhead in the current interval. This method is robust, but requires global synchronization for monitoring at every time step. The effect of the global synchronization on the performance is dependent on the application and the machine, and may or may not be significant.

We propose a *predictive* method (Chapter 10) that does not require monitoring. This method assumes that the average rate of increase of load imbalance does not vary much from one interval to the following interval. It uses the average rate from the previous interval to estimate the length of the next interval. This method is not as robust as the monitoring method, but does not require global synchronization.

7.4 Comparison with the Existing Load Balancing Systems

In Table 7.1, we compare *Ādhāra* with the following runtime systems with respect to their support for dynamic load balancing: PARTI, LPAR, DYNO and DIME.

7.5 Summary

In this chapter, we looked at the issues in application induced load balancing: how to partition the space, how to load balance and when to load balance. In the next chapter, we describe how *Ādhāra* chooses a partitioning scheme based on the application and machine characteristics.

Table 7.1: Comparison with the Existing Load Balancing Systems

	How is Partitioning Scheme Chosen?	How is Load Balanced?	When is Load Balanced?
PARTI	specified by the user; supports rectangular and non-rectangular regions, block/scatter decompositions	managed by the user	specified by the user
LPAR	specified by the user; supports rectangular and non-rectangular regions, block/scatter decompositions	managed by the user	specified by the user
DYNO	automatic & dynamic; specialized for irregular mesh applications	automatic	automatic
DIME	automatic & dynamic; specialized for irregular mesh applications	automatic	automatic
ADHARA	automatic & dynamic; rectangular regions; block decomposition; hierarchical; specialized for dynamic space-based applications	automatic load estimation by non-uniform, adaptive discretization of space; hierarchical	automatic; adaptive frequency predictive method

Chapter 8

CHOOSING A HIERARCHICAL PARTITIONING SCHEME BASED ON APPLICATION CHARACTERISTICS

Ādhāra partitions space into rectangular regions using a hierarchical block decomposition scheme. *Ādhāra* optimizes the load balancing overhead if there is preferential global load movement. It uses a TYPE-2 hierarchical scheme where the number of levels is equal to the number of dimensions (Section 7.1.3).

Ādhāra represents a partitioning scheme using four parameters:

1. Type of the decomposition (block, beam or slice)

2. Dimensions along which the space is decomposed (applicable for beam and slice decompositions)

3. Hierarchical order in which the dimensions are partitioned

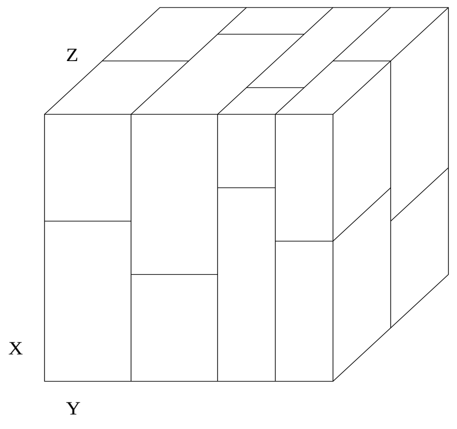
4. Number of partitions in each dimension

For example, BLOCK[YZX][4x2x2] represents a three-dimensional partitioning scheme where the space is partitioned into four regions first along Y, then into two regions along Z and finally into two regions along X. BEAM-Z[YX][4x4] represents a two-dimensional scheme where the space is partitioned into four regions first along Y and then into four regions along X (beams along Z dimension). Figure 8.1 illustrates these two schemes.

Ādhāra uses heuristics that are based on execution characteristics to automatically choose a good partitioning scheme. The following two sections describe the heuristics and the methods of extracting execution characteristics.

8.1 Heuristics for Choosing a Good Hierarchical Partitioning Scheme

Ādhāra's heuristics are based on the following parameters:



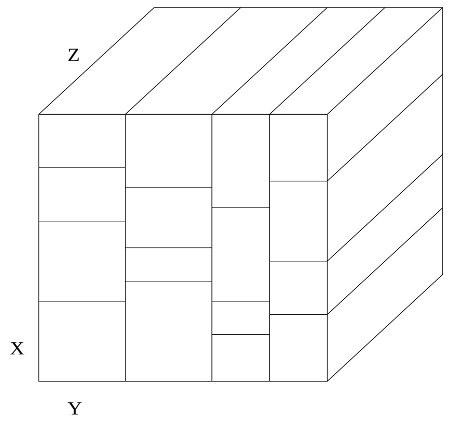


Figure 8.1: Examples of Partitioning Schemes

- Application execution characteristics, such as the communication patterns and the movement and distribution of spatial objects

- Number of processors allocated to the application

- Machine specific characteristics, such as the latency and bandwidth of message communication

A partitioning scheme is chosen by eliminating bad choices. The heuristics used by *Ādhāra* are given below:

- If the load movement along dimension D is much more than the movement in the other dimensions, then avoid partitioning along D. It is better not to partition along D because it results in higher communication and load balancing overheads, since there would be considerable movement across the partition boundaries.

- If the load density along dimension D [1] is much higher than the density along the other dimensions and if data is shared across partition boundaries along

[1] By load density along dimension D, we mean the density of the particles in the cross-section along D

D, then avoid partitioning along D, because the amount of data communicated along D is very high.

- If the number of processors allocated to the application is large, then avoid the slice scheme because thin slices result in high communication overhead for sharing the data along the boundaries.

- If there is large amount of data sharing along the partition boundaries, then avoid schemes (such as the Slice and Beam) that result in partitions with large surface area.

- If the latency of message communication is large compared to the transfer rate, then avoid the block scheme because it results in too many messages. On the other hand, if the transfer rate is small, then avoid the slice scheme because it results in large messages.

- If the load movement along dimension D1 is more than movement along dimension D2, then partition the space first along D2 and then along D1. If the movement along D2 is very small, then the load balancing overhead can be reduced by balancing the load only along D1.

8.2 Extracting Execution Characteristics

Ādhāra estimates the following parameters dynamically:

- Load movement along each dimension

- Load density along each dimension

The estimation method used by *Ādhāra* is described here. The coordinates of the spatial objects are discretized by a fine regular grid called the *CS-Mesh*. ('CS' stands for computation space.) The discretized coordinates of an object represent the grid-cell in which the object is located. *Ādhāra* measures the load movement in terms of the average distance (number of grid cells) traveled by the objects per time step. (Recall that the simulation proceeds in series of time steps). Load density along dimension D is measured in terms of the number of particles per slice along D (slices are induced by the CS-Mesh). *Ādhāra* takes a sample of the objects to compute the load movement and density in each dimension.

8.3 Parameters for the Heuristics

In the current implementation, we use the following parameters for making decisions:

- We say that A is much greater than B if $A > 2B$. We use this equation for comparing load movement, load density and data sharing along different dimensions.

- We avoid the slice scheme along dimension D, if the number of processors is greater than or equal to the number of cells along D. Here, cells refer to either the cells of a regular grid, or the cells used for partially sorting the particles.

- We say that the latency of message communication is large, if the latency is greater than the transfer time for a message of average length that is communicated to share data along the partition boundaries.

- For estimating load movement and density, we sample 10% of the objects.

8.4 Summary

In this chapter, we discussed the heuristics used by *Ādhāra* for choosing a good partitioning scheme based on the application execution characteristics and machine characteristics. In the following chapter, we describe *Ādhāra*'s dynamic load balancing scheme.

Chapter 9

DYNAMIC LOAD BALANCING USING NON-UNIFORM, ADAPTIVE DISCRETIZATION OF SPACE

Ādhāra discretizes the space by means of a fine, uniform, static grid called the *CS-Mesh*. The load is estimated using a load discretizing mesh, which we call the *LD-Mesh*. The LD-Mesh is represented using the CS-Mesh, as described in the following section. Throughout this chapter, by *CSM-Slots*, we refer to the slots induced by the CS-Mesh, and by *LDM-Slots*, we refer to the slots induced by the LD-Mesh.

This chapter is organized as follows: Section 9.1 describes how the non-uniform load discretizing mesh is represented using the CS-Mesh. Section 9.2 describes how the load is estimated using the LD-Mesh. Hierarchical load balancing using the LD-Mesh is explained in Section 9.3. Heuristics for adapting the LD-Mesh are given in Section 9.4. Section 9.5 describes how to take advantage of preferential load movement. Section 9.6 gives the adaptive distributed algorithm for hierarchical load balancing. Adaptation to a new partitioning scheme is discussed in Section 9.7.

9.1 Representing a Non-Uniform, Adaptive Load Discretizing Mesh

The LD-Mesh is defined by the following parameters:

- $Width_D$, for each dimension D
- $NSlots_D^+$ and $NSlots_D^-$, for each dimension D

Each processor maintains a portion of the LD-Mesh, as shown in Figure 9.1. $Width_D$ represents the width of the LDM-Slots with respect to the width of CSM-Slots along dimension D. (LDM-Slots are at most as fine as CSM-Slots.) $NSlots_D^+$ and $NSlots_D^-$ represent the number of LDM-Slots used along the positive and negative directions of a region, along dimension D.

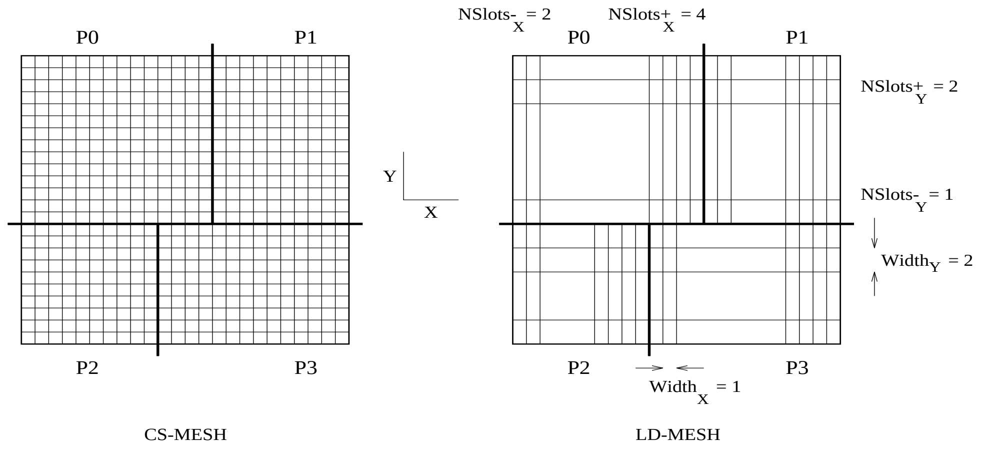


Figure 9.1: Parameters of the Non-Uniform Load Discretizing Mesh
(The figure illustrates how an LD-mesh is represented using the CS-Mesh. The two dimensional space is partitioned onto four processors.)

9.2 Estimating Load Distribution

Each processor estimates the load distribution along the boundaries of its region. To make the selection of boundary objects efficient, each processor partially sorts the objects into two bins (Figure 9.2). The *exterior* bin contains the boundary objects whose coordinates need to be discretized, and the *interior* bin contains the non-boundary objects whose coordinates need not be discretized. Each processor sorts the objects a time step before load balancing, at the time of checking the coordinates

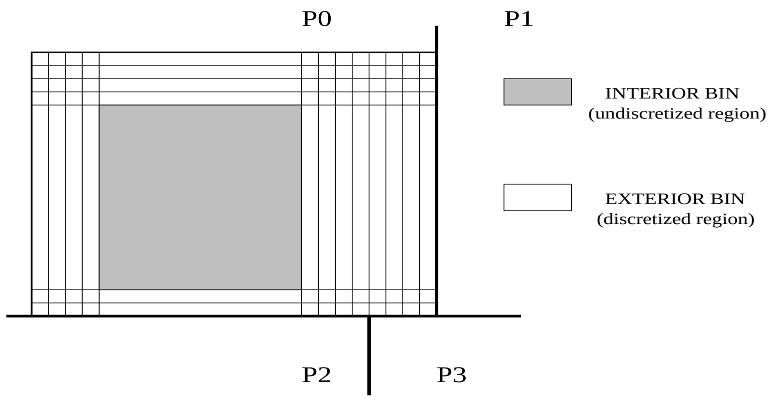


Figure 9.2: Using Bins to Separate Boundary and Non-Boundary Objects

of the objects for possible movement out of its assigned region [1].

The load is discretized along the dimension in which it is going to be balanced. For example, consider a two dimensional space partitioned into four regions using a BEAM[YX] partitioning scheme (Figure 9.3). The load is first balanced along Y, and then balanced along X. In the first step, each processor discretizes the load along Y by partially sorting the boundary objects into the LDM-Slots along the Y dimension. The resulting load estimates computed by the four processors are shown in Figure 9.3. The figure also shows the load estimates along the X dimension. These estimates are used if the load is balanced only along X (see Section 9.5).

9.3 Hierarchical Load Balancing

We explain the distributed hierarchical load balancing algorithm using the simple example illustrated in Figure 9.4. In this example, a two dimensional space is partitioned into four regions using a BEAM[YX][2x2] partitioning scheme. The line C1 divides the space along the Y dimension into two level-1 regions, and the lines C2 and C3 divide the level-1 regions along the X dimension into four level-2 regions. The four regions are assigned to processors P0, P1, P2 and P3, as shown in the figure.

The load is balanced hierarchically, in two steps. In the first step, the load is balanced along Y by moving the line C1 so that the computational load in each level-1 region is approximately the same. In the second step, the load is balanced along X by moving the lines C2 and C3 so that the computational loads in the level-2 regions are approximately equal.

In each step, the computation and communication is distributed across the machine by making use of the processor hierarchy induced by the hierarchical partitioning scheme (Figure 9.5). Each processor at *processor-level-0* represents the level-2 region (created by the lines C1 and C2/C3) that it is assigned. Processors P0 and P2 at *processor-level-1* represent the level-1 regions created by the line C1, and are responsible for updating C2 and C3 for balancing the load along X. Processor P0 at *processor-level-2* represents the whole space, and is responsible for updating C1 to balance the load along Y.

The distributed algorithm is given in Figures 9.6 and 9.7, and is explained below.

[1] When an object moves from the region of processor P1 to the region of processor P2, P1 needs to send the object to P2 to maintain spatial locality.

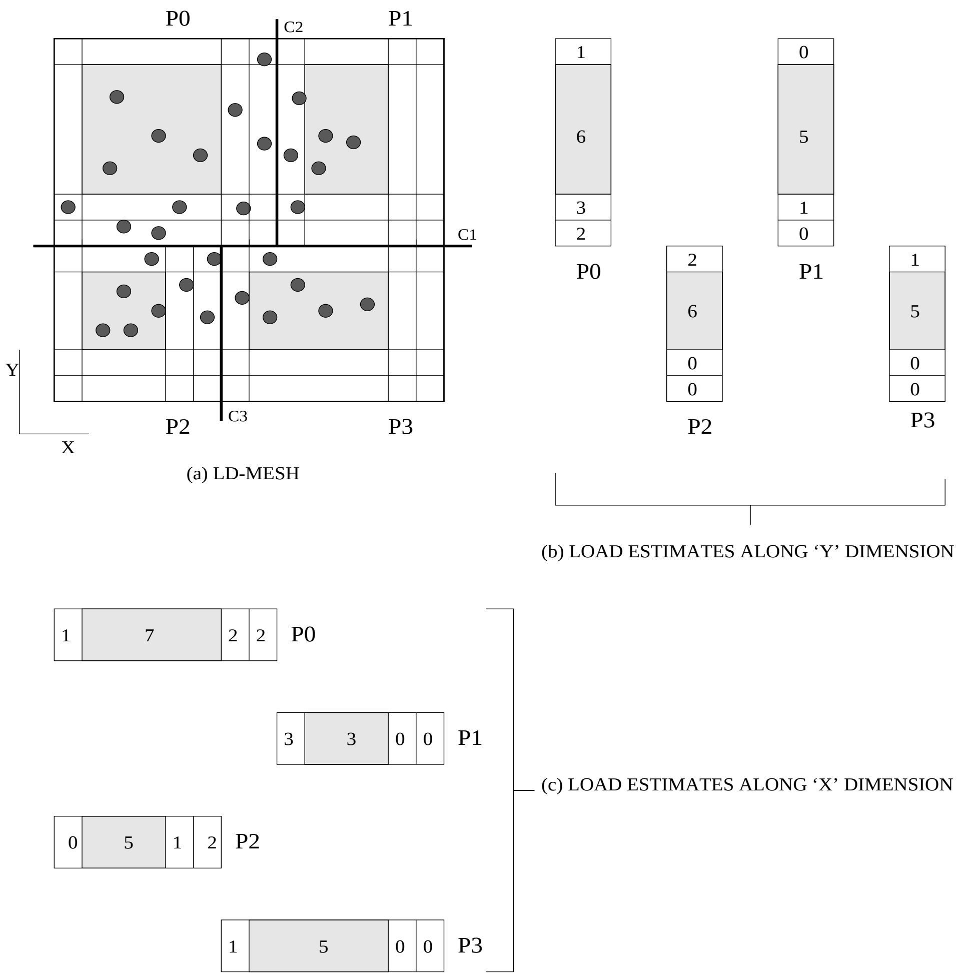


Figure 9.3: Load Estimation using a Non-Uniform Load Discretizing Mesh
(The figure illustrates how each processor in a four-processor system estimates the load using a load discretizing mesh. The two dimensional space is partitioned using a BEAM[YX][2x2] scheme. In this example, each particle represents unit load.)

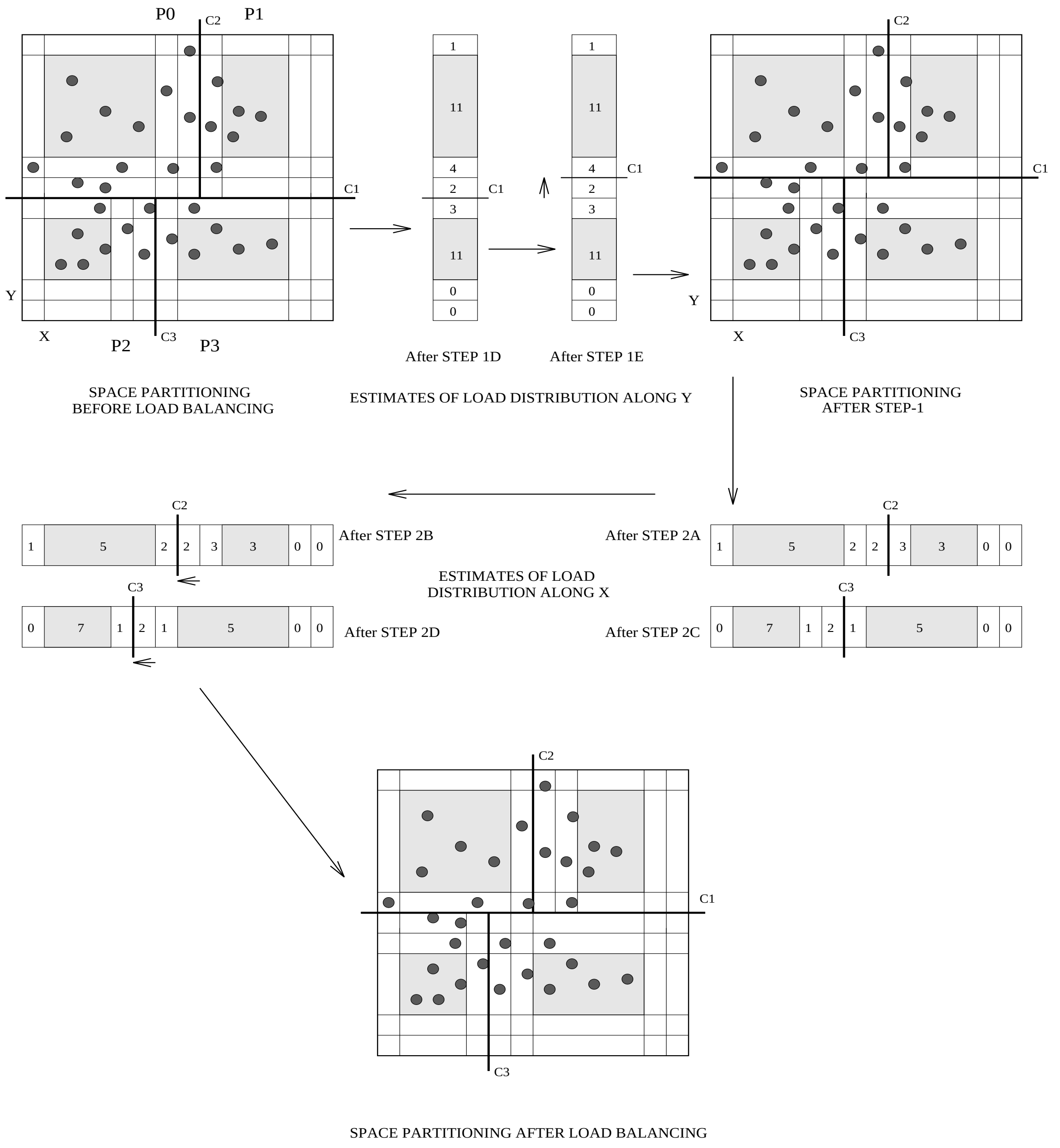

P0
P1
C1
C2
C3
P2
P3
X
Y
After STEP 1D
After STEP 1E
SPACE PARTITIONING BEFORE LOAD BALANCING
ESTIMATES OF LOAD DISTRIBUTION ALONG Y
SPACE PARTITIONING AFTER STEP-1
After STEP 2A
After STEP 2B
After STEP 2C
After STEP 2D
ESTIMATES OF LOAD DISTRIBUTION ALONG X
SPACE PARTITIONING AFTER LOAD BALANCING

Figure 9.4: Hierarchical Load Balancing

(This figure illustrates hierarchical load balancing using a non-uniform load discretizing mesh. A two dimensional space is partitioned onto four processors using a BEAM[YX][2x2] partitioning scheme. In this example, each particle represents unit load.)

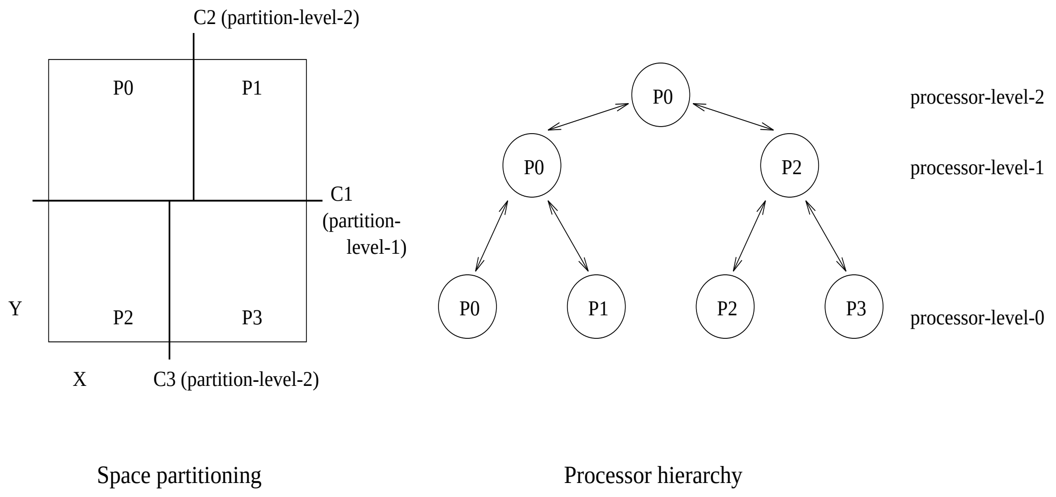


Figure 9.5: Processor Hierarchy for Distributed Load Balancing
(The figure shows an example of a two dimensional space partitioned onto four processors, and how the processors are organized into a hierarchy for communicating load estimates and for computing a new partition.)

In Step 1, each processor estimates the load distribution along Y. The resulting load estimates (after STEP 1A of the algorithm) are shown in Figure 9.3. Processor P0(P2) at processor-level-1 combines the estimates computed by P0(P2) and P1(P3) using the addition operation (STEP 1B(1C)). Processor P0 at processor-level-2 combines the resulting estimates using the concatenate operation (STEP 1D) and computes a new C1 that divides the space into level-1 regions of approximately equal load (STEP 1E). The result of these two operations is shown in Figure 9.4. The new C1 is broadcast using the processor hierarchy. The processors then exchange data to adjust to the new level-1 regions.

In Step 2, each processor estimates the load distribution along X. Processor P0(P2) at processor-level-1 combines the estimates computed by P0(P2) and P1(P3) using the concatenate operation (STEP 2A(2C)) and computes new C2(C3) that divides the level-1 region into level-2 regions of approximately equal load. Processor P0 then combines the new partition information ({C2,C3}) and broadcasts using the processor hierarchy. The processors then exchange data to adjust to the new level-2 regions.

```
/*
   STEP 1: UPDATE LEVEL-1 REGIONS (UPDATE LINE C1)
   NOTE: Pn(L-m) means processor 'n' at level-'m'
         Px means all processors
*/

Px(L-0): estimate local load distribution along Y           .. STEP 1A

P1(L-0): send load estimates to P0
P3(L-0): send load estimates to P2

    P0(L-1): recv load estimates from P1
             add estimates of P0 and P1                     .. STEP 1B

    P2(L-1): recv load estimates from P3
             add estimates of P2 and P3                     .. STEP 1C
             send level-1 estimates to P0(L-2)

        P0(L-2): recv level-1 estimates from P2
                 concatenate level-1 estimates of P0 and P2 .. STEP 1D
                 compute new C1 (level-1 regions)           .. STEP 1E
                 send C1 to P2(L-1)

    P0(L-1): send C1 to P1(L-0)
    P2(L-1): recv C1 from P0(L-2)
             send C1 to P3(L-0)

P1(L-0): recv C1 from P0
P3(L-0): recv C1 from P2

Px(L-0): exchange data to move C1
```

Figure 9.6: Distributed Algorithm for Hierarchical Load Balancing: STEP 1
(This figure gives STEP 1 of the hierarchical load balancing algorithm for a simple case where a two dimensional space is partitioned into four regions using a BEAM[YX][2x2] partitioning scheme. The result of executing this algorithm on a specific example is given in Figure 9.4.)

```
/*
   STEP 2: UPDATE LEVEL-2 REGIONS (UPDATE LINES C2 AND C3)
   NOTE: Pn(L-m) means processor 'n' at level-'m'
         Px means all processors
*/


Px(L-0): estimate local load distribution along X

P1(L-0): send load estimates to P0
P3(L-0): send load estimates to P2

    P0(L-1): recv load estimates from P1
             concatenate estimates of P0 and P1         .. STEP 2A
             compute new C2 (level-2 regions)           .. STEP 2B

    P2(L-1): recv load estimates from P3
             concatenate estimates of P2 and P3         .. STEP 2C
             compute new C3 (level-2 regions)           .. STEP 2D
             send C3 to P0(L-2)

        P0(L-2): recv C3 from P2
                 send {C2,C3} to P2

    P0(L-1): send {C2,C3} to P1(L-0)
    P2(L-1): recv {C2,C3} from P0(L-2)
             send {C2,C3} to P3(L-0)

P1(L-0): recv {C2,C3} from P0
P3(L-0): recv {C2,C3} from P2

Px(L-0): exchange data to move C2/C3
```

Figure 9.7: Distributed Algorithm for Hierarchical Load Balancing: STEP 2
(This figure gives STEP 2 of the hierarchical load balancing algorithm for a simple case where a two dimensional space is partitioned into four regions using a BEAM[YX][2x2] partitioning scheme. The result of executing this algorithm on a specific example is given in Figure 9.4.)

9.4 Adjusting Parameters of the Load Discretizing Mesh Based on Application Characteristics

The parameters of the LD-Mesh must be chosen so as to minimize the amount of memory and time required for load balancing, while achieving good load balance.

Good load balance can be achieved by choosing a load discretizing mesh that is fine enough and by discretizing a sufficiently wide boundary region. Memory usage can be minimized by discretizing just the necessary amount of the boundary region. The time required for load balancing can be minimized by reducing the number of load estimates, i.e., by using as few and as wide LDM-Slots as possible.

Ādhāra dynamically adjusts the parameters of the LD-Mesh based on the load movement and density. The following subsections describe how the parameters are adjusted by *Ādhāra*.

9.4.1 Adjusting the size of the boundary region to discretize

The boundary region discretized along dimension D is specified by the $Width_D$, $NSlots_D^-$ and $NSlots_D^+$ parameters. The following paragraph describes how *Ādhāra* adjusts these parameters for a particular dimension (the dimension suffix is omitted).

When the load is balanced, partition boundaries (for example, line C1 in Figure 9.8) tend to move in the direction of load movement. Hence *Ādhāra* adjusts the $NSlots$ parameters based on the maximum number of slots used for moving the partition boundaries. We explain the heuristics used by *Ādhāra* by means of the example shown in Figure 9.8. As a result of load balancing, the partition boundary C1 moved by two slots in the negative direction, C2 by one slot in the positive direction and C3 by one slot in the negative direction. The maximum number of $NSlots^+$ used is 2 (due to C1) and the maximum number of $NSlots^-$ used is 1 (due to C2). Since the current $NSlots^+$ is 2, and they are used up for moving C1, *Ādhāra* predicts that more $NSlots^+$ might be used the next time the load is balanced. Hence, the $NSlots^+$ parameter is increased from 2 to 4. Similarly, since the current $NSlots^-$ is 4 and only one of them is used up, *Ādhāra* decreases the $NSlots^-$ parameter from 4 to 2. The heuristic algorithm to adjust the $NSlots$ parameters is given in Figure 9.9.

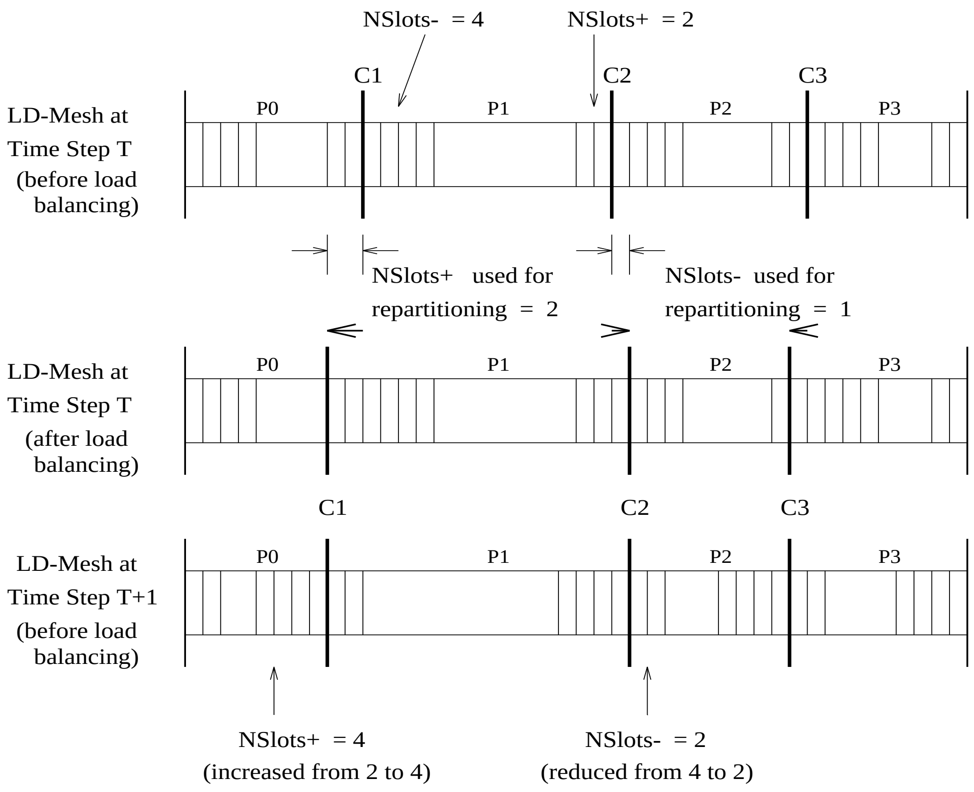

NSlots- = 4
NSlots+ = 2
C1
C2
C3
LD-Mesh at Time Step T (before load balancing)
P0
P1
P2
P3
NSlots+ used for repartitioning = 2
NSlots- used for repartitioning = 1
LD-Mesh at Time Step T (after load balancing)
C1
C2
C3
LD-Mesh at Time Step T+1 (before load balancing)
NSlots+ = 4 (increased from 2 to 4)
NSlots- = 2 (reduced from 4 to 2)

Figure 9.8: Adjusting Parameters of the Load Discretizing Mesh

(The figure shows how the NSlots parameters of the load discretizing mesh are adjusted based on the application characteristics. A one-dimensional space is used for simplicity of illustration. The space is partitioned into four regions.)

9.4.2 Adjusting fineness of the load discretizing mesh based on the desired quality of load balance

The fineness of the LD-Mesh in dimension D is specified by the $Width_D$ parameter. The *threshold_load_per_LDM_slot* parameter, which is the limit on the total computational load per LDM-Slot along dimension D, represents the desired quality of load balance. (*Ādhāra* sets this threshold to 0.5 percent of the total load.)

Ādhāra adjusts the $Width$ parameter based on the load density. Load density is represented by $max_i\{load\_per\_LDM\_slot_i\}$. Note that only the load distribution along the partition boundary is considered for estimating the load density. Consider the example given in Figure 9.4. From the estimates of the load distribution along Y (computed at STEP 1D of the load balancing algorithm), we can see that the load density is 4 (maximum of the load in the LDM-Slots along Y: 1, 4, 2, 3, 0, and 0).

Ādhāra adapts the $Width$ such that $max_i\{load\_per\_LDM\_slot_i\}$ is approximately equal to the *threshold_load_per_LDM_slot*. Once the boundary region is fixed by adjusting the $NSlots$ parameters (as described in the previous subsection), when the $Width$ is increased(decreased), the $NSlots$ parameter must be appropriately decreased(increased), as given in the heuristic algorithm (Figure 9.9).

9.5 Taking Advantage of Preferential Load Movement

If the load moves preferentially along a particular dimension, say X, and if there is very little movement along Y, then the load along Y does not need to be balanced as often as it is balanced along X. That is, most of the time, the space can be repartitioned just by balancing the load along X.

Consider the example given in Figure 9.4. If there is very little movement along Y, the line C1 does not need to be updated; it is sufficient to update C2 and C3 to balance the load. In this case, STEP 1 of the load balancing algorithm can be omitted, thereby reducing the load balancing overhead.

9.6 Adaptive Distributed Algorithm for Hierarchical Load Balancing

Here we describe the distributed algorithm that not only balances the load but also adapts the load discretizing mesh and the partitioning hierarchy. This algorithm is

```
ADJUST BOUNDARY REGION:
    let $2^i \leq NSlots_{used} < 2^{i+1}$.
    if ($NSlots_{used} < 1.5 * 2^i$)
        $NSlots = 2^{i+1}$;
    else
        $NSlots = 2^{i+2}$;

ADJUST FINENESS OF THE LD-MESH:
    $density = max_i\{load\_per\_LDM\_slot_i\}$;
    $limit = threshold\_load\_per\_LDM\_slot$;
    if ($density > limit$)
        while ($density > limit$) {
                $Width = Width/2$;
                $density = density/2$;
                $NSlots^+ = NSlots^+ * 2$;
                $NSlots^- = NSlots^- * 2$;
        }
    else
        while ($density < limit$) {
                $Width = Width * 2$;
                $density = density * 2$;
                $NSlots^+ = NSlots^+/2$;
                $NSlots^- = NSlots^-/2$;
        }
```

Figure 9.9: Heuristic Algorithm for Adjusting Parameters of the Load Discretizing Mesh

a modified version of the algorithm discussed in Section 9.3 (the code is given in Figures 9.6 and 9.7.

STEP 1 and STEP 2 of the new algorithm are given in Figures 9.10 and 9.12. STEP 2 of the algorithm uses a heuristic procedure to adapt the partitioning type, hierarchy and the dimensions in which the load needs to be balanced at the next balancing step. This heuristic procedure is given in Figure 9.11.

At STEP 1A, the processors estimate the load distribution as well as the load movement and density (together called as the *load_info*). At STEP 1B+(1C+), processor P0(P2) combines the load characteristics of P0(P2) and P1(P3). Processor P0 combines the resulting load characteristics at STEP 1D+ and adjusts the LD-Mesh parameters along Y (STEP 1E+) and broadcasts the new parameters.

Similarly, in STEP 2, LD-Mesh parameters along X are adjusted by processor P0 (see steps 2A+, 2C+ and 2E+). At STEP 2F+, P0 uses the heuristic procedure (Figure 9.11) to determine if the partitioning scheme needs to be adjusted. The runtime system adapts to the new partitioning scheme after the completion of the current time step, as discussed in the following section.

9.7 Adapting to a New Partitioning Scheme

To adapt to a new partitioning scheme, the runtime system determines a new space partitioning and redistributes the data. The existing non-uniform load discretizing mesh may not be suitable for estimating the load distribution because the partition boundaries may move substantially. Hence, a *uniform load discretizing mesh*, which estimates the load throughout the space, is used for balancing the load. Once the system adapts to the new partitioning scheme, a new non-uniform load discretizing mesh is created and then adapted in the subsequent time steps.

9.8 Summary

In this chapter, we described the novel hierarchical load balancing scheme implemented in the *Ādhāra* runtime system. In the following chapter, we describe the *predictive method* for adapting the frequency of load balancing.

```
/* STEP 1: UPDATE LEVEL-1 REGIONS (UPDATE LINE C1)
   Pn(L-m) means processor 'n' at level-'m', Px means all processors
   load_info means load estimates and characteristics (movement and density)
   LD-Mesh-Y means parameters of the LD-Mesh along dimension Y
*/

   If the load needs to be balanced only along X, then go to STEP 2

   Px(L-0): estimate local load_info along Y                  .. STEP 1A
   P1(L-0): send load_info to P0
   P3(L-0): send load_info to P2

       P0(L-1): recv load_info from P1
                add load estimates of P0 and P1               .. STEP 1B
                combine load characteristics of P0 and P1     .. STEP 1B+

       P2(L-1): recv load_info from P3
                add load estimates of P2 and P3               .. STEP 1C
                combine load characteristics of P2 and P3     .. STEP 1C+
                send level-1 load_info to P0(L-2)

           P0(L-2): recv level-1 load_info from P2
                    concatenate level-1 estimates of P0 and P2 .. STEP 1D
                    combine load characteristics of P0 and P2  .. STEP 1D+
                    compute new C1 (level-1 regions)           .. STEP 1E
                    adjust LD-Mesh-Y                           .. STEP 1E+
                    send C1 and new LD-Mesh-Y to P2(L-1)

       P0(L-1): send C1 and new LD-Mesh-Y to P1(L-0)
       P2(L-1): recv C1 and new LD-Mesh-Y from P0(L-2)
                send C1 and new LD-Mesh-Y to P3(L-0)

   P1(L-0): recv C1 and new LD-Mesh-Y from P0
   P3(L-0): recv C1 and new LD-Mesh-Y from P2

   Px(L-0): exchange data to move C1 and update LD-Mesh-Y
```

Figure 9.10: Adaptive Algorithm for Hierarchical Load Balancing: STEP 1

```
procedure_decide_what_to_adapt() {

    Use heuristics (given in Chapter 8) to determine if the
                    partitioning scheme needs to be changed

    if the type of partitioning scheme (BEAM/SLICE) needs to be changed {
        Determine new partitioning scheme
        At the next load balancing step, balance the load along
               all the dimensions in which the space is partitioned
    }
    else { /* no change in the type of partitioning scheme */

        if (load movement along Y >> load movement along X) {
            Change the hierarchy from [YX] to [XY]
            At the next load balancing step, adapt to the new hierarchy
                          and balance the load along X and Y
        }
        else {
            if there is preferential movement along X
                    and very little movement along Y
                    and the current C1 balances the load well along Y {
                At the next load balancing step, balance the load only along X
            }
            else /* no change */
                At the next balancing step, balance the load along Y and X
        }
    }
    Store the result in info_what_to_adapt
}
```

Figure 9.11: Heuristic Procedure for Adapting Partitioning Scheme and the Dimensions in which Load Needs to be Balanced
(This figure gives the heuristic procedure for a simple case where a two dimensional space is partitioned into four regions using a BEAM[YX][2x2] partitioning scheme.)

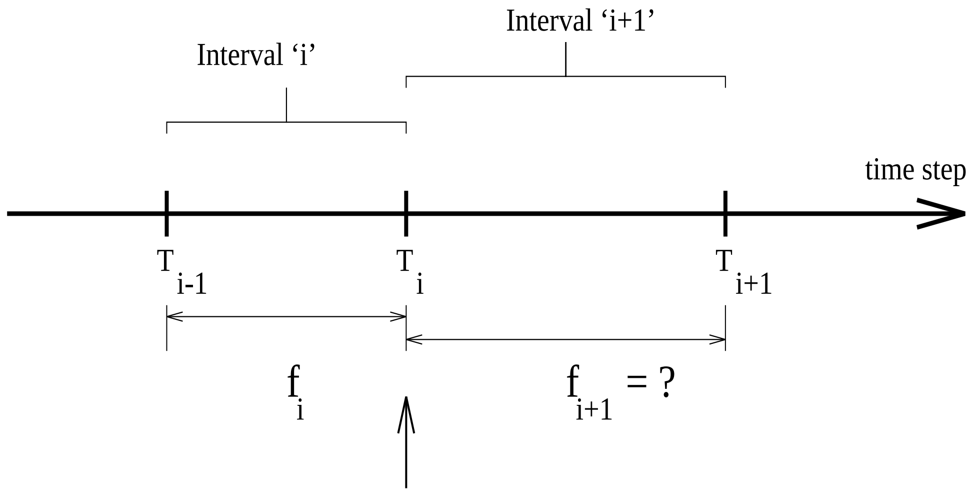
```
/* STEP 2: UPDATE LEVEL-2 REGIONS (UPDATE LINES C2 AND C3) */
   Pn(L-m) means processor 'n' at level-'m', Px means all processors
   load_info means load estimates and characteristics (movement and density)
   LD-Mesh-X means parameters of the LD-Mesh along dimension X
*/

Px(L-0): estimate local load_info along X
P1(L-0): send load_info to P0
P3(L-0): send load_info to P2

    P0(L-1): recv load_info from P1
             concatenate load estimates of P0 and P1        .. STEP 2A
             combine load characteristics of P0 and P1      .. STEP 2A+
             compute new C2 (level-2 regions)               .. STEP 2B

    P2(L-1): recv load_info from P3
             concatenate load estimates of P2 and P3        .. STEP 2C
             combine load characteristics of P2 and P3      .. STEP 2C+
             compute new C3 (level-2 regions)               .. STEP 2D
             send C3 and level-1 load characteristics to P0(L-2)

        P0(L-2): recv C3 and level-1 load characteristics from P2
                 combine load characteristics of P0 and P2
                 adjust LD-Mesh-X                           .. STEP 2E+
                 procedure_decide_what_to_adapt()           .. STEP 2F+
                 send {C2,C3}, new LD-Mesh-X and info_what_to_adapt to P2

    P0(L-1): send {C2,C3}, new LD-Mesh-X and info_what_to_adapt to P1(L-0)
    P2(L-1): recv {C2,C3}, new LD-Mesh-X and info_what_to_adapt from P0(L-2)
             send {C2,C3}, new LD-Mesh-X and info_what_to_adapt to P3(L-0)

P1(L-0): recv {C2,C3}, new LD-Mesh-X and info_what_to_adapt from P0
P3(L-0): recv {C2,C3}, new LD-Mesh-X and info_what_to_adapt from P2

Px(L-0): exchange data to move C2/C3 and update LD-Mesh-Y
         after finishing the computation in the current time step,
           adapt to the new partitioning scheme based on info_what_to_adapt
```

Figure 9.12: Adaptive Algorithm for Hierarchical Load Balancing: STEP 2

Chapter 10

PREDICTIVE METHOD FOR ADAPTING FREQUENCY OF LOAD BALANCING

Ādhāra uses a *predictive method* for deciding when to balance the load. This method attempts to minimize the sum of load balancing and load imbalance overheads per time step of the simulation, which we call the LILB overhead. If the load is balanced too often, then the load balancing overhead is very high, and if the load is balanced too infrequently, then the load imbalance overhead is very high. The predictive method dynamically adjusts the frequency of load balancing to balance these two overheads together.



Let us suppose that the load is balanced at time steps T_{i-1} and T_i. Let f_i represent the interval-i in which the load is not balanced ($f_i = T_i - T_{i-1}$). At time step T_i, the predictive method determines the next interval f_{i+1} based on the load balancing cost incurred at T_i and load imbalance in the interval-i. This method assumes that the load imbalance in interval-$(i+1)$ changes in the same way as it did in the interval-i.

Let R_{LI} be the average rate of increase of load imbalance in the interval-i, and L_i be the load imbalance at time step T_i just after the load is balanced. Let C_{LB} be the load balancing overhead at time step T_i. Let $C_{compute}$ be the computation time per

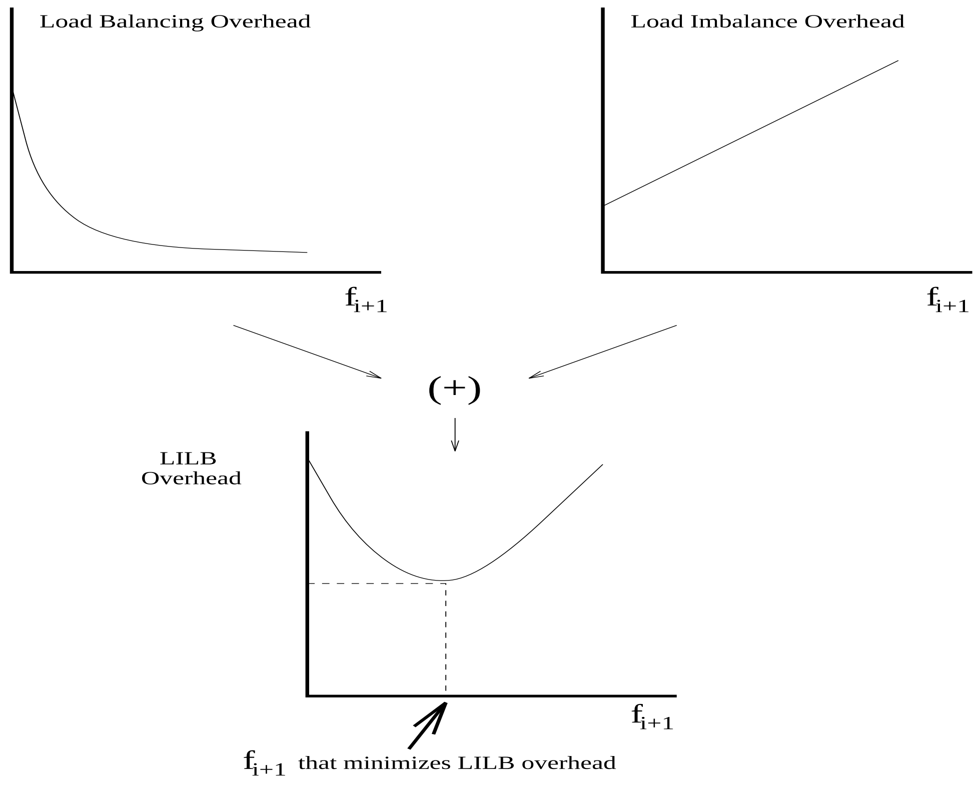


Figure 10.1: Effect of the length of the load balancing interval on load balancing and load imbalance overheads

object (per unit load). Then

Predicted load balancing overhead per time step in interval-$(i+1)$

$$= \frac{C_{LB}}{f_{i+1}}$$

Predicted average load imbalance overhead per time step in interval-$(i+1)$

$$= C_{compute}(L_i + \frac{R_{LI} f_{i+1}}{2})$$

Predicted LILB overhead in interval-$(i+1)$

$$= \frac{C_{LB}}{f_{i+1}} + C_{compute}(L_i + \frac{R_{LI} f_{i+1}}{2})$$

Figure 10.1 shows how the LILB overhead is predicted to vary with f_{i+1}. The goal of the predictive method is to determine an f_{i+1} that minimizes the LILB overhead. The value of f_{i+1} that minimizes the above expression can be shown to be

$$f_{i+1} = \sqrt{\frac{2C_{LB}}{C_{compute} R_{LI}}}$$

The cost parameters C_{LB} and $C_{compute}$ are estimated dynamically by measuring the time taken for balancing the load and the time taken for the actual computation. The rate R_{LI} is estimated by taking the difference in the load imbalance at the beginning and the end of the previous interval, and dividing by f_i.

This method does not require global synchronization at very time step. However, it cannot handle drastic changes in the load distribution. In dynamic space-based simulations, most of the time, load distribution changes gradually. Sudden changes occur only when new objects enter the system, for example, when a cluster of atoms drop onto a crystal in a molecular dynamics simulation. The application programmer knows about such changes, so she can inform *Ādhāra* so that the runtime system can balance the load as soon as drastic load changes occur.

Summary

In the last three chapters, we described *Ādhāra*'s approach to automatic data partitioning and dynamic load balancing. In the following chapter, we discuss some implementation details, and in chapter 12, we present some performance results.

Chapter 11

IMPLEMENTATION OF THE PORTABLE RUNTIME SYSTEM

In this chapter, we describe how the *Ādhāra* runtime system is implemented. This chapter is organized as follows: Section 11.1 describes the organization of the *Ādhāra* runtime software. Portability of the runtime system is discussed in Section 11.2. Some important implementation issues are given in Section 11.3. Sections 11.4- 11.9 describe the software modules.

11.1 Design of Ādhāra *Runtime System Software*

The *Ādhāra* runtime system is implemented as a C-library. The application program, which is written using the *Ādhāra* programming constructs, is first converted into C-code by a pre-processor. The resulting code is then compiled and linked with the *Ādhāra* runtime library and the system dependent message passing library to produce an executable parallel code (Figure 11.1).

Figure 11.2 shows how the *Ādhāra* runtime software is organized into various modules. The application communicates with the runtime system through the *application interface* module. The runtime system and the operating system communicate through the *operating system interface* module. The main functions of the runtime system are handled by six modules: *message handler, phase manager, spatial data manager, data redistributor, load balancer and processor reallocation adapter.* These modules are described in Sections 11.4- 11.9.

11.2 Portability

The runtime system assumes simple message passing support from the operating system. This support is provided either directly by the operating system (on non-shared memory machines such as the Intel Paragon and nCUBE) or by message passing interfaces such as the PVM (on networks of workstations and shared memory

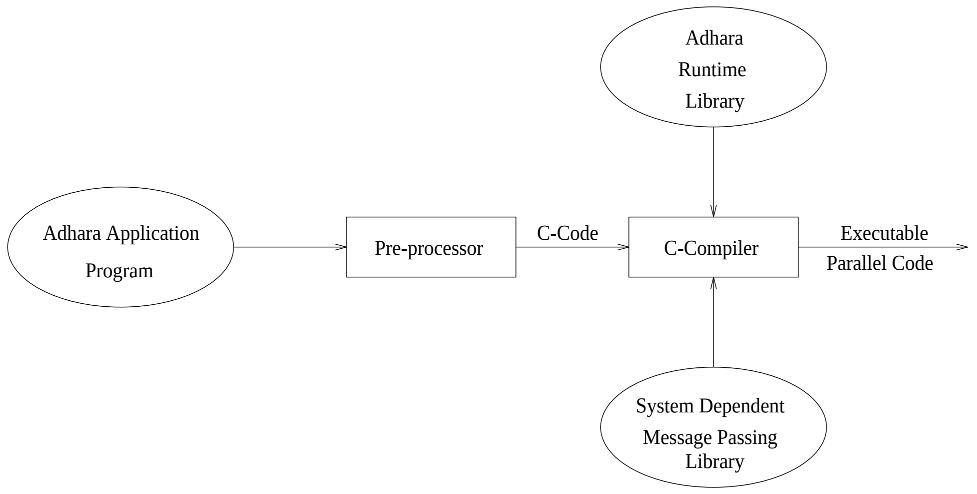


Figure 11.1: Compiling an $\bar{A}dh\bar{a}ra$ Program

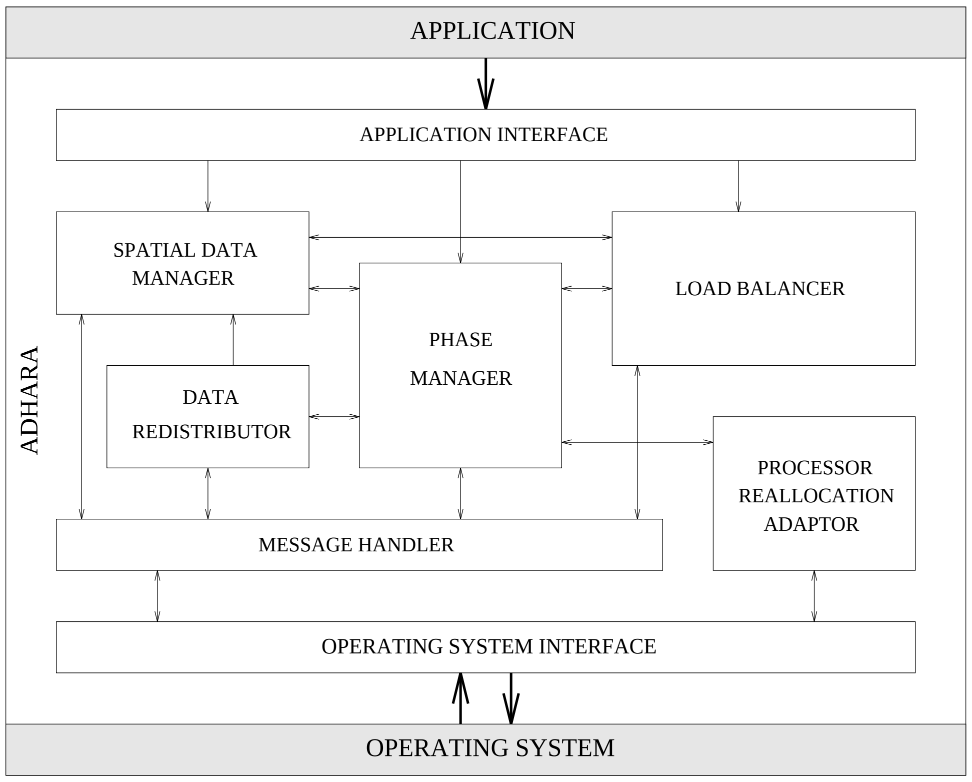


Figure 11.2: Organization of the $\bar{A}dh\bar{a}ra$ Runtime Software

machines such as the KSR-1). *Ādhāra* expects support for the following machine dependent functions:

- Load an executable code on a set of processors
- Name the processes for sending and receiving messages
- Send an asynchronous message of given type to a specific process
- Receive messages asynchronously
- Poll and read in the message of a specific type sent by a specific process
- Combine distributed data using global operations such as ADD and MAX

The *operating system interface* module of the runtime system provides a machine independent interface to these functions, so this is the only module that is machine dependent. The code required to provide this interface is less than 2% of the runtime system code, and it is straightforward to port this module to different machines. The remaining modules do not use any system dependent functions, so they are portable across different machines.

Ādhāra automatically tunes the performance of an application not only based on the application execution characteristics but also based on the machine characteristics such as the latency and transfer rate of message communication and the relative processor speed. Hence *Ādhāra* is portable both in terms of adapting the code and the performance.

11.3 Implementation Issues

Ādhāra uses a very fine, regular, static grid called the *CS-Mesh* to discretize the space. It uses this mesh for:

- discretizing coordinates of the spatial objects
- representing space partitioning
- representing the load discretizing mesh that is used for load balancing

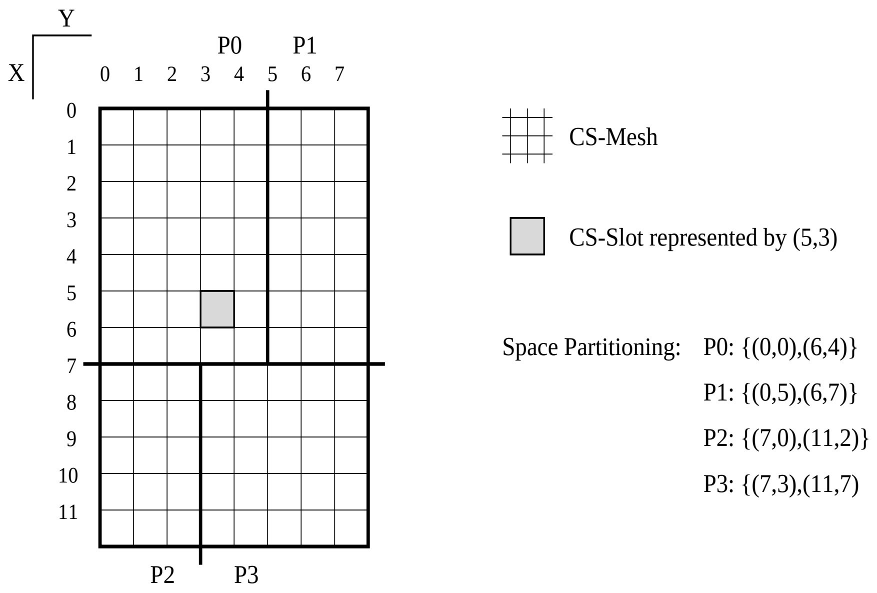


Figure 11.3: Representing Space Partitioning of a Two Dimensional Space: An Example
(This figure given an example of a two dimensional space partitioned onto four processors P0-P3. The space is discretized by a 12 x 8 CS-Mesh.)

Space partitioning is represented by *PartitionArray*, an array of sub-cubes which are defined using the discretized coordinates. Let the size of the *CS-Mesh* be $CS_X \times CS_Y \times CS_Z$. The region assigned to node i is given by

PartitionArray[i]: $\{(X1_i, Y1_i, Z1_i), (X2_i, Y2_i, Z2_i)\}$,

where $0 \leq D1_i \leq D2_i < CS_D$ for $D = X, Y, Z$. $(X1_i, Y1_i, Z1_i)$ and $(X2_i, Y2_i, Z2_i)$ specify the diagonally opposite corners of the subcube assigned to node i. Figure 11.3 illustrates a two dimensional example.

11.4 Message Handler

The *message handler* module provides support for high-level message passing functions tailored to the needs of dynamic space based applications. The features supported by this module are listed below.

- **Combining messages:** It buffers up requests for sending/receiving data and combines messages going to the same processor, thereby reducing the number

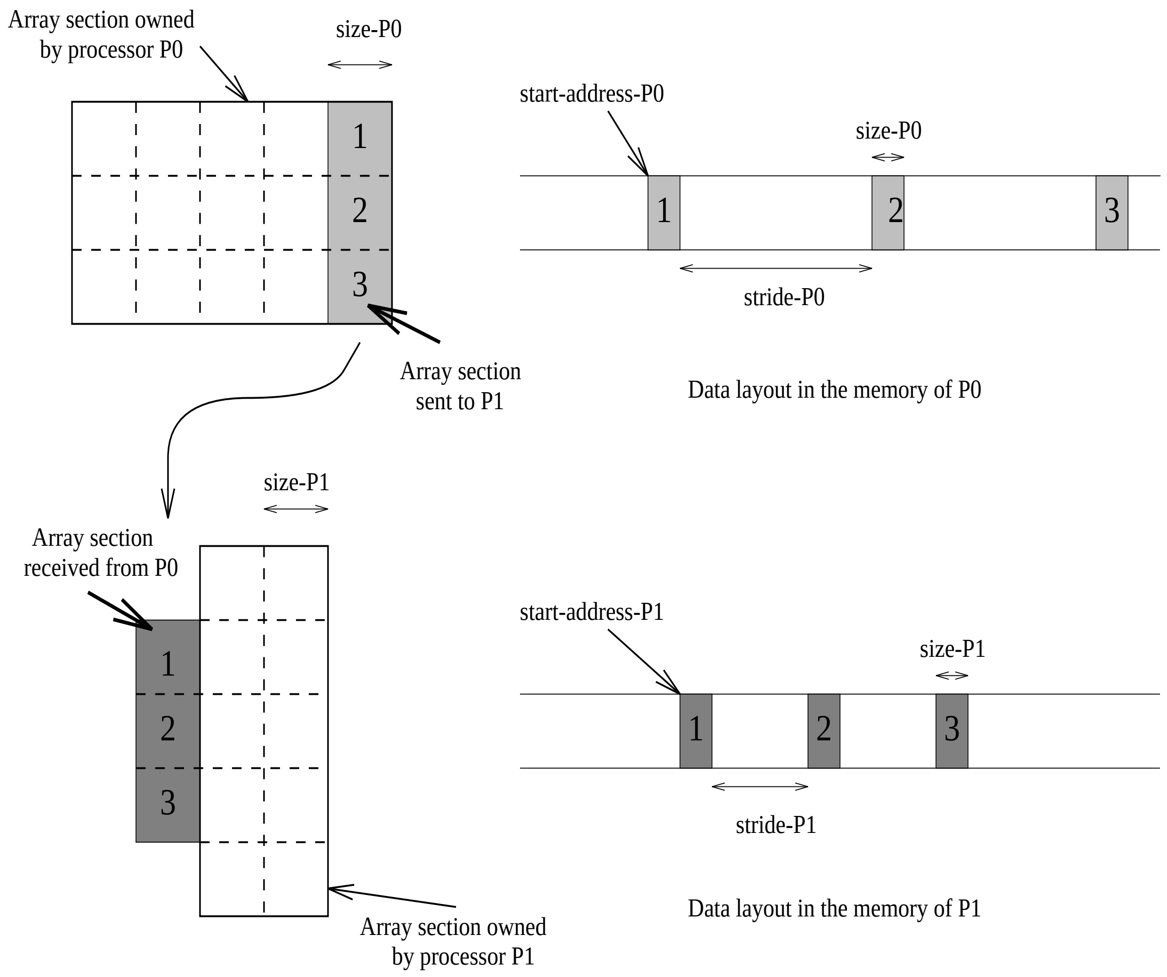

Array section owned by processor P0
size-P0
1
2
3
Array section sent to P1
start-address-P0
size-P0
1
2
3
stride-P0
Data layout in the memory of P0
size-P1
Array section received from P0
1
2
3
Array section owned by processor P1
start-address-P1
size-P1
1
2
3
stride-P1
Data layout in the memory of P1

Figure 11.4: Communicating Array Sections: An Example

(This figure shows an example of communicating a section of a two dimensional array.)

of operating system level messages. Requests for send/receive are buffered up using msg_need_to_send/receive___() functions, and the data is actually sent/ received using the msg_send/receive() function. Consider the following example code:

```
PROCESSOR P0: msg_need_to_send___( P1, data1, ... );
              msg_need_to_send___( P2, data2, ... );
              msg_need_to_send___( P1, data3, ... );
              msg_need_to_send___( P2, data4, ... );
              msg_send();

PROCESSOR P1: msg_need_to_recv___( P0, buffer1, ... );
              msg_need_to_recv___( P0, buffer2, ... );
              msg_recv();

PROCESSOR P2: msg_need_to_recv___( P0, buffer1, ... );
              msg_need_to_recv___( P0, buffer2, ... );
              msg_recv();
```

When the above code is executed, the *message handler* module sends *data1* and *data3* together in a single operating system level message from processor P0 to processor P1, and *data2* and *data4* in a single message to processor P2. Processors P1 and P2 receive one message each, and demultiplex the data into *buffer1* and *buffer2*.

- **Gather/Scatter operations:** The *message handler* provides support for gathering/scattering data from/to regular sections of three-dimensional arrays. For example, a column of a two-dimensional array (Figure 11.4) can be communicated using the following code:

```
PROCESSOR P0: msg_need_to_send_2D_array_section(
                                        P1, start-address-P0,
                                        size-P0, stride-P0 );
              msg_send();
```

```
PROCESSOR P1: msg_need_to_recv_2D_array_section(
                                   P1, start-address-P1,
                                   size-P1, stride-P1 );
              msg_recv();
```

- **Processing received data:** The *message handler* gives different options for processing received data:
  - Scatter into a 2D/3D array section (just copies the data)
  - Scatter into a 2D/3D array section and combine the old and new data with the specified operation (such as ADD and MAX)
  - Copy into a buffer, or combine the new data with the data existing in the buffer
  - Invoke the specified handler to process incoming data

11.5 Phase Manager

The *phase manager* executes the application routines within the *phases* specified by the user. When the EXECUTE-PHASE construct is executed, the phase manager goes through the following steps:

1. Invoke the load balancer. If necessary, the load balancer balances the load.

2. For each spatial data structure that is going to be used in the current phase, if the space partitioning in the current phase is different from the partitioning in the previous phase in which this data structure was used, then redistribute the data.

3. Execute the application routine.

4. If the load balancer suggests a different partitioning scheme for this phase, redistribute the data to adapt to the new partitioning scheme.

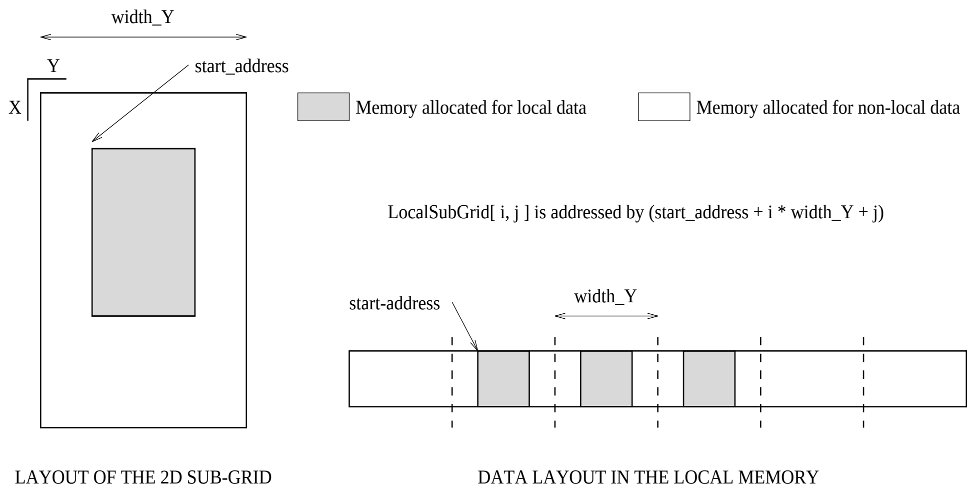

width_Y
Y
X
start_address
Memory allocated for local data
Memory allocated for non-local data
LocalSubGrid[i, j] is addressed by (start_address + i * width_Y + j)
start-address
width_Y
LAYOUT OF THE 2D SUB-GRID
DATA LAYOUT IN THE LOCAL MEMORY

Figure 11.5: Storing and Accessing a REGULAR-GRID Data Structure
(This figure illustrates how a node stores and accesses its portion of a two dimensional regular-grid)

11.6 Spatial Data Manager

This module maintains the distributed spatial data structures. Each node maintains its portion of the REGULAR-GRID data structure as a dynamic array within its local memory. Sufficient memory is reserved for storing non-local data that is received from the neighboring processors along the partition boundary. Figure 11.5 shows how a node stores and accesses its portion of a two-dimensional regular grid.

The data elements of a PARTICLE data structure are maintained, at each node, in a complex local structure. This local data structure keeps the particles partially sorted (according to their X, Y, Z coordinates) into *CS-Slots*, the slots induced by the *CS-Mesh* that is used to discretize the space. Partial sorting makes the operation of moving particles from region to another easier. (This operation is performed when the particle data is repartitioned to balance the load.) Partial sorting also makes it easier to pair up closely located particles for applications such as Molecular Dynamics that operate on pairs of particles located within a cut-off distance.

11.7 Data Redistributor

The *data redistributor* module supports redistribution of the spatial data in the following situations:

1. when the execution proceeds from one *phase* to another and the space partitioning in the previous phase is different from the partitioning in the current phase

2. when the data is shared across partition boundaries

3. when the load is balanced and the space needs to be repartitioned

4. when the spatial objects move from one region to another

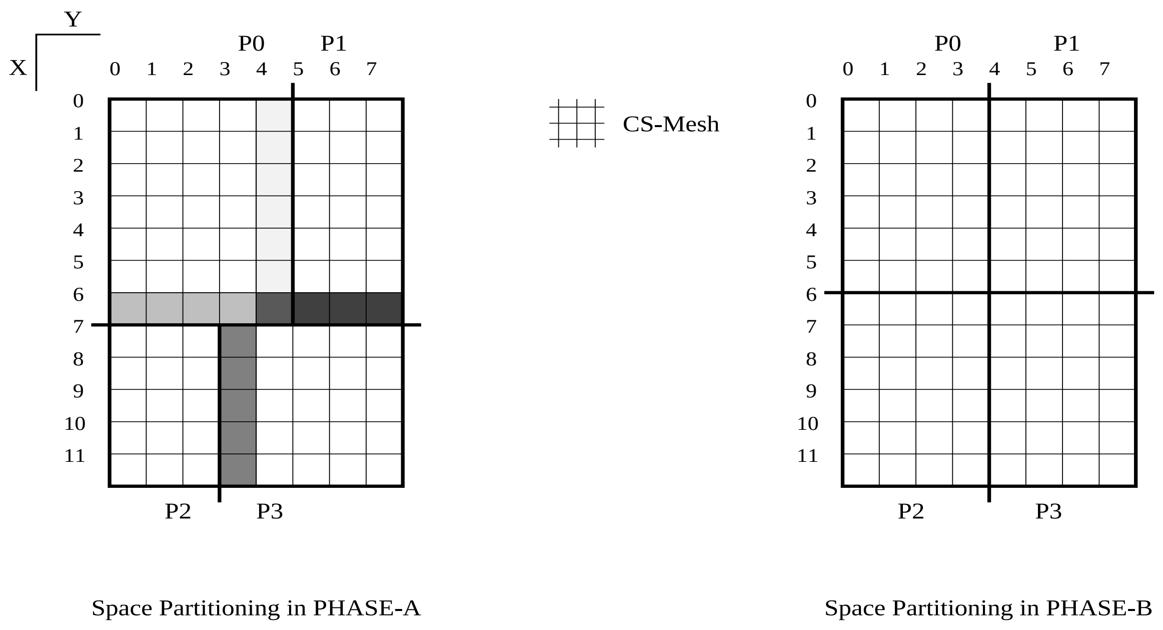
In the first two situations, the redistribution schedule could be repeatedly used, so the *data redistributor* module saves the schedules and reuses them. A redistribution schedule is represented by the set $\{(S,P)\}$, where S is the sub-cube that needs to be sent to (or received from) processor P. In the case of a REGULAR-GRID data structure, S represents a sub-grid, whereas in the case of a PARTICLE data structure, S represents a sub-region from which particles must be taken (or a sub-region into which particles must be placed). Figure 11.6 shows an example of a redistribution schedule used when the phase changes.

11.8 Load Balancer

The *load balancer* module decides when to balance the load, estimates the load distribution using a non-uniform, adaptive load discretizing mesh, and balances the load hierarchically. The functionally of the load balancer module is described in detail in Chapter 9.

11.9 Processor Reallocation Adapter

The *processor reallocation adapter* module interacts with the operating system to adapt to the processor reallocations performed by the kernel scheduler. The functionality of this module is described in Chapter 13.



REDISTRIBUTION SCHEDULE FOR PHASE CHANGE (A->B)

P0:	send {(0,4),(5,4)} to P1 ▢	P1:	send {(6,5),(6,7)} to P3 ■
	send {(6,0),(6,3)} to P2 ▒		recv {(0,4),(5,4)} from P0
	send {(6,4),(6,4)} to P3 ■		
		P3:	send {(7,3),(11,3)} to P2 ▓
P2:	recv {(6,0),(6,3)} from P0		recv {(6,4),(6,4)} from P0
	recv {(7,3),(11,3)} from P3		recv {(6,5),(6,7)} from P1

Figure 11.6: Example of a Redistribution Schedule

11.10 Summary

In this chapter, we discussed some details about the implementation of the *Ādhāra* runtime system. In the next chapter, we present some performance results.

Chapter 12

PERFORMANCE RESULTS

To evaluate the performance of the *Ādhāra* runtime system, we used applications from three scientific fields: plasma physics, aeronautics and materials science. We refer to the electro-magnetic particle-in-cell simulation by EMPIC, rarefied fluid flow by MP3D and molecular dynamics by MD. The EMPIC application was developed by hand converting a sequential Fortran program (written by David Walker, Oak Ridge National Laboratory) to a sequential C program and then to an *Ādhāra* program. The MP3D application was developed by extracting the sequential program from the shared-memory parallel C program, which is one of the SPLASH benchmarks, and converting it into an *Ādhāra* program. The MD program was developed by converting a sequential C program developed in the department of Materials Science at the University of Washington. The computational characteristics of these applications are given Table 12.1.

We executed the applications on 2 to 16 nodes of an Intel Paragon. Each simulation was performed for 200 time steps (leading to 1 - 30 minute execution times, depending on the application and the number of nodes), unless otherwise specified. We measured all overheads in terms of the percentage of optimal compute time, which is the execution time of the sequential program divided by the number of processors on which the parallel program is executed.

We use all the three applications for studying the effect of *Ādhāra*'s heuristics for choosing a partitioning scheme and the effect of *Ādhāra*'s dynamic load balancing scheme on the overall performance. For studying the effect of the non-uniform, adaptive load discretizing mesh and the effect of the predictive method, we do not use the MD application, the reason for which is discussed below. Both EMPIC and MP3D use *Ādhāra*'s automatic load balancing facility, whereas MD does not. The MD application, in order to optimize the time required for partially sorting the particles into cells, informs *Ādhāra* of spatial changes only once every few time steps, instead of every time step (see Section 5.3). It explicitly invokes the load balancer after the changes in the coordinates are given to the runtime system. Since the frequency of load balancing is under the control of the application, *Ādhāra* uses a static load discretizing mesh for balancing the load.

This chapter is organized as follows. Section 12.1 describes how the performance is affected by the partitioning scheme. The effectiveness of *Ādhāra*'s heuristics for choosing a good partitioning scheme is discussed in Section 12.2. Section 12.3 gives the overhead of dynamically changing the partitioning scheme. The effect of the coarseness of the load discretizing mesh is discussed in Section 12.4. The performance of the non-uniform, adaptive mesh scheme and adaptation of the parameters of this mesh are presented in Sections 12.5 and 12.6. The performance of the predictive method is given in Section 12.7. Section 12.8 discusses the effect of *Ādhāra*'s load balancing scheme on the overall execution time. Section 12.9 concludes by summarizing the important results.

12.1 Sensitivity of Application Performance to the Partitioning Scheme

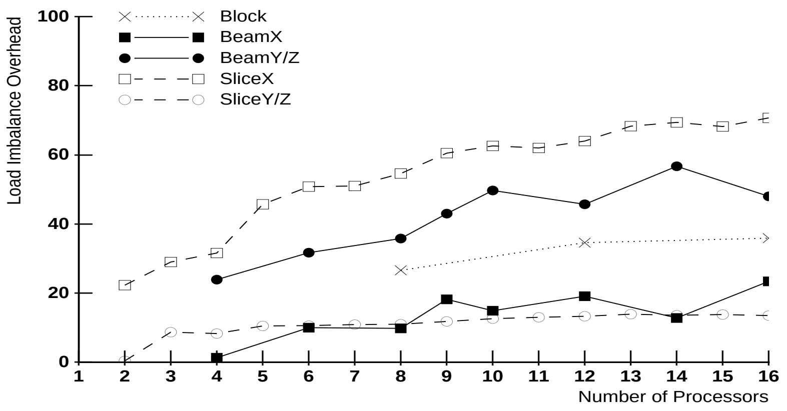
In this section, we show that application performance can be very sensitive to the partitioning scheme. We use figures 12.1, 12.2 and 12.3 to illustrate the importance of choosing a good partitioning scheme. These figures give the overheads and parallel efficiencies for different number of processors and for different partitioning schemes, when no load balancing is performed. (The effect of dynamic load balancing on overall performance is discussed in section 12.8.) Note that the Block scheme is applicable only when the number of processors has at least three factors (e.g., 8, 12 and 16) and the Beam schemes are applicable only when the number of processors has at least two factors (e.g., 9, 14 and 16). For the MD program, the Slice scheme is not used when the number of processors is greater than 10, since *Ādhāra* does not allow very thin slices. (The length of the simulation box used for the 16K input is only ten times the cut-off distance used for forming pairs of particles).

12.1.1 Effect of the Partitioning Scheme on the Load Imbalance Overhead

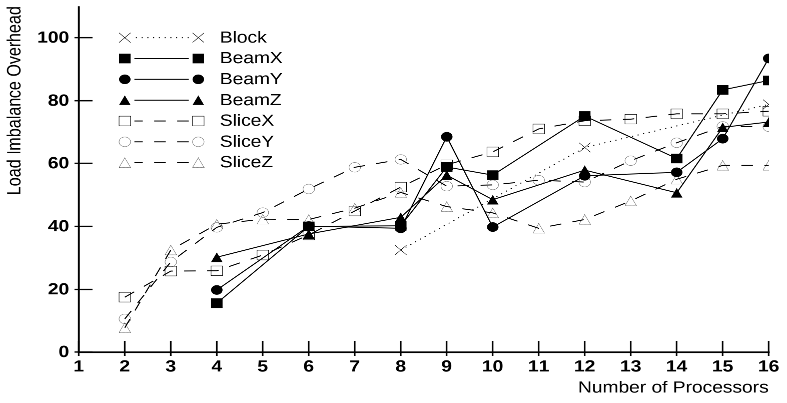
Figure 12.1 shows how the partitioning scheme affects load imbalance for the example applications. For the EMPIC application, since there is high load movement and density along the X dimension, the SliceX scheme results in very high load

Table 12.1: Comparison of the Application and Execution Characteristics

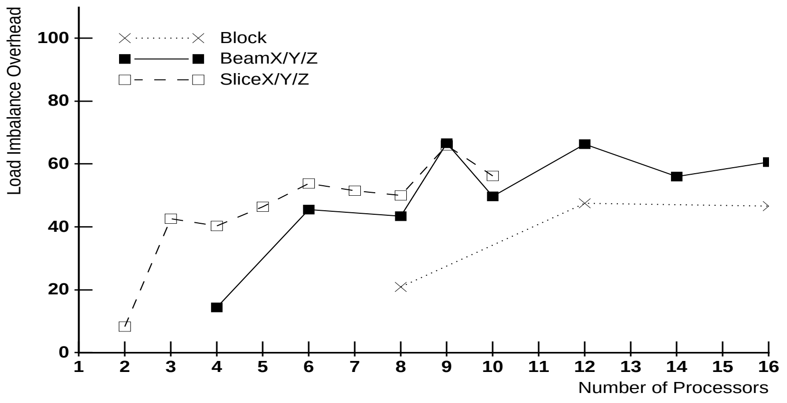
	EMPIC	MP3D	MD
Spatial data structures	three *regular-grid*s and one *particle*	one *regular-grid* and one *particle*	one *particle*
Data size per regular-grid cell	small (24 bytes)	medium (72 bytes)	
Data size per particle	medium (100 bytes)	medium (84 bytes)	large (252 bytes)
Types of interactions	*particle-grid* and *grid-grid*	*particle-grid* and *particle-particle* (random pairing within a grid-cell)	*particle-particle* (pairing based on cut-off distance)
Sharing of grid data along partition boundaries	four times per time step	once per time step	
Sharing of particle data along partition boundaries (per time step)	once for maintaining data locality	once for maintaining data locality	once for maintaining data locality and once for forming pairs
Granularity (average computation per particle per time step)	medium (225 μsec.)	small (70 μsec.)	large (900 μsec.)
Input used for the experiments (unless otherwise specified)	32K particles, 16x32x16 grid; initial particle distribution is uniform	32K particles, 32x16x16 grid; space object is part of an aeroplane wing	16K atom crystal with a spherical hole; symmetric load distribution
Execution characteristics for the above inputs	high load movement and density along X; load variation is steady	high load movement along X; load variation is unsteady due to obstructions	high communication overhead; load variation is slow and steady
Load Balancing	automatically performed by *Ādhāra*	automatically performed by *Ādhāra*	application specifies when load balancing must be done
Load Estimation	unit load per particle	unit load per particle	based on the number of pair-wise interactions on a particle



(a) EMPIC



(b) MP3D



(c) MD

Figure 12.1: Effect of the Partitioning Scheme on the Load Imbalance

(Load imbalance is given in terms of the percentage of optimal compute time.)

imbalance overhead, whereas the BeamX and SliceY/Z schemes result in low load imbalance overhead (since for these two schemes, the space is not partitioned along X). For MP3D, load imbalance is not that sensitive to the partitioning scheme, since the load variation is high and unsteady along all the three dimensions. The MD simulation does not exhibit preferential movement along any dimension, and the distribution of the particles is such that the Block scheme partitions the load more evenly than the Beam and Slice schemes.

12.1.2 Effect of the Partitioning Scheme on the Communication Overhead

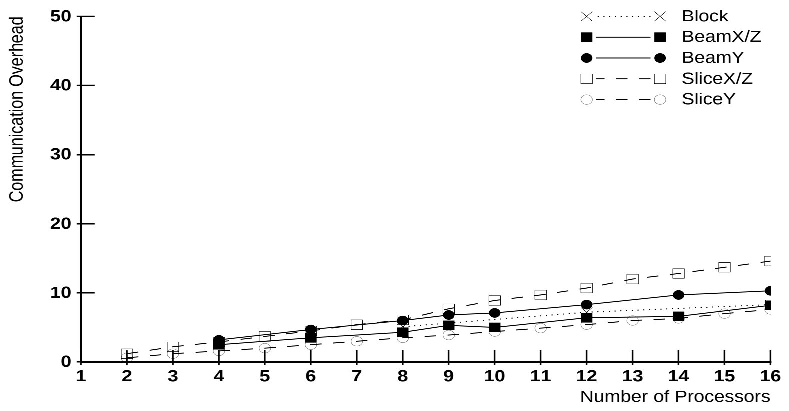
Figure 12.2 shows how the partitioning scheme affects the communication overhead. For the EMPIC and MP3D applications, the communication overhead is not significant, where as for the MD application, the overhead is high due to large amount of data sharing along the partition boundaries every time step. For EMPIC and MP3D, the SliceZ scheme results in higher overhead because of the larger surface area of the regions. For MD, the Block scheme results in regions with smaller surface area, hence, smaller communication overhead.

12.1.3 Effect of the Partitioning Scheme on the Parallel Efficiency

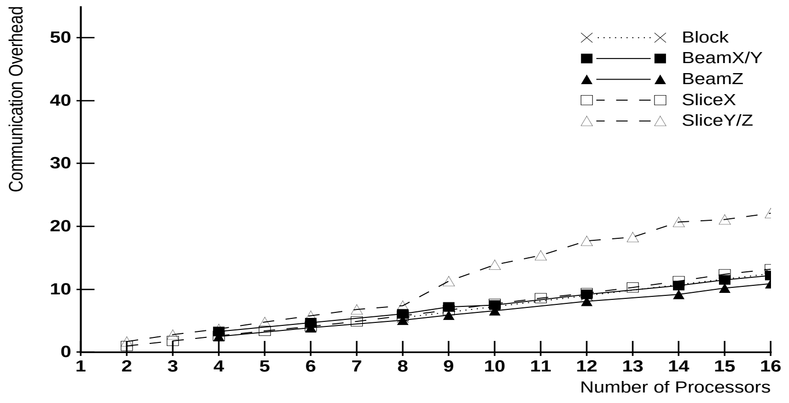
Figure 12.3 show how the partitioning scheme affects the parallel efficiency. For the EMPIC application, the BeamX and SliceY schemes perform much better than the SliceX scheme because they take advantage of the preferential movement along X. For the MP3D application, the performance is not that sensitive to the partitioning scheme, because there is no preferential movement and the communication overhead is not that significant. For the MD application, communication overhead is significant, hence the Block scheme performs the best since it results in smaller amount of data shared along the partition boundaries.

12.1.4 Importance of Choosing a Good Partitioning Scheme

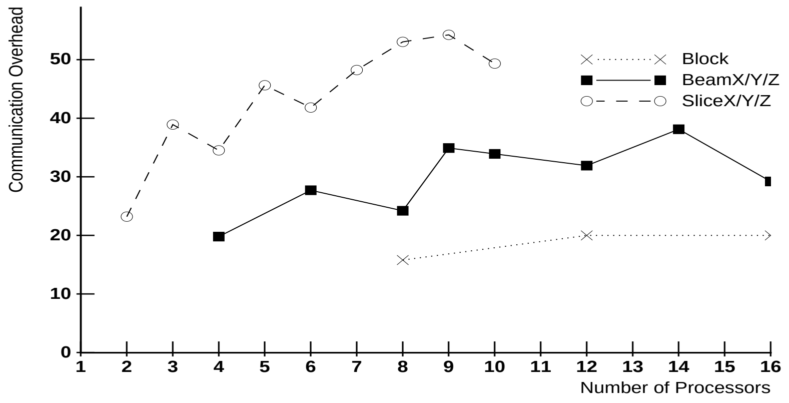
From the figure 12.3, we can see that the performance can be very sensitive to the partitioning scheme. By choosing the right scheme, parallel efficiency can be improved by as much as 30% (see the performance of EMPIC for 16 processors). In the next section, we see how *Ādhāra* selects a good scheme using heuristics that as based upon



(a) EMPIC



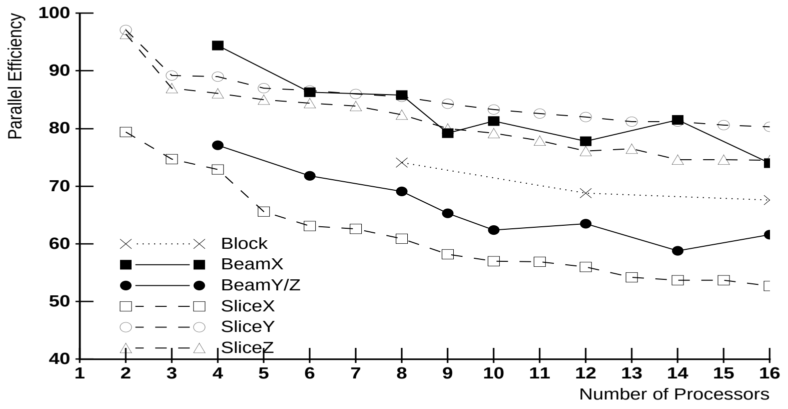
(b) MP3D



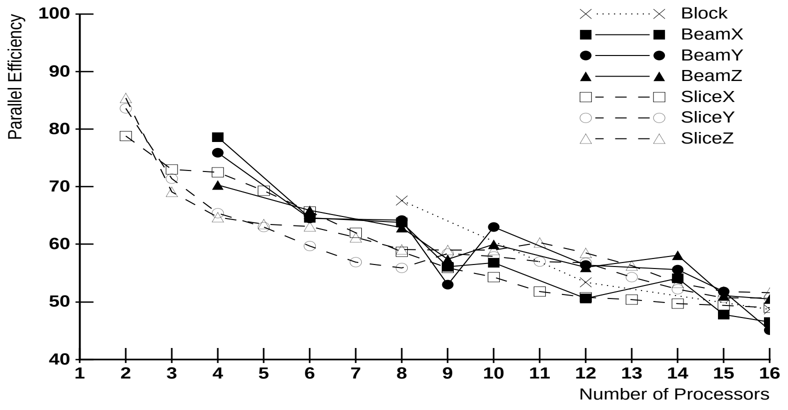
(c) MD

Figure 12.2: Effect of the Partitioning Scheme on the Communication Overhead

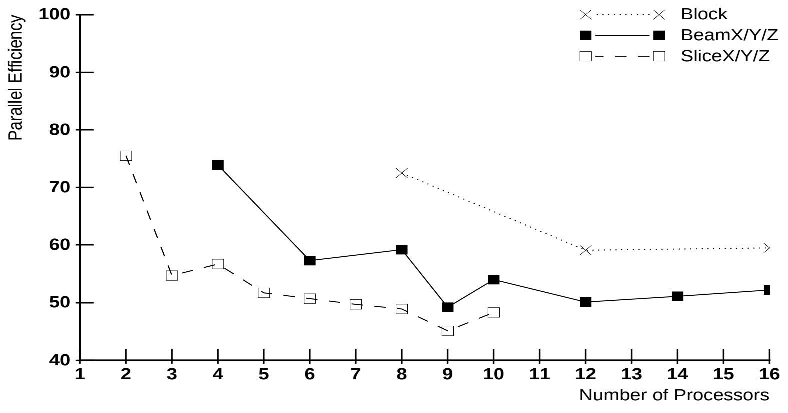
(Communication overhead is given in terms of the percentage of optimal compute time.)



(a) EMPIC



(b) MP3D



(c) MD

Figure 12.3: Effect of the Partitioning Scheme on the Parallel Efficiency

Table 12.2: Applying Heuristics for Choosing a Partitioning Scheme
(The load movement and density are given with respect to the grid cells of a fine 16384^3 *CS-Grid*. The RESULT column gives the decision of the *Ādhāra*'s heuristic algorithm. The last two columns compare the parallel efficiency of the scheme chosen by *Ādhāra* with the parallel efficiency of the best and worst schemes.)

	Load Movement	Load Density	Data Sharing Overhead	No. of Processors	RESULT	Diff. from the best scheme	Diff from the worst scheme
EMPIC	X: 119 Y: 43 Z: 28	X: 43 Y: 29 Z: 48	small	4	BeamX	0%	21%
				16	BeamX	-6%	15%
MP3D	X: 960 Y: 460 Z: 308	X: 93 Y: 46 Z: 103	small	4	BeamX	0%	14%
				16	BeamX	-5%	2%
MD	very small	high, same along X, Y, Z	large	4	BeamY	0%	18%
				16	Block	0%	8%

application execution characteristics.

12.2 *Effectiveness of* Ādhāra*'s Heuristics for Choosing a Good Partitioning Scheme*

Table 12.2 gives the decisions made by the *Ādhāra* data partitioner for the three applications. The heuristics that are used to make the decisions are given below.

- **EMPIC**: Since the load movement along X is very high, the four schemes that partition along X (Block, BeamY, BeamZ and SliceX) are eliminated.
  - *Four processors*: Since the load density and movement along Y and Z are relatively small and not much different, *Ādhāra* chooses the BeamX scheme. Note that there is no strong reason to eliminate the SliceY and SliceZ schemes. In general, *Ādhāra* prefers Beam to Slice schemes, because

Beam schemes are a good compromise in terms of the surface area of the regions and the number of neighbors of each region.

  - *Sixteen processors*: Since the simulation uses a 16x32x16 regular grid, there are only 16 grid cells along Z. To avoid very thin slices, *Ādhāra* eliminates the SliceZ scheme. Since the load density and movement along Y and Z are relatively small and not much different, *Ādhāra* chooses the BeamX scheme. Note that there is no strong reason to eliminate the SliceY scheme.

- **MP3D**: Since both the load density and movement along X are high, the four schemes that partition along X (Block, BeamY, BeamZ and SliceX) are eliminated.

  - *Four processors*: Since the load movement along Y and Z are relatively small and not much different, *Ādhāra* chooses the BeamX scheme.
  - *Sixteen processors*: Since the simulation uses a 32x16x16 regular grid, there are only 16 grid cells along Y and Z. To avoid very thin slices, *Ādhāra* eliminates SliceY and SliceZ schemes, thereby choosing BeamX.

- **MD**: Since communication overhead is significant due to large amount of data shared along the partition boundaries, *Ādhāra* eliminates all of the Slice schemes. Since the load movement and density along the three dimensions do not differ much, the choice is between Block and BeamY. (BeamY is randomly chosen from the three Beam schemes.)

  - *Four processors*: Since there is no Block scheme for this case, *Ādhāra* chooses the BeamY scheme.
  - *Sixteen processors*: *Ādhāra* chooses Block scheme because this scheme minimizes the amount of data communicated along the partition boundaries.

The last two columns of Table 12.2 show that the *Ādhāra*'s heuristics are very effective in eliminating the bad schemes, and in choosing schemes that perform very close to the best schemes.

Table 12.3: Overhead of Dynamically Changing the Partitioning Scheme
(The measurements are made on 16 nodes of an Intel Paragon. The overheads given here are the averages of the overheads of changing from block to beam scheme, slice to beam slice, etc.)

	Absolute time (millisec)	Relative to compute time per time step
EMPIC		
32K input	265	0.55
320K input	2935	0.65
MP3D		
32K input	320	2.10
320K input	3570	2.45
MD		
16K input	250	0.30

12.3 Overhead of Dynamically Changing the Partitioning Scheme

Table 12.3 gives the overhead of dynamically changing the partitioning scheme, which involves redistribution of the data to adjust to the new scheme. We can see that the overhead is quite small compared to the execution time. The MP3D application has larger relative overhead compared to the EMPIC because of its smaller computation granularity. The partitioning scheme is typically changed only once at the beginning of the execution (after *Ādhāra* estimates the execution characteristics), unless the the execution characteristics change dynamically. The simulations are typically run for hundreds of time steps, hence the overhead of changing the partitioning scheme is negligible compared to the overall execution time.

12.4 Effect of the Coarseness of the Load Discretizing Mesh on the Performance

In this section, we show that performance is sensitive to the coarseness of the load discretizing mesh. Note that the objective of the load balancer is to minimize LILB overhead, the sum of load imbalance and load balancing overheads. A very fine load discretizing mesh results in a good load balance, but high load balancing overhead. On the other hand, a coarse mesh results in a small load balancing overhead but a large load imbalance overhead. The load imbalance overhead depends on the load movement and density, which is specific to the application.

Figure 12.4 gives LILB overhead (in terms of the percentage of optimal compute time) for uniform, static load discretizing meshes of varying coarseness. For these measurements, we represent the coarseness relative to a 16384^3 mesh, that is, a mesh with coarseness n is of size $(\frac{16384}{n})^3$.

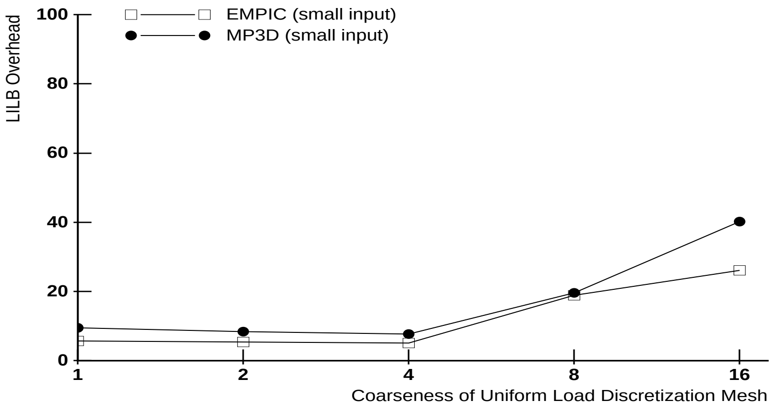
We can make the following observations:

- Coarseness of the load discretizing mesh has significant impact on performance.
- As the coarseness increases, the LILB overhead decreases, comes to a minimum and then increases. The optimal coarseness depends on the application, the input, and the number of processors.
- For the same input size, the impact is more pronounced for larger number of processors, because the optimal compute time is smaller.
- For the same input size, the impact is more pronounced for an application with smaller computation granularity (MP3D), because the optimal compute time is smaller.
- There is a bigger penalty to overestimating the coarseness than to underestimating it, since the cost of load imbalance resulting from using a very coarse mesh is much more than the cost of using a very fine mesh.

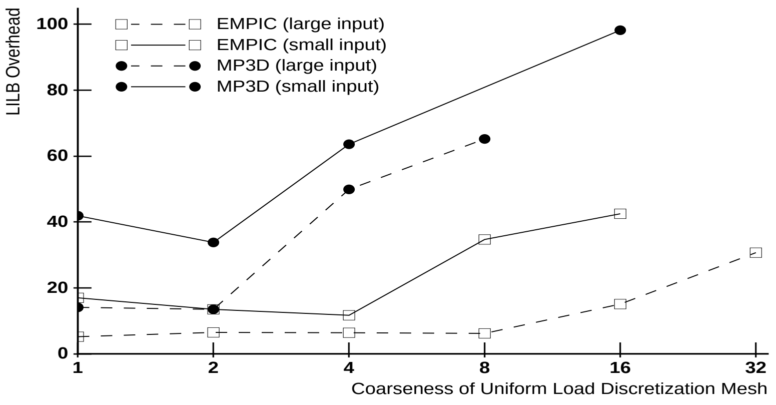
The results indicate the need for an adaptive scheme that dynamically chooses the coarseness based on the application execution characteristics. The following section describes the performance of the novel scheme that uses a non-uniform, adaptive load discretizing mesh for balancing the load.

12.5 Performance of the Non-Uniform, Adaptive Load Discretizing Mesh Scheme

The load balancing overhead can be broken up into three parts: (1) overhead for estimating the load distribution, (2) overhead for computing a new partition, which includes communication of the local load estimates, and (3) overhead for redistributing the data for adapting to the new partition. Compared to a uniform, static mesh scheme, a non-uniform, adaptive mesh scheme reduces the first two overheads by estimating the load only along the partition boundaries and thereby communicating



(a) Four Processors



(b) Sixteen Processors

Figure 12.4: Effect of Coarseness of Load Discretizing Mesh on the Sum of Load Imbalance and Load Balancing Overheads

(The overheads are given in terms of the percentage of optimal compute time. Coarseness of the mesh is relative to a 16384^3 mesh, i.e., a mesh with coarseness n is of size $(\frac{16384}{n})^3$. A Block partitioning scheme is used in all the cases. Small input refers to a input containing 32K particles, whereas large input refers to 320K particles. The load is balanced once every 5 time steps.)

smaller amount of data. Since the third overhead is common for both the schemes, we compare the schemes based on the first two overheads.

Table 12.4 compares the performance of the uniform and adaptive mesh schemes. We can make the following observations:

- By estimating the load only along the boundaries, the non-uniform mesh scheme reduces the load balancing overhead by a significant amount.

- The heuristics used to adapt the non-uniform mesh (in other words, the boundary region that is used to estimate the load) work very well – the resulting load imbalance overhead is close to the overhead incurred by the fine uniform mesh, i.e., the adaptive scheme does not trade off much load balance by dynamically determining just the sufficient amount of boundary region.

- The effect of the adaptive scheme is more pronounced for the MP3D application due to its smaller computation granularity.

The results indicate that the non-uniform, adaptive scheme can improve the performance by as much as 24%. In the next section, we show how the heuristic algorithm adapts the parameters of the load discretizing mesh.

12.6 Adaptation of the Non-Uniform Load Discretizing Mesh

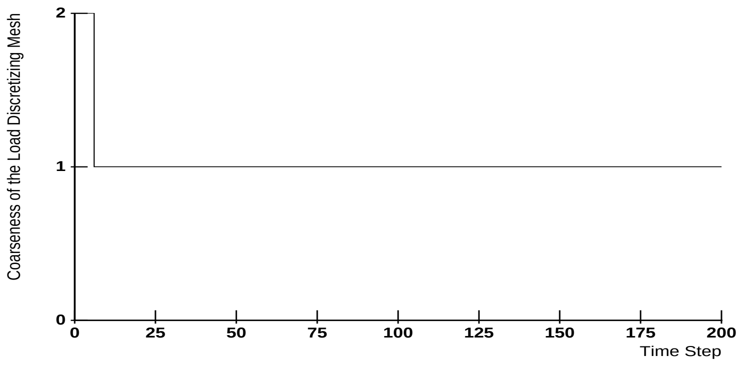
Figure 12.5 shows how *Ādhāra* adapts the coarseness of the load discretizing mesh for the EMPIC application. Coarseness along dimension D is chosen based on the load density along D. *Ādhāra* adapts the mesh whenever load balancing is performed. For the EMPIC simulation, since there is high load movement along X, the load density along X becomes high, hence *Ādhāra* chooses very thin slots along X. Along the Y dimension, since the load density is initially low and the load movement is small, *Ādhāra* uses thicker slots until the time step 176. Along the Z dimension, *Ādhāra* changes the coarseness of the mesh earlier, at time step 55.

Figures 12.6 and 12.7 show how *Ādhāra* adapts the number of slots of the load discretizing mesh. The number of slots used along dimension D is based on the load movement along D.

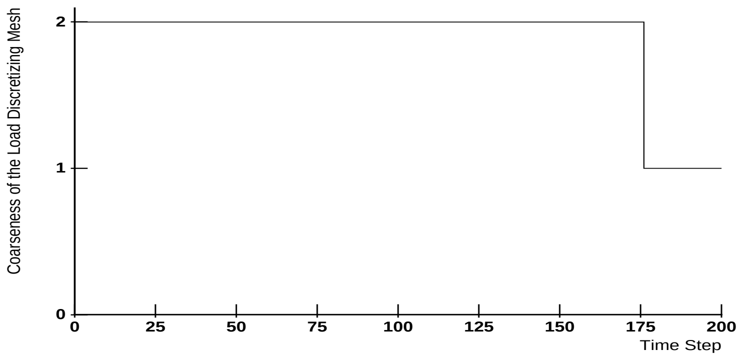
For the EMPIC application, *Ādhāra* uses more slots along X than along Z, because there is more load movement along X (Figure 12.6). In the initial part of the

Table 12.4: Performance of the Non-Uniform, Adaptive Load Discretizing Mesh
(The experiments are done on 16 processors. All overheads are given in terms of the percentage of the optimal compute time, unless otherwise specified. Coarseness of the uniform load discretizing mesh is represented by the coarseness factor *cf* with respect to a grid of size 16384^3. For example, *cf* $= n$ represents a grid of size $(\frac{16384}{n})^3$. LILB overhead is the sum of load imbalance and load balancing overheads. The *performance improvement* column gives the difference in the LILB overhead between the adaptive mesh and the optimal uniform mesh.)

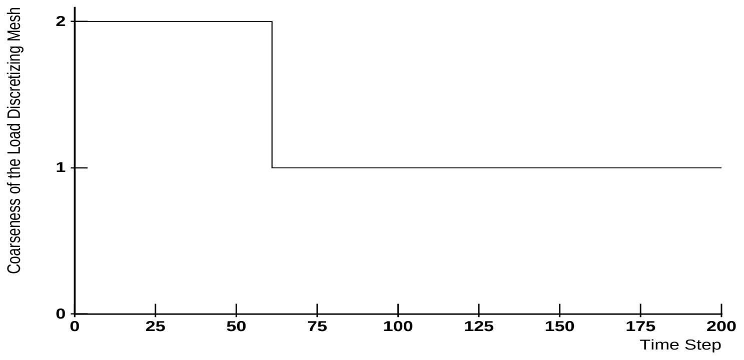
Overhead	Uniform, Static Load Discretizing Mesh					Non-Uniform Adaptive Mesh	Performance Improvement
	cf=1	cf=2	cf=4	cf=8	cf=16		
Cost of computing a new partition (in milliseconds)	7440	4730	3345	2705	2400	850	
EMPIC							
Load Estimation	4.0	4.0	4.0	4.0	4.0	0.5	3.5
Computing Partition	7.8	5.0	3.6	2.6	2.5	0.9	
Load Imbalance	3.9	3.9	3.9	27.5	35.8	5.5	
LILB	15.7	12.9	**11.5**	34.1	42.3	6.9	4.6
MP3D							
Load Estimation	10.0	10.0	10.0	10.0	10.0	0.9	9.1
Computing Partition	25.3	16.0	11.1	8.8	7.9	2.8	
Load Imbalance	3.5	5.0	41.7	57.1	80.2	3.6	
LILB	38.8	**31.0**	62.8	75.9	98.1	7.3	23.7



(a) Coarseness Along the X Dimension



(b) Coarseness Along the Y Dimension



(c) Coarseness Along the Z Dimension

Figure 12.5: Adaptation of the Coarseness of the Load Discretizing Mesh for the EMPIC Application

simulation, the particles move preferentially along the positive X direction, and in the later part, along the negative X direction. *Ādhāra* adapts to the dynamics effectively by initially choosing more slots along the negative X boundary and later choosing more slots along the positive X boundary. (Note that if load moves in the positive direction, the partition boundary on the negative side moves towards the positive side, thereby using the slots on the negative side of the region.) The slow movement along the Z dimension results in very little variation of slots along Z.

In the MP3D simulation, there is more load movement along X, so *Ādhāra* chooses more slots along X. The load movement is unsteady along the positive directions, that is, the difference in the load movement from one time step to the next varies a lot. This is reflected in the fast variation in the slots in the negative X and Z directions. (For example, see the variation in the slots along the negative Z at time step 50 in Figure 12.7(b).)

From the results, we can see that the heuristic algorithm implemented by *Ādhāra* is very effective in adapting the load discretizing mesh in response to the changes in the load distribution and movement.

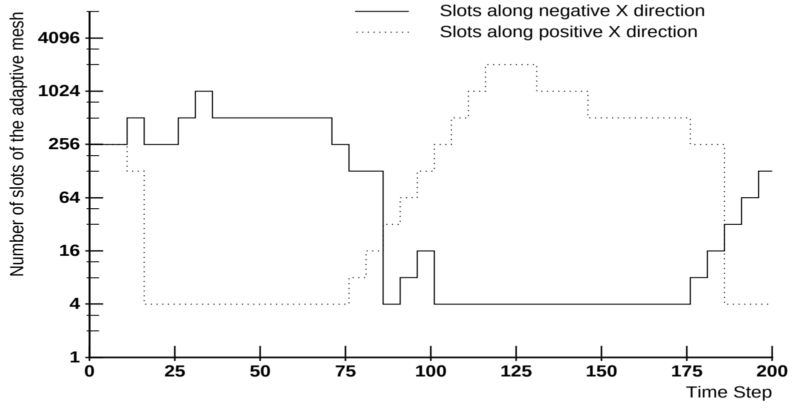
12.7 Performance of the Predictive Method

In this section, we compare the predictive method with the fixed frequency method. Figure 12.8 shows that the performance is sensitive to the frequency of load balancing. We can make the following observations from the figure:

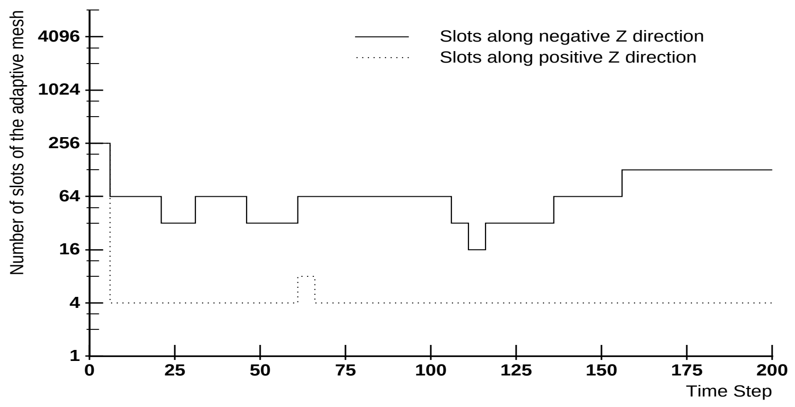
- As the interval between successive load balancing steps increases, the LILB overhead first decreases, comes to a minimum, and then increases. Unsteady load variation can result in multiple local minimas (see MP3D).

- The optimal fixed frequency depends on the application, the input, and the partitioning scheme.

- For high frequencies (that is, when the interval is less than 10), the impact of the frequency on the performance is higher for those applications that have smaller computation granularity (such as the MP3D) and that use small inputs. This is because, at high frequencies, the load balancing cost dominates, and the relative overhead is higher when the optimal compute time per time step is

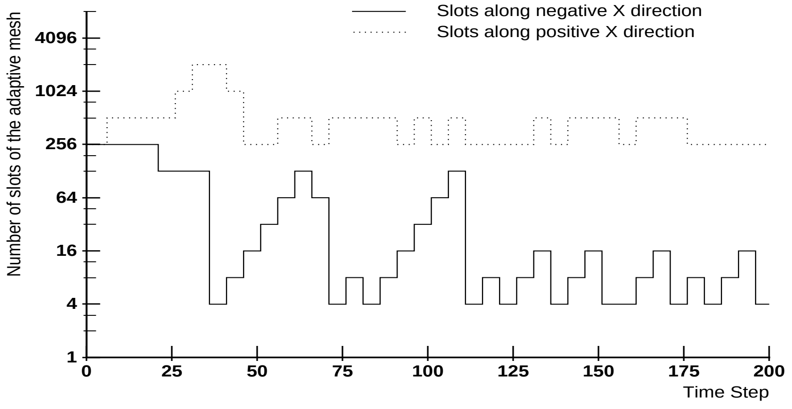


(a) Slots along the X dimension

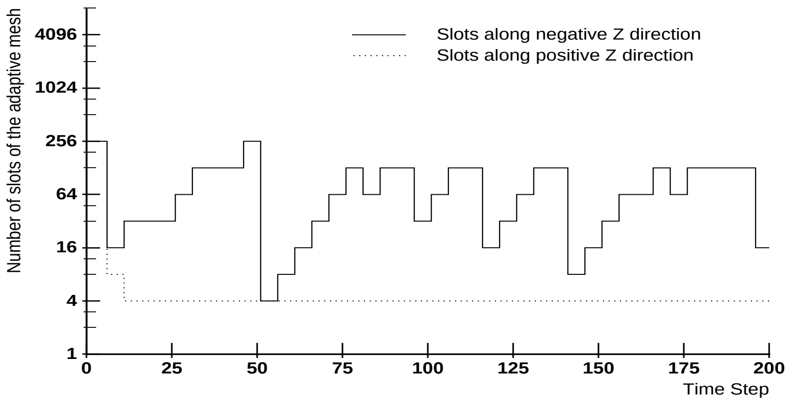


(b) Slots along the Z dimension

Figure 12.6: Adaptation of the Number of Slots of the Load Discretizing Mesh: EMPIC Application



(a) Slots along the X dimension



(b) Slots along the Z dimension

Figure 12.7: Adaptation of the Number of Slots of the Load Discretizing Mesh: MP3D Application

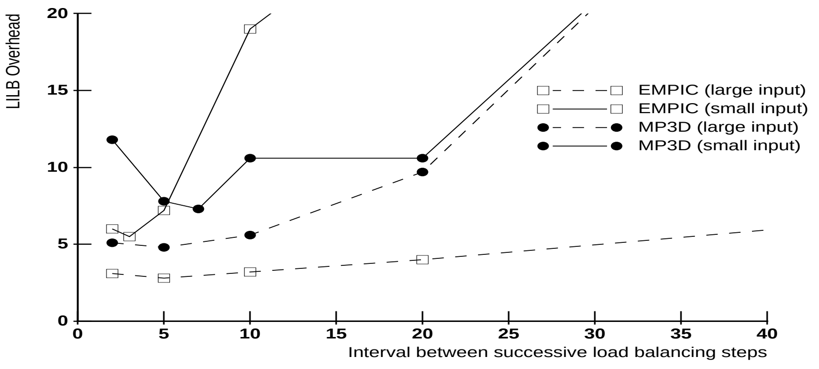
Table 12.5: Performance of the Predictive Scheme for Adapting Frequency of Load Balancing

	Optimal Fixed Interval	LILB Overhead for Optimal Fixed Frequency Method	LILB Overhead for Predictive Method	Average Interval Induced by the Predictive Method
EMPIC				
Block Scheme	3	5.5	7.0	4.3
BeamX Scheme	4	3.8	3.7	6.9
MP3D				
Block Scheme	7	7.3	7.9	8.3
BeamX Scheme	8	7.5	7.8	9.1

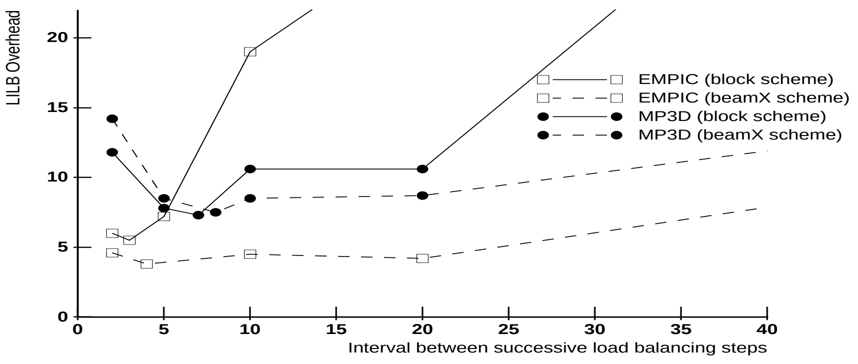
smaller. We use figure 12.8(c) to illustrate this point, by using a hypothetical application that is exactly same as the MP3D application in terms of execution characteristics, except that it has a very small computation granularity (about 25 μ seconds).

The results indicate the need for a scheme that adapts the frequency based on the costs of load balancing and load imbalance.

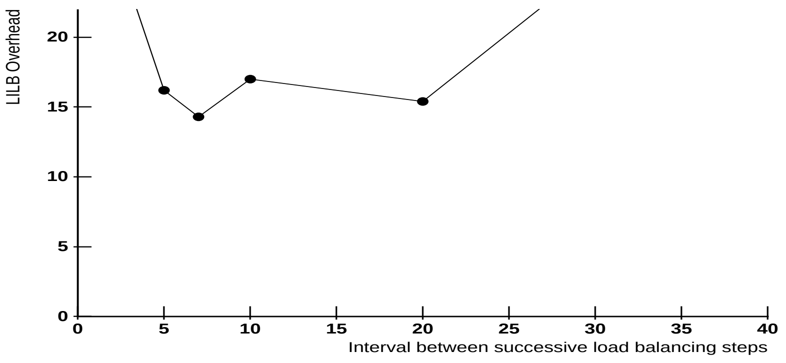
Table 12.5 gives the performance of the predictive method. From the results, we can see that predictive method performs very well – the LILB overhead is very close to that of the optimal fixed frequency method. In some cases, it performs better (for example, BeamX scheme for the EMPIC application), because it adapts to the variation in the load movement. The average frequencies induced by the predictive method reflect the execution characteristics. For the EMPIC application, the interval between successive load balancing steps is higher for the BeamX scheme than for the Block scheme, because BeamX scheme results is much lower load imbalance (see Figure 12.1(a)). The average interval for MP3D is larger than the interval for EMPIC, because the computation granularity is smaller and as a result, the cost of load imbalance is relatively small compared to the cost of load balancing. Figure 12.9 shows how the predictive method adapts the frequency of load balancing for different partitioning schemes.



(a) Large vs. Small Inputs



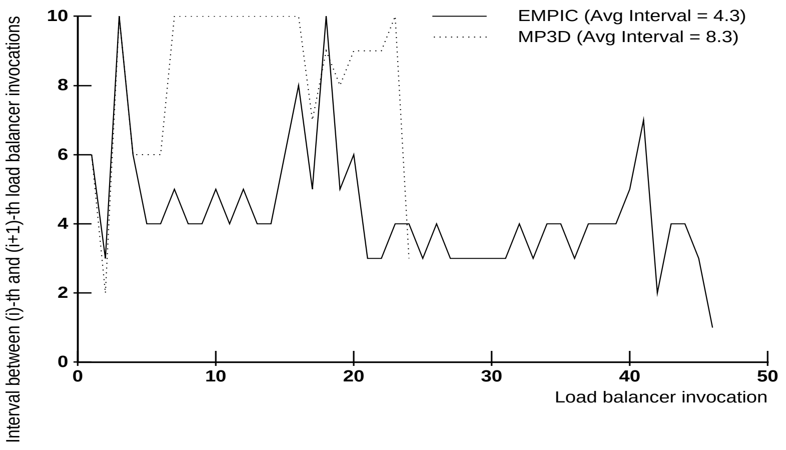
(b) Block vs. Beam Schemes



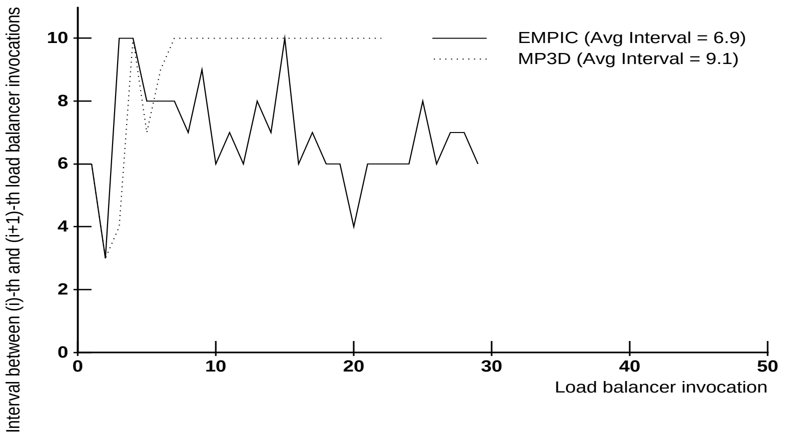
(c) Hypothetical Application with a Very Small Computation Granularity

Figure 12.8: Effect of the Frequency of Load Balancing on the Sum of Load Imbalance and Load Balancing Overheads

(The overhead is given in terms of the percentage of optimal compute time. Small input refers to the input containing 32K particles and large input refers to 320K particles. Figure (c) shows the performance of a hypothetical application whose execution characteristics are similar to the MP3D application, but whose computation granularity is very small.)



(a) Block Scheme



(b) Beam Scheme

Figure 12.9: Adaptation of Load Balancing Frequency

12.8 Effect of the Ādhāra*'s Load Balancing Scheme on Overall Execution Time*

Table 12.6 shows the effect of the *Ādhāra*'s load balancing scheme on the performance of the three applications for different inputs. The overheads incurred by a parallel execution are described below:

- *Parallelization overhead*: This is due to the computation required for maintaining spatial locality. Whenever changes to the spatial coordinates are transmitted to the runtime system, *Ādhāra* needs to compare the coordinates of each particle to the region boundaries to check whether or not the particle is still within the local region. Out of bound particles need to be sent to the appropriate processors.

- *Communication overhead*: This is due to exchanging of particle data for maintaining spatial locality, sharing grid and particle data along the partition boundaries and redistributing data between phases.

- *Load imbalance overhead*: This is due to uneven distribution of computation across the processors.

- *Load balancing overhead*: This is the overhead of repartitioning the space for balancing the computational load.

From the results, we can see that the *Ādhāra*'s load balancing scheme is very effective in reducing the load imbalance overhead while incurring very small load balancing overhead. In the case of MP3D application (large input case), there is a performance improvement of as much as 32%. The results shown here use only 200 time-step simulations. For longer simulations, the load imbalance becomes worse if dynamic load balancing is not done, hence *Ādhāra*'s load balancing scheme improves performance even more.

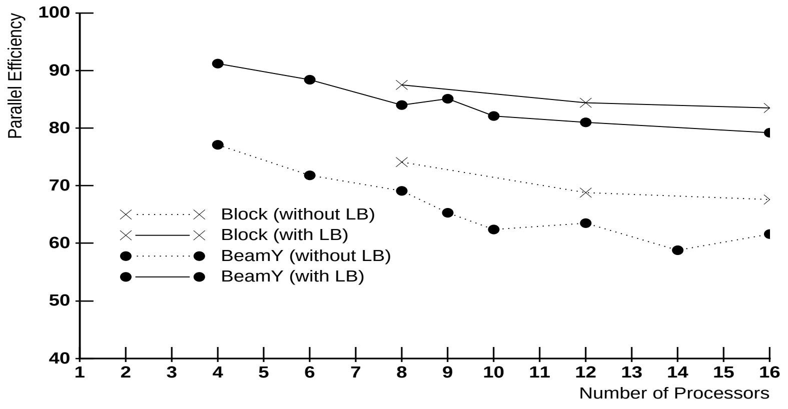
Ādhāra's load balancing scheme does not perform as well for the MD application, because load balancing is under the control of the application. (See the beginning of this chapter.) However, for longer simulations, the performance improvement will be more significant.

Figure 12.10 shows the effect of the *Ādhāra*'s load balancing scheme on parallel efficiency for two partitioning schemes and for different numbers of processors. We

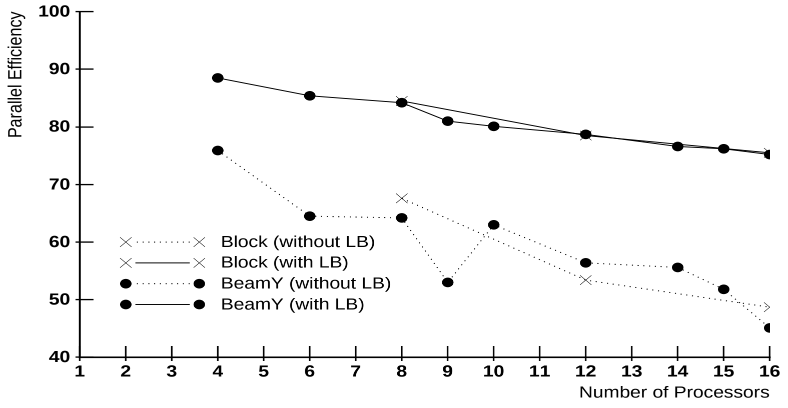
Table 12.6: Performance of the *Ādhāra*'s Load Balancing Scheme

(All overheads are given in terms of the percentage of the optimal compute time. Small input refers to the input containing 32K particles and large input refers to one containing 320K particles. For MD, a 16K particle input is used. The Block partitioning scheme is used in all the cases on 16 processors. "No LB" means no load balancing and "Dynamic LB" means dynamic load balancing using the novel adaptive schemes, except for the MD case in which uniform load discretization mesh is used.)

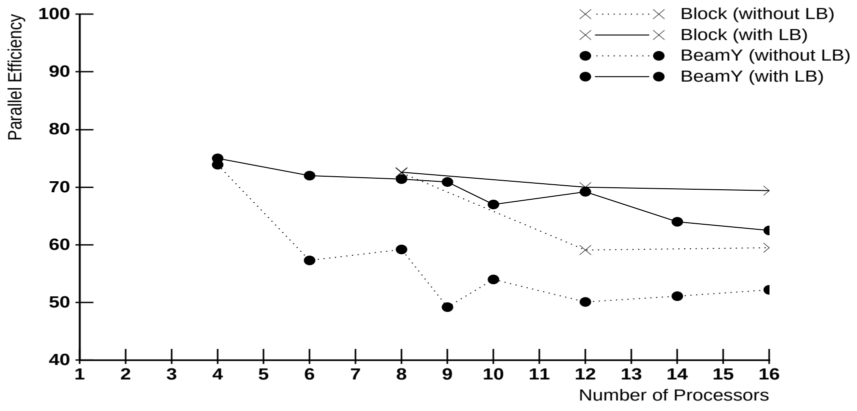
Application	Parallelization Overhead	Communication Overhead	Load Imbalance Overhead	Load Balancing Overhead	Parallel Efficiency
EMPIC: small input					
No LB	2.9	8.3	35.9	-	67.6%
Dynamic LB	2.6	9.8	5.3	1.8	83.5%
EMPIC: large input					
No LB	2.6	2.3	26.8	-	75.8%
Dynamic LB	2.5	2.8	2.7	0.4	92.2%
MP3D: small input					
No LB	13.2	12.5	78.8	-	48.7%
Dynamic LB	9.1	13.8	4.7	3.2	76.0%
MP3D: large input					
No LB	13.2	3.7	73.3	-	52.5%
Dynamic LB	9.5	3.4	4.1	1.3	84.4%
MD					
No LB	1.4	20.0	46.6	-	59.4%
Dynamic LB	1.2	22.0	11.9	4.3	71.4%



(a) EMPIC



(b) MP3D



(c) MD

Figure 12.10: Effect of the *Ādhāra*'s Load Balancing Scheme on Parallel Efficiency

can see that the dynamic load balancing scheme has significant impact on the overall performance.

12.9 Summary

We summarize the important results below:

- The partitioning scheme has a significant impact on the performance of those applications that exhibit preferential load movement and/or high data sharing along the partition boundaries.

- *Ādhāra*'s space partitioning heuristics are very effective in choosing good schemes.

- The coarseness of the load discretizing mesh has significant impact on the performance of those applications that have small computation granularity and that execute on large numbers of processors.

- *Ādhāra*'s novel scheme using a non-uniform, adaptive load discretizing mesh is very effective in reducing the overheads of estimating the load distribution and computing a new partition, while maintaining good load balance.

- The frequency of load balancing has a significant impact on performance.

- The predictive method implemented by *Ādhāra* is very effective in dynamically adapting the frequency of load balancing for minimizing the sum of the overheads due to load imbalance and load balancing.

- *Ādhāra*'s dynamic load balancing scheme is very effective in reducing the load imbalance while incurring very small overhead for load balancing.

Part IV

Operating System Induced Load Balancing

Chapter 13

ADAPTING TO PROCESSOR REALLOCATIONS

In some environments, the operating system may change an application's processor allocation its during the execution in order to improve throughput and average response time of the system workload. In this chapter, we look at the application's response to the changes in the processor allocation.

Policies that can be used by an operating system to determine when and how to reallocate the processors are discussed in detail in a recent thesis [McCann 94, McCann & Zahorjan 93], and we do not discuss them here. Figure 13.1 shows the basic approach taken by the operating system. When the allocation to a job is changed, the job's threads are relocated onto its new processor set as evenly as possible. On the assumption that the computational load is balanced across the threads, this allocation of threads to processors implements the best load balancing achievable by the operating system.

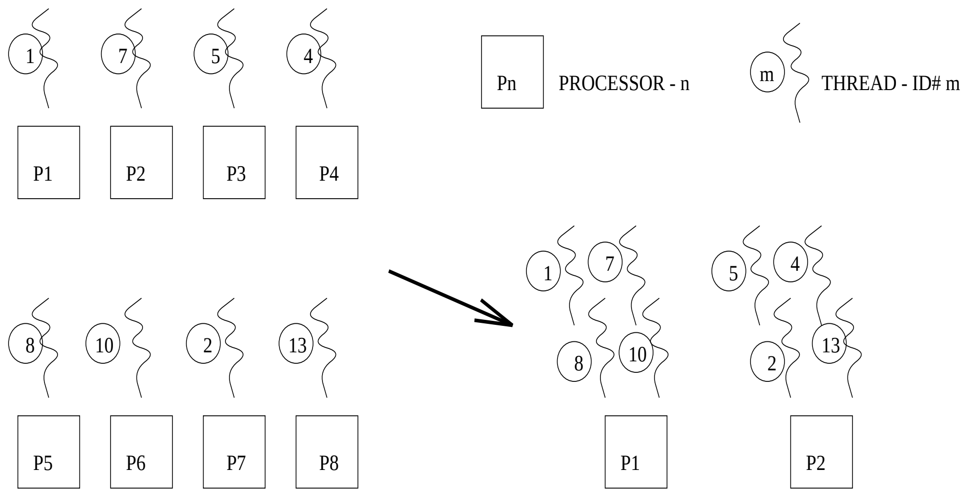


Figure 13.1: Processor Reallocation by the Operating System

When the allocation changes, the runtime system can react in either of two ways. In the *non-adaptation* scheme, the runtime system ignores the changes and continues to execute with the current threads. In the *dynamic adaptation* scheme, the runtime

Table 13.1: Non-Adaptation and Dynamic Adaptation Schemes: Advantages and Disadvantages

	Advantages	Disadvantages	
		One time cost	Continuing Cost
Non-Adaptation Scheme	Simple; no application level cost at the time of reallocation		Context switching overhead; sub-optimal load balance; increased load balancing and communication costs
Dynamic Adaptation Scheme	Minimal on-going overhead	Cost of remapping the data and adjusting the number of threads	

system remaps the computation such that the number of working threads is same as the number of processors currently allocated to the application.

Table 13.1 lists the advantages and disadvantages of these two schemes. The non-adaptation scheme incurs the continuing costs of context switching and increased execution overhead due to running multiple threads per processor instead of a single thread per processor. The latter overhead is simply a reflection of the fact that applications typically run more efficiently at lower parallelism than at higher.

Figure 13.2 shows how the execution overhead increases with the number of application threads for three example dynamic space-based applications, when there is single thread per processor. The purpose of the figure is to quantify how execution overhead grows with increasing parallelism. In the figure, execution overhead is the sum of the overheads due to communication, load balancing and load imbalance. As the number of threads increases, the communication cost typically increases due to the larger number of messages required, the load balancing cost increases due to more work involved in computing the new partitions, and the load imbalance increases because the computation needs to be partitioned into many small pieces instead of a few larger pieces.

Figure 13.3 shows how the performance on two processors degrades due to running multiple threads per processor. As the number of threads per processor increases,

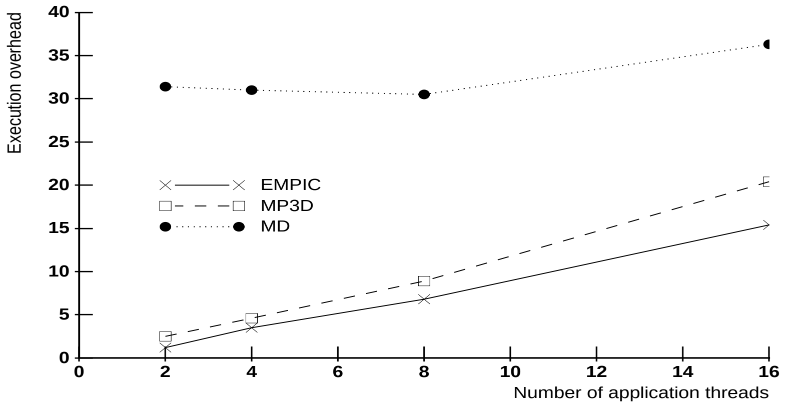


Figure 13.2: Effect of the Number of Threads on the Execution Overhead (*The overhead is given in terms of the optimal compute time. There is a single thread per processor.*)

the effective speed-up decreases due to increased execution overhead.

Compared to the non-adaptation scheme, the dynamic adaptation scheme incurs smaller execution overhead, but incurs, once per reallocation, the cost of remapping the application data and adjusting the number of threads. Additionally, it is more complicated. Note, however, that application level remapping fits naturally into the *Ādhāra* runtime system.

Depending on the costs, one approach may be preferable to the other. Our goal is to get a handle on the application level costs based on the execution characteristics of the dynamic space-based applications, and then compare the performance of these two approaches. For our measurements, we use the three example applications, EMPIC, MP3D and MD, which are described in Chapters 5 and 12.

13.1 Assumptions and Cost Model

For comparing the two schemes, we make the following assumptions:

1. The overhead of context switching among multiple threads of a single application running on a single processor is negligible.

2. The execution overhead for an application with N threads per processor running on P processors is same as the execution overhead for the application with one

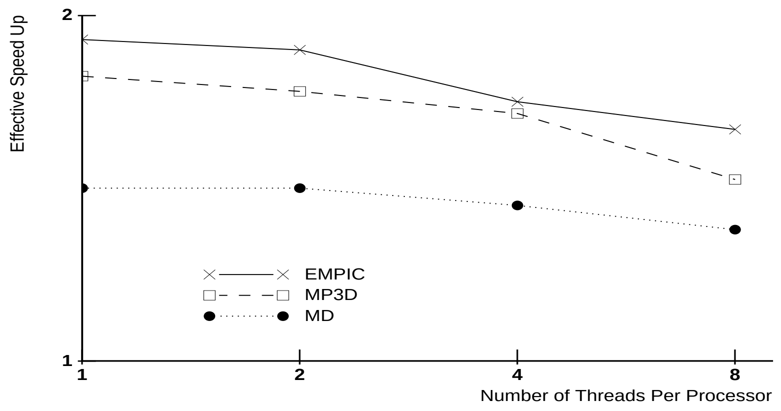


Figure 13.3: Effective Speed-Up on Two Processors Induced by the Non-Adaptation Scheme

thread per processor running on NP processors.

The first assumption favors the non-adaptation scheme, since the context switching overhead for kernel-level threads cannot be ignored. The second assumption favors the dynamic adaptation scheme. When there are multiple threads per processor, some of the communication is local to the processors, so the message processing time can be reduced by clever schemes. However, the second assumption ignores this optimization and overestimates the execution overhead induced by the non-adaptation scheme.

Let $C_{LI,M}$, $C_{LB,M}$ and $C_{comm,M}$ be the costs due to load imbalance, load balancing overhead and communication respectively, when the application executes on M processors with one thread per processor. Using assumption (2) above, we define $O_{non-adaptation}$, the execution overhead per time step for the non-adaptation scheme, by

$$O_{non-adaptation} = C_{LI,NP} + C_{LB,NP} + C_{comm,NP}$$

where P is the number of processors and N is the number of threads per processor. We compute $O_{dynamic}$, the execution overhead per time step for the dynamic adaptation scheme, by

$$O_{dynamic} = C_{LI,P} + C_{LB,P} + C_{comm,P}$$

Let $R_{dynamic}$ represent the remapping overhead incurred by the dynamic adaptation scheme. Let T be the number of time steps until the sooner of the next reallocation or completion of the job. We compare the two schemes using the relative overhead of the non-adaptation scheme, which is given by

$$\text{Relative overhead} = (O_{non-adaptation} - O_{dynamic}) \times T - R_{dynamic}$$

When the relative overhead is negative, the non-adaptation scheme is preferable to the dynamic adaptation scheme, and vice versa.

In the next section, we present measurements obtained using *Ādhāra* that let us parameterize this model. In Section 13.2.3, we use those measurements to comment on the effectiveness of the dynamic adaptation scheme.

13.2 Measurements

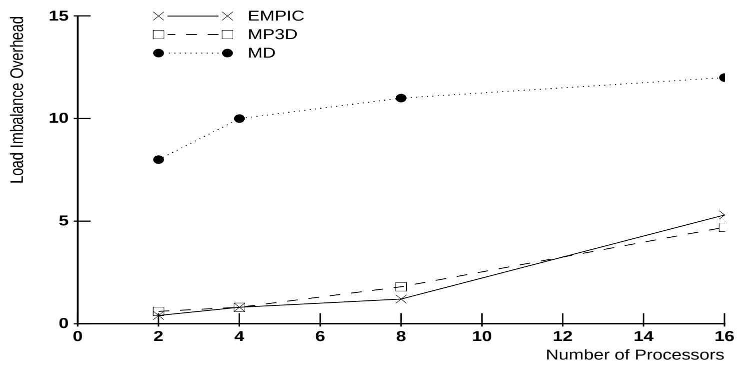
13.2.1 Execution Overheads

Figure 13.4 shows the effect of the number of processors on the execution overheads of the three example applications. As the number of processors increases, the overheads due to communication, load imbalance and load balancing increase, the reasons for which are discussed earlier. Note that the MD application exhibits unusual behavior – the communication cost first decreases and then increases. This is because, large amount of data is shared along the partition boundaries. For small number of processors (2 and 4), *Ādhāra* can use only the Slice and Beam schemes, which result in a high amount of data sharing due to large surface area of the space partitions. For larger number of processors (8 or more), *Ādhāra* can partition the space using the Block scheme, which minimizes the surface area of the partitions.

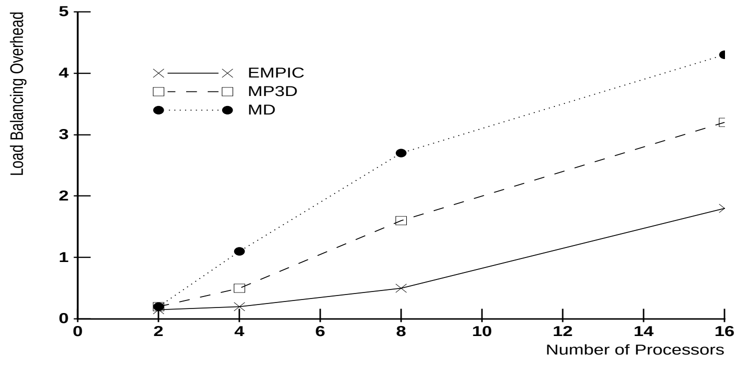
The effect of the number of processors on the total execution overhead is given in figure 13.2. Table 13.2 extracts the necessary data from this figure and presents the execution overheads incurred by the two schemes for different contractions from an initial allocation of 16 processors. Execution overhead for the non-adaptation scheme depends only on the total number of application threads (16), hence, it does not change when the allocation changes. On the other hand, execution overhead for the dynamic adaptation scheme depends only on the number of processors, hence, it decreases when the application is reallocated onto a smaller set of processors.

13.2.2 Reallocation Overhead

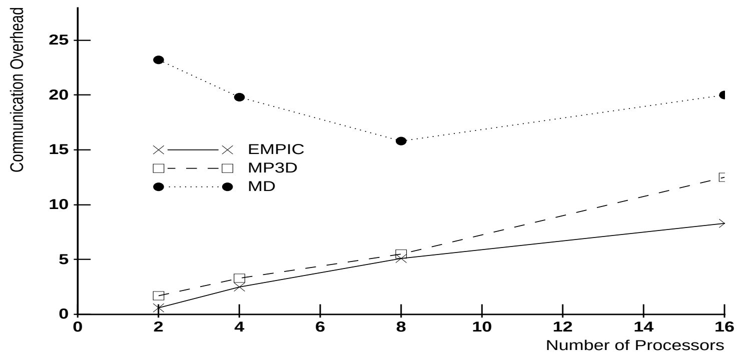
Table 13.3 gives the remapping overheads for the three applications when the initial allocation is 16 processors. These measurements are made by implementing the



(a) Load Imbalance Overhead



(b) Load Balancing Overhead



(c) Communication Overhead

Figure 13.4: Execution Overheads

(The overheads are given relative to the compute time per time step.)

Table 13.2: Execution Overheads for Different Processor Contractions
(*The overheads are given in terms of the percentage of the optimal compute time. The measurements are made using 32K particle inputs for the EMPIC and the MP3D applications, and 16K inputs for the MD application.*)

	Reallocation from 16 to 8 processors		Reallocation from 16 to 4 processors		Reallocation from 16 to 2 processors	
	$O_{non-adapt}$	$O_{dynamic}$	$O_{non-adapt}$	$O_{dynamic}$	$O_{non-adapt}$	$O_{dynamic}$
EMPIC	15.4	6.8	15.4	3.5	15.4	1.2
MP3D	20.4	8.9	20.4	4.6	20.4	2.5
MD	36.3	30.5	36.3	31.0	36.3	31.4

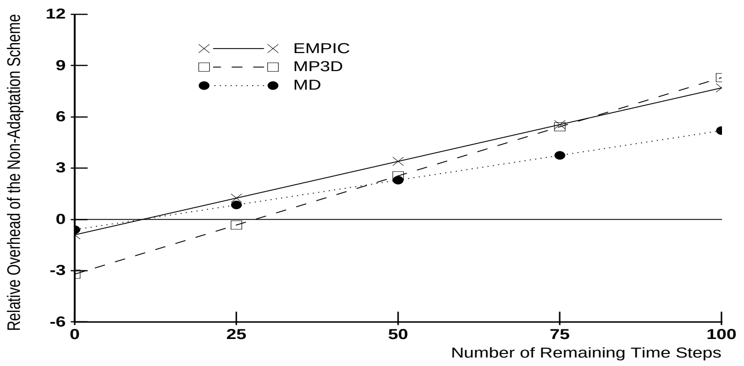
Table 13.3: Overhead for Adapting to Processor Reallocations
(*These measurements are made on 16 nodes of an Intel Paragon using 32K particle inputs for the EMPIC and the MP3D applications, and 16K inputs for the MD application.*)

Application	16 to 8 Processors		16 to 4 Processors		16 to 2 Processors	
	Absolute time (millisec)	Relative to compute time per time step	Absolute time (millisec)	Relative to compute time per time step	Absolute time (millisec)	Relative to compute time per time step
EMPIC	435	0.9	600	1.3	815	1.7
MP3D	475	3.2	615	4.1	785	5.2
MD	525	0.6	700	0.8	865	1.0

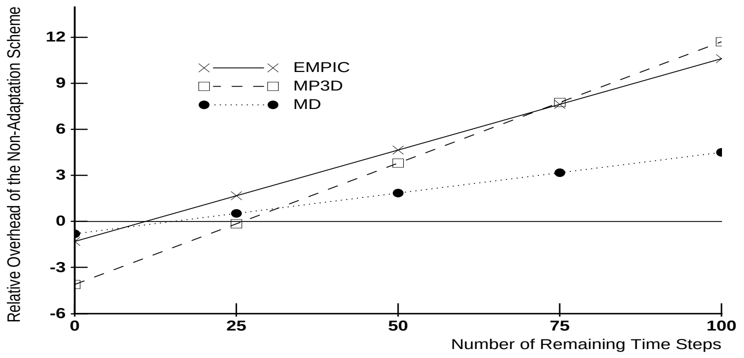
remapping facility in the *Ādhāra* runtime system and running experiments on a 16 node Intel Paragon. The results show that the reallocation overhead is quite small compared to the execution time. The relative reallocation cost is larger for the applications that have smaller computation granularity (such as MP3D).

13.2.3 Relative Overhead of the Non-Adaptation Scheme

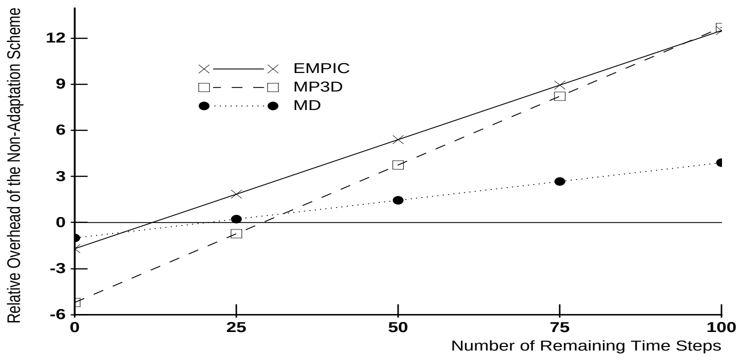
Figure 13.5 gives the relative overhead of the non-adaptation scheme for different values of T (the number of time steps until the next reallocation or completion of the job) and different processor contractions from an initial allocation of 16 processors. Note that all the overheads are measured relative to the compute time per time step. From the results, we can see that the dynamic adaptation scheme outperforms the non-adaptation scheme, unless the job is very close to completion (with less than 25



(a) Reallocation from 16 to 8 processors



(b) Reallocation from 16 to 4 processors



(c) Reallocation from 16 to 2 processors

Figure 13.5: Overhead of the Non-Adaptation Scheme relative to the Dynamic Adaptation Scheme

(*The overheads are given relative to the compute time per time step.*)

time steps remaining) or the next reallocation happens within the time required to execute 25 time steps. For our applications, this corresponds to a threshold of 4 to 20 seconds for small (32K) inputs and 40 to 200 seconds for large (320K) inputs.

13.3 Conclusions

In this chapter, we focussed on the question of how the runtime system should adapt to changes in processor allocation. The non-adaptation scheme, in which the application continues to execute with multiple threads per processor, incurs continuing cost due to sub-optimal load balance and increased communication. On the other hand, the dynamic adaptation scheme, in which the runtime system remaps the application data and adjusts the number of threads, incurs the cost of remapping. Based on the execution characteristics of the three dynamic space-based applications that we used for our experiments, we show that the dynamic adaptation scheme performs better than the non-adaptation scheme as long as the job is not too close to completion or the reallocations are not performed too often. For the applications that we used, the threshold is 25 time steps, which corresponds to 20 seconds for small (32K) inputs and 200 seconds for large (320K) inputs.

Part V

Summary and Conclusions

Chapter 14

CONCLUSIONS

This thesis investigates the issues in the design of a specialized runtime system for supporting dynamic space-based applications on distributed memory multiprocessors. It addresses both the issues in designing an appropriate programming interface and implementing efficient runtime support.

This chapter summarizes the contributions of this thesis and proposes some interesting topics for future research in runtime support on distributed memory multiprocessors.

14.1 Summary

This thesis addresses the issues in the runtime support for dynamic space-based applications on distributed memory multiprocessors, with emphasis on the issues in dynamic load balancing.

Dynamic space-based applications are simulations of objects moving through a closed k-dimensional space subject to mutual forces. There are a wide variety of such applications, differing in the kinds of objects and forces being simulated. A space-based simulation proceeds in series of time steps, each of which consists of one or more data-parallel computation phases.

We propose a new programming model, called *Ādhāra*, that supports mechanisms to operate on spatial data structures such as regular-grids and particles, to specify short-range spatial interactions, and to specify data-parallel phases. Data partitioning and load balancing are performed automatically by the system. The *Ādhāra* programming system enables the programmer to develop parallel codes involving very few additional lines of code and very few additional concepts beyond those required to construct a sequential version of the application.

We evaluate the *Ādhāra* programming model by parallelizing three dynamic space-based applications from different scientific fields: electro-magnetic particle-in-cell simulation from plasma physics, rarefied fluid flow simulation from aeronautics and molecular dynamics simulation from materials science. We measure the programming effort in terms of the additional lines of code that need to be written, in order to convert a sequential program into a parallel program. We show that the first two applications can be parallelized with less than 7% programming effort. The programming effort for the third application is negative, i.e., it is easier to develop an *Ādhāra* program than to develop a sequential program, since this application takes advantage of the high-level support provided by *Ādhāra* for operating on spatial data structures.

For choosing a good partitioning scheme automatically, we propose heuristics based on the application execution characteristics. By using three sample applications, we show that these heuristics are very effective in eliminating bad schemes. By measuring the overhead of changing the partitioning scheme for these applications, and showing that it is very small compared to the execution time, we emphasize the potential benefit of using these heuristics for dynamically adapting the partitioning scheme.

We propose a novel scheme based on non-uniform, adaptive discretization of the problem space for estimating the load distribution. This scheme, in which the load is discretized only along the boundaries of the regions, reduces the time for load estimation and computing a new partition, by minimizing the amount of information collected and shared among the processors. It uses a non-uniform, adaptive mesh to discretize the space, and adapts this grid dynamically, based on the load movement and density. We show that, if the load distribution changes gradually, this scheme maintains good load balance, while balancing the load six times faster than the traditional schemes that use a uniform, static mesh for discretizing the space.

We implement a hierarchical load balancing scheme for taking advantage of the preferential load movement. We show that, if the particles preferentially move along one dimension, this scheme reduces the load balancing time by balancing the load only along the dimension in which there is noticeable load movement.

We propose a predictive method for deciding how often to balance the load. This methods assumes that the average rate of increase of load imbalance does not vary much from one interval to the following interval. It uses the average rate from the previous interval to estimate the length of the next interval. We show that this method is very effective in dynamically adapting the frequency of load balancing based on the costs of load balancing and load imbalance.

Finally, we study the issues in adapting to processor reallocations performed by the operating system on a multiprogrammed parallel machine. We propose an approach for remapping the application data. We measure the reallocation overhead for the sample applications and show that it is advantageous for the runtime system to remap the data, unless the processor allocation changes too often or there is very little time left over to finish the job.

14.2 Future Research Directions

14.2.1 Supporting Other Types of Space-Based Data Structures

The programming model that we designed supports only regular-grids and particles. This model can be extended to support other space-based structures, such as adaptive, multi-level grids and irregular grids. The characteristics of the load distribution induced by these structures are different, hence the load balancing scheme and the heuristics for choosing a partitioning scheme need to be modified. The model could also be extended to support long-range forces which are used by several N-body applications.

14.2.2 Experimentation on Different Types of Large Scale Machines

We ran experiments only on an Intel Paragon consisting of 16 nodes. The effectiveness of the schemes that we proposed needs to be examined on larger number of processors and on a variety of distributed memory machines.

14.2.3 Using Other Applications

In our experiments, we used only three applications. The effectiveness of the *Ādhāra* system needs to be studied for other dynamic space-based applications that exhibit different execution characteristics.

Bibliography

[Agrawal et al. 93] G. Agrawal, A. Sussman, and J. Saltz. Compiler and Runtime Support for Structured and Block Structured Applications. In *Proceedings of the Supercomputing Conference*, pages 578–587, November 1993.

[Allen & Tildesley 87] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids.* Clarendon Press, Oxford, 1987.

[Anderson et al. 92] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, Volume 10(1), pages 53-79, February 1992.

[Ashok & Zahorjan 94] I. Ashok and J. Zahorjan. Adhara: Runtime Support for Dynamic Space-Based Applications on Distributed Memory Multiprocessors. *Proceedings of the Scalable High Performance Computing Conference*, May 1994.

[Baden 91] S. Baden. Programming Abstractions for Dynamically Partitioning and Coordinating Localized Scientific Calculations Running on Multiprocessors. *SIAM Journal of Science and Statistical Computation*, Volume 12, Number 1, pages 145–157, January 1991.

[Baden & Kohn 91] S. Baden and S. Kohn. A Comparison of Load Balancing Strategies for Particle Methods Running on MIMD Multiprocessors. *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, March 1991.

[Baden & Kohn 94] S. Baden and S. Kohn. A Robust Parallel Programming Model for Dynamic Non-Uniform Scientific Computations. *Proceedings of the Scalable High Performance Computing Conference*, May 1994.

[Belkhale & Banerjee 90] K. P. Belkhale and P. Banerjee. Recursive Partitions on Multiprocessors. *Proceedings of the 4th Distributed Memory Computing Conference*, pages 930–938, April 1990.

[Berger & Bokhari 87] M. Berger and S. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Transactions on Computers*, Volume C-36, Number 5, May 1987.

[Berryman et al 91] H. Berryman, J. Saltz, and J. Scroggs. Execution Time Support for Adaptive Scientific Algorithms on Distributed Memory Machines. *Concurrency: Practice and Experience*, Volume 3(3), pages 159–178, June 1991.

[Birdsall & Langdon 85] C. K. Birdsall and A. B. Langdon. Plasma Physics via Computer Simulation. McGraw-Hill International, New York, 1985.

[Bokhari et al. 93] S. Bokhari, T. Crockett, and D. Nicol. Parametric Binary Dissection. *ICASE Techincal Report No. 93-39*, NASA Langley Research Center, July 1993.

[Bozkus et al. 94] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation and Performance Results. *Proceedings of the Supercomputing Conference*, pages 351–360, November 1993.

[Bozkus et al. 94] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M. Wu. Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers. *Journal of Parallel and Distributed Computing*, Volume 21, Number 1, April 1994.

[Brugè & Fornili 90] F. Brugè and S. Fornili. A distributed Dynamic Load Balancer and its Implementation on Multi-transputer Systems for Molecular Dynamics Simulation. *Computer Physics Communications*, Volume 60, pages 39-45, 1990.

[Campbell et al. 90] P. Campbell, E. A. Carmona, and D. W. Walker. Hierarchical Domain Decomposition With Unitary Load Balancing for Electromagnetic Particle-In-Cell Codes. *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 943–950, April 1990.

[Chapman et al. 93a] B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima. Dynamic Data Distributions in Vienna Fortran. *Proceedings of the Supercomputing Conference*, pages 284–293, November 1993.

[Chapman et al. 93b] B. Chapman, P. Mehrotra, and H. Zima. High Performance Fortran Without Templates: An Alternative Model for Distribution and Alignment. *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, May 1993.

[Cybenko & Allen] G. Cybenko and T. Allen. Multidimensional Binary Partitions: Distributed Data Structures for Spatial Partitioning. *International Journal of Control*, Volume 54, Number 6, pages 1335–1352, 1991.

[Fallavollita et al. 92] M. A. Fallavollita, J. D. McDonald, and D. Baganoff. Parallel Implementation of a Particle Simulation for Modeling Rarefied Gas Dynamic Flow. *Computing Systems in Engineering*, volume 3, pages 283–289, 1992.

[Ferraro et al. 93] R. Ferraro, P. Liewer, and V. Decyk. Dynamic Load Balancing for a 2D Concurren Plasma PIC Code. *Journal of Computational Physics*, Volume 109, pages 329-341, 1993.

[Fincham 87] D. Fincham. Parallel Computers and Molecular Simulations. *Journal of Molecular Simulation*, Volume 1, 1987.

[Griswold et al. 90] W. G. Griswold, G. A. Harrison, D. Notkin, and L. Snyder. Scalable Abstractions for Parallel Programming. *Proceedings of the 5th Distributed Memory Computing Conference*, April 1990.

[Gupta & Banerjee 92] M. Gupta and P. Benerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers.

IEEE Transactions on Parallel and Distributed Systems, Volume 3, Number 2, March 1992.

[Gupta & Banerjee 93] M. Gupta and P. Benerjee. PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers. *Proceedings of the International Conference on Supercomputing*, July 1993.

[Hanxleden & Scott 91] R. Hanxleden and R. Scott. Load Balancing on Message Passing Architectures. *Journal of Parallel and Distributed Computing*, Volume 13, pages 312–324, 1991.

[Hinz 90] D. Y. Hinz. A Run-Time Load Balancing Strategy for Highly Parallel Systems. *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 951-961, April 1990.

[Hiranandani et al. 91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An Overview of the Fortran D Programming System. *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[Hiranandani et al. 94] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluating Compiler Optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, Volume 21, Number 1, April 1994.

[Hockney & Eastwood 88] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles.* Adam Hilger, Bristol, England, 1988.

[Kohn & Baden 93] S. Kohn and S. Baden. An Implementation of the LPAR Parallel Programming Model for Scientific Computations. *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, March 1993.

[Lanoski et al. 93] D. Lanoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, Volume 4, Number 1, January 1993.

[Leuze et al. 89] M. Leuze, L. Dowdy, and K. H. Park. Multiprogramming a Distributed-Memory Multiprocessor. *Concurrency: Practice and Experience*, Volume 1, pages 19–33, September 1989.

[Liewer & Decyk 89] P. C. Liewer and V. K. Decyk. A General Concurrent Algorithm for Plasma Particle-In-Cell Simulation Codes. *Journal of Computational Physics*, Volume 85, 1989.

[Lin & Snyder 90] C. Lin and L. Snyder. A Comparison of Programming Models for Shared Memory Multiprocessors. *Proceedings of the International Conference on Parallel Processing*, pages II 163–180, 1990.

[Lin & Snyder 93] C. Lin and L. Snyder. ZPL: An Array Sublanguage. *Proceedings of the Languages and Compilers for Parallel Computing Conference*, pages 96–114, 1993.

[McCann et al. 92] C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed, Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, Volume 11(2), pages 146-178, May 1993.

[McCann & Zahorjan 93] C. McCann and J. Zahorjan. Processor Allocation Policies for Message-Passing Parallel Computers. *Proceedings of ACM SIGMETRICS Conference*, pages 19-32, May 1994.

[McCann 94] C. McCann. Processor Allocation Policies for Message-Passing Parallel Computers. *PhD Thesis*, University of Washington, Seattle, 1994.

[McDonald 89] J. D. McDonald. A Computationally Efficient Particle Simulation Method Suited to Vector Computer Architectures. *Ph.D. Thesis*, Department of Aeronautics and Astronautics, Stanford University, December 1989.

[Ngo & Snyder 92] T. Ngo and L. Snyder. On the Influence of Programming Models on Shared Memory Computer Performance. *Proceedings of the Scalable High Performance Computing Conference*, 1992.

[Nicol & Saltz 88] D. M. Nicol and J. H. Saltz. Dynamic Remapping of Parallel Computations with Varying Resource Demands. *IEEE Transactions on Computers*, Volume 37, Number 9, pages 1073–1087, 1988.

[Nicol & Saltz 90] D. M. Nicol and J. H. Saltz. An Analysis of Scatter Decomposition. *IEEE Transactions on Computers*, Volume 39, pages 1337–1345, 1990.

[Nicol 91] D. M. Nicol. Rectilinear Partitioning of Irregular Data Parallel Computations. *ICASE Technical Report No. 91-55*, NASA Langley Research Center, July 1991.

[Park & Dowdy 89] K. H. Park and L. W. Dowdy. Dynamic Partitioning of Multiprocessor System. *International Journal of Parallel Programming*, Volume 18, Number 2, pages 91–120, 1989.

[Pilkington & Baden 94] J. Pilkington and S. Baden. Partitioning with Spacefilling Curves. *Technical Report No. CS94-349*, Department of Computer Science and Engineering, University of California, San Diego, March 1994.

[Pinches et al. 91] M. R. S. Pinches, D. J. Tildesley, and W. Smith. Large Scale Molecular Dynamics on Parallel Computers Using the Link-Cell Algorithm. *Journal of Molecular Simulation*, Volume 6, 1991.

[Raine et al 89] A. R. C. Raine, D. Fincham, and W. Smith. Systolic Loop Methods for Molecular Dynamics Simulation Using Multiple Transputers. *Computer Physics Communications*, Volume 55, 1989.

[Rapaport 91] D. C. Rapaport. Multi-million Particle Molecular Dynamics II. Design Considerations for Distributed Processing. *Computer Physics Communications*, Volume 62, pages 217–228, 1991.

[Reed et al. 87] D. Reed, L. Adams, and M. Patrick. Stencils and Problem Partitioning: Their Influence on the Performance of Multiple Processor Systems. *IEEE Transactions on Computers*, Volume C-36, Number 7, July 1987.

[Rogers 91] A. M. Rogers. Compiling for Locality of Reference. *Ph.D. Thesis*, Department of Computer Science, Cornell University, 1991.

[Rogers & Pingali 94] A. M. Rogers and K. Pingali. Compiling for Distributed Memory Architectures. *IEEE Transactions on Parallel and Distributed Systems*, Volume 5, Number 3, March 1994.

[Rosing et al 91] M. Rosing, R. B. Schnabel, and R. P. Weaver. The DINO Parallel Programming Language. *Journal of Parallel and Distrbitued Computing*, Volume 13, Number 9, pages 30-42, September 1991.

[Saltz et al. 91] J. Saltz, H. Berryman and J. Wu. Multiprocessors and Runtime Compilation. *Concurrency: Practice and Experience*, Volume 3(6), pages 573–592, December 1991.

[Singh et al 90] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory.

[Singh et al. 93] J. P. Singh, T. Joe, J. L. Hennessy, and A. Gupta. An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors. *Proceedings of the Supercomputing Conference*, pages 214–225, November 1993.

[Smith 91] W. Smith. Molecular Dynamics on Hypercube Parallel Computers. *Computer Physics Communications*, Volume 62, pages 229–248, 1991.

[Snyder 89] L. Snyder. The XYZ abstraction levels of Poker-like languages. *Proceedings of the Second Workshop on Parallel Compilers and Algorithms*, Urbana, Illinois, August 1989.

[Snyder 93] L. Snyder. Foundations of Practical Parallel Programming Languages. *Proceedings of the Second International Conference of the Austrian Center for Parallel Computation*, 1993.

[Setia et al. 93] S. Setia, M. S. Squillante, and S. Tripathi. Processor Scheduling on Multiprogrammed, Distributed Memory Parallel Systems. *Proceedings of ACM SIGMETRICS Conference*, pages 158-170, May 1993.

[Tucker & Gupta 89] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 159-166, December 1989.

[Walker 90] D. W. Walker. Characterizing the Parallel Performance of a large-scale Particle-In-Cell Plasma Simulation Code. *Concurrency: Practice and Experience*, Volume 2, pages 257–288, 1990.

[Weaver & Schnabel 92] R. Weaver and R. Schnabel. Automatic Mapping and Load Balancing of Pointer-Based Dynamic Data Structures on Distributed Memory Machines. *Proceedings of the Scalable High Performance Computing Conference*, April 1992.

[Williams 91a] R. Williams. DIME: A users manual. *Caltech Concurrent Computation Project report C3P 861*, February 1991.

[Williams 91b] R. Williams. Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations. *Concurrency: Practice and Experience*, Volume 3(5), pages 457–481, October 1991.

[Williams 92] R. Williams. Voxel Database: A Paradigm for Parallelism with Spatial Structure. *Concurrency: Practice and Experience*, Volume 4(8), pages 619-636, December 1992.