High Performance Computing Issues in Large-Scale Molecular Statics Simulations

Gautam Pulla

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Calvin J. Ribbens, Chair
Diana Farkas
Dennis Kafura

May 3, 1999
Blacksburg, Virginia

Keywords: High Performance Computing, Parallel Computing, Distributed Computing, Computational Environments

Copyright 1999, Gautam Pulla

High Performance Computing Issues in Large-Scale Molecular Statics Simulations

Gautam Pulla

(ABSTRACT)

Successful application of parallel high performance computing to practical problems requires overcoming several challenges. These range from the need to make sequential and parallel improvements in programs to the implementation of software tools which create an environment that aids sharing of high performance hardware resources and limits losses caused by hardware and software failures. In this thesis we describe our approach to meeting these challenges in the context of a Molecular Statics code. We describe sequential and parallel optimizations made to the code and also a suite of tools constructed to facilitate the execution of the Molecular Statics program on a network of parallel machines with the aim of increasing resource sharing, fault tolerance and availability.

Acknowledgement

I would like to offer my thanks to Dr. Calvin Ribbens for his valuable advice and encouragement throughout the course of this research work. His guidance was the cornerstone of this thesis. Dr. Diana Farkas played a major role in the shaping of this thesis, indeed, most of the work grew out of and was supported by the Materials Science department's project in molecular statics. Dr. Yuri Mishin of the Department of Materials Science at Virginia Tech. was kind enough to answer my questions on the functioning of the molecular statics code authored by him. I thank him for his cooperation and patience. I am indebted to Dr. Dennis Kafura for agreeing to serve on my comittee.

None of this would have been possible without the constant support of my family, them I thank the most of all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Issues in High Performance Computing

High performance scientific computing is a field which draws upon a diverse number of areas. In recent years it has moved in fascinating new directions. Of great importance and interest are developments in the field of computational environments which make it possible to use parallel machines with greater ease and efficiency [1, 9, 16]. These are systems or tools which provide a convenient interface for users of parallel machines. The ability to link several machines together and access them remotely through a network has made it possible to build such environments so that they span multiple machines. However, tools for supercomputers are still in a nascent state; there is still considerable room for improvement in systems that facilitate ease and efficiency of use of parallel machines.

Looking at another aspect of high performance computing, the problem of writing programs to run efficiently in time and space occurs here with increased importance. This is due to the fact that parallel applications are usually major consumers of computer time and resources, and thus good algorithms and implementations are vital. Writing good parallel programs is a task that draws on techniques not usually found in writing sequential programs, and can be a hard undertaking.

The object of the research described in this thesis is to identify and solve the most significant computational challenges which must be met in order to do efficient large scale simulations in a particular scientific application area, namely molecular statics. Successful high performance computational

science requires working within a variety of areas — from building systems to provide an effective environment for the parallel computation, to parallel enhancements made on the programs themselves. The underlying motif is that we need to work in several different directions to solve real problems.

Accordingly this thesis is as much about system building as it is about program optimization, and as much about distributed computing as it is about parallel computing.

1.2 The Problems

Scientists trying to solve problems on supercomputers have a universal refrain: as soon as a large problem is solved, the scientist almost immediately wants to solve a problem ten times as large. Consequently it is extremely important to use algorithms and data structures that scale well in terms of memory usage and computational needs. In the context of molecular statics, the main problem parameter which controls problem size is the number of atoms in the simulation. Informally "scalability" means that the program which ran a million atoms on a hundred processors yesterday ought to run with ten million atoms on a thousand processors today, or run with the million atoms ten times as fast. This performance scalability, in both space and time, is the first of the problems that we examine in this thesis.

The second broad problem that we address has to do with the computational environment. Applications for high end machines are almost by definition compute intensive, and need to use the machine for prolonged periods of time. This presents several administrative problems in a multi-user environment. Some users may have to wait for extended periods of time to allow others to finish running their programs, before they get a chance to access the machine. Specifically, a program that takes a very short time to run, may be held up indefinitely by one that ties up the machine for very long. Another problem is due to the number of processors needed by an application. Imagine an application that is started up on 80 processors on a 100 processor machine. If a second program needs 25 processors, it cannot run because only 20 are available (Note that we are assuming each application acquires exclusive control of a set of processors. This is a typical scheduling policy on massively parallel machines such as the Intel Paragon.) Another serious problem is that the results of extremely long simulations are lost if the machine crashes before the simulation finishes. Finally, many computational

scientists have access to machines of various sizes (under various workloads). Hence, there can be the need to move a long running simulation from one machine to another.

Given the expense of parallel machines, and their usefulness, a system which solves these problems would be invaluable. We describe the Parallel Application Control Environment (PACE), developed as part of this research, which solves the management problems for our particular computational environment.

The thesis is broadly divided into two logical parts. One part describes our work in optimizing the molecular statics program for parallel execution on a distributed memory multiprocessor. The other part of the thesis describes the design and implementation of the PACE system.

1.3 Molecular Statics Application

1.3.1 Basic Algorithm

The molecular statics program[1] used in this research simulates the physics of a metal crystal under various conditions, computing inter-atomic forces and finding the position of the atoms by minimizing the total energy using the conjugate gradient method. The code is written for distributed memory machines using the Message Passing Interface (MPI) [14, 15] in Fortran. Problems of interest use $O(10^5)$ to $O(10^6)$ atoms.

Briefly speaking, the program proceeds by representing internally various characteristics of the atoms in a three dimensional crystal, such as position, electron density, proximity of other atoms in the crystal, and so forth. A perturbation such as a defect or a crack is then introduced by changing the positions of some atoms. The forces on each atom, as a function of its neighbors attributes, are calculated and the position of each of the atoms is recomputed so as to minimize the total energy. This process is carried through several iterations of disturbance and relaxation.

[1]The sequential molecular statics program we describe in this thesis was written by Yuri Mishin of the Department of Materials Science, Virginia Tech.

1.3.2 Parallelizing the Computation

Parallelizing the code proceeds by dividing the atoms into groups and assigning the responsibility of computing the relaxation process on each group of atoms to a different processor. However each processor cannot proceed independently of the others. Since an atom assigned to one processor may affect atoms on another processor, any updates in the locations of atoms on a particular processor may need to be sent to other processors as well.

An existing implementation of statics was used as the source code base [2] [2]. However, this code had the shortcoming that though computation was parallelized in the manner described above, the space allocation was not. Every processor allocated storage for all atoms in the crystal. On every iteration, a processor would compute new coordinates for the atoms assigned to it, then all processors would communicate the updated locations to processors which needed them, and go on to the next iteration.

This approach permits a simple communications structure. If the atom coordinates are stored in an array for instance, a given atom has the same index on every processor. Using a message passing library, it is fairly easy to transmit updates to whoever needs them.

1.3.3 Size Scalability

A shortcoming of the above approach is that the problem size is bounded by the memory available to one processor. If all the atoms do not fit on a single processor, then no matter how many processors are made available, the problem cannot be solved. Thus the program does not scale well in terms of size.

Our aim is to modify the program and improve its size scalability while maintaining good parallel performance. This necessitates the design of data structures that are evenly divided amongst the processors. Since the problem is not "embarrassingly" parallel, we must also keep in mind the efficiency and simplicity of the communication algorithms.

Lastly, as we shall see, the problem of distributing atoms to processors is nontrivial. Some thought is needed to avoid gross imbalances in the load on each processor. Both computational and storage loads are important; we describe our approach to solving these problems in Chapters 3-5.

[2] This implementation was due to Brian Blount of the Department of Computer Science, Virginia Tech.

1.3.4 Message Passing Interface

The Message Passing Interface (MPI) is a specification for a message passing library intended to run on distributed memory multicomputers. The chief advantages of MPI are its portability and efficiency.

MPI provides a rich variety of functions and subroutines to exchange information between parallel tasks. The implementations described in this thesis occasionally use simplified versions of MPI function calls. The reader is referred to the book "MPI — The Complete Reference" by Dongarra, et al. [15] for more information on MPI.

1.4 Computational Environments

Computational environments are systems that make it easier to use, control or analyze the results of programs. In the context of high performance computing, computational environments can range over a sizeable domain, from full fledged distributed operating systems, to tools that make it easier to visualize results, to systems for enhancing fault tolerance. In this section we will describe some recurring themes in the area of computational environments for high performance scientific computing. The next chapter explores some example systems in depth.

1.4.1 Checkpointing

For our purposes, a checkpoint is the complete and *relevant* state of a computation, saved to non-volatile storage such as a hard disk. The process of taking a checkpoint is checkpointing. *Relevant* means that the entire core image of the executing computation need not be checkpointed. Often it is sufficient to consider only a subset of the process data.

The primary motivation for checkpointing is to guard against machine failures. High performance machines often have unstable operating systems and hardware [3]. Given that some computations can take days to run, the cost of failure is high. A checkpoint which can be used to resume a computation from a point before the failure, forestalls much of the harm caused by a crash.

[3]The HPC market for them is so small that there is relatively less work and fewer people working on making systems more robust. This is in contrast to operating systems used on personal computers and workstations for instance. Furthermore, massively parallel machines simply have more components that can fail.

Checkpointing has greater subtlety when one considers applying it to parallel computations. It is possible to view the state of a parallel computation as merely the union of the states of each executing parallel task within it. Or, taking a more sophisticated view, the state may be taken to be the state of the problem being solved, in a way independent of the number of tasks. Taking checkpoints in a way that is independent of the number of tasks also has the advantage that a checkpoint can be used to resume the computation with a different number of tasks (or processors) than what it was originally started on. As will be seen later, discovering such a state of the computation requires some analysis.

1.4.2 Migration of Processes

Process migration follows naturally from the concept of checkpointing. If we have versions of the same program on different machines, migrating a process is simply a matter of checkpointing it, transferring the checkpoint to another machine and loading the program on that machine with the transferred checkpoint.

An important consideration here is the possibility of differing data formats on the machines. This needs to be resolved by a translation mechanism from one format to another.

1.4.3 Queueing Environments and Distributed OS's

The design of systems that allow users to use migration effectively is a complicated and extensively researched area. In the following paragraphs we describe some relevant considerations in such systems.

A queuing environment is a system that lets users submit jobs to be executed at a queue for a network of machines. A queue manager repeatedly removes the job at the head of the queue, and schedules it for execution on a free machine selected using some load balancing/mapping algorithm.

Sometimes queueing environments may support task migration as well to enhance performance, but migration is often expensive, and considerable thought needs to be put into the question of when its use is appropriate. Some systems place control over this aspect into the users hands, while others employ sophisticated algorithms. All this is further complicated by the possibilities of machine and network failures. Security too is an issue when multiple users attempt to access multiple machines.

Distributed operating systems typically have many more functions than queuing systems, and many of those functions are irrelevant to our purpose. We will only look at those areas in distributed OS's which have some bearing on migration.

There are many queuing environments available, and many distributed operating systems, both as systems in research and commercially. We study some of these closely in Chapter 2.

1.4.4 Computational Steering

Computational steering has to do with modifying the internal state or variables of a running computation. This has great value if the computation takes very long to execute. A typical example involves a large search space that needs to be explored. If a scientist recognizes that the program is looking at a part of the search space that is uninteresting, he may choose to look in another direction. This is done by modifying certain parameters in the running program.

While computational steering is not directly relevant to molecular statics or to PACE, several steering systems also support checkpointing and migration. Hence, we briefly describe a well known system in Chapter 2. Also, in Chapter 10, on future work, we suggest certain enhancements to PACE that have to do with steering.

1.4.5 Metasystems

A metasystem is defined as a system formed by the composition of several systems, written in possibly different programming languages, running on possibly different architectures. An example we will explore later is the Earth-Science metasystem [13], which combines a program written for modeling the atmosphere and another for modeling the ocean, to create a more complete picture of weather in terms of these systems. Often such programs cannot be simply combined, as they are written and optimized for different machines. The coupling must therefore be loose and distributed.

Tools for building metasystems draw on the areas of distributed computing, language design and parallel computing. It is the author's opinion that PACE can be extended further and combined with the Unix shell scripting language, to produce a tool for building some types of metasystems. This possibility is more completely considered in Chapter 10.

1.5 Hierarchy of the Systems

This thesis can be viewed as attacking the molecular statics problem at several different levels, building many layers of software, as it were, one on top of the other. This is depicted in the figure below.

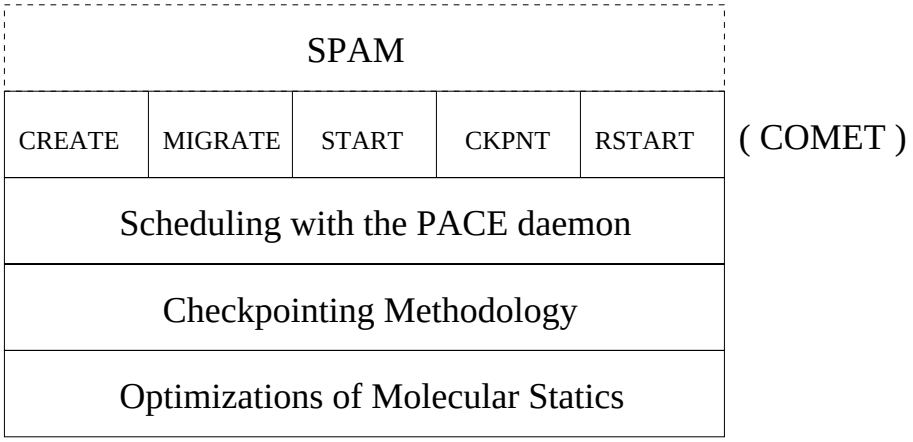

Figure 1.1: Hierarchy

At the bottom layer of the diagram are the optimizations on the parallel program. The checkpointing is a methodology (and also has a mini-API) for parallel programs generally, and has been tried out on the molecular statics program. Further above is the PACE scheduling daemon, which runs on a single machine. The COMET tools give users the abstraction of multiple machines running PACE, which can migrate programs between them. COMET thus serves to *distribute* PACE. The smaller boxes into which the COMET box is broken up, are the most important of the programs available.

Finally at the highest level is SPAM, the tool to build metasystems. This is shown in a dotted box as it is yet to be built. It will consist of some extra tools in addition to those currently in COMET and their combined use with shell scripts and PACE.

1.6 Organization of the Thesis

The thesis consists of two major parts, one describing the optimizations on the molecular statics program and another the design and implementation of PACE and COMET.

We discuss several contemporary systems similar to PACE, in part or whole, in Chapter 2. The idea will be to develop a feel for design choices.

Chapters 3 through 5 are about the optimizations on the statics program. We discuss in turn, the sequential version, the first parallelized version (due to Brian Blount, see [2]) and finally our improvements.

Chapters 6 through 9 discuss the various components of PACE, the rationale behind their design and some implementation details.

Finally Chapter 10 talks about possible further improvements in the statics code and some extensions to PACE to increase its usefulness.

Chapter 2

Contemporary Systems

2.1 Introduction

It is instructive to examine a number of computational environments before plunging into the design of PACE-COMET. The systems we describe shall provide a context for the design decisions we make with PACE-COMET. Some of the computational environments discussed here may appear to differ substantially from the system we propose. However, all the systems that we look at share with PACE-COMET the broad goals of improving utilization of high performance computing resources with an eye on distribution. Our aim is also to look at several important issues in high performance computing so that we may develop a feel for the special needs that arise in real applications. It is useful even to look at systems whose goals vary from PACE-COMET so we have a better understanding of both the *problem* and the *design* choices available to us; it is as important to pick the right problem to be solved as to find the right solution to a problem.

2.2 Checkpointing Systems

2.2.1 Binary Checkpointing

Binary checkpointing systems save the entire core image of the program as a checkpoint. Some systems that use this form of checkpointing are described in [4, 17].

Taking core images for checkpoints is a simple, un-complicated process

and there are no significant differences between the major systems that do this kind of checkpointing. Some method must exist to re-send messages that were in transit while the checkpoint was being taken, or a checkpoint must only be taken while no messages are in transit; this is the only major issue.

Binary checkpointing has the advantage that it is completely transparent to the user and the programmer and is language neutral. However, compared to the other possibilities we discuss below, binary checkpoints are unnecessarily large, particularly in the case of high performance computational applications that have large quantities of data. Also unlike the other strategies below, binary checkpoints restrict the program to use the same number of processors across checkpoints and preclude the possibility of process migration between heterogeneous machines.

2.2.2 Paradigm Oriented Checkpointing

Paradigm Oriented Checkpointing [19] is a system of checkpointing that provides some parallel programs the facility to take a checkpoint which could be used to restart them on a different number of processors than they were checkpointed on.

POC is used with MPI programs written using the Parallel Utilities - Regular Decomposition (PUL-RD) suite by the Edinburgh Parallel Computing Center. Essentially, PUL-RD allows programmers to write parallel programs by specifying a data decomposition and a skeletal operational interface for a parallel program. POC exploits this decomposition information to take checkpoints and decompose the data of a checkpoint differently when starting on a number of processors different from the number on which the checkpoint was taken.

The model of checkpointing used by POC is *user directed*. This means that the programmer has to explicitly write checkpointing code in the program by making function calls to a checkpointing API. When the checkpoint is taken is also controlled by the programmer explicitly. While this is a disadvantage so far as transparency goes, this strategy allows us to exploit the programmer's semantic knowledge of the program to take checkpoints that are portable across the number of processors. In addition, the checkpoints will usually be smaller than binary checkpoints, since only those program variables are saved which need to be.

Our checkpointing model is similar to the POC model in that we employ a user commanded approach and in that we take checkpoints that are portable

across the number of processors. We build a migration system around a POC like system, and our core checkpointing process is designed so it can easily coexist with the migration system. This implies building facilities like remote checkpoint triggering and restart, designing ways for the migration facility to communicate and control the checkpointing facility and so on.

2.2.3 CUMULVS

The CUMULVS project [12, 18] is an effort by the Oak Ridge National Laboratories to implement a distributed and fault-tolerant computational environment that provides migration, steering and visualization of parallel applications.

The approach taken by CUMULVS is to provide user directed checkpointing with a checkpointing API. The programmer must explicitly name the variables that must be saved as a checkpoint. The CUMULVS system then decides how to schedule and migrate the program on a network of high performance machines.

Another thrust area of CUMULVS is *computational steering*. This involves allowing a scientist to change parameters within the parallel program while it is running. Using this capability, the scientist can control the program's "direction" or *steer* the program.

Implementations of CUMULVS exist for programs using PVM and MPI. In many ways CUMULVS is a superset of the COMET-PACE system. The chief differences are in the user commanded nature of COMET-PACE and the ability to use COMET-PACE commands in a shell script. COMET-PACE is at a lower level than CUMULVS in that it puts a greater burden on the user than CUMULVS, but it affords a finer degree of control.

2.3 Migration

2.3.1 Overview of Design Issues

Process migration [7, 22] is a widely researched topic, and there is a plethora of design issues in building a migration system. Migration systems differ drastically in purpose and design, so we will not try to look at the entire spectrum of possibilities. Instead we shall concentrate only on those aspects of migration which seem most relevant to our situation.

An important concept while designing migration systems is that of *mechanism* and *policy* [10]. The mechanism is the means of moving context between machines while the policy specifies when the context should be migrated. One possibility for the policy is to have a load distribution algorithm, which attempts to use the resources in the "best" way possible. The definition of "best" can be varied — some typical definitions are maximizing utilization, maximizing throughput, maximizing availability, and so forth.

2.3.2 NQE

The Network Queueing Environment(NQE) [6] is a product of Silicon Graphics International/Cray Computer Corporation. This is a system designed to be installed on a network of computers. Users of NQE submit *batch requests* to a Network Queueing System (NQS) server, which schedules the requests on an appropriate machine using data supplied by a Network Load Balancer (NLB) about the load on the machines. NQE also provide fault tolerant file transfer through a File Transfer Agent (FTA) which can be instructed to repeatedly try a transfer till success in case a link fails.

An NQE *batch request* is a shell script and is submitted to the NQS using a GUI based submission program. There is also a command line based submission tool. From the point of submission onwards the request is completely managed by NQE. NQE also provides programs (GUI as well as command line) that enable a user to monitor the progress of a request.

NQE takes the route of controlling the lifetime of the program completely, deciding when to checkpoint, restart, migrate, etc. This is in contrast to our system which puts this responsibility entirely in the user's hands. The tradeoff here is whether the load balancing subsystem knows enough to make sound scheduling decisions. For our application this is not so. Moreover we also need a way for users to make use of time slots that have been pre-booked by them on particular machines. Finally the question of heterogeneous migration needs to be considered. NQE relies on binary checkpointing, which makes it impossible to migrate between differing machines.

2.3.3 Charlotte

Charlotte is a distributed operating system developed at the University of Wisconsin for a network of computers on an ethernet. In our discussion we only look at the process migration subsystem of Charlotte.

All computers on a network running Charlotte run an OS kernel that implements a process migration mechanism. The mechanism involves the three steps of negotiating the details of a migration between the source and destination machines, transfer of binary contexts and finally transfer of kernel data structures. The policy used by Charlotte is *automated* as opposed to a *manual* policy as in Sprite [17]. An automated policy uses an internal algorithm to decide on candidates for migration, possible destinations and also when to save the context of a migrating process and to halt it. A manual policy, on the other hand, relies on the user to supply all this data.

A key feature in Charlotte worth emulating (for some systems) is the separation of mechanism and policy. Charlotte implements the mechanism as part of the OS kernel, whereas the policy is a utility. This has the disadvantage of slight inefficiency, but it permits a cleaner design and also allows for changes to be made easily in future versions of policy. This is a principle followed in COMET-PACE.

2.4 Metasystems

Metasystems [1, 9, 13, 16] are a newly emerging sub-area in high performance computing. A metasystem attempts to seamlessly combine distributed resources such as programs, databases, machines and people[1]. The objective of most metasystems is to make it possible to build *wide area applications* by providing middle-ware that fuses distributed resources into one single virtual machine, that provides some or all of the following: scheduling, fault tolerance and great computational power.

Metacomputing — as computational science using metasystems is called — draws on many of the techniques outlined previously in this chapter, such as checkpointing, process migration (possibly heterogeneous), scheduling and load balancing, queuing systems, etc.

An important area in which metasystems are being developed and used is that of building component based metasystems. These are metasystems built by combining several parallel programs running on different machines connected by a network. Often the source codes of the component programs in such a metasystem are written in different languages by different research groups, thus making it a huge effort to combine them into a single program.

[1]When we speak of combining people, we imply that metasystems make it easier for people to collaborate.

Additionally, these programs may be optimized for execution on the specific computer for which they were written. It is thus invaluable to have some middleware that permits these programs to exchange data through a network, while they continue to run on their home machines. The programs need be modified very little; they only need to incorporate "entry points" which receive and send data, from and to the other components of the metasystem. A related issue in metasystems is that of security. Since the components may belong to different groups, they often represent intellectual property. Also since the metasystem is spread over a network, this introduces all kinds of possibilities for breaches in security that must be resolved by the middleware.

As excellent example of a metasystem is the Earth Science component metasystem [13]. It attempts to create a unified oceanographic and atmospheric model of the weather by combining the individual simulations for these models. Both components are written by groups of vastly differing specialties, making collaboration hard. The programs are also written for different machines, and run using different resolutions in time and space. Metacomputing provides the solution.

Metasystems embody many important principles in high performance computing nowadays: the need to support heterogeneity, exploiting distributed computer networks for higher performance and greater fault tolerance, the interdisciplinary nature of high performance computing and so on. It is useful to review them for a better understanding of which issues are of importance.

Chapter 3

Molecular Statics Simulation Code

3.1 Introduction

Before we look at optimizations of the Molecular Statics code, it is important to gain an understanding of the algorithms and data structures of the sequential version of the code. It is not necessary to examine the entire program as most of the execution time is spent only in some parts of the code, making it necessary to modify only those portions to improve performance. We will therefore identify a *kernel* — a portion of the code which is critical to performance — and focus our energies on that.

It is also helpful to look at a parallelization of the basic sequential code due to Brian Blount [2] which distributes computation among multiple processors. Our optimizations are based upon this parallel version of the code which we shall refer to as PMOLSTAT(this code written by Brian Blount is based on a sequential code written by Yuri Mishin which is referred to as MOLSTAT.)

3.2 Sequential Algorithms

To simplify our description of the algorithms of Molecular Statics, we define the following terms.

- *Position of an Atom.* A triple of floating point numbers representing the x, y and z coordinates of an atom in space. Each atom is associated

with a single position. Where the context is clear we shall refer to this attribute simply as the *position*.

- *Force on an Atom.* A triple of floating point numbers representing the force exerted on an atom as the three components of a vector in space. Each atom is associated with a single force vector acting on it. We shall use the term *force* for short, when we wish to refer to this attribute and it is clear that we use the term to refer to an attribute of an atom.

- *Electron Density on an Atom.* A single floating point number associated with each atom. This shall be referred to as the *electron density* for short, when it is clear that we are referring to an attribute of an atom.

The reader is cautioned not to attach too much physical significance to these attributes. They are important because they occur as data that the program manipulates. This definition is only correct and complete as far as it pertains to the molecular statics algorithms.

3.2.1 The Crystal and its Generation

The first significant step of the statics code is to initialize a two dimensional array of coordinates sized *Number of atoms* $\times$ *3*, which we shall call `co[]`. Each triple of elements of the array — `(co[i,1],co[i,2],co[i,3])` where `i` is some integer — holds the position of an atom of the crystal, and is initialized to an x, y and z coordinate respectively, by the `Crystal` subroutine.

In the future, when we talk of "atom `i`", we shall take it to mean the atom whose position is given by the coordinate (`co[i,1],co[i,2],co[i,3]`), where `i` is an integer.

The `Crystal` subroutine generates three types of atoms — *free*, *buffer* and *fixed*. An atom falls exclusively into one of these categories depending on its position. The regions over which these types extend are depicted in Figure 3.1 in two dimensions. The statics code operates on these atoms and the end result is the attributes of the free atoms. However we cannot have only free atoms in the crystal, for the free atoms close to the boundaries of the crystal would then not have as many atoms near them as atoms close to the center. This would lead to the simulation behaving incorrectly as far as the physics goes. The purpose of the fixed and buffer atoms is to provide the effect of having many atoms near these free and buffer atoms at the boundary.

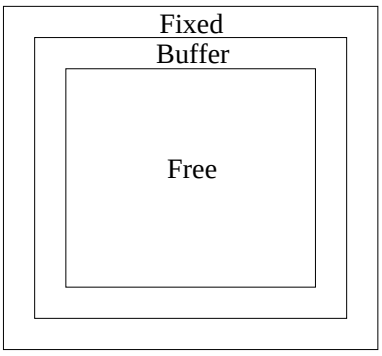

Figure 3.1: Two dimensional illustration of crystal structure.

The position and electron density of the fixed atoms are not modified, but fixed atoms contribute to the force exerted on free atoms. The buffer atoms have a similar purpose. They are interposed as a layer between the free and the fixed atoms to provide a gradual transition. Only the electron density of the buffer atoms is modified as the simulation proceeds; their position does not vary at all. In the case of the free atoms, we modify all three attributes — position, force and electron density.

The algorithm for the `Crystal` subroutine proceeds as outlined below. We use the notation `Array[*]` to refer to all elements of an array named `Array`. Thus `co[i,*]` refers to the position of atom `i` — (`co[i,1]`, `co[i,2]`, `co[i,3]`).

```
do i = 1, Number of Atoms
   Generate an x, y, z coordinate.
   Initialize an auxiliary array co1[i,*] to the coordinate.

Sort the atoms into three types so that all the free atoms
occur first, followed by all the buffer atoms and finally the
fixed atoms. The resulting sorted list is in co[].
```

Generation of the `x`, `y` and `z` coordinates is done by a simple algorithm, which takes as input — the dimensions of the (rectangular) crystal in terms of the number of atoms along the x, y and z directions, the number of chemical species of atoms, the interatomic distances and so forth, The algorithm then generates coordinates spaced in a regular fashion in space. The details of coordinate generation are irrelevant to the parallelization effort, and they only take up a minor portion of the execution time, so we will not discuss this

any further. The sort at the end of `Crystal` employs an algorithm similar to *count sort* [5]. The sorting is done so that there is a straightforward way to determine whether an atom `i` is free, fixed or buffer without a time consuming check as to what region it lies in. In a sorted list of atoms, all it takes to check the atom's type is an index comparison. For instance if `i` > *The Number of Free Atoms*, then `i` cannot be free.

3.2.2 Table of Neighbors

The `Table` subroutine is responsible for the creation of a *table of neighbors* for the atoms. This table of neighbors is a two dimensional array, which for every free and buffer atom in `co[]`, lists all other atoms whose position is within a *search radius* `R`. The neighborhood relation is formally stated in Definition 3.1.

Definition 3.1 We say that the atom `j` is a neighbor of `i` if and only if,

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \leq R$$

$$\begin{aligned} \text{where,} \quad & x_1 = co[i,1] \qquad x_2 = co[j,1] \\ & y_1 = co[i,2] \qquad y_2 = co[j,2] \\ & z_1 = co[i,3] \qquad z_2 = co[j,3] \end{aligned}$$

and `R` is the search radius.

This relation is expressed in the statics algorithms as,

$$neighbors[i,k] = j, \quad \texttt{for some } k \leq \texttt{The Number of Neighbors of } i$$

In other words, the neighbors of i are stored as neighbors[i,1], neighbors[i,2], neighbors[i,3]... neighbors[i, n], where n is the number of neighbors.

Note that the relation of neighborhood is symmetric, that is, `i` is a neighbor of `j` $\Leftrightarrow$ `j` is a neighbor of `i`.

The `Table` subroutine[1] proceeds by dividing the entire crystal into cubic blocks of side `R`. The blocks are numbered from 1 to the number of blocks.

[1]The sequential algorithm discussed here is by Yuri Mishin, but with a further space optimization. The original algorithm proceeded by dividing the crystal into cubical cells and generating a table of neighbours of these cells. Neighbors of atoms in a cell (of side

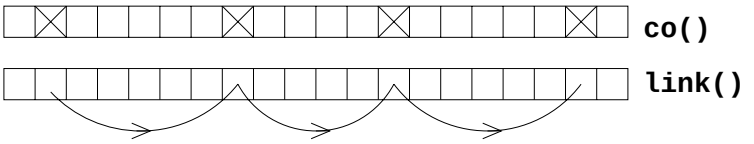

Figure 3.2: Linking blocks.

Therefore, two atoms can be neighbors only if they are located in the same block or in physically adjacent blocks. Thus, given an atom, the algorithm for the `Table` subroutine proceeds by retrieving its block number, and the block numbers of the blocks immediately adjacent. Then it needs to examine only the atoms in the adjacent blocks for the neighborhood condition. The algorithm for `Tables` is as follows.

```
do over all free and buffer atoms i
   k = 0
   b = block to which i belongs
   do over all c = {b and blocks neighboring b}
      do over all atoms j in c
         if(i and j are close enough)
            neighbors[i,k] = j
            k = k + 1
```

For the algorithm to work well, it should be possible to efficiently iterate through all the atoms of a block when given its block number. This is done by maintaining a linked list for each block in `co[]`. This is shown in Figure 3.2. The Xs mark atoms in the same block. A "block assignment" subroutine makes a pass through `co[]` initially (before `Tables`), building up the linked lists. Note that the array `link[]` can be used to hold linked lists for all blocks simultaneously. The overhead of computing blocks and linked lists of atoms is well worth it because it improves the table creation algorithm from $O(n^2)$ to $O(n)$ which has a significant impact on its performance in large cases.

equal to search radius) would thus lie in neighbor cells as given by the table of neighbors of cells. In the sequential code this table of neighbors of cells is not a big space bottleneck as compared to the arrays of atom attributes, but in the parallel code, which optimizes away the space occupied by the atom attributes, this becomes a very big liability. Therefore this table of neighbors of cells was eliminated by slightly changing the sequential table of neighbors algorithm. For the purposes of the discussion in this thesis we may safely ignore the changes after noting that they were made.

One thing should be noted — the algorithm presented above is actually a simplicfication of the algorithm used in MOLSTAT. We do this to simplify our discussion. In MOLSTAT, there is the notion of "displacements" of atoms. The position of each atom n_1 is diplaced by a multiples of a certain distance to yield positions $p_1, p_2, p_3, \ldots$ Then for each atom n_2, we check if it is a neighbour of n_1 where the position of n_1 is taken as one of the p_i. If n_1 and n_2 are indeed neighbours, they are listed as such in `neighbors[]`, except that they are not neighbours in the sense described earlier, since this neighbourhood relationship is defined in terms of the displaced position of n_1 rather than its real position. This is indicated by setting the corresponding element of a logical array, `realn`[$\mathtt{n}_1$][$\mathtt{n}_2$] to `FALSE` (otherwise this is `TRUE`). The amount of displacement is stored in an integer array in the element `idisp`[$\mathtt{n}_1$][$\mathtt{n}_2$]. Since the displacements in the X, Y and Z directions are only integral multiples of constant displacements `dispx`, `dispy` and `dispz` (one for each direction), we only store the integer factor in `idisp[][]` to conserve space. The motivation for having displacements is as follows. In a typical simulation we get best results when we have more atoms. However, more atoms mean more storage, so there is a limit to how high we can go in terms of the number of atoms. Instead, we use the atoms we already have and displace them to get many more imaginary atoms. The results are closer to the physical reality when we use these imaginary displaced atoms. The algorithm described above needs to be modified slightly to take the displacement strategy into account.

```
do over all free and buffer atoms i
   k = 0
   do id = -ndispx, ndispx
      displace atom i by id*ndispx
      b = block to which the displaced i belongs
      do over all c = {b and blocks neighboring b}
         do over all atoms j in c
            if(i and j are close enough)
               neighbors[i,k] = j
               if(id is not 0)
                  realn[i,k] = FALSE
                  idisp[i,k] = id
               else
                  realn[i,k] = TRUE
               k = k + 1
```

In the above algorithm `ndisp` is the "number of displacements" in the X direction. To keep the algorithm short we only consider displacements in the X direction. `dispx` is the unit displacement and all other displacements are multiples of `dispx`.

We will not discuss the displacement strategy again even though it affects almost all the other algorithms. This is due to two things — first, once we have explained the concept of displacements, and given an example of how it can be easily incorporated into the statics code, it should be easy for the reader to see how the rest of the code is similarly modified to deal with the displacement strategy. Secondly, our improvements to the code are not greatly affected by the displacement strategy. In other words, the algorithms as we describe them in this thesis are not significantly different in their implementation when we incorporate displacements. Given these facts we choose to ignore the changes introduced by the displacement strategy and instead opt to present a simplified version of the algorithms.

3.2.3 Aberrations

Once the table of neighbors has been initialized, the next step is to introduce some kind of aberration in the crystal. This may be a crack, a point defect, a dislocation or a planar defect. The subroutines to do this are not important from our point of view. They are neither dominant in terms of execution time, nor do they require any non-trivial changes during parallelization. In essence all they do is change the positions of some atoms in `co[]` by displacing them, or removing atoms from `co[]` by marking them as removed in an auxiliary array.

3.2.4 Relaxation

The relaxation subroutine uses the Conjugate Gradient method to compute a new set of positions for the free atoms to minimize the total energy of the crystal. This works in two steps — first, a subroutine `Enforce` is called which calculates the force and electron density for the atoms[2], and the total energy of the crystal. Then this atom attribute information is used by the conjugate gradient subroutine to try different positions for the atoms and attempt to

[2]Not for *every* atom. As outlined in 3.2.1 we may be interested in only some attributes of atoms depending on whether the atoms are free, buffer or fixed.

Table 3.1: Typical execution times for MOLSTAT.
t_{10} is the time taken by MOLSTAT for 10 iterations, and $t_{Enforce}$ is the time spent in `Enforce`.

Free Atoms	$t_{Enforce}$	t_{10}	Percentage Time
450	67.8	68.5	99.0%
1250	164.0	166.0	98.8%
1800	227.7	230.2	98.9%
5000	589.7	596.7	98.8%
9800	2190.9	2217.8	98.8%

find a configuration that minimizes the total energy of the crystal. The conjugate gradient subroutine makes repeated iterations of calling `Enforce` and modifying the atom positions. In effect it traverses a search space to find a minimum energy set of atom positions, and each step exploring the search space is guided by the atom attributes in the current iteration.

`Enforce` is the most time consuming step in the molecular statics code (This is clearly apparent in Table 3.1). Therefore, it is `Enforce` which must be parallelized first and best.

The energy of the crystal is the sum of the energy contributions due to each free and buffer atom in the crystal. We call this contribution γ_i where `i` is a free or buffer atom. Each such γ_i is made up of contributions that we shall call α_{ij}, made by an atom `j` which is a neighbor of `i`. The α_{ij} contribution depends upon the position of atoms `i` and `j`[3]. The total energy of the crystal is given by the formula below.

$$E = \sum_{i \in Free} \gamma_i \tag{3.1}$$

$$= \sum_{i \in Free} \sum_{j \in Nbors(i)} \alpha_{ij} \tag{3.2}$$

Free is the set of free atoms in the crystal and *Nbors(i)* is the set of atoms which are neighbors of `i` according to Definition 3.1.

The force on an atom `i` is a function of the positions of `i` and `j`, where `j` is a neighbor of `i`. The array `G[]` keeps track of the force on each free atom.

[3]Specifically, it depends upon the interatomic distance and the species of atoms.

`G[i]` holds the force on atom `i`. For simplicity only one component of the triple for the force vector is shown in the algorithms that follow. Calculations are analogous for the other components.

Electron density is similar to force — given a free or buffer atom `i`, each neighbor `j` which is a free or buffer atom, makes a contribution to the electron density of `i`. The array `R[]` keeps track of the electron density on an atom, with `R[i]` holding the electron density of atom `i`.

The algorithm for `Enforce` appears below.

```
1  E = 0
2  do over all free and buffer atoms i
3    G[i] = 0
4    do over all neighbors j of i
5       E = E + e(i,j) * 0.5
6       if(i <= nfree)
7          G[i] = G[i] + g(i,j)
8       if(i <= nfree + nbuffer)
9          R[i] = R[i] + r(i,j)
```

The function `e(i,j)` returns the energy due to the interaction of the atoms `i` and `j` and corresponds to the α_{ij} of Equation 3.2. The order of the arguments does not matter. The reason for the 0.5 factor in line 5 is that each pair of neighbors is listed twice over in `neighbors` due to symmetry. However the energy due to the interaction between `i` and `j` must be added only once per pair of neighbors, thus the factor of one half.

The quantities `nfree` and `nbuffer` stand for the number of free and buffer atoms respectively. The test in line 6 checks if an atom is of type free and the one in line 8 checks to see if it is either free or buffer. This is the reason for sorting the atoms during crystal generation, so that there is a quick and easy way to check the type of an atom. The alternative test, using the coordinate of the atom to find out which region it lies in is too inefficient to be used in the "inner loop" of `Enforce`.

The function `g(i,j)` gives the force exerted by atom `j` on atom `i`. When the order of the arguments is reversed, the sign of `g` is reversed.

The function `r(i,j)` is the contribution of atom `j` to the electron density on atom `i`. Like `g`, the sign of `r` is reversed when the order of its arguments is reversed.

3.2.5 Symmetry

We have noted the symmetry of the neighborhood relation. This suggests one space saving optimization — if atoms `i` and `j` are neighbors, then by Definition 3.1, both the atoms will have listed the other as neighbors in `neighbors[]`. This approach uses up twice as much space as is necessary to convey the same *information*, that `i` and `j` are neighbors.

The space used by `neighbors[]` can be cut by half if we set `neighbors[i,k] = j` (for some `k`) only when `i` and `j` are neighbors and `i` $<$ `j`. Since the relation "$<$" is not symmetric, this ensures that `i` is not listed as `j`'s neighbor if `j` is listed as `i`'s neighbor. This optimization is shown in the modified algorithm for `Tables` shown here.

```
do over all free and buffer atoms i
   k = 0
   b = block to  which i belongs
   do over all blocks c = {b and blocks adjacent to b}
      do over all atoms j in c
         if(i < j AND i and j are close enough)
            neighbors[i,k] = j
            k = k + 1
```

On the other hand, if we only list atom `j` as atom `i`'s neighbor, but not `i` as `j`'s neighbor, then the relaxation algorithm from Section 3.2.4 would not work right. This is because, while computing the force on atom `j` the algorithm iterates only through the atoms listed as neighbors of `j` in `neighbors`. Since atom `i` is not listed as a neighbor, the computation is incorrect. Similarly the calculation of total energy and electron density produces incorrect results.

A simple modification of `Enforce` allows symmetry to be exploited here as well. The contribution of the force exerted by a neighbor `j` on an atom `i` is added in as before, but at the same time, the force exerted by `i` on `j` is computed and added to the force on `j`. A similar modification is made to the computation of the electron density. Correct calculation of energy under this scheme is done by eliminating the 0.5 factor from the sequential algorithm, for each contribution. This is because each pair of neighbors is only listed once in our new scheme. The modified algorithm is as follows.

```
E = 0
G[*] = 0
R[*] = 0
do over all free and buffer atoms i
   do over all neighbors j of i
      E = E + e[i,j]
      if(i <= nfree)
         G[i] = G[i] + g(i,j)
      if(j <= nfree)
         G[j] = G[j] + g(j,i)   /* g(j,i) = -g(i,j) */
      if(i <= nfree + nbuffer)
         R[i] = R[i] + r(i,j)
      if(j <= nfree + nbuffer)
         R[j] = R[j] + r(j,i)   /* r(j,i) = -r(i,j) */
```

3.3 Parallelizing Computation

We now turn to the parallel molecular statics code, PMOLSTAT, developed by Brian Blount. In PMOLSTAT dividing computation among processors is done by "assigning" some atoms to each processor. Each processor is then responsible for applying the same steps outlined in the sequential algorithm to the atoms assigned to it. Although the atoms are divided among the processors for the sake of parallelizing computation, each processor holds the location of every atom. This is because an atom assigned to a processor may be affected by an atom assigned to another processor. Thus it may be necessary for a given processor to store the positions of atoms not assigned to it. PMOLSTAT deals with this situation by storing all atoms on all processors.

3.3.1 Crystal Generation and Dividing Atoms

The crystal generation is as before, only each processor calls the `Crystal` subroutine independently and redundantly. At the end of `Crystal`, each processor has a complete copy of the positions of all the atoms in `co[]`.

Now each processor assumes responsibility for some of the free and buffer atoms in the crystal. This set is determined by two integers — `start` and `end`. A processor owns all atoms `i` such that `start` $\leq$ `i` $\leq$ `end`, where

`start` and `end` are different on different processors. The exact algorithm for computing `start` and `end` is only of passing interest with respect to our main purpose. As long as it allocates roughly equal numbers of atoms to each processor so that the load imbalance is not too great, the exact method used is not important. Each processor gets a contiguous subset of atoms to work on.

3.3.2 Table of Neighbors and Enforce

The parallel algorithm used by PMOLSTAT for the `Table` subroutine is given below. The table of neighbors is distributed across the processors, with each processor holding the neighbors for only the atoms assigned to it.

```
1  do i = start, end
2     k = 0
3     b = block to which i belongs
4     do over all blocks c = {b and blocks adjacent to b}
5        do over all atoms j in b
6              if(i < j OR i and j are on different processors)
7                 if(i and j are close enough)
8                    neighbors[i,k] = j
9                    k = k + 1
```

Observe how this algorithm compares with the sequential algorithm in 3.2.5. One change is that the loop over all atoms in the sequential algorithm is replaced by a loop iterating over atoms from `start` to `end`. The more interesting change is the test in the `if` statement at line 6 checking if `i` and `j` are on different processors. The reason for this is that in the sequential version of `Enforce`, the program modifies `G[j]` at the same time as it modifies `G[i]`, where `j` is listed as a neighbor of `i`. However in the parallel version it is quite possible that `i` and `j` are on different processors. Assuming `i` $>$ `j`, if we used symmetry as in Section 3.2.5, then `G[i]` would not get updated by the processor which was assigned atom `j` even if `i` and `j` were neighbors. We can solve the problem by not taking advantage of symmetry when the neighboring atoms are on different processors. By setting up the table of neighbors in this selectively symmetric way, the `Enforce` subroutine works correctly. Pseudo code describing how the parallelized version of `Enforce` works is shown below. `Enforce` is executed in parallel on each processor.

```
1  E = 0
2  G[*] = 0
3  R[*] = 0
4  do i = start, end
5     do for all neighbors j of i
6        if (i and j are on the same processor)
7           factor = 1.0
8        else
9           factor = 0.5
10       E = E + e(i,j) * factor
11       if(i <= nfree)
12          G[i - start + 1] = G[i - start + 1] + g(i,j)
13          if(j <= nfree AND j is on this processor)
14             G[j - start + 1] = G[j - start + 1] + g(j,i)
15       if(i <= nfree)
16          R[i - start + 1] = R[i - start + 1] + r(i,j)
17          if(j <= nfree + nbuffer AND j is on this processor)
18             R[j - start + 1] = R[j - start + 1] + r(j,i)
19  call mpi_allreduce(Etot, E)
```

The final call to `mpi_allreduce` at line 19 is a *reduction* step. This adds up the values of all the `E`'s on the different processors and puts the result into `Etot`.

The `factor` at line 6 and 9 is used to account for our selectively taking advantage of symmetry. Where we do not use symmetry and the same pair of neighbors is listed twice over (each time on a different processor), the value returned by `e()` must be halved to account for the contribution of the interaction only once.

Each processor holds only the values of `G[]` and `R[]` corresponding to atoms assigned to it. Therefore, when we use the `G` attribute and the `co` attribute of an atom, we must translate the indices to the atom in `co[]` which are "global" indices (from 1 to the number of atoms) to "local" indices (ranging from 1 to the number of atoms assigned to the processor running `Enforce`). This is the reason why we do not use `i` and `j` as the indices to `G[]` and `R[]` in the algorithm described above.

3.3.3 Communication

The one remaining problem to be solved is that of communication. We have mentioned earlier that the `Conjug` subroutine modifies the attributes of the free and buffer atoms depending on the attributes of the atoms in the crystal. In PMOLSTAT, this translates to each processor modifying the positions of the atoms assigned to it. This means that the modifications are not visible to other processors. After these modifications, the changed positions of the atoms must be communicated to the other processors before `Enforce` is called again in the next iteration. If not, this will lead to incorrect results if a processor calling `Enforce` uses the position of any atom not assigned to it and the position of that atom was modified by another processor in the previous iteration. This happens if the neighbor of an atom assigned to a processor p_1 lies on another processor p_2. Thus if p_2 modifies the neighbor's position, the calculation of the energy (for instance) due to interaction of the atoms, is based upon the incorrect position of the neighbor at p_1.

In PMOLSTAT communication is achieved by each processor sending the positions of atoms assigned to it, after `Conjug` has modified them, to all other processors whose atoms have neighbors assigned to this processor. Only the free atoms need be sent, as the positions of the fixed and buffer atoms are not changed. Keeping track of which processors need which atoms is done in `Tables`. The algorithm for tables is modified slightly. When `Tables` decides that two atoms n_1 and n_2 are neighbours, if they are on different processors, say p_1 and p_2, then `Tables` should should register somehow that p_1 needs an atom on p_2 and vice versa. This is done by means of an array `nprocs_to_send_to[]` which keeps a list of all processors which have neighbouring atoms on the processor holding the array. For example, both p_1 and p_2 would list each other as an entry in `nprocs_to_send_to[]`. In the communication step during PMOLSTAT, each processor scans through its entries in `nprocs_to_send_to[]` and posts an asynchronous send for all the atoms assigned to it (that is, the attributes of the atoms assigned to it) and also a receive for the atoms assigned to the processor listed in `nprocs_to_send_to[]`. Thus when communication completes each processor has the correct and up to date attributes for all the atoms. One more thing to note is that each processor needs to know the `start` and `end` values for any processor it receives from. This is easy to ensure, since we divide the atoms into equal sized contiguous chunks and assign a chunk to each processor.

The costs incurred to record the communication information are small,

since the table creation routine where this information is collected is not a significant bottleneck and the collection of information occurs only once.

Other Details

Parallelizing of the rest of the conjugate gradient method in PMOLSTAT is straightforward. We do not discuss the methods used since simple loop parallelization suffices to produce a good parallel version of the code. The cost of these portions of the code is also negligible, so there is not much point in examining them. For the rest of this thesis the focus will be on the `Enforce` subroutine.

Chapter 4

Optimization of PMOLSTAT

4.1 Introduction

In this chapter we describe our optimizations to PMOLSTAT. We identify bottlenecks in PMOLSTAT, why they arise and how to eliminate them. The main emphasis is on reducing the per-processor memory requirements and reducing the cost of communication. Our intention is to design a highly scalable version of this molecular statics simulation code.

The idea of scalability in parallel programs is a very important one. Essentially all parallel algorithms try to solve problems faster than sequential algorithms by using more processors. This however, introduces the necessity to divide the work among the processors in a "sensible" fashion. If for instance, one processor is given almost all the work, clearly there will be little improvement in performance. In addition to computational work it is also important to divide space requirements evenly (assuming a distributed memory context). A strategy that divides the work and space of a problem evenly among processors, irrespective of the number of available processors and the size of the problem, is a necessary condition for scalability. In real life many parallelization strategies fail to work equally well for all ranges of processor numbers and problem sizes. Careful analysis and design may be needed to discover a good way to parallelize computation and space in a scalable manner.

4.2 Scalability

When considering the implications of memory requirements on scalability, it is useful to distinguish between two types of large data structures: *sequential* and *parallel*. A *sequential data structure* grows in size in direct proportion to the problem size, and is not influenced by the number of processors. It consumes $O(n)$ memory on each processor, where n is the problem size. In contrast, a *perfectly parallel data structure* uses up $O(n/p)$ memory per processor, where n is as before and p is the number of processors being used. The per processor memory requirement of a program with only sequential and parallel data is therefore given by the following Equations.

$$m_{parallel} = \alpha\frac{m}{p}, \tag{4.1}$$

$$m_{sequential} = (1-\alpha)m, \tag{4.2}$$

where,

α is the proportion of the data which is perfectly parallel.
m is the memory requirement of the problem on a single processor.
p is the number of processors.
$m_{parallel}$ is the memory requirement of the parallel data.
$m_{sequential}$ is the memory requirement of the sequential data.

If a program has a sequential data component, the largest problem size that can be tackled is limited by the memory requirement of the sequential portion of the data, even if we can indefinitely increase the number of processors available to solve the problem. The program is thus limited in how well it scales towards larger problem sizes.

The parallel molecular statics code divides computation evenly among processors, and succeeds in achieving some size scalability by breaking up the table of neighbors — `neighbrs[]`. However there is one set of data structures — the `co[]` array and a family of arrays which are used in a way similar to `co[]` — which are sequential. For convenience we shall refer to this set of arrays, namely `co[]`, `co1[]`, `lco[]` and `lco1[]` as *CO* collectively.

Table 4.1: Space requirements of PMOLSTAT.

Array	Size
`neighbrs[ ]`	$\frac{200ni}{p}$
`idisp[ ]`	$\frac{600ni}{p}$
`workspace[ ]`	$\frac{15nf}{p}$
`R[ ]`	$\frac{nf}{p}$
`fi[ ]`	$\frac{3nf}{p}$
`G[ ]`	$\frac{3nf}{p}$
`co[ ]`	$3nf$
`co1[ ]`	$3nf$
`lco[ ]`	$3ni$
`lco1[ ]`	$3ni$

4.2.1 Sequential Bottleneck

Our immediate objective was to run the code well on about a 100 processors of the Intel Paragon. It is important to know exactly what the memory bottleneck is for that number of processors before we proceed with any optimization.

Table 4.1 lists the major data structures in the statics code and the amount of space they occupy. The i represents the space occupied by an integer and f represents the space occupied by a double precision floating point number. The symbols n and p, as before, stand for the problem size (number of atoms) and the number of processors used to solve the problem, respectively.

It is apparent from Table 4.1 that `neighbrs[]`, `idisp[]`, `workspace[]`,

`R[]`, `Fi[]` and `G[]` are perfectly parallel, and everything else is purely sequential. For a hundred processors, the ratio of space occupied by parallel data to sequential data is given by

$$\begin{aligned} \frac{m_{sequential}}{m_{parallel}} &= \frac{3nf + 3nf + 3ni + 3ni}{\frac{200ni+600ni+15nf+nf+3nf+3nf}{100}} \\ &= 2.132 \end{aligned} \tag{4.3}$$

assuming that $f = 2i$.

It is clear that CO is the major consumer of memory when about 100 processors are used. If we were to successfully parallelize these arrays, their per processor memory requirement would also scale down as the number of processors increased. A strategy that perfectly parallelized the sequential arrays would reduce the ratio in Equation 4.3 to the following.

$$\begin{aligned} \frac{m_{sequential}}{m_{parallel}} &= \frac{\frac{3nf+3nf+3ni+3ni}{100}}{\frac{200ni+600ni+15nf+nf+3nf+3nf}{100}} \\ &= 0.021 \end{aligned} \tag{4.4}$$

In other words, perfect parallelization would reduce the memory requirement of the sequential data component from being the dominant component to being an insignificant component. It is also clear from these formulae that CO becomes an increasingly significant bottleneck as the number of processors is increased. In PMOLSTAT the limit on the number of atoms possible for a simulation, in real memory (no virtual memory) is about 10,000.

4.3 Data Structures

Our analysis suggests that we must decompose CO evenly among the processors. This turns out to be equivalent to the problem of decomposing `co[]` evenly among the processors, for the statics code performs very similar operations on all arrays in CO . Therefore, in the algorithms and data structures we discuss, we will only speak about `co[]`[1].

[1]We only discuss algorithms and data structures that are the most *representative* of the optimizations made. In fact, the statics code is considerably more complex than what

4.3.1 CBlocks

We tackle the problem of decomposing `co[]` by dividing the entire crystal into a number of *Communication Blocks* or *CBlocks*. These CBlocks are cuboids (rectangular boxes), and their dimensions are picked to suit the problem[2]. An atom is located in a CBlock if its position falls within the CBlock. CBlocks do not overlap, so each atom can be located in in one and only one CBlock. We shall refer to an atom as *belonging* to a CBlock when we wish to indicate that it is located in the CBlock.

Atoms are affected by other atoms located within a distance δ, the search radius which defines an atom's neighborhood. So given a CBlock m, it is possible to identify a set of CBlocks which have some atoms that influence atoms in m and in turn are influenced by some atoms in m. This is illustrated in Figure 4.1 in two dimensions. Essentially, we draw a rectangle centered around m and extending in each of the three directions by a further distance δ. Then we find out which CBlocks this rectangle intersects. This set of CBlocks is the region of influence for m which we denote as R_m. The concept of identifying the region of influence using CBlocks is similar to the idea of using blocks in the algorithm for the table of neighbors (Section 3.2.2), but not exactly the same, because CBlocks can have varying sizes as opposed to blocks which are always cubes of side δ. At this point of time we make no assumptions about the relative magnitudes of the CBlock dimensions and δ.

Each processor is assigned a set of CBlocks such that every CBlock is assigned to some processor, and no two processors are assigned the same CBlock. These CBlocks are called the $owned_p$ set of CBlocks for a processor p. A processor must apply the conjugate gradient method to the atoms belonging to CBlocks in $owned_p$. In applying the conjugate gradient method, some of the atoms in $owned_p$ may be influenced by atoms not in the set. Therefore each processor must also keep track of the attributes[3] of atoms belonging to CBlocks which are in the region of influence of the CBlocks in $owned_p$. The set of CBlocks which are assigned to other processors, but which are in the region of influence of CBlocks assigned to this processor, is

we describe here. For simplicity of discussion we only talk of simplified versions of the algorithms and data structures which are used to optimize the statics code. Thus the algorithms and data structures we describe are used in more parts of the code than we discuss.

[2]The effect of CBlock dimension on program execution is discussed in Chapter 5.

[3]Recall that we defined the attributes of an atom as position, electron density and force in Section 3.2.

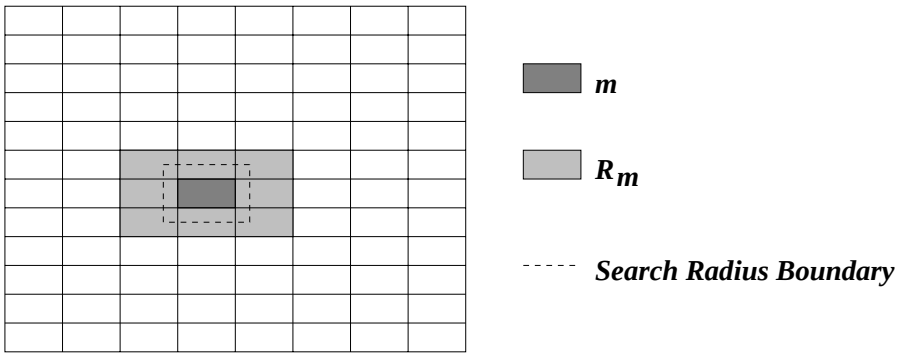

Figure 4.1: CBlocks within the region of influence of a CBlock m.

called $needed_p$.

While applying the conjugate gradient method to the atoms in $owned_p$, a processor can modify the attributes of those atoms. However the attributes of atoms in $needed_p$ are read-only. The processor which modifies attributes of atoms in a particular CBlock must send the updated attributes to all processors which have that CBlock in their $needed_p$ sets.

Each CBlock is uniquely identified by a *CBlock number* which is an encoding of the x, y and z coordinate for the system of CBlocks within the crystal. We perform the following operation $x \times 1000000 + y \times 1000 + z$ to encode the coordinate. Thus the CBlock number of a CBlock located at $(20, 10, 3)$ in the CBlock coordinate system would be 020010003.

4.3.2 Send and Receive Tables

The communication algorithm in a strategy using CBlocks can be complicated since we try to communicate only some of all atoms assigned to a processor. This is handled by having each processor initialize a `CBlock_send[]` table and a `CBlock_recv[]` table. Both of these tables are arrays of integers such that,

`CBlock_send[p, j]` gives the CBlock number of the `j`th CBlock to be sent to processor `p`, where `j` ranges from 1 to the number of CBlocks to be sent to `p`. `CBlock_recv[i]` gives the CBlock number of the `i`th CBlock to be received, where `i` ranges from 1 to the number of CBlocks to be received. A separate array is used to keep track of which processor owns the `i`th CBlock in `CBlock_recv[]`.

The determination of which CBlocks to send and receive must be done so that every processor p receives the contents of the CBlocks in $needed_p$.

4.4 Parallel Algorithms

4.4.1 Overview

The main difference from the PMOLSTAT code, introduced by using CBlocks, is that the atoms are assigned to processors in a different manner. Communication is built upon the principle that each processor only needs CBlocks that influence CBlocks assigned to it.

Each processor is assigned a set of CBlocks whose atom attributes it updates during each iteration of the conjugate gradient method. The atoms in these CBlocks may be influenced by atoms in CBlocks assigned to other processors if those atoms are close enough to be neighbors. These extra atoms (their attributes, that is) are needed by a processor even though it may not modify their position or any other attribute. After each iteration of conjugate gradient, processors may modify the locations of atoms that are in CBlocks assigned to them. These processors must then convey the new locations to other processors which may need those atoms. Once the communication step is successfully accomplished, conjugate gradient can go on to the next iteration.

4.4.2 CBlock Creation

Assigning CBlocks

The first step that the statics code must take is to compute $owned_p$ for each processor. For the moment we ignore the specifics of the algorithm for computing $owned_p$; this is explored later (See Chapter 5). For now we concentrate on making our design as general as possible, i.e., adaptable to any strategy of computing $owned_p$. The set of CBlocks $needed_p$ is initialized to empty. Next each processor must calculate the region of influence for each CBlock in $owned_p$. Every CBlock in the region of influence is tested to see if it is already in $owned_p$ or $needed_p$. If not, it is added to $needed_p$. At the end of this step, each processor knows all the CBlocks it needs.

Generating Send and Receive Tables

The algorithm for initializing the send and receive tables is actually embedded in the CBlock assignment algorithm. Once the CBlocks in $owned_p$ have been assigned to each processor, the following is executed on every processor.

```
1  do p = 1, Number of processors
2     needed_p = ∅
3     do over all mb ∈ owned_p
4        Get the region of influence  R_mb  for mb
5        do over each CBlock mbr ∈ R_mb
6           Get p_mbr, the owner of mbr
7           if(mbr ∉ owned_p ∪ needed_p)
8              Add mbr to needed_p
9              if(p is equal to myrank)
10                Write mbr as
                    being received from p_mbr
11             else
12                if(myrank is equal to p_mbr )
13                   Write mbr as being sent to p
```

Each processor executes the algorithm above. Consider an example — suppose processor 3 has CBlock 10 assigned to it which is close enough to CBlock 15 assigned to processor 6 to be able to influence atoms in 15. Thus, processor 3 must note that CBlock 10 needs to be sent to processor 6, and

likewise processor 6 must note that CBlock 10 needs to be received from processor 3. Initially $10 \in owned_3$. Thus when the `do` loop in line 1 iterates with p set to 6, both processor 3 and 6 would enter the `do` loop in line 3 and iterate over $owned_6$. Now both processors would calculate $\mathtt{R_{mb}}$ for each CBlock in $\mathtt{mb} \in owned_6$. Since CBlock 10 is in the region of influence of CBlock 15, it follows that for $mb = 15$, $10 \in \mathtt{R_{15}}$. Thus when $\mathtt{R_{15}}$ is iterated through, both processor 3 and 6 eventually set the the loop variable of the `do` loop in line 4 (that is — `mbr`), to 10 eventually, at line 5. The $\mathtt{p_{10}}$ at line 6 is of course processor 3, the owner of CBlock 10. Now CBlock 10 is certainly not in $owned_6$ as it has been assigned to processor 3, not 6. Thus if it has not yet been added to $needed_6$ it will be now, on both processors 3 and 6 at line 8. At lines 9 and 12, the `if` tests evaluate differently for processors 3 and 6.

- For processor 3, line 9 evaluates to false (`myrank` is simply the ID of the processor executing the program). So it executes line 12 where the test evaluates to true. Thus processor 3 adds `10` to the list of CBlocks to be sent to `p`, that is processor 6, by writing this into the send table.
- For processor 6, line 9 evaluates to true. So CBlock 10 is written to the receive table as being received from from its owner, that is $\mathtt{p_{10}} = 3$

It is easy to see that other processors which have nothing to do with CBlock 10 will fail the tests in both line 9 and line 12 and thus will not modify their send or receive tables.

4.4.3 Modifying Crystal Generation

The `Crystal` subroutine must be modified to store only the atoms that are in $owned_p \cup needed_p$. The modified algorithm is shown below.

```
j = 1
do i = 1, Number of Atoms
   Generate an x, y and z coordinate
   Get mb, the CBlock to which (x, y, z) belongs
   if(mb ∈ owned_p ∪ needed_p )
      co1[j, *] = (x, y, z)
      iactual[j] = i
      j = j + 1
Sort the atoms so that free atoms come first,
followed by buffer atoms and finally fixed atoms.
Within these classes of atoms (free, fixed and
buffer) the atoms owned by this processor should
precede all atoms needed by it.  This is a stable
sort.  Output array is co[].
```

The stability of the sort method in the final step of the `Crystal` algorithm ensures that if any two atoms n_1 and n_2 occur in a CBlock m, then if $m \in owned_p \cup needed_p$ for more than one processor p — on all processors, n_1 and n_2 will occur in the same order — in fact the order in which `Crystal` generated them. This fact is crucial to the design of the communication algorithm and we formally prove it in Section 4.4.4.

Finally, the array `iactual[]` stores for each atom, an index number that is the same on each processor for a given atom. This index is the loop variable `i`, which is the iteration in which an atom was generated, and can be thought of as a global atom number. We use `iactual[]` during the symmetry optimization in table creation. The use of `iactual[]` is discussed further in Section 4.4.5.

The sorting method used in `Crystal` is a variation of *count sort*, which is $O(n)$. Thus the total complexity of the initialization phase is unaffected by sorting.

4.4.4 Compacting CBlocks

At the end of the `Crystal` subroutine the `co[]` array contains the position of all atoms in the CBlocks assigned to each processor or needed by each processor. However, these atoms are not ordered in any straightforward way. The atoms of a CBlock would typically be scattered all over `co[]`.

This complicates matters if we wish to send the contents of a CBlock to another processor, since message passing favors sending a CBlock as a single, contiguous chunk of data. Therefore we must rearrange the positions of atoms in `co[]` so that all atoms in a CBlock are situated contiguously. This is done by the `Compact` subroutine. `Compact` is a stable sort over `co[]` using the CBlock number as the key — with the exception that CBlocks in $owned_p$ are considered to have keys less than CBlocks in $needed_p$.

Another requirement of communication is that for a given CBlock occurring on multiple processors, the atoms should occur in the exact same order on all processors. This is so that if a processor sends the positions (for example) of those atoms as a single message, and another processor receives those positions as a single message, the *nth* atom position within the chunk should correspond to the same atom on both processors. By using a stable sort, both when we sort the atoms in `Crystal` and in `Compact` we ensure this. To see that this is true consider atoms n_1 and n_2 which occur on a CBlock m that is on processors p_1 and p_2.

- If n_1 and n_2 are of differing atom types.

  Then since atoms are sorted as free, buffer and fixed, in that order, within a CBlock, n_1 and n_2 occur in the same order on p_1 and p_2.

- If n_1 and n_2 are of the same atom type.

  The first sort through `Crystal` is a sort by type. As the types are the same, the sort being stable preserves the order of occurrence of n_1 and n_2 (and this is the order in which `Crystal` generates them, which is identical on both processors). The second sort which occurs during `Compact` is a sort by CBlock number. Since both atoms have the same CBlock number m, the original order is preserved.

Another reason for compaction is that it is important to have an efficient way to perform two operations. These operations, described below, are used in a critical part of the statics code. The precise algorithms used to perform these operations are discussed in Section 4.4.6.

- Is a given atom free, fixed or buffer?

  The motivation for this test is that depending on the type of atom, we may or may not modify a particular attribute. This was discussed in Section 3.3.2. This can be accomplished by ensuring that atoms in a

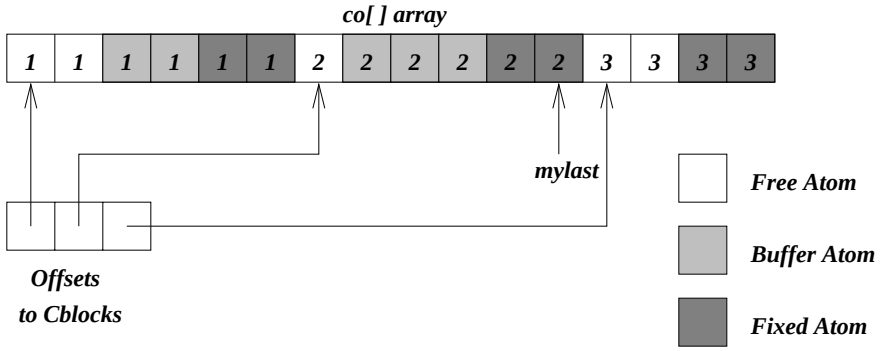

Figure 4.2: `co[]` after compaction.

CBlock occur in the following order — free first, buffer next and finally fixed. If this is so, then we can determine the type of an atom by a simple index comparison.

- Does a given atom belong to a CBlock in the set $owned_p$?

  This operation is used during our selective employment of the symmetry optimization (Section 3.2.5). The ownership test is efficient if all the (positions of) atoms belonging to CBlocks in $owned_p$ occur before all the atoms belonging to CBlocks in $needed_p$, within the array `co[]`. Thus there is an index `mylast` before which all atoms are in a CBlock from $owned_p$ and after which all atoms are in a CBlock from $needed_p$.

The output of `Compact` is illustrated in Figure 4.2. The numbers in the boxes represent the CBlock numbers of the atoms. There are three CBlocks, numbered 1, 2 and 3 on the processor for which `co[]` is shown. The atoms are color coded to signify their type as free, buffer or fixed. The set $owned_p = \{1, 2\}$, thus `mylast` = 12. The atoms within any particular CBlock and within any particular atom type, are in the same order as they were generated by `Crystal`.

The diagram also shows an array of offsets. We need to be able to quickly access the atoms of any given CBlock to iterate through its atoms. This we do by maintaining a table of offsets into `co[]` which gives us the index of the first atom of any CBlock occurring in `co[]`. This table of offsets is initialized by `Compact`. Other data structures initialized by `Compact` include a table for the number of atoms per CBlock and tables of the number of free, buffer

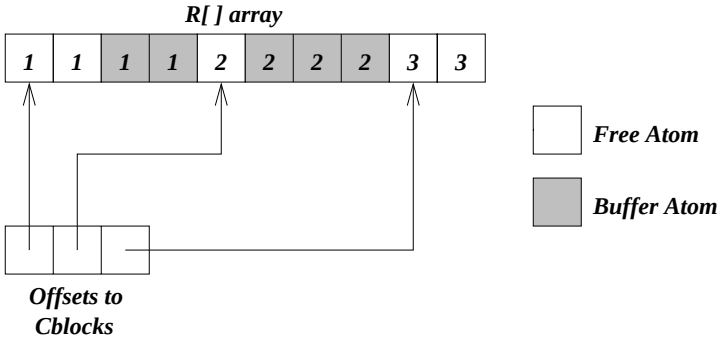

Figure 4.3: Arrangement of atoms in `R[]`.

and fixed atoms in a CBlock. All of these are used to access the atoms in a CBlock.

The array `G[]` which stores the force vector and `R[]` which stores the electron density, store those attributes of atoms in the same arrangement as `co[]`. `G[]` is meaningful only for free atoms and `R[]` is meaningful for just free and buffer atoms. For example, the equivalent arrangement of the `R[]` array should be as depicted in Figure 4.3.

4.4.5 Tables of Neighbors

The table of neighbors algorithm is modified very little from the basic parallel version of MOLSTAT except that now it must find the neighbors for the atoms in $owned_p$. The neighbors are looked for among $owned_p \cup needed_p$, that is, all the atoms on the calling processor. The atoms on each processor are split up into blocks (not to be confused with CBlocks) as in the previous parallel version of neighbors. The algorithm is given below, followed by an explanation of the main operations.

```
1  do over all atoms i = iterfreebuf(owned_p)
2     b = block to which i belongs
3     k = 0
4     do over all c ∈ {b and blocks neighboring b}
5        do over all atoms j in c
6           if(iactual[i] < iactual[j] OR
                    j > mylast)
7              if(i and j are close enough)
8                 neighbrs[i,k] = j
9                 k = k + 1
```

Iteration

The function `iterfreebuf` at line 1 returns the free and buffer atom indexes in a serial fashion. Since the arrangement of atoms while using CBlocks is different from when we don't, iteration is different too. Most importantly, the free and buffer atoms owned by a processor do not occur contiguously. Thus while iterating through them, the offset and count tables described in Section 4.4.4 must be used. The `iterfreebuf` function implements the following algorithm.

```
'i' is a static variable initialized to CBlock_offset[1].
'mb' is a static variable initialized to 1.
if(i <= CBlock_offset[mb] + CBlock_nfree[mb] +
                    CBlock_nbuffer[mb])
   return_value = i
   i = i + 1
else
   mb = mb + 1
   if(mb > Number of owned CBlocks)
      return_value =  -1 signifying that all free/buffer atoms
      have been iterated through.
   else
      return_value = CBlock_offset[mb]
return return_value
```

Note that `mb` is not the CBlock number. Rather it is an index into the

tables which hold CBlock information — `CBlock_offset[]` which holds the offset (within `co[]`) to the CBlock, `CBlock_nfree[]` which holds the number of free atoms for each CBlock and `CBlock_nbuffer[]` which stores the number of buffer atoms for each CBlock. The CBlock numbers are poorly suited for use as indices into these tables. For instance, consider a crystal divided into $2 \times 2 \times 2$ CBlocks. The CBlock at (1, 1, 1) would have a CBlock number of 100100100 which is too large an index to be used given that there are only 4 CBlocks in the crystal.

Same Processor Test

In Figure 4.2 we illustrate how all atoms assigned to a processor have index less than or equal to `mylast` due to compaction. Thus the test `j > mylast` checks to see if `j` is on the same processor as `i`.

Symmetry

Since compaction rearranges `co[]` we can no longer use the original test for symmetry described in Section 3.2.5. Instead we must use the `iactual[]` array. If any given atom `n` has index $\mathtt{n}_\mathtt{p}$ into `co[]`, on processor `p`, then `iactual`$[\mathtt{n}_{\mathtt{p}_1}]$ and `iactual`$[\mathtt{n}_{\mathtt{p}_2}]$ have the same order (with respect to the $>$ relationship) for any pair of processors $\mathtt{p}_1$ and $\mathtt{p}_2$ on which `n` is present.

4.4.6 Relaxation

The relaxation algorithm `Enforce` is almost the same as given in Section 3.3.2. The algorithm is given below. An outline of the main differences from the previous version follows.

```
1  E = 0
2  G[*] = 0
3  R[*] = 0
4  do over all atoms i = iterfreebuf(owned_p)
5     do for all neighbors j of i
6        if(j <= mylast)
7           factor = 1.0
8        else
9           factor = 0.5
10       E = E + e(i, j) * factor
11       if(i is free)
12          ifree = get_free_index(i)
13          G[ifree] = G[ifree] + g(i, j)
14          if(j is free AND j  ∈ owned_p)
15             jfree = get_free_index(j)
16             G[jfree] = G[jfree] + g(j, i)
17       if(i is free or buffer)
18          ifreebuf = get_free_buf_index(i)
19          R[ifreebuf] = R[ifreebuf] + r(i, j)
20          if(j is free or buffer AND j ∈ owned_p)
21             jfreebuf = get_free_buf_index(j)
22             R[jfreebuf] = R[jfreebuf] + r(j, i)
23 call mpi_allreduce(Etot, E)
```

Type Test

Due to the compaction step, arrays like `co[]` are no longer organized so that all the free atoms come first, followed by the buffer atoms and finally the fixed atoms. Therefore determining whether an atom is of type free or buffer, such as in lines 11, 14, 17 and 20 is not as simple as the previous version of `Enforce`. The following algorithm ascertains whether an atom `i` is of type free.

```
'b' is the CBlock to which atom i belongs
'offset' is the offset to first atom of b
'nfree' is the number of free atoms in b
if(i <= offset + nfree)
   i is a free atom
else
   i is not a free atom
```

The steps should be self explanatory. The algorithm for determining whether an atom is free or buffer is very similar.

```
'b' is the CBlock to which atom i belongs
'offset' is the offset to first atom of b
'nfree' is the number of free atoms in b
'nbuffer' is the number of buffer atoms in b
if(i <= offset + nfree + nbuffer)
   i is a free or buffer atom
else
   i is not a free or buffer atom
```

Index Conversion

Now consider line 12-13. These calculate a new value for `G`, the force on an atom, if it is free. The array `G[]` is arranged in a fashion similar to `co[]` after compaction except that it does not have any elements for either buffer or fixed atoms. Thus it has its own table of offsets, and given some `i`, `co[i, *]` and `G[i]` do not correspond to the same atom. There must be a translation of an index into `co[]` for a free atom, to an index into `G[]` for that same atom. The formal expression for this translation is given below.

> `b` = CBlock to which i belongs
> Index of `b` into offset tables = `offset_index[i]`
> Index into `G[]` = `i` − Offset of `b` into `co[]` + Offset of `b` into `G[]`

A table, `offset_index[]` (initialized by `Compact`), stores for each atom, the index into the offset tables (a given CBlock has the same index for all offset tables).

The operation for translating an index into `co[]` to an index into `R[]` is similar — only we use the offset tables for `R[]` rather than `G[]`.

4.4.7 Communication

Each processor uses the send and receive tables described in 4.3.2 to determine which CBlocks to send to which processor, and also which CBlocks to receive from which processor. When we talk about sending or receiving CBlocks, we of course refer to sending updates of the positions of atoms belonging to those CBlocks[4]. The algorithm to do this is expressed as follows. It should be self explanatory.

```
do p = 1, Number of processors
   n = Number of CBlocks to send to this processor
   do i = 1, n
      b = CBlock_send[p, i]
      off = Offset of b into co[]
      num = Number of free atoms in b
      Nonblocking send num locations in co[] starting from off,
      to p
do i = 1, Number of CBlocks to receive
   b = CBlock_recv[i]
   p = Owner of CBlock b
   off  = Offset of b into co[]
   num = Number of free atoms in b
   Nonblocking receive num locations in co[] starting from off,
   from p
Wait for all Nonblocking sends and receives to complete
```

[4]In the statics code, actually the attribute *electron density*, stored in `R[]`, also needs to be communicated. However the algorithm for communication there is very similar to the one for communicating `co`, so we do not describe it here.

Chapter 5

Performance Analysis

5.1 Introduction

The optimizations outlined till now have been mainly in the form of basic changes to the algorithms and data structures from PMOLSTAT. In addition to this some more fine-tuning may be required to achieve our target, namely, to fit around 10^6 atoms on 100 processors (without using virtual memory). These optimizations are relatively minor, but achieving our goal is impossible without them.

In addition we also need to analyze the OPMOLSTAT code to verify that there are no significant performance bottlenecks. This consists of looking at three aspects of OPMOLSTAT — load balance, communication cost and the overhead imposed by our new data structures. If we can show that the costs due to each of these are reasonable, we shall have gone a long way towards understanding the parallel performance of OPMOLSTAT and its scalability.

Our aim is to compare the performance OPMOLSTAT with a hypothetical, "best possible" parallelization of MOLSTAT. If MOLSTAT could indeed be perfectly parallelized, such a parallelization would be characterized by the following attributes.

- The total work to solve the problem should not have increased. In other words, any additional data structures or algorithms used should not be incur too great a cost. Additional work may be added in two ways — maintenance of parallel data structures and communication costs.

- The work should be divided perfectly among processors. This ensures

that we shall get better performance (in time) as we use more processors, in direct proportion to the number of processors used.

- The total storage requirements should not have increased. This means that any additional data structures should not take up more than a minimal amount of space. This criterion also considers data that is stored redundantly (replicated) on multiple processors as "additional" data. For example, in our case the $needed_p$ set of atom attributes is redundant.

- The storage requirements should be evenly divided among the processors. If this is so, we can solve increasingly larger problems by simply adding more processors.

In this chapter we shall measure the performance of OPMOLSTAT with respect to all these criteria. Ideally these criteria should be satisfied for all problem sizes and for all numbers of processors. In practice very few algorithms approach this ideal. We shall instead concentrate on looking at the "operating range" of OPMOLSTAT. Our objective is to calibrate the performance of OPMOLSTAT for $O(10^5) - O(10^6)$ atoms on $O(10) - O(10^2)$ processors of the Intel Paragon.

Further Memory Bottlenecks

In Table 5.1 we show the sizes of various data structures in OPMOLSTAT. The parameter δ is due to the distribution of CO being imperfect. In Section 5.3.2 we discuss it further and show that it is relatively small. For now we will assume that the contribution by δ is minor enough to ignore.

Assuming as usual that $f = 2i$, it is plain that the dominant cost is the array `idisp[]`. This array holds displacements for each neighbor of an atom such that the `j`th neighbor of `i` is displaced by:
`idisp`$[$`i`$,$ `j`$,1] \times$ `xdisp` in the x-direction,
`idisp`$[$`i`$,$ `j`$,2] \times$ `ydisp` in the y-direction and
`idisp`$[$`i`$,$ `j`$,3] \times$ `zdisp` in the z-direction,

where `xdisp`, `ydisp` and `zdisp` are constants. Since the displacements are three dimensional, there are three elements of `idisp[]` for each pair of neighboring atoms `i` and `j`.

Table 5.1: Space requirements of OPMOLSTAT after optimizations.

Array	Size
neighbrs[ ]	$\frac{200ni}{p}$
idisp[ ]	$\frac{600ni}{p}$
workspace[ ]	$\frac{15nf}{p}$
R[ ]	$\frac{nf}{p}$
fi[ ]	$\frac{3nf}{p}$
G[ ]	$\frac{3nf}{p}$
co[ ]	$\frac{3nf}{p} + \delta$
co1[ ]	$\frac{3nf}{p} + \delta$
lco[ ]	$\frac{3ni}{p} + \delta$
lco1[ ]	$\frac{3ni}{p} + \delta$

To reduce the cost of storing `idisp[]` we note that typically the integers stored in this array are small, ranging from -10 to 10. Thus it is possible to shrink the `idisp[]` array by a factor of 3 by packing three displacements into a single integer as follows.

$$\textit{Packed Displacement} = \sum_{k=1}^{3} (\texttt{idisp[i, j, k]} + 10) \times 10^{2(i-1)}$$

Extraction of the displacements from a *packed displacement* is a trivial exercise.

5.2 Division into CBlocks

How we cut the crystal into CBlocks can significantly affect the computation. For example if we choose CBlocks that are too small, then communication may be slowed down. This happens because, in message passing computers, each message has an overhead attached to it and each CBlock communicated from one processor to another has an overhead associated with it. A strategy using small CBlocks must have more CBlocks, and thus incur greater message passing overhead. On the other hand, having excessively large CBlocks is not a very good idea either. If a CBlock is very large, then it may mean that we are communicating the attributes of more atoms than are needed by other processors, since large CBlocks have more atoms. Thus we need to have a balance between either extreme.

We choose a strategy of slicing the crystal into CBlocks perpendicular to the X-axis. This is shown in Figure 5.1 where a case with 4 CBlocks per processor is illustrated. The shaded regions are the CBlocks which a particular processor must send to its neighbors. We expect this strategy to yield good results since the overhead is at the minimum possible — every processor sends exactly one CBlock to each neighboring processor — if there are no displacements in a direction perpendicular to the interprocessor "cuts". If there are such displacements, extra communication is unavoidable. Essentially the idea is to cut the crystal into CBlocks in a direction where there are no displacements. If displacements occur in all three directions we are forced to communicate more than a case where there are no displacements no matter what direction we make the cuts in. Note also that usually only two directions are available for us to make cuts in. This is because the crystal is usually only a few layers of atoms thick in one direction (the Z axis in the

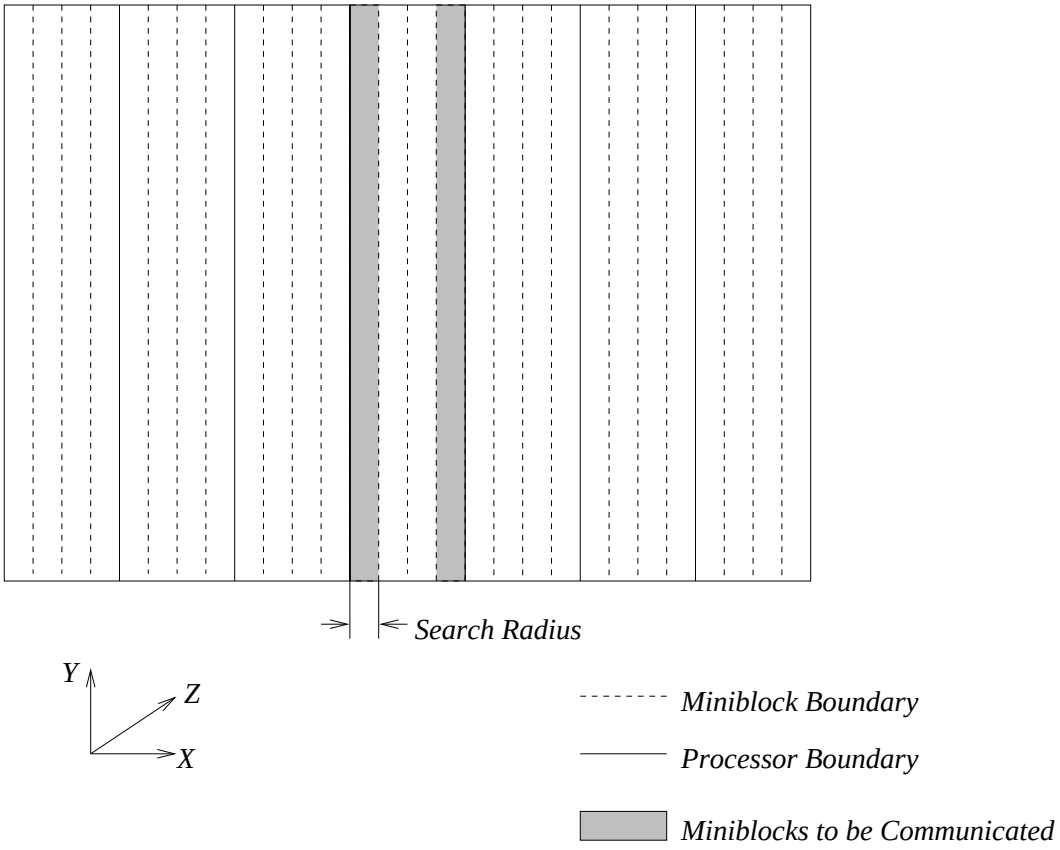

Figure 5.1: CBlock slicing strategy.

illustration) in real experiments. Also, since the CBlocks are of thickness equal to the search radius along the X axis, we do not incur an excessively high cost in communication by sending more atom attributes than needed.

5.3 Measuring Performance

5.3.1 Computational Load Balance

By measuring computational load balance we intend to examine the effect of some processors having more work to do than the rest. When this happens, these processors take longer to finish than the others, and they become a

Table 5.2: Time spent in `Enforce` compared to total time in CONJUG in OPMOLSTAT.

Free Atoms	Processors	$t_{Enforce}$	t_{Conjug}	Total Time	%Time
320,000	40	555	585	95	781
	60	333	342	97	495
	90	230	248	93	383
500,000	60	567	605	94	821
	80	396	426	93	619

Table 5.3: Time per iteration for a problem with 320,000 atoms on 40 processors.

Iteration Number	Time in seconds
1	135.5
2	85.1
3	83.0
4	83.1
5	83.5
6	83.4

performance bottleneck.

It is important for us to examine first the part of the code where the maximum time is spent — the computational kernel. This turns out to be the `Enforce` subroutine in OPMOLSTAT. The amount of time spent in `Enforce` as a percentage of total time in `Conjug` is given in Table 5.2 for 5 iterations. When compared to the total time spent in the program, $t_{Enforce}$ may not look as overwhelming, but as the number of iterations increases (typically, problems run for 100-200 iterations) the time spent in `Enforce` and `Conjug` increases linearly. The time spent outside of `Conjug` (for table creation and other initialization) is constant with respect to the number of iterations.

As can be seen from Table 5.3 the amount of time per iteration is more or less the same save the first start up iteration which is slower on the Paragon due to hardware dependent reasons. From the table it is easy to extrapo-

late that over the length of a typical 100-iteration problem the time spent in `Enforce` would be about $85 \times 100 = 8500$ seconds whereas the time spent outside of `Conjug` remains constant no matter how many iterations the problem takes to converge to a solution. This time is about $781 - 555 = 231$ seconds according to Table 5.2. So only about $\frac{231}{8500+231} = 2.65\%$ of the total time is spent in portions of the code apart from `Enforce`.

From this it should be apparent that from the point of view of time, it is sufficient to focus our attention on `Enforce`.

To measure the degree of load imbalance in OPMOLSTAT, we must first determine how much time a perfectly load balanced execution of OPMOLSTAT would take. This time corresponds to the case where the computational work is evenly divided among the processors. In both the balanced and unbalanced case, the total work done by all the processors is roughly equal[1]. Thus, if we measure the amount of time for one call to `Enforce` on each processor *individually*, neglecting time spent in communication, and add all these times together, we get a fairly reliable estimate of total work. Dividing this total time by the number of processors should give us a good estimate of how long a perfectly load balanced case should take to run. Let us call this time $T_{balanced}$. The unbalanced case on the other hand, takes up as much time to run as the processor which finishes last. Let this time be $T_{unbalanced}$. The difference between these two times, $T_{delay} = T_{unbalanced} - T_{balanced}$, gives us a good estimate of the amount of time lost due to load imbalance. T_{delay} as a percentage of $T_{balanced}$ is an indication of the degree of load imbalance. Some typical measurements for these quantities are tabulated in Table 5.4. All times are given in seconds. The times are averaged over 5 calls to `Enforce`. It is apparent from this table that the load is evenly distributed among processors.

Justifying the Total Work Metric

We have used the sum of all the times across all processors, for computation, as a metric of "total work". Further, while computing the hypothetical time taken by a perfectly balanced load, we have assumed that the total work (as we measure it), can be redistributed straightforwardly. This is not entirely true as the total work depends upon how the load is distributed among the

[1]Not *exactly* equal. Due to selective employment of symmetry as described in Section 3.2.5, the total amount of computation is affected by the distribution of atoms, i.e., the distribution of work. We examine this situation later in this section.

Table 5.4: Effect of load imbalance.

$FreeAtoms$	Processors	$T_{balanced}$	Delay %
180,000	30	60	0.1
	40	48	3.0
	50	39	3.3
	60	33	4.1
320,000	30	120	1.0
	40	93	2.2
	50	68	2.6
	60	56	1.9
500,000	60	95	0.7
	70	74	2.3
720,000	90	97	2.2

processors. Let us look at an illustration to clarify this concept. Suppose a set of processors $\{p_1, p_2, p_3, \ldots, p_n\}$ are assigned workloads $\{l_1, l_2, l_3, \ldots, l_n\}$ respectively. Our assumption is that a hypothetical perfect distribution of load would assign a load of $\sum_i l_i/n$ to each processor, so that every processor has the same load while the total load remains the same. However, in the case of OPMOLSTAT, distributing loads in a different way to processors also affects the total load by a small amount l_δ.

The change in the workload is because in OPMOLSTAT, symmetry is not exploited for atoms on different processors. If atom n_1 is on processor p_1 and atom n_2 is on processor p_2, and the atoms are close enough to be neighbors, then since we do not use the symmetry optimization for pairs of atoms on different processors,

- p_1 computes the force f_{21} exerted by n_2 on n_1, and adds f_{21} into the net force on n_1.
- p_2 computes the force f_{12} exerted by n_1 on n_2, and adds f_{12} into the net force on n_2 (note that $f_{12} = -f_{21}$).

On the other hand, if n_1 and n_2 are both on processor p_1,

- p_1 computes the force f_{21} exerted by n_2 on n_1.

Table 5.5: Additional work due to non-utilization of symmetry.

Free Atoms	Processors	Symmetry Nonuse%
180,000	30	4.1
	50	7.2
	70	10.2
320,000	50	5.3
	70	7.2

- f_{21} is added to the net force on n_1.
- $-f_{21}$ is added to the net force on n_2.

Clearly the distribution of atoms influences the total work to some extent. However the difference in total work l_δ, introduced by a different distribution, is small. To show this we must measure the extra work added by the loss of symmetry for atoms on different processors. If this is a small fraction of the total work, we will have shown that our work metric is valid. In OPMOLSTAT work grows in proportion to the number of pairs of neighbours in the crystal. Each pair can interact, and requires (with some restrictions as to free, buffer and fixed) extra computation during relaxation. Thus if we show that the number of extra pairs of neighbours added due to nonuse of symmetry is small, then that would be strong evidence that not much extra work is done by not utilizing symmetry across processor boundaries. Table 5.5 lists the number of extra pairs of neighbours due to not using symmetry, as a percentage of total pairs of neighbours. The neighbour pairs are counted in the inner loop of `Enforce` where the computations of energy occur. The results in the table reflect the variable part of the total work, the part which depends upon the distribution of atoms. We see that the total increase in work due to extra pairs of neighbours being added is not more than 10%.

As we would expect, the proportion of extra evaluations grows as the number of processors is increased.

5.3.2 Scalability

In order to measure scalability we look at two things. First we need to consider memory constraints to see if the program can tackle larger problem sizes as more processors are made available. This is size scalability. Secondly, we need to measure how well the computation speeds up as the number of processors is increased. Ideally, for a given problem, the speedup should increase linearly with the number of processors. However, *classical fixed size speedup*, which is what we mean when we measure speedup in this way, is not the best measure of scalability. It fails to take into account the behavior of the program as the problem size grows. Instead we shall attempt to measure the amount of time taken by OPMOLSTAT when the problem size grows with the number of processors.

Size Scalability

Recall that OPMOLSTAT assigns two kinds of atoms to each processor — $owned_p$ and $needed_p$. The $owned_p$ sets are disjoint, whereas copies of the $needed_p$ atoms may be present on more than one processor. Also, since we assign the blocks of $owned_p$ to each processor equally, and given that each block has almost the same number of atoms, we can easily see that $owned_p$ scales well in terms of size. In other words, if the total number of atoms grows proportionally to the number of processors, the sizes of the $owned_p$ sets on each processor should stay roughly constant. From this we conclude that what we need to examine for size scalability is the $needed_p$ set. As we can see from Figure 5.1, the atoms for $needed_p$ on any processor come from the shaded CBlocks on the neighboring processor. As the problem size increases the dimensions of the crystal increase in the X and Y direction (typically the crystal is kept quite thin in the Z direction). But the CBlock dimension increases only in the Y direction. Thus the set $needed_p$ should decrease in proportion to $owned_p$ as the problem size increases. Table 5.6 summarizes data for $needed_p$ compared to $owned_p$ for a number of typical cases. Here, we evaluate over all processors p the maximum value of the ratio

$$\frac{needed_p}{owned_p} \tag{5.1}$$

This is an upper bound on the fraction of atoms that are needed by any processor. Table 5.6 also shows values for the maximum numbers of owned,

needed and total (number of atoms stored on a processor) across all processors for a number of cases.

It is clear from Table 5.6 that for typical problem sizes $needed_p$ is smaller than $owned_p$ and the ratio of Equation 5.1 decreases as the problem size is increased when the number of processors is constant. The number of atoms in the set $needed_p$ is the parameter δ in Table 5.1; δ is also a measure of communication overhead since it is the $needed_p$ CBlocks that are communicated.

Time Scalability

The program can be considered to scale well in time if given the same number of atoms per processor (across different problem sizes), each iteration takes more or less the same amount of time. Here we consider the total number of atoms, including free, buffer and fixed atoms in both $owned_p$ and $needed_p$. The results are tabulated in Table 5.7.

From this table we see that the average amount of time to relax 10,500-11,000 atoms per processor is about 112 seconds. We consider the free and buffer atoms owned by a processor as a measure of the size of the problem. This is because the work is contributed due to these atoms. The column headed "D" shows the difference between the time per call to `Enforce` and the mean. The final column lists D as a percentage of the time to call `Enforce`. The figures indicate good scalability in time.

5.3.3 Communication

In Table 5.8 we compare the time spent exchanging messages with the total time spent executing `Enforce` (averaged across times for each individual processor). Since `Enforce` is by far the dominant computational kernel, we need not look anyplace else for communication costs. For instance, even if communication in `Conjug` has substantial cost, relatively little time is spent in the part of `Conjug` outside `Enforce`; there is little overall benefit by identifying bottlenecks in `Conjug`. Timings are in seconds and are averaged across all processors for 5 iterations.

It is clear by looking at Table 5.8 that communication incurs only miniscule costs, i.e. less than1%.

Table 5.6: Scalability in size.

p is the number of processors. Note that $max(owned_p)+$ $max(needed_p)$ is not necessarily $max(total)$. Neither is $max(\frac{needed_p}{owned_p})$ the same as $\frac{max(needed_p)}{max(owned_p)}$.

Free Atoms	p	$max(\frac{needed_p}{owned_p})$	$max(owned_p)$	$max(needed_p)$	$max(total)$
320,000	30	0.31	11564	4946	16107
	40	0.41	8673	5782	14455
	50	0.43	7021	4956	9499
	60	0.51	5782	5782	8673
	70	0.52	4956	4956	9912
	80	0.67	4543	8673	8673
	90	0.68	4130	7847	7847
405,000	40	0.34	11112	5556	16205
	50	0.42	8797	6019	14353
	60	0.51	7408	7408	14353
	70	0.52	6482	6482	9260
	80	0.52	5556	5556	8334
	90	0.68	5093	9723	9723
500,000	50	0.41	10773	7182	17955
	60	0.41	9234	6156	14877
	70	0.53	7695	8208	15390
	80	0.54	6669	7182	9747
	90	0.52	6156	6156	9234
720,000	70	0.41	11034	7356	18390
	80	0.52	9808	9808	19003
	90	0.52	8582	8582	17164

Table 5.7: Scalability in time.

Total Atoms	Processors	Owned free − buffer atoms per Processor	Time per `Enforce` call	D	% D
320,000	30	11,100	118	6	5
405,000	40	10,500	98	−14	−14
500,000	50	10,400	124	12	10
720,000	70	10,400	106	−6	−6

Table 5.8: Communication costs.

Free Atoms	Processors	Communication Time	Total Time in `Enforce`
320,000	30	0.04	118
	60	0.05	55
500,000	60	0.1	94
	70	0.1	73

Table 5.9: Optimization overhead costs.

Free Atoms	Time to run OPMOLSTAT	Time to run MOLSTAT	% Time Lost
800	103	85	21
5,000	510	430	19

5.3.4 Overhead

Finally, we must consider the cost of the new data structures and algorithms in OPMOLSTAT. The question to be answered is — how much additional overhead do they add? One way to determine this is to run OPMOLSTAT on 1 processor and compare it with MOLSTAT's performance. The main additional cost introduced by OPMOLSTAT is that free, buffer and fixed atoms are not contiguously placed in arrays. For instance MOLSTAT can iterate through all free atoms by running through a single loop with loop index taking values 1, 2, 3,...,*nfree*, where *nfree* is the total number of free atoms. However OPMOLSTAT needs to make "jumps" between CBlocks. After iterating through all the free atoms on the nth CBlock, the loop index skips over to the first free atom of the $n+1$th block on any particular processor. Secondly, we incur an additional cost due to the packing strategy used for `idisp[]` (see Section 5.1) in the inner loop of `Enforce`. Since these schemes are the same even with 1 processor, any slowdown in the OPMOLSTAT running on one processor is due to the cost imposed by the new data structures.

Table 5.9 summarizes the timings for a few typical cases. The timings are for a run lasting 5 iterations in either case.

5.4 Conclusion

The data indicate that there is 20% or less slowdown due to the additional data structures introduced in OPMOLSTAT over MOLSTAT. OPMOLSTAT also scales well in time and space. Imbalance due to computational load and space requirements is very low, infact quite near optimal. The cost of communication is negligible. The algorithmic and data structure improvements introduced score well in all areas that have to do with parallel performance.

Chapter 6

Checkpointing

6.1 Motivation

Machine failure is a catastrophic event for long running scientific computations. The longer the computation runs, the more there is to be lost by a crash. As an extreme case, if the average time between successive crashes is much less than the time needed for the computation to complete, then it is unlikely that the computation ever will finish. Unfortunately for some computations, this is quite true. High performance systems tend to crash quite frequently, due to the fact that high performance computing hardware and software is oftimes new and not widely used, and thus not quite as robust as more commonly used lower end hardware and software. Parallel computations are often long running as well, since scientists tend to try solving the largest problems possible.

Another characteristic of supercomputer installations is that they tend to be shared by large numbers of users. The usual scheduling policy employed to share the machine is to reserve time in advance. The conventional multi-user operating system model of running each user process for a small time slice does not work well here because users may like to have the exclusive use of many processors for performance reasons. The additional overhead of process scheduling wastes too much time while running already heavily compute intensive applications. Additionally, most parallel programs are written so that they can run on any number of processors, but once they start execution there is usually no easy way to switch to fewer or more processors.

One disadvantage with the usual method of scheduling is that users need

to have a very good idea of how long their computations will take, well in advance, so as to book a specific time slot on a machine. If they overshoot their time limit, then they would have to kill their computations, so that other users can run theirs, and later start again from scratch.

Also, if the duration of the maximum obtainable time slot is less than the program execution time, the program will never be able to execute to completion. This is unfortunate considering that it may be quite possible for the user to obtain a number of discontinuous, smaller time slots, which would sum up to the total time desired.

Another disadvantage of typical HPC environments is that if more processors become available as time goes on, the user cannot use them. He is stuck with using the same number of processors he started off with.

Finally, if the user has access to multiple machines, he may be able to book time slots on more than one. The usual one-machine-multi-user model cannot exploit this extra computer time, available in a distributed way.

6.2 A Different Model

The underlying idea of the system we propose, is that the number of processors, and the time available to run on them, are the fundamental resources needed by a parallel program. In the conventional model of sharing the machine, resources are obtained as a tuple, p processors for t hours on machine m, or, (m, p, t). However, in a distributed network of parallel machines, it is possible to obtain computer time as a set of tuples, for example, $\{(m, p, t_1),\ (n, q, t_2)\}$. This means that p processors are available on machine m for time t_1 and q processors on machine n for time t_2. A model that assumes that computer resources are structured in this way closely fits the actual situation in many cases and thus promotes better use of computational resources and gives users better service.

We propose to resolve these issues by a three layered approach: a system of checkpointing, a system for managing scheduling, and a system which manages distribution. The bottom-most layer (for the layers above this depend on the functionality it provides), checkpointing, is described in this chapter.

We will describe a method of taking checkpoints *portably* (Term due to Joao De Silva, [19]). As described earlier (Chapter 2), a portable checkpoint is one that is independent of the number of processors or the machine. Our

checkpointing method is also *user directed* [7, 12, 18]. We will also see how portable, user directed checkpointing supports migration.

The checkpointing strategy outlined here has been successfully incorporated in the the molecular statics code. However, describing the precise nature of our implementation is a difficult task, since it requires detailed knowledge of how the code operates. In an attempt to keep the description of the principles of checkpointing (and indeed PACE and COMET as well) clear and simple, we have abstained from including any details of making the statics code checkpointable, into our present discussion.

6.3 A Model of Scientific Computation

To simplify the design of the checkpointing process, we define a simple model for how scientific programs execute. This model is by no means complete, in that it does not describe every scientific program. There are however enough computations which fall in the purview of such a model to make it worthwhile. The following fragment of code illustrates the structure of a common (generic) scientific computation.

```
Program Scientific
  Initialization()
  while(not solved)
     RefineSolution()
  enddo
  Termination()
End
```

The program starts off with an initialization phase, which may be reading input, distributing data among processors, or computating data that is used in the later phases. Then there is a phase where the computation iteratively refines the data, or explores a search space, or some similar operation. For checkpointing to be effective, this must be the dominant phase, in terms of time spent. Finally, when a solution has been found, or the computation abandoned, the termination phase is entered. This may perform actions such as outputting results.

6.4 Types of Variables

6.4.1 Definitions

Portable checkpointing entails the saving of program variables as its state. However, not all variables need to be saved, and not all should be saved the same way. Portability also implies that the variables should be amenable to being redistributed among the processors, so that the checkpoint can be started on a number of processors that is different from the number on which it was taken. Thus "portable" here is taken to mean that not only is the checkpoint portable across different machines, it is also portable across executions employing different numbers of processors.

Redistributable variables are defined as arrays that are distributed across the different processors, but if concatenated, the resulting array is the same for every execution of the program (irrespective of the number of processors.) Redistributable variables are saved as a concatenation of all corresponding variables on the processors, and on restarting on a different number of processors, these are distributed across the processors differently. Redistributable variables are present in programs written for distributed memory machines only. In the case of shared memory machines, they appear as *uniform* variables described below.

Uniform variables are those which are the same on every processor, and in the corresponding steps of different executions (with different numbers of processors) they are identical.

Meta-data variables are those which represent information about how program data is distributed among the processors. They do not need to be saved and are initialized when the data is assigned to different processors.

Non-checkpointable variables are any variables that do not fall into the above categories. Barring rare circumstances, the presence of these variables prevents checkpointing.

6.4.2 Example Scientific Computation

Here we take a look at an example fragment of code illustrating the various types of variables. Consider the program below which increments each element of an array of size `M` by 1, for `N` iterations (This trivial example is used only for illustration). The parallel version of this program would divide the array evenly among the processors available, and each processor would apply

the increments to the portion of the array assigned to it.

```
Program Foo
   dimension A(M)

C Initialization
   numprocs = GetNumberOfProcessors()
   myid = GetRankOfThisProcessor()
   ReadInput(A)
   mychunksize = M/numprocs
   istart = myid * mychunksize
   iend = istart + mychunksize

C Iteration
   do i = 1, N
      do j = istart, iend
         A(j) = A(j) + 1
      enddo
   enddo

C Termination
   do j = istart, iend
      write(outputfile) A(j)
   enddo
End
```

In the above example, the variables can be categorized as follows.

- *Redistributable*— The array `A`. This is broken up between the processors differently depending on `numprocs`, but `A(i)` is always the same at any given iteration, no matter how many processors the program is run on.
- *Uniform*— `M`. The size of the problem remains the same no matter how many processors are used.
- *Meta-data*—`numprocs`, `myid`, `istart`, `iend` and `mychunksize`. These are different depending on the number of processors. They represent information about the distribution of data, and do not convey anything about the state of the computation.
- *Non-Checkpointable*— There are no such variables in this example.

6.5 The Checkpointing interface

In this section we describe the general structure of a checkpointing program. If we assume that the program to be made checkpointable follows the model outlined in Section 6.3, a version of the program that incorporates checkpointing should look as follows.

```
Program Checkpoint
   Initialization()
   if(Restarting())
      ReloadCheckpoint(PARAMS)
   while(not solved)
      RefineSolution()
      if(Checkpointing(exit))
         SaveCheckpoint(PARAMS, exit)
   Termination()
End
```

Brief definitions of the function calls are given below.

- `Restarting()`

  This function returns a boolean value which is false if the current run is a start from the beginning, and true if the intent is to load a prior checkpoint.

- `ReloadCheckpoint()`

  The parameters to this function, `PARAMS`, are the redistributable variables. The function reads the variables from the checkpoint files, thus restoring the state of the computation[1] to what it was at the time the checkpoint was taken.

- `Checkpointing()`

  This function returns a true boolean value if a checkpoint needs to be taken during the current iteration, and false otherwise. The return value could be dependent on an interaction with the user, i.e., the

[1] The use of the term *computation* as opposed to program is significant here. Whenever we use this term, we are talking of the state of the problem independent of the number of processors working on it.

program could ask the user what needs to be done, or, there may be some logic which checks the iteration number and decides whether a checkpoint must be taken. The boolean parameter `exit` is output by this function too, and is true if the program must exit after taking the checkpoint and false otherwise.

- `SaveCheckpoint()`

  This function writes to disk the portable checkpoint that must be read by a subsequent invocation of `ReloadCheckpoint()`. The `PARAMS`, as before, are the redistributable variables. The parameter `exit` is used by this function to decide whether the program must terminate after taking this checkpoint.

We will look at the operation of this code when it is about to take a checkpoint and also when it is about to restart by loading a past checkpoint.

- *Taking a Checkpoint*

  During a normal run of the program, when no checkpoint is being taken, the function `Restarting()` and `Checkpointing()` return `false` so the program proceeds correctly. After some iterations, if a checkpoint is to be taken, the `Checkpointing()` function returns true. This causes the checkpoint to be written to the disk.

- *Restarting*

  Now, if we intend to restart the program, on a different number of processors, this is done by setting an input parameter[2] to the program that causes `Restarting()` to return `true`. The initialization step proceeds as is usual for this number of processors. Before the iterations start, the state of the computation is loaded by the `ReloadCheckpoint()` function. Thus the computation proceeds as though it had always been running on the same number of processors as it is now.

It bears repeating that for checkpointing as we define it here to be practical, most of the program's time must be spent in the iterative phase. This is why we place the checkpointing calls in the iterative loop. It is apparent

[2]In our implementation of checkpointing, the `Restarting()` function checks for the presence of a file named `restart`. The presence or absence of this file conveys the neccessary information about whether the program is restarting.

that it is much harder to place checkpointing calls in either the initialization or termination phases. Because there is no iteration in those phases, we would have to place many checkpointing calls at many different locations in the program, and it is a harder task to figure out what the contents of the checkpoints should be.

6.6 Checkpointing Example

To demonstrate exactly how the functions `Checkpoint()` and `ReloadCheckpoint()` must be written, we will use the example in (Section number here) and turn it into a checkpointing program. The two functions are written as follows for this program.

The `Checkpoint()` function when called, needs to write the state of the computation, which is represented by the contents of `A` across the processors. `A` must be written in a redistributable way, that is, it should be possible to use the data written to reload the state on a different number of processors. This means that we concatenate the segments of `A` from each processor to get the state.

One further observation — since we write the state of the computation to the checkpoint, the array `A` must be written as the concatenation of its distributed segments. Thus the processors must serialize writing to the checkpoint file, in the same order in which they have segments of `A`. In this case that is ordering by rank; processor 1 writes to the checkpoint file first, followed by processor 2 appending, and so on. In our example we have used messages to coordinate this.

```
Function Checkpoint(exit, myid, istart, iend, A)
   if(myid.NE.0)
      RecvMessageFrom(myid - 1)
   OpenForAppending(checkpointfile)
   do i = istart, iend
      write(checkpointfile)A(i)
   enddo
   if(myid.NE.numprocs - 1)
      SendMessageTo(myid + 1)
   if(exit)
      Stop
   return success
End
```

Reloading the checkpoint is the opposite process. The checkpoint data needs to be read into a different decomposition. In our implementation of `ReloadCheckpoint()`, the parameters `istart` and `iend` represent information about the new decomposition.

Since reading the correct segment of the checkpoint files can be done independently of other processors, no synchronization is needed in `ReloadCheckpoint()`.

```
Function ReloadCheckpoint(istart, iend, A)
   Open(checkpointfile)
   Seek(checkpointfile, istart)
   do i = istart, iend
      read(checkpointfile)A(i)
   enddo
   return success
End
```

Note that the values of `istart` and `iend` in both function calls are different if the program is started on a different number of processors. This is because the initialization step distributes the array `A` depending on the number of processors.

Nested Function Calls

So far we have implicitly assumed that in the program to be checkpointed, the place to position the calls to the checkpointing functions is the main

program function. But this is usually not the case, and checkpoints may be best positioned in a function that is not at the bottom level of the call stack. The way to deal with this situation is to save all the redistributable variables in all the function calls below the one that takes the checkpoint. These variables must be passed to the top level function that takes the checkpoint, or made global, so that the checkpointing function can access them.

6.7 Migration

Once we can portably checkpoint a program, migration of context is just a matter of transferring checkpoint files from one machine to another. There are several reasons why migration is well supported by user directed, portable checkpointing.

Since the checkpoints are not just a meaningless byte stream, as the user actually writes the code to store and reload checkpoints, it is possible to write checkpoints in a data format independent of machine architecture. In other words, we can exploit the fact that type information is known about the checkpoints to introduce a translation mechanism into the migration system.

Portability has another, not so obvious benefit. Parallel machines vary across a wide range in terms of the number of processors in a machine, and the computational power of each processor. Suppose for example, that a program needs to be migrated from a 1000 processor machine to a 10 processor one, where each processor of the latter is many times as powerful as one of the former. Consider specifically that each processor on the second machine has 100 times as much memory available. Clearly, a program that needs 300 processors on the first machine, needs only 3 processors on the second. Migration between both machines would be severely limited, or tremendously inefficient, if checkpoints were not portable.

Parallel machines can also vary in another important way — some can have shared or distributed shared memory (which do not need explicit message passing) and others may be message passing, distributed memory. Programs for both kinds of architectures are written differently. In many cases, there is a version of the same program for many different kinds of machines. By separating the checkpointing concept from the program, and defining it as the state of the computation, we make checkpoints portable between different kinds of machines and different implementations (programs).

To sum up, portable checkpoints have many characteristics that are of use

in the kind of distributed heterogeneous environment that migration systems are built upon.

6.8 Disadvantages

Admittedly checkpointing as it is outlined above is not free of pitfalls. For one, it is an intrusive method and requires modifying the source code of the program to use it. This obviously makes it unavailable for situations where sources are not available. Fortunately in the case of most scientific codes this is not so. Secondly, our checkpointing approach demands knowledge of the inner workings of the program, to discover how to save variables so they represent a checkpoint that captures the state of the computation, independently of the number of processors. Finally checkpointing is not possible even theoretically for certain computations. Consider the following involving computing the dot product of two vectors in parallel.

```
Program DotProduct
   myid = GetMyId()
   numprocs = GetNumProcs()
   step = vectorlength/numprocs
   do i = myid * step, (myid + 1)*step - 1
       sum = sum + A[i] * B[i]
   enddo
   ReduceAdd(sum)
   write(outputfile)sum
End
```

Clearly, in the iterative part of the computation, the state of the program cannot be represented as anything but as a partial sum for each task, which is dependent on the number of tasks. There is no straightforward way to checkpoint and redistribute the work on restarting for this program. The argument we advance is that even if not all computations can be portably checkpointed, enough of them can be, making this a method of some value.

Chapter 7

Parallel Application Control Environment

7.1 Introduction

Given that we can write portably checkpointing programs, it is useful to have software which uses this ability to share a parallel machine between users. Users must be able to request this program for time slots, which it assigns based on some internal policy, and then enforces those time slots by queuing programs submitted by the user and killing programs which overshoot their time limit. A program getting killed is not a disaster in the scenario where checkpoints can be taken, because the computation can be restarted from the last saved state. The scheduling software can send a message to a program about to be killed thus giving it a window of time during which it can write its state to a checkpoint.

Such a controlling program, which we shall call the *Parallel Application Control Environment Daemon* (abbreviated hence as the PACE daemon), is described in this chapter.

We will not go into the implementation details of the PACE daemon yet; the current chapter describes the external interface and the functionality of the daemon, and how that gives us an effective way to harness the power of checkpointing.

The PACE daemon implements a *policy* to effectively use the *mechanism* of checkpointing. To some extent, this policy is programmable by the system administrator. These two terms, policy and mechanism, are concepts of

considerable importance in designing a system cleanly, and we shall see these recur throughout this thesis in various contexts.

7.2 An Example Machine – The Intel Paragon

Here we describe the model of processor sharing of a sample parallel machine operating system — OSF/1 for the Intel Paragon. The Paragon bases application management on the concept of *partitions* of groups of processors, and the idea of rolling computations in and out of memory.

7.2.1 Partitions and Processor Sharing

A partition is defined as a "named group of nodes" [8]. Users are allowed to group sets of nodes and designate them as a partition. Applications running on the nodes in a partition can be scheduled in three ways.

1. *Standard Scheduling*

   In this mechanism the usual OSF/1 scheduling mechanisms are used. Multiple applications may use a single node, and the operating system uses a 100 millisecond scheduling quantum for each. Additionally, applications may be de-scheduled when they make system calls.

2. *Space Sharing*

   Nodes in this type of partition run only one application at a time. Once an application has acquired a particular node, it has exclusive use of the node till it finishes running.

3. *Gang Scheduling*

   In a gang scheduled partition, applications are run for a period of time called a *rollin quantum*. The rollin quantum is typically a very long period of time (may be as long as a day), and is set for the partition by the system administrator. The purpose of a long rollin quantum is to minimize the amount of time lost in context switching.

7.2.2 Shortcomings

Standard scheduling is rarely used for parallel applications because it causes poor performance in compute or I/O intensive applications. The frequent traps to the kernel while scheduling and de-scheduling processes are a significant drain on the CPU time available to the application. Space sharing is the most commonly used scheme. It however causes a few problems which are outlined below.

1. Since a user has exclusive use of a set of nodes, another user who needs some of those nodes will have to wait for the first user to finish running his application. If that application takes days to run, then, clearly it is unfair on the second user if his application takes only minutes to run.

2. The usual solution to the fairness problem — restricting users to run on large numbers of nodes for only short periods of time — is a poor one. Following this policy means that long running applications that use many processors cannot run at all. Many installations require the users to sign up for computer time in advance if they have applications like this, but this doesn't really solve the problem. All it does is to allow other users to schedule their computations some other time.

3. In addition, there is no way in this system to queue jobs in advance so that they are run during the signed up time slot. Many installations do have queuing environments though, and queueing environments that can be installed on top the the OS exist, so this isn't a major drawback. Still, even a queueing environment has some tough restrictions on when it can run some applications if they are not portable in terms of how many processors they use.

Gang scheduling partially addresses this problem. It deals successfully with the fairness problem. The queueing problem is also partially dealt with – if an application is submitted that needs processors currently in use by a higher priority application, the low priority application waits. The solution is partial because it does not admit of the possibility of application priorities changing dynamically, which is essentially what happens when users sign up for time slots. When a sign up policy is in operation, a request made by a user who is signed up for a particular time period has the highest priority for the duration of that period. When the time period expires, the priority

of the request drops off to some lower value. There is no way to have this kind of behavior in gang scheduled applications. Moreover, gang scheduling does not exploit the possibility of using the saved contexts for fault tolerance purposes.

Since the contexts are core images, the programmer doesn't have to do any work to make his program 'gang schedulable', but there is also less benefit, as gang scheduling does not allow restarting the program on fewer or more processors than what it was started on. Also, there is no support for migration between heterogeneous machines. The contexts are core images, so it is virtually impossible to translate a context on one machine to an equivalent context on another.

7.3 Operation of PACE

The basic algorithm for the PACE daemon is quite simple. The daemon periodically repeats the following actions.

1. Wait for a user request to come in. A user request consists of the path for a program to be executed and the number of processors on which it is to be run. Upon getting a request, the daemon puts it in a queue of requests waiting to be serviced. This is a priority queue and the request made by the user with the highest *request priority* is first. Request priority is a function of a *user priority* set by the system administrator and the sign-up schedule. Thus if all users have equal user priority, the request priority of the user who is signed up for processors at the current time is highest. From this point on, whenever we use the word priority, it will be taken to mean request priority. Where we are referring to user priority, we will explicitly use that term.

2. Check to see if enough processors are available to service the highest priority request. If not, the daemon must see if it is possible to kill some running programs of lower priority, to free up enough processors. In this case the daemon sends a warning message to those processes before terminating them.

3. Check to see if any of the user priorities or time slots alloted to the users have changed. Update the priority queue to reflect this. Use the new priorities to do scheduling. Priorities change because the time

slots of users may expire or users may sign up for new time slots. Another reason is that the system administrator may update user priority information while the PACE daemon is running.

7.3.1 Attached and Unattached Users

An important design issue for the PACE daemon is the way it deals with users who run programs without going through the process of submitting requests to the daemon. Since the daemon is not a part of the operating system, but only a service, this is quite a likely scenario; in fact, this is even a desirable situation at times. To see this, consider the case where a user has requested a time slot, but due to some reason does not use it (maybe because he finds bugs in the program). It is a waste of computer time to restrict other users from not using this time slot.

Also, it is work to sign up for a schedule and submit requests to the PACE daemon, and all users may not consider the benefit sufficient to be worth the effort. This is true, for example, when users are using the computer for debugging as opposed to production runs. Therefore it is essential to let users use the computer in the normal way.

These users who have no special scheduling needs are called *unattached* users, as opposed to *attached* users who request executions through the PACE daemon. Unattached users are dealt with by assigning their requests the lowest request priority of all. In other words, their applications are permitted to run till such a time as a higher priority request comes in needing those processors, at which time they are killed.

Most users would not take too kindly to their debugging runs being suddenly terminated, so in practice, the PACE daemon must be configured so that it only has a subset of all processors under control. This way there are always some processors for unattached users to work on uninterruptibly. Alternatively, since many institutions have more than one parallel machine, the most powerful one which is typically used for "production runs" could be exclusively reserved for PACE like policies.

7.3.2 System Administrator Interface

The PACE daemon must admit of a certain amount of programmability. The system administrator must be able to configure its operation to control priority. This is done by having a configuration file, which contains the user

priorities of the different users, as assigned by the super user. This file is read at startup and again periodically, at intervals of time set by the super user.

Another parameter is the time slots assigned to each user. These are also kept in a separate *scheduling information file*, which is periodically read by the PACE daemon. How this file gets updated is out of the scope of this thesis, but it may be possible for users to sign up for time slots using a program which updates the scheduling file.

7.3.3 The Role of PACE

PACE is a minimal system with very few frills. Partly this is because of the preliminary state of its development. The chief reason is that PACE is mainly intended to supplement COMET and portable checkpointing. Other queuing systems may provide greater functionality and a better user interface, but that very complexity makes them poor candidates to function well in conjunction with COMET or portable checkpointing. Some of this will become clear later as we go on to describe these three systems in Chapter 9 on implementation — but as a simple example, consider how PACE must checkpoint a program. It must interact with the program through some pre-designed protocol via the checkpointing API, to inform the program that it needs to take a checkpoint. This means that PACE must be designed in conjunction with the checkpointing API. Also most queuing systems schedule jobs "automatically" in that they attempt to schedule jobs as per some load distribution algorithm. However each parallel program may have unique constraints as to where it can run. Additionally, many HPC installations schedule user jobs by allocating specific time slots to users, as already discussed. A queueing system needs to fit in well with this kind of framework to be truly effective. This is what PACE attempts to do.

Chapter 8

Computational Environment Template

8.1 Introduction

This chapter describes the set of tools that manipulate a Computational Environment Template, or COMET, so called because it defines an abstract object consisting of sites which correspond to parallel machines. A *COMET object* is defined as a group of *sites* where a particular program can be executed. A site, in turn, is defined as a network connected host computer and a directory containing files essential to running the program. The COMET object acts as a pattern or template for the set of networked parallel machines which are available for running programs.

The tools perform manipulations on the COMET object, of which the main operations are — starting up programs at sites, checkpointing programs at sites, restarting or migrating checkpointed programs between sites, and so on.

The COMET tools are designed to work with the PACE daemon, but some of them can also be used when no daemon is present. However, to get the most out of the tools, they must be used in conjunction with PACE and portable checkpointing. The principle is that the daemon provides control of resources on a single computer, while the COMET tools permit a user to initiate the same actions from a remote site, and in addition, provide support for migration of processes.

Thus, if a user has time slots to run his portably checkpointable program

on a number of different machines, as well as checkpoint-compatible versions of the program on each machine, he can use the COMET tools to run his program in those time slots.

In this chapter we will concentrate on the design choices available to us, and explain the features of the most important of the tools. In Chapter 10 we will look at the implementation of all the tools in detail.

8.2 Design Issues

Each of our three layers — portable checkpointing, PACE and COMET — are very simple systems, with the bare minimum of functionality. They depend on being used with each other to deliver the maximum benefit to the user. Consequently, the design of each is linked with the design of the others. The idea is to keep these three systems as orthogonal as possible, and to build them with features that can easily be exploited by the others. With this in mind, the current section will talk about the design of checkpointing and PACE even as we deal with issues involved in the design of COMET.

8.2.1 Command Orientation of Checkpointing

One decision that must be made when using checkpointing is whether the checkpointing mechanism is to be commanded by the user or triggered automatically. Command orientation means that the user issues a particular command, or uses a tool to order a program to take a checkpoint (This also allows the user to write programs that decide when to take checkpoints internally, without any intervention from any tool. In such a case, the user would not have any need of the COMET tool to take checkpoints). This is as opposed to system supplied triggering, where the system automatically issues a checkpoint command. Our choice was for the former, for a number of reasons.

To start with, it is simpler to put the responsibility for taking checkpoints into the hands of the user, as opposed to PACE, since this would clutter up its implementation. The alternative, of building a separate module that runs as a checkpointing daemon is without justification, for if checkpoint triggering was to be supplied as a system facility, the logical place to put it would be with the scheduling module, which is PACE, as it is PACE which is concerned with the "lifetime" of a program.

Also, there is little uniformity in the checkpointing needs of programs. Some may checkpoint far more frequently than others, if they have very small checkpoints for example, whereas such a policy would be disastrous to programs which wrote large checkpoints. It would be tiresome to have PACE interpret user commands to control the checkpointing needs of their programs, when, as we shall see, with the expenditure of minimal effort, the user can automate command oriented checkpointing.

Thirdly, any automated checkpointing system can be easily implemented if need be, as an additional layer which sits on top of PACE and COMET.

8.2.2 Command Orientation of Migration

The justifications for keeping migration command directed are similar to those for checkpointing, with one additional reason. There is no simple way for any system to keep track of which machines the users may have accounts on, and whether or not those machines have PACE or COMET installed on them. The problem essentially is that users may have accounts on distinct sets of machines, which they may want to use for their applications. An automated migration system would have to know where the users had accounts, and coordinate with those sites, or it could restrict the users to utilize only a particular set of machines. It is simpler and more flexible to have the user control migration between arbitrary sites, using tools purely at the user level.

8.3 COMET

8.3.1 Basic Concept

The central concept is that a COMET is an object consisting of a collection of sites with *workspaces*. A site is a parallel machine which the user plans to use for computation, and which can be accessed through a network, by the user, and other sites in the COMET object. A workspace is a directory on any site, containing all the files needed to run a program, including data files and executables. The workspace is also where a program writes checkpoint files, and any output files.

8.3.2 Description of Tools

Below is a short description of each of the COMET tools.

- `cometcreate`

  This is called with an input file as parameter, which contains the hostnames of the machines at which a workspace must be created. The input file must give the path on the remote host, where the program executables and the data files are located, and the names of the checkpoint files that the program outputs.

  The command then creates a COMET file, which is used by subsequent commands to refer to the COMET object.

- `activate`

  Once a COMET object has been created, the next step is to *activate* the program at any one site of the COMET; in other words, to run the program on a certain number of processors, or possibly submit a request to the PACE daemon if there is one installed on the machine. At most one site of a COMET may be active at any point of time.

- `freeze`

  This tool portably checkpoints a running program by sending it a notification which is received by the `Checkpointing()` function of the checkpointing API. It also has options which specify whether the program must exit after taking the checkpoint, or continue to run. `freeze` must be compatible with the checkpointing mechanism.

- `restart`

  This is used to reload a prior portable checkpoint and continue the computation from the state saved in the checkpoint. The checkpoint may be reloaded on any number of processors.

- `transfer`

  The `transfer` tool is used to move the context of a checkpointed program to another site. It handles complexities such as translating between differing data formats if needed[1]. The simplest way to implement

[1]Our implementation of COMET does not include the translation mechanism as we judged it to be of peripheral importance. However we mention it here to indicate how such a tool can be incorporated into the architecture of COMET in a fuller implementation.

translation is to have a local translation program at each site which translates checkpoint data to and from a common global format, and which is invoked by the transfer tool at source and destination sites. This could be speeded up if there was a database listing which pairs of sites needed no translation thus skipping translations at both ends of the transfer. A more efficient and more complicated tool may have a translation program for each pair of incompatible machines, and a database indicating which program is to be used for a given pair of machines.

- `cometls`

  This tool displays the state of the COMET, which site is active, which sites have had checkpoints taken and timing information such as how long the site has been active, or when the checkpoint was taken. Essentially this is a monitoring tool for the state of the COMET.

8.3.3 Example

To illustrate the workings of the system, we look at an example of how a user may go about using the PACE-COMET setup to simplify running a large program, of which there are versions on two separate machines. Assume that this hypothetical user has unrestricted access to one powerful machine, but the policy of the computing installation limits access to a second, highly powerful machine for overnight runs only. Naturally, the user would try to use the powerful machine as much as possible, and in the day, run the program on the first machine. This user could use the following script to run the program.

```
cometcreate inputfile
foreach day in Sunday Monday Tuesday Wednesday
  at 10:00am day activate cometobj weakersite
  at 8:00pm day
     freeze cometobj weakersite
     transfer cometobj weakersite strongersite
     activate cometobj strongersite
  at 8:00am day+1
     freeze cometobj strongersite
     transfer cometobj strongersite weakersite
```

```
    activate cometobj weakersite
endfor
```

The script essentially creates a COMET object (which is referred to as a file of name `cometobj`) by reading information from the file `inputfile`. The contents of an example `inputfile` are as follows (the numbering of lines is for purposes of explanation).

```
1  weakersite.cc.vt.edu
2  /home/pulla/workspace
3  /home/pulla/statics/code/a.out
4  /home/pulla/statics/data/inputfile.dat
5  /home/pulla/statics/data/forces.dat
6  *
7  strongersite.cc.vt.edu
8  /usr/pulla/workspace
9  /usr/pulla/statics/code/a.out
10 /usr/pulla/statics/data/inputfile.dat
11 /usr/pulla/statics/data/forces.dat
12 %
13 checkpointfile1 float
14 checkpointfile2 float
15 checkpointfile3 integer
```

Lines 1-12

Line 1 tells the `cometcreate` tool the location of the first of the sites. Lines 2-5 give information on how at the first site (the less powerful of the two machines) is to be set up. Line 3 tells `cometcreate` that the workspace is to be created in a directory called `/home/pulla/workspace`. Lines 4-5 give the names of files that must be copied into the workspace directory. Line 6 has a separator, marking the end of information for the first site. Lines 7-11 repeat the same pattern as 1-5. Finally line 12 is occupied by a different separator marking the end of all information about sites.

Lines 13-15

The last three lines give the names of the checkpoint files that are created, and the type of data that is stored in each of them. The information on data types is useful in case any translation is needed at any of the sites.

Coming back to the script, after creating the COMET, it uses the UNIX `at` command to schedule an activation of the program, by means of the `activate` command at 10:00 a.m., from Sunday through Wednesday. At 8:00 p.m. the same day, the program will be checkpointed using the `freeze` command. The `freeze` command does not return till the checkpointing is complete. This is followed by the transfer command which copies the checkpoint files to `strongersite` with any translation if suitable.

8.4 Types of Checkpointing

An immediate benefit of making checkpointing command driven, that is, as part of COMET rather than PACE, is the ability to support users who do not checkpoint in the way prescribed by COMET. If checkpointing had been part of PACE, the user would have to either forego using PACE, or be forced to use the form of portable checkpointing that we have talked about. Our system allows these different classes of users to use PACE-COMET to varying degrees. Thus users who, for example, have programs which simply dump core images, and can reload them later (albeit on the same number of processors and the same machine) are able to use some features of PACE and COMET selectively. Such users would use some other (non-COMET) mechanism to trigger checkpointing, but use the COMET tools for managing remote sites, and use the PACE daemon to schedule their computations, since the PACE daemon makes no assumptions about the checkpointing method of the users (or even the absence of it.)

Hence the PACE-COMET system as it stands can support users who subscribe to it only partially, as well.

8.5 Fault Behavior

8.5.1 Some Typical Faults

Now that we have looked at checkpointing, PACE and COMET, we are in a position to examine how these systems can provide better fault tolerance.

We will look at the reaction of the system to network or machine failures in a variety of situations. In the following, when we refer to failure of a site, it is taken to mean either that the site has crashed, or that the network has failed making it impossible to reach the site.

1. *Failure of Active Site*

   If the active site fails or becomes unreachable, this will become apparent to the user when he queries the state of the COMET using `cometls`. The user would now have two choices — wait for the site to come up, so that he could restart from the last checkpoint taken there, or, activate or restart (if a checkpoint has been taken) the program at some other functioning site. The user must constantly monitor the active site for failure if he wishes to take action as soon as failure is detected. In 8.5.2 we look at a shell script that makes it possible to automate this sort of failure recovery.

2. *Failure of Checkpointed Site*

   The case of a checkpointed site failing is not as complicated as the case of an active site failure. For one thing, it does not need immediate detection (at least for most imaginable applications). Failure is detected as soon as one tries to perform any COMET operation involving the site, and the user can then take any remedial action he chooses.

3. *Failure of Site while Checkpointing*

   The checkpoint operation would be reported as failed if any of the sites involved or the network crashed during the operation, when the user queries the state of the COMET object with `cometls`. The `cometls` tool would be unable to connect to the malfunctioning site and would print an error message indicating the fact.

4. *Failure of Site while Transferring*

   The behavior of this case, and how to handle the fault, is identical to the previous one.

8.5.2 Extensibility of Fault Tolerance

Neither PACE nor COMET have any automatic recovery capability (beyond that the checkpoints allow the user to restart from a pre-failure state). However, it is possible for the user to create applications that work with these systems, and have the ability to detect and recover from failures. A simple way to do so is to write a shell script that checks for the different types of failures and takes appropriate action. COMET provides the tools for recovering state, and rescheduling computations. The user could, with some

expenditure of effort, write the controlling shell script to account for certain kinds of failures.

Sample Shell Script

A sample pseudo shell script is given below for purposes of illustration. This script is written for a situation where the user has access to a number of parallel machines, and versions of his program on those machines. He wishes to run the program on any one machine. If that machine crashes, he must be able to detect the crash and start running the program at a site where the latest checkpoint has been stored.

```
1  currentsite = ‘fgrep active status | awk ’{print $1}’‘
2  cometls cometobj > status
3  if(fgrep $currentsite status | fgrep ’failed’)
4  then
5        cometls -t cometobj | fgrep -v ’failed’ > sitelist
6        fgrep ’checkpointed’ sitelist | sort -nk2 > sortedckpnts
7        currentsite = ‘head -1 sortedckpnts‘
8        activate $currentsite
9     fi
10 fi
```

The shell script above must be periodically invoked, possibly using the `at` command of Unix. Now we will examine this script, line by line to see what it does.

Lines 1-2

The file `status` represents the current status of the comet, which site is active, which sites may have failed, which sites have checkpoints on them and so forth. The contents of the `status` file are as output by the `cometls` function, and a sample is given below.

```
site1.cc.vt.edu active
site2.cc.vt.edu checkpointed 10:00am January 8
site3.cc.vt.edu checkpointed  2:00pm January 8
site4.cc.vt.edu failed
site5.cc.vt.edu unstarted
```

Line 1 of the shell script looks for the line in `status` which lists the active site and initializes the shell variable `currentsite` to the hostname. Then in line 2, the latest status of the comet is stored to the file `status`.

Lines 3-8

In these lines, the script checks to see if the site that was active formerly has failed. In case this is so, the output of `cometls` from line 2 would list the site as `failed`. If this is indeed so, then a new site must be chosen to execute the program, and the `if` conditional is entered.

Lines 5-8 extract the list of checkpointed sites and sort them to see which one was the most recent. The `-t` option of cometls displays the checkpoint time as a single number, such that the numbers displayed for the checkpoints fall in the same order as the times they were taken at. Finally the most recently checkpointed site is activated[2].

Generic Fault Tolerance

Writing such a script is a complicated business, and needs programming ability as well as a facility with the Unix shells. However users who need customizable fault tolerant behaviour may be willing to pay the price of the increased flexibility of this approach.

For other users, it should be possible to write systems using shell scripts, similar to the above, only more generic. Such a shell script could take as input a file similar to the input file to the `cometcreate` command and essentially hide the details of schedule and fault management from the user by itself invoking the COMET tools. We would thus be adding an automated fault tolerance layer on top of COMET, PACE and checkpointing.

8.6 Conclusion

An important thing to realize is that the entire checkpointing and COMET-PACE systems are not necessarily an end in themselves, but provide a suitable set of building blocks that can be used to build systems that have more

[2]One case we have neglected for conciseness is where no site has any checkpoint. In this situation we would have to check if no output was produced in line 6, and if that was so, choose one of the `unactivated` sites.

features such as automated recovery from faults and automated scheduling and load balancing. The central theme in building these tools has been that parallel applications have wide ranging fault tolerance and scheduling requirements, and users have access to widely different types of machines. A single system that tries to reconcile this heterogenity has a demanding task in front of it. A cleaner option is to provide basic tools that can be combined in a way suitable to the needs of the application.

One way to put this is to say that the trio of methods/systems — checkpointing, PACE and COMET — provide the mechanism for better fault tolerance but do not dictate any policy on how to go about it. A user can use them in conjunction with the Unix shell scripting languages to implement a suitable policy.

Chapter 9

Systems Internals

9.1 Introduction

So far we have discussed the design and the features of the trio of systems — checkpointing, PACE and COMET — in depth, but said little about how they work internally. In this chapter we will examine the algorithms that define the operation of these systems, and where appropriate, describe their implementation in the Unix environment.

These systems need to be built so that they work well with each other, so an implementation must tackle the issues of interfacing them. The question of security comes up as well, since we have an environment consisting of multiple users and multiple machines. Care must be taken to ensure that the users are restricted to accessing only those resources to which they have the right of access.

The implementations we describe make considerable use of Unix features and standard Unix programs. Our philosophy has been to keep implementation straightforward by reusing existing frameworks as much as possible. For instance, we use the `rhosts` framework extensively, to ensure the authenticity of users within the COMET system. This approach has the advantage of freeing us from the chore of implementing a secure system from scratch. Moreover, this sort of reuse ensures that the final system is conceptually better integrated with the rest of the operating system.

The internals we describe are necessarily tied up with Unix concepts, of which some may be obscure. The reader is referred to W. Richard Stevens' *UNIX Network Programming* [20] for a detailed description of these concepts.

9.2 Specification of Checkpointing

The checkpointing subsystem is mostly constructed by the user himself, and is in fact less of a system and more of a specification. As such, there are no "implementation algorithms". However checkpointing needs to follow certain rules in order to work with PACE and COMET. These rules are described here. It will become clear later, when we examine the implementation of PACE and COMET, how exactly our specification of checkpointing helps their implementation.

9.2.1 The Checkpointing API

A checkpointable program must make some subroutine calls during certain events to be fully usable with PACE and COMET. These subroutine calls are different from the subroutine calls which write the checkpoint and reload it. Both of those subroutines are written by the user himself with some conventions imposed on him as to what names to use for the checkpoint files. These conventions are explained in Section 9.2.2.

Below we describe the subroutines of the checkpointing API, categorized by the situations in which they must be invoked. These functions perform the task of communicating information between the checkpointing subsystem on one side, and PACE and COMET on the other. Such communication is needed because checkpointing is controlled by PACE and COMET. An important entity used by the checkpointing subsystem is the *checkpointing log*. This is simply a file, present in the same directory as the checkpointing files. Many of the functions of the API write to this file, thus logging the current status of the program, and the COMET tools read from it to monitor the progress of the checkpointing program.

- *Program Startup*

  During program startup the `Log_Startup` subroutine must be called. This writes an entry to the checkpointing log that the program has successfully started up. The function also checks whether the current run is a normal start or a restart, and makes the entry into the checkpointing log appropriately.

  Internally, the `Log_Restart` function uses the `Restarting` function (described in Chapter 6) and later on in this section) to distinguish between an ordinary start and a restart.

- *Checkpointing*

  The `Checkpointing` function has been described earlier in Chapter 6. This function returns information to the program on whether COMET or PACE has commanded a checkpoint.

  In addition, the `LogCheckpointStart` function must be called just before a checkpoint is actually taken. This function writes an entry to the checkpointing log that the checkpointing process has started, and also at what time it started.

  The function `LogCheckpointEnd` is to be called when the checkpoint taking is finished. This function writes the entry to the checkpointing log that the checkpoint was successfully taken, and what time it was taken.

- *Restarting*

  We have already discussed the `Restarting` function in Chapter 6. This function essentially conveys information to the program from COMET, telling it whether the user requested a normal execution, or a restart.

- *Termination*

  Before the program finishes execution, it needs to call the `Log_Terminate` routine. The fact of the program's having finished execution is written to the checkpointing log by this routine.

9.2.2 Checkpointing Files

Because COMET transfers checkpoint files implicitly, that is, without the user having to specify how to copy them, it must have certain advance information about the files.

Firstly, COMET must have information on how the files are named. File names are picked by the user when he writes the subroutines to take the checkpoint, and to reload it. The input file to the `cometcreate` command must list these file names. It is important that the names of the checkpoint files be identical at all sites in a COMET.

COMET also needs type information about the checkpoint files so that it can translate them across machine architectures if necessary. The type information is conveyed by the extension to the checkpoint filename. A file with a `real8` extension consists of double precision floating point data, and

so on. Thus, the checkpoints are stored in multiple files, each of which may contain just one data-type, indicated by the file extension.

9.3 Implementation of the PACE Daemon

9.3.1 Operation

The algorithms used in the PACE daemon can be expressed best as a set of functions which call each other. We describe these in a top-down fashion below, using C-like pseudo code.

Top Level

At the highest level, the PACE daemon constantly accepts requests from users and enqueues them in a priority queue, ordered by the priority[1] of the request. It also repeatedly makes calls to the function `dsptchrq()` whose task is to try to run pending requests from the priority queue. The PACE daemon may also kill some processes to free processors. To be able to kill processes when it needs to, the PACE daemon runs in super user mode.

```
void main ()
{
   while (TRUE)
     {
       Get the next request, if any, from a user and enqueue it.
       dsptchrq ();
     }
}
```

Dispatching Requests

The operation of `dsptchrq()` is also quite simple. It checks each of the enqueued requests in priority order, and tries to service them by calling `tryservice()`. Note that the request is not removed from the priority queue by this function.

[1]The reader is reminded that when we talk of the priority of a request, we refer to request priority, which is a function of user priority and sign up schedule.

```
void dsptchrq ()
{
  for (Scan through queue in priority order)
     tryservice (req);
}
```

Scheduling Requests

The `tryservice()` function is a slightly more complicated one. It checks to see if enough processors are available to satisfy the request, and if that is so, it can execute the request straightaway.

There is a potential problem here though, namely a race condition. Since another, unattached user may start up a job between the time that `tryservice()` checks the number of processors, and the time that it tries to execute the program, the execution may fail even if `tryservice()` finds enough processors to service the request initially.

Failure to service the request may also occur due to security reasons, if the user does not have permission to execute the program. This is discussed in Section 9.3.3. In this situation an error condition must be reported back to the user.

To avoid the race condition, `tryservice()` needs to check whether a request is successfully serviced even if it is able to ascertain beforehand that enough processors are available. This is accomplished by looking at the checkpointing log, since the program makes an entry into the log at startup.

If on the other hand, there are not enough processors available, the daemon must try to free some up by killing some lower priority processes. The priority of a process is the user priority if the process is signed up for the number of processors it is currently using. Otherwise the priority is set to the lowest possible value.

The function `trykill()` checks to see if there are any processes of lower priority than the request which needs to be serviced, and if so it kills them. If `trykill()` fails to find enough processors in use by low priority processes, the request is put back in the queue and the `tryservice()` fails. If `trykill()` succeeds in freeing up the needed number of processors, the daemon must once again try to satisfy this request.

```
int tryservice (req)
{
   while (TRUE)
     {
       if (There are enough processors available)
         {
           Execute the program;
           if (Execution failed due to security violation)
             {
               dequeue (req);
               return FAILURE;
             }
           if (Execution succeeded)
             {
               dequeue (req);
               return SUCCESS;
             }
          Otherwise, request is tried again.
         }
       else
         {
           trykill (pr, n);
           if (trykill succeeded)
              continue;
           enqueue (req);
           return FAILURE;
         }
     }
}
```

Freeing Processors

Before we examine the `trykill()` algorithm, let us take a look at the following C structure containing information about a running process.

```
struct process_info
{
   int uid;
   int pid;
   int upriority;
   int procs;
   int atime;
};
```

The fields of `process_info` correspond to various attributes of a running process: `uid` is the user id of the owner of the process, `pid` is the process id, `upriority` is the priority of the owner as set by the system administrator, `procs` is the number of processors being used by the process and finally, `atime` is the total time that the process has been actively running so far (This excludes the time that the process may have been waiting in a queue to be serviced.)

The algorithm for `trykill()` checks to see if any processes can be killed to free up enough processors to service a request. If this is so, then it sends an message to those processes, notifying them that they will be killed within a particular time period, during which they must take checkpoints or perform whatever other actions they may choose to. At the end of the time interval, the processes are sent a `SIGKILL` signal.

The function is as follows.

```
int trykill (pr, n)
{

   Get information about all running processes with
   priority < pr into the array running[ ]. Suppose
   the size of this array is 'items'.
   Sort running[ ] in the following order,
       Ascending by priority of running[i]->uid
       Ascending by running[i]->atime
       Descending by running[i]->procs
```

```
   for (i = 0; i < items; i++)
     {
       av_procs = av_procs + running[i]->proc[i];
       if (av_procs >= n)
         {
           for (j = 0; j <= i; j++)
              killmessage (running[i]->pid);
              return SUCCESS;
         }
     }
   return FAILURE;
}
```

The arguments to `trykill()` are `pr`, the priority of the request which the daemon is attempting to schedule, and `n`, the number of processors needed by the request. `trykill()` starts off by obtaining information about all processes that are currently active in the system with lower priority than `pr`. Then it sorts these processes first by their priority, then by running time and last by number of processors available.

The rationale for the sort order is that it is preferable to kill the lowest priority processes which have been running for the shortest period of time, to ensure that the owners of those processes lose as little work as possible. It is also desirable to kill as few processes as possible, to satisfy the needs of the request, so we sort the "kill list" in descending order by number of processors used.

Then `trykill()` checks to see if in fact the request can be satisfied, since there is no point in killing low priority processes unless we are certain that this will free up enough processors. If the check succeeds, the processes are killed.

To allow the processes a chance to save their context, the `killmessage()` function actually only sends the process an message of the impending kill. After a timeout period the process is killed. The daemon writes an entry logging the kill and the time of the kill to the checkpointing log of the checkpointing program. This is useful information in case the daemon kills the checkpointing program before it is able to complete the checkpoint. From such an entry, the user can know what happened if the checkpoint did not complete.

The `trykill()` function is not a perfect one, as it is possible for a high priority user (of priority higher than the current request) to intervene between a successful check to determine if enough processors are available, and the actual killing of processes, and start up a large job which uses up enough processors that it prevents the current job from starting up. This is essentially the same sort of race as was discussed in the `tryservice()` algorithm, but occurring here now. The conclusion is that a successful return from `trykill()` does not necessarily imply that the request will be satisfied.

9.3.2 Super User Control

The super user must be able to control the scheduling policy of the PACE daemon to some extent. He must be able to specify at what time slots certain users are allowed to run jobs. He must be able to update this information, the sign up schedule, periodically while the daemon is up and running. This is accomplished by submitting a schedule file to the daemon, which it periodically checks.

The super user may also assign the user priority attribute to each user, which controls the PACE daemon in the manner already discussed. Like the sign up schedule, the user priorities are read by the PACE daemon periodically from a file.

9.3.3 Security in the PACE daemon

An important issue is to ensure that Unix semantics for security be supported by the PACE implementation. In the PACE daemon, this means that no user must be able to submit a request on behalf of another user, or affect his requests in any way (except of course through the scheduling by the daemon). Only the owner of a program must be allowed to successfully execute it. The communication channel through which requests are submitted to the PACE daemon must be secure and allow PACE to verify the identity of the requester. This is done by making the communication channel a message queue [20] with only super user read-write access. The channel is accessed, for submitting requests, by a program installed in super user mode[2]. The mechanisms to enforce security are in two parts and are described below.

[2]Readers familiar with the Unix `passwd` utility will note the similarity with this scheme.

Submitting Requests

The problem here is ensuring that a user cannot masquerade as another while submitting a request. A request consists of the information to satisfy it (like the number of processors and executable path), and the identity of the user. Other attributes of the request, such as priority of the user, must be initialized by the daemon.

The solution for Unix is as follows. The request submission program is the only one permitted to access the request channel, a message queue. The program runs with the effective user id of the super user so that this is possible. This can be done as it is installed in super user mode. The user id of the invoker can be obtained with the `getuid()` system call. Thus the invoker can in no way fake his identity.

```
void main ()
{
   seteuid (SUPER_USER);
   userid = getuid ();
   Enter (request_date, userid) into message queue.
}
```

Executing Requests

Once the system can obtain the correct identity of the requester, it must use this knowledge to enforce that the program runs with that user id. This is non-trivial because the daemon runs with user id set to super user, and so would any processes spawned by it, including of course the one resulting from the request. This problem is tackled in a straightforward manner using the `setreuid()` call below which sets the "real user id". Note that once the user id of the program is set to non-super user, the program can no longer execute any actions permitted to a super user process[3].

```
int acchk_execute (userid, request)
{
   exe_owner = getowner (executable);
   if (exe_owner != userid)
      return ILLEGAL_ACCESS;
```

[3]This is identical to what the login process in a typical Unix system does.

```
   if (fork () == CHILD)
     {
       setreuid (userid);
       execute (request);
     }
}
```

9.4 Implementation of COMET

In this section we will discuss the implementation of the various COMET tools. We will also see how the implementation successfully maintains Unix security semantics in a distributed environment. Security while using COMET means that a user only be able to access his own accounts on machines. In addition, COMET must fully comply with the security model of PACE.

9.4.1 Operation of the Tools

Creation

The creation of a COMET object is done by using the `cometcreate` tool. This tool takes as an argument the name of a description file, which tells it the names of the hosts where the sites are to be created, and also what files are to be copied into the sites.

The operation of the tool is outlined below.

```
void main ()
{
   Read comet description file.
   for (iterate through hosts)
     {
       Use Unix rsh program to create site directories and copy
       files into those directories.
     }
   Create COMET file, write data to it.
}
```

The COMET file contains information such as the data formats on the different machines, information which is used by the `transfer` tool to decide whether translation is needed.

Activation

This tool submits a request for execution at a particular site, to the PACE daemon present there. Like the `cometcreate` tool, this tool uses the `rsh` program to accomplish remote startup. The request submission is done through the `submit` command, which may submit the request to the PACE daemon.

Checkpointing

Checkpointing is done by the `freeze` tool by creating a file named `res` in the directory of the program to be checkpointed. The checkpointing program checks periodically for the presence of this file and when it finds it, writes a checkpoint. Again, the `rsh` program is used to make it possible to employ the `freeze` tool remotely.

The `freeze` tool must not return until the program is either successfully checkpointed or until there is an error. It does this by examining the checkpointing log repeatedly till the checkpointing program has written an entry to it signifying the success of the checkpoint. Alternatively, the checkpointing program may be killed by the PACE daemon, in which case the daemon makes an entry into the log. If this is so, then `freeze` must return and notify the user of the situation.

Restarting

Restarting a program has two components. First the program needs to be activated. Then it needs to be notified that it is being restarted (as opposed to a regular run). We accomplish this by using the `rsh` program to create a file named `res` at the remote site in the directory of the program to be restarted. The `restart` tool is implemented as a shell script, and invokes the `activate` tool.

Migration

The `transfer` tool can copy checkpoint files from one site to another using the `rcp` command. If the data formats used by the source and destination are different, translation of the checkpoint data from one format to another is needed.

Information about the data formats of the sites is stored in the COMET file. The `transfer` tool checks to see if the formats are different, and if so,

uses a locally installed translation program to convert the checkpoint files to a machine independent format before copying it to the remote site. Then the `rsh` program is used to invoke the translation program at the remote site, to convert the checkpoint to the data format there. Different, machine specific versions of the translation program need to be installed at each site.

This `transfer`[4] tool returns when either all checkpoint files have been transferred, or if an error occurs. Such behavior is straightforward to implement as a series of `rcp` commands.

Querying COMET State

The function of the `cometls` tool is, as we have seen, to retrieve information about the status of the various sites in a COMET. The status of any site is obtained by looking at the last entry in the checkpointing log of the site. Below is a list of the various possible statuses and a short description of what they mean.

- *Active*

  A program is running at the site. This is distinct from having a request merely submitted to PACE for scheduling at an appropriate time.

- *Checkpointed*

  The last action at this site was of taking a checkpoint. `cometls` reports the time at which the checkpoint was taken. The site is currently inactive.

- *Computation Terminated*

  The program has finished executing at this site.

- *Killed by Daemon*

  The program was killed by the PACE daemon before it had a chance to take a checkpoint.

- *No Transaction*

  This means that nothing has occurred at this site since creation.

[4] Our implementation does not include the `transfer` tool. We mention it here for the sake of completeness and to demonstrate how the problem of differing data formats may be addressed.

- *Unreachable*

  This indicates machine or network failure.

In addition `cometls` has various switches that allow the user to retrieve more specific information about the remote sites. `cometls` acts as a tool to conveniently examine the checkpointing logs of the various sites.

9.4.2 Interfaces

The COMET tools need to work in conjunction with checkpointing. For instance, in order to obtain the status of a site, `cometls` would need to have some means of knowing what the program was doing. The `checkpoint` tool needs to know when the checkpointing program has finished taking a checkpoint. Therefore,

9.4.3 Security

Security in the case of the COMET tools consists essentially of ensuring that unauthorized users cannot transfer files to machines or accounts that they do not have access to. This is guaranteed by implementing the entire remote command and file transfer mechanism based on the Unix remote hosts system.

What this means is that each host which the user wishes to use as part of the PACE-COMET system must have present on it a file named `.rhosts` which lists the names of all other hostnames in the group, and the user-id of the user on those hosts.

The guiding principle here is to use as much of the existing Unix infrastructure as possible to create a secure system.

Chapter 10

Conclusions and Future Work

10.1 Future Work on Molecular Statics

We have shown that the optimizations described in this thesis have successfully eliminated the `co[]` array and associated arrays as a memory bottleneck. The memory bottleneck now is the `neighbrs[]` array (and other related arrays). Table 10.1 lists the memory requirements of the important data structures after our optimizations (this table is the updated version of Table 4.1).

The δ term in Table 10.1 is because the parallelization of `co[]` is not perfect. However, as we have already seen in Chapter 5, δ is not a dominating factor. Assuming $f = 2i$ (i is the size of an integer and f the size of a double precision floating point number), it is clear that `neighbrs[]` and `idisp[]` dominate heavily. Therefore any improvements in the statics code must be made with a view to reducing the size of these arrays.

10.2 Future Computational Environment Work

10.2.1 SPAM to build Metasystems

A key design goal for PACE-COMET has been to build it as far as possible as a collection of small simple programs which can be combined to accomplish various tasks. The glue which holds these programs together is a UNIX shell. PACE-COMET can also be extended as a set of utilities which accomplish different functions in a metasystem. These utilities can be used in a coherent

Table 10.1: Space requirements of OPMOLSTAT after optimizations.

Array	Size
neighbrs[ ]	$\frac{200ni}{p}$
idisp[ ]	$\frac{200ni}{p}$
workspace[ ]	$\frac{15nf}{p}$
R[ ]	$\frac{nf}{p}$
fi[ ]	$\frac{3nf}{p}$
G[ ]	$\frac{3nf}{p}$
co[ ]	$\frac{3nf}{p} + \delta$
co1[ ]	$\frac{3nf}{p} + \delta$
lco[ ]	$\frac{3ni}{p} + \delta$
lco1[ ]	$\frac{3ni}{p} + \delta$

manner by invoking them from a shell script. We will call applications built in this manner as Shell-PACE-COMET Application Metasystems (SPAM). Seamless integration of the PACE-COMET tools with the UNIX environment is hence very important.

Implementing tools that permit extraction of some of the data in checkpoints so that the user can base further scheduling decisions on that data would be a valuable addition. For instance, this allows the user to take a look at a snapshot of a simulation's state, and decide based on the snapshot whether to go on with the simulation or not. Being able to modify the checkpoint contents is an even more powerful way of affecting the course of the simulation. The checkpoint data files are also a useful way to exchange information between programs. Tools which enabled us to perform such exchanges would aid greatly in building component based metasystems (discussed in Section 2.4).

10.2.2 Controlling Programs

A typical materials science example is observing the behavior of a crystal of a metal alloy under stress, varying its composition. The scientist may wish to obtain the composition that gives the "best" result under some criterion. Additionally, several machines may be available, at varying times, and for varying durations and varying numbers of processors may be available.

Let us suppose for specificity that the scientist needed to find the composition that is the best by some metric. Assume also that the scientist can examine an output parameter `strength` which gives a measure of that metric. The scientist could write a script which controlled the computation using PACE-COMET.

1. Generate multiple cases, α_i with different compositions across the entire possible range. $i = 1, 2, 3....$
2. Queue α_i till completion.
3. Wait till all simulations complete.
4. Transfer all output files to local site.
5. Examine the `strength` parameter of each result to determine which is closest to the solution.

6. Generate a new set of cases close to the best result and goto 2.

Step 2 is really an invocation of another shell script that manages the scheduling of the α_i in a fault tolerant way and in such a way that the time slots available to the user are effectively used. The above script illustrates the fact that writing a few tools which make PACE-COMET more useful provides much benefit.

10.3 Experience of doing HPC

Our experience of working with high performance computational applications has been that the problems of using and maintaining them are very varied in character. Ranging from the design and implementation of parallel algorithms to the issues of recovery from faults and scheduling on a parallel computer, the problems we have described are all too real to be ignored.

The importance of thoroughly understanding a parallel program in order to be able to extend and improve it further cannot be emphasized too much. Unlike many software programs, parallel programs (atleast the molecular statics code) are constructed with many interdependencies. To put it in other words, nearly every part of the program has a bearing on the efficient working of other parts of the program. For example we have seen, in the statics code, how crystal generation is connected with miniblock creation, which in turn is intricately connected with the relaxation process and communication algorithms. The pattern we observe is that as a parallel program is optimized more and more, we tend to use increasingly complicated data structures. Partly this arises because parallel programs have to deal with three aspects of manipulating data:

1. Communication.
2. Distribution.
3. Computation.

Only the third aspect is relevant in the case of sequential programs. It would not be far from the truth to say that efficiently using data structures is thrice as important and thrice as difficult in parallel programs as it is in sequential programs.

We are also convinced of the need to have better tools for running parallel programs. Checkpointing, PACE and COMET are only a start, to what we hope are better, more comprehensive systems. In fact this is the reason why this triad of systems were constructed very simply and without any frills. The UNIX philosophy — "do one job and do it well" — has been the guiding principle in designing these tools, making us opt for a number of small tools rather than combining the functionality of those tools into a single monolithic system.

A final note: none of the systems by themselves — portable checkpointing, queueing (PACE) or migration (COMET) — are unique by themselves. Indeed they are all very simple systems. However, their being used in conjunction with each other, and the ability to combine them with shell scripts makes this a powerful system overall.

Bibliography

[1] E. Arge, A. Bruaset, H. P. Langtangen, *Modern Software Tools for Scientific Computing*, Birkhauser, 1997.

[2] Brian Blount, *Parallelizing a Molecular Statics Code*, B. S. thesis, VPI & SU, 1997.

[3] J. Casas, D. L. Clark, P. S. Galbiati, R. Konuru, S. W. Otto, R. M. Prouty, Jonathan Walpole, *MIST: PVM with transparent migration and checkpointing* Presentation at the 3rd annual PVM user's group meeting, Pittsburgh, PA, May 7-9, 1995.

[4] Condor Team, *Condor System Summary*, http://www.dur.ac.uk/c̃ondor/condor.html#Summary.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to Algorithms*, MIT Electrical Engineering and Computer Science Series, July 1990.

[6] Cray Research Inc., *Introducing NQE*, Publication IN-2153 2.97, 1993, 1997.

[7] Geert Deconinck, *User Triggered Checkpointing and Rollback in Massively Parallel Systems*, Ph.D. dissertation, Catholic University of Belgium, 1996.

[8] Digital Equipment Corporation, *User Guide to the Paragon*.

[9] J. Dongarra, B. Tourancheau, *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, SIAM, 1994.

[10] Raphael A. Finkel, Michael L. Scott, Yeshayahu Artsy, Hung-Yang Chang, *Experience with Charlotte: Simplicity versus function in a Distributed Operating System*, IEEE Transactions on Software Engineering, Vol. 15, No. 6, June 1989.

[11] Ian Foster, *Designing and Building Parallel Programs*, Addison Wesley, February 1995.

[12] G.A. Geist, James Arthur Kohl, Philip M. Papadopulous. *CUMULVS: Providing Fault Tolerance, Visualization and Steering of Parallel Applications*, Environments and Tools for Parallel Scientific Computing, August 21-23, 1996.

[13] Andrew Grimshaw, Adam Ferrari, Greg Lindahl, Katherine Holcomb, *Metasystems*, Communications of the ACM, Vol. 41, No. 11, November 1998.

[14] William Gropp, Ewing Lusk, Anthony Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, December 1994.

[15] William Gropp, Marc Snir, Bill Nitzberg, Ewing Lusk, *MPI: The Complete Reference*, MIT Press, October 1998.

[16] P. Messina, T. Sterling, *System Software and Tools for High Performance Computing Environments*, SIAM 1993.

[17] John K. Ousterhout, Andrew R. Chrenson, Fredrick Douglis, Michael N. Nelson, Brent B. Welch, *The Sprite Network Operating System*, IEEE Computer, February 1998.

[18] P.M. Papadopoulos, J. A. Kohl, B. D. Semeraro, *CUMULVS: Extending a Generic Steering and Visualization Middleware for Application Fault Tolerance*, Proceedings of the 31st International Conference on System Sciences, Kona, 1998.

[19] Luis M. Silva, Joao G. Silva, Simon Chapple, Lyndon Clarke, *Portable Checkpointing and Recovery*, 1995.

[20] Richard W. Stevens, *Unix Network Programming*, Prentice Hall, April 1990.

[21] Alan Sussman, Susan Graham, James Demmel, Scott Baden, Jack Dongarra, Joel Satz *Programming tools and Environments* Communications of the ACM, Vol. 41, No. 11, November 1998.

[22] Andrew S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, January 1995.

Gautam Pulla

Gautam Pulla graduated from Osmania University, Hyderabad, in May 1997 with a B.E. degree in Computer Science and Engineering. Subsequently he pursued an M.S. degree at Virginia Tech., Blacksburg. He is currently working at Microsoft Corporation, Redmond, as a software development engineer.