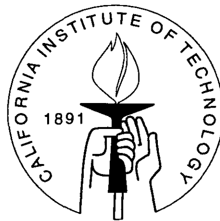
Dynamic Load Balancing and Granularity Control on Heterogeneous and Hybrid Architectures

Thesis by

Jerrell R. Watts

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy



California Institute of Technology
Pasadena, California

1998
(Submitted May 22, 1998)

© 1998
Jerrell R. Watts
All Rights Reserved

Acknowledgements

This author is deeply indebted to two people who greatly assisted me while I conducted the research described in this thesis. I count both of them not only as colleagues, but also as close friends.

I could have asked for no better advisor than Stephen Taylor. Steve's contributions extended far beyond commentary and insight. His infallible cheerfulness and support made graduate life considerably more pleasant than it might otherwise have been. In the end, I am proudest not of having graduated from Caltech, but rather of having had Steve as my advisor.

Fellow graduate student and comrade-at-arms, Marc Rieffel, also made major supporting contributions to this work. It was Marc's direct simulation monte carlo (DSMC) application that provided the bulk of the experimental evidence for this thesis. He assisted in conducting the experiments that provided those results. Finally, he was always willing to listen to any idea I might have and to offer his thoughts.

I also owe thanks to a number of other players. Mikhail Ivanov and Guenadi Markelov, of the Siberian Branch of the Russian Academy of Sciences, provided performance results from their DSMC code, which used a variant of the load balancing techniques presented here. Sergey Gimelshein, also of the Russian Academy of Sciences, contributed substantially to broader aspects of that collaboration. Robie Samanta Roy, formerly at the Massachusetts Institute of Technology, provided insight into his particle-in-cell (PIC) application, which was also targeted by this thesis. Alan Stagg, formerly of Cray Research, assisted in porting the PIC application to the Cray T3D. The homogeneous static partitioner used for the majority of the experiments in this thesis was implemented by John Maweu, an undergraduate at Clark Atlanta University. Jeremy Monin, an undergraduate at Syracuse University, extended the partitioner for heterogeneous networks of computers. My fellow graduate students at Caltech, Daniel Maskit and Michael Palmer, provided valuable comments early on

in my research. I would also like to thank the members of my thesis defense committee, Jim Arvo, Mani Chandy, and Robert van de Geijn, for their comments and suggestions.

This research was sponsored primarily by the Advanced Research Projects Agency under contract number DABT63-95-C-0116. The author was also partially supported by a Graduate Research Fellowship from the National Science Foundation. Collaboration with our Russian colleagues was sponsored by the U.S. Civilian Research and Development Foundation under award number RE1241. Computing resources and infrastructure were provided by the Ballistic Missile Defense Organization under contract number DAAH04-96-1-0319 and the National Science Foundation under contract number AFS-91576590. Additional computing resources were provided by Avalon Computer Systems, Inc., Silicon Graphics, Inc., and the Intel Corporation. Access to a Cray T3D was provided by the NASA Jet Propulsion Laboratory. Access to an Intel Paragon was provided by the Caltech Center for Advanced Computing Research.

Abstract

The past several years have seen concurrent applications grow increasingly complex, as the most advanced techniques from academia find their way into production parallel applications. Moreover, the platforms on which these concurrent computations now execute are frequently heterogeneous networks of workstations and shared-memory multiprocessors, because of their low cost relative to traditional large-scale multicomputers. The combination of sophisticated algorithms and more complex computing environments has made existing load balancing techniques obsolete. Current methods characterize the loads of tasks in very simple terms, often fail to account for the communication costs of an application, and typically consider computational resources to be homogeneous. The complexity of current applications coupled with the fact that they are running in heterogeneous environments has also made partitioning a problem for concurrent execution an ordeal. It is no longer adequate to simply divide the problem into some number of pieces per computer and hope for the best. In a complex application, the workloads of the pieces, which may be equal initially, may diverge over time. On a heterogeneous network, the varying capabilities of the computers will widen this disparity in resource usage even further. Thus, there is a need to dynamically manage the granularity of an application, repartitioning the problem at runtime to correct inadequacies in the original partitioning and to make more effective use of computational resources.

This thesis presents techniques for dynamic load balancing in complex irregular applications. Advances over previous work are three-fold: First, these techniques are applicable to networks comprised of heterogeneous machines, including both single-processor workstations and personal computers, and multiprocessor compute servers. Second, the use of load *vectors* more accurately characterizes the resource requirements of tasks, including the computational demands of different algorithmic phases as well as the needs for other resources, such as memory. Finally, runtime repartition-

ing adjusts the granularity of the problem so that the available resources are more fully utilized. Two other improvements over earlier techniques include improved algorithms for determining the ideal redistribution of work as well as advanced techniques for selecting which tasks to transfer to satisfy those ideals. The latter algorithms incorporate the notion of task migration *costs,* including the impact on an application's communications locality. The improvements listed above are demonstrated on both industrial applications and small parametric problems on networks of heterogeneous computers as well as traditional large-scale multicomputers.

Contents

Acknowledgements **iii**

Abstract **v**

1 Introduction **1**

1.1 Problem Statement 2

1.2 Contributions 3

1.2.1 Nomenclature 4

1.2.2 Notation 5

2 Basic Methodology and Algorithms **6**

2.1 Algorithms 7

2.1.1 Load Evaluation 7

2.1.2 Profitability Determination 9

2.1.3 Load Transfer Calculation 11

2.1.4 Task Selection 14

2.1.5 Task Migration 20

2.2 Results 21

2.3 Related Work 25

2.4 Summary 26

3 Improved Load Transfer Calculation **27**

3.1 General Diffusion Framework 27

3.2 Algorithms 30

3.2.1 First-Order Diffusion Algorithm 31

3.2.2 Second-Order Diffusion Algorithm 34

3.2.3 Adaptive-Timestepping Diffusion Algorithm 36

3.3 Results 37
3.4 Related Work 42
3.5 Summary 43

4 Cost-Driven Task Selection **44**

4.1 Cost-Driven Algorithm 44
4.2 Results 46
4.2.1 Comparison for Communication Cost Metrics 46
4.2.2 Comparison for Other Cost Metrics 50
4.3 Related Work 50
4.4 Summary 51

5 Vector-based Load Balancing **52**

5.1 Algorithmic Modifications 54
5.1.1 Load Evaluation 54
5.1.2 Profitability Determination 54
5.1.3 Load Transfer Calculation 56
5.1.4 Task Selection 57
5.1.5 Task Migration 64
5.2 Results 64
5.2.1 Applications with Multiple Phases 65
5.2.2 Applications with Disparate Computation and Memory Requirements 68
5.2.3 Applications with Rapidly Changing, but Predictable Computation Times 71
5.3 Related Work 74
5.4 Summary 76

6 Dynamic Granularity Control **78**

6.1 Algorithmic Modifications 79
6.1.1 Load Evaluation 79

6.1.2 Profitability Determination 79
6.1.3 Load Transfer Calculation 80
6.1.4 Task Selection 80
6.1.5 Task Migration 81
6.1.6 Granularity Adjustment 81
6.2 Vector Extensions 81
6.2.1 Load Evaluation 82
6.2.2 Profitability Determination 82
6.2.3 Load Transfer Calculation 82
6.2.4 Task Selection 82
6.2.5 Task Migration 83
6.2.6 Granularity Adjustment 83
6.3 Results 83
6.3.1 Synthetic Application 83
6.3.2 Direct Simulation Monte Carlo Application 84
6.4 Related Work 85
6.5 Summary 86

7 Heterogeneous Systems **87**
7.1 Algorithmic Modifications 87
7.1.1 Load Evaluation 87
7.1.2 Profitability Determination 89
7.1.3 Load Transfer Calculation 90
7.1.4 Task Selection 92
7.1.5 Task Migration 92
7.1.6 Granularity Adjustment 93
7.2 Vector Extensions 93
7.2.1 Load Evaluation 93
7.2.2 Profitability Determination 94
7.2.3 Load Transfer Calculation 95

7.2.4 Task Selection 96

7.2.5 Task Migration 96

7.2.6 Granularity Adjustment 96

7.3 Results 96

7.3.1 Heterogeneous Testbed 96

7.3.2 Parametric Experiments 97

7.3.3 Application Experiments 105

7.4 Related Work 107

7.5 Summary 109

8 Conclusions **111**

A Scalable Concurrent Programming Library **113**

A.1 Programming Model 113

A.2 Implementation of Dynamic Load Balancing and Granularity Control 116

A.3 Related Work 118

A.4 Summary 119

B Face-based Finite Element Field Solver **120**

B.1 Derivation 121

B.1.1 Overview of the Finite Element Method 121

B.1.2 Face-based Finite Element Method 123

B.1.3 Conjugate Gradient Method 128

B.2 Concurrent Implementation 128

B.3 Validation 130

B.3.1 Infinite Conducting Plates 130

B.3.2 Conducting Box 130

B.3.3 Infinite Conducting Plates with Intermediate Charge 131

B.3.4 Conducting Box with Interior Charge 131

B.4 Integration into the DSMC Algorithm 132

B.5 Related Work 132

B.6 Summary 132

Bibliography **135**

List of Figures

2.1 Example of hierarchical balancing method on 5-computer array. . . . 12
2.2 Example of pseudopolynomial subset sum algorithm. 18
2.3 Example of fully polynomial subset sum algorithm. 18
2.4 Utilization distributions for DSMC application before and after load balancing. 23
2.5 140,000-tetrahedra grid of the GEC reactor. 24

3.1 Relative load graph for 2×2 mesh of computers. 30
3.2 Worst-case total load transfer (top) and execution times (bottom) of transfer vector algorithms for varying numbers of processors. 40
3.3 Average-case total load transfer (top) and execution times (bottom) of transfer vector algorithms for varying numbers of processors. 41

4.1 Average distance between communicating tasks as a function of load balancing steps for various locality metrics (top) and the improvement of initially poor locality (bottom). 49

5.1 Example of low efficiency in a "balanced" system. 53
5.2 Example of pseudopolynomial vector subset sum algorithm. 62
5.3 Example of fully polynomial vector subset sum algorithm. 63
5.4 ESEX/Argos geometry and cutplanes for ion density, charge-exchange ion density and electric field. 68
5.5 Variation in memory usage after several load balancing rounds with a time-only scalar load metric and a time-memory vector load metric. . 70
5.6 Step times without load balancing and with load balancing for three load metrics. 75

6.1 Efficiency versus the number of partitions per computer for static and dynamic partitionings. 84

6.2 Performance versus the number of partitions per computer for static and dynamic partitionings. 85

7.1 Computer load assignments for single-phase box grid problem on heterogeneous testbed. 99

7.2 Computer load assignments for memory-intensive box grid problem on heterogeneous testbed. (Computers are sorted by memory capacity instead of processing rate.) 104

A.1 A concurrent graph and the internal structure of one of the nodes in that graph. 114

List of Tables

5.1 Summary of the run time and efficiency for the DSMC application without load balancing and with the scalar and vector views of load. . 67

5.2 Summary of the run time and efficiency for the PIC application without dynamic load balancing and with the scalar and vector views of load. 69

5.3 Summary of the efficiency for linear and quadratically varying step times with different load metrics. 73

5.4 Results without load balancing and before and after load balancing for three load metrics. 75

7.1 Descriptions of computers in the heterogeneous testbed. 97

7.2 Results of load balancing two-phase box grid problem on heterogeneous testbed. 100

7.3 Results of load balancing rapidly evolving box grid problem on heterogeneous testbed. 101

7.4 Results of load balancing box DSMC problem on entire heterogeneous testbed. 106

7.5 Results of load balancing box DSMC problem on heterogeneous testbed without computer 1. 106

7.6 Results of load balancing GEC reactor DSMC problem on heterogeneous testbed. 108

List of Programs

2.1 First-order implicit diffusion algorithm for computer i. 15

2.2 Second-order implicit diffusion algorithm for computer i. 16

2.3 Concurrent DSMC algorithm for a single grid partition. 22

3.1 Generalized first-order implicit diffusion algorithm for computer i. . . 34

3.2 Generalized second-order implicit diffusion algorithm for computer i. 36

3.3 Generalized adaptive timestepping diffusion algorithm for computer i. 38

5.1 Vector first-order implicit diffusion algorithm for computer i. 58

5.2 Vector adaptive-timestepping diffusion algorithm for computer i (part 1). 59

5.3 Vector adaptive-timestepping diffusion algorithm for computer i (part 2). 60

5.4 Concurrent DSMC algorithm, with self-consistent fields, for a single partition. 66

B.1 Sequential preconditioned conjugate gradient algorithm. 129

B.2 Concurrent DSMC algorithm, with integrated field solver, for a single partition. 134

Chapter 1 Introduction

Two trends have dominated high-performance scientific computing over the past few years: Applications have grown increasingly complex as practitioners implement state-of-the-art numerical simulation techniques on concurrent hardware, and networks of workstations and shared-memory multiprocessor computers ever more frequently serve as production computing platforms. The impetus for sophisticated computational techniques has primarily come from industry, which finds itself faced with mounting research and development costs. One way to reduce these costs is to use realistic simulations instead of traditional physical prototyping. Unfortunately, to achieve a sufficient level of accuracy in such simulations, implementors must incorporate a wide variety of physics, chemistry and/or biology, often on disparate spatial and temporal scales. For example, a simulation of particle flow in a silicon-wafer etching and deposition chamber (plasma reactor) involves not only tracking the motion of particles but also the chemical reactions that may occur when particles collide with one another or with the silicon wafer. Moreover, particles may contribute to and in turn be influenced by electromagnetic fields. The extent to which the numerical methods associated with these aspects of a simulation contribute to an application's resource requirements, both in terms of processor time and memory, may well depend on the very unknown quantities that the simulation is attempting to resolve. For example, the resource needs for simulating a particular region of a plasma reactor will depend on the changing concentration of particles in that region—a quantity which is specified only as an initial condition. Therefore, a static division and mapping of such a problem will seldom result in uniform resource usage across a set of computers throughout the duration of the computation.

Disparities in resource usage due to the original problem mapping widen when the environment is composed of heterogeneous machines. Because of differences in the capabilities of the machines in such a network, resource usage may vary greatly.

Situations will undoubtedly arise in which light computational workloads are assigned to the most powerful machines and heavy workloads are assigned to the least powerful, effectively magnifying the load imbalance. To handle these situations, a load balancing framework must take into account the relative capacities of the machines, both in terms of their computing power and available memory, to guarantee that load is reassigned appropriately.

Finally, a simple remapping of the existing components of a problem may fail to balance the loads among computers because the initial problem decomposition is too coarse. Just as the loads of problem partitions cannot, in general, be determined a priori, neither can a particular decomposition be deemed adequate because the load of any particular piece of the problem may become arbitrarily high during the course of the computation; any computer to which such a task is mapped will find itself hopelessly overloaded. A solution to this problem is to adjust the granularity of the problem dynamically, subdividing the problem at runtime to increase options for reassigning load from one computer to another and to make more effective use of multiprocessor computers.

1.1 Problem Statement

The load balancing problem as addressed by this thesis is as follows:

Given a set of tasks mapped to a collection of computers, find a remapping of the tasks, or subdivisions thereof, that results in more uniform utilization of the resources of the computers.

Note that the concept of a "task" in the above statement is rather loose. A task need not be an actual thread or process. A task in a fluid dynamics calculation, for example, might be thought of as an operation on a single grid cell or on a collection of grid cells. Such decisions affect the degree to which a load balancing framework must explicitly interact with application data structures. If tasks are considered to be processes, for example, then it may be possible to automatically transfer them

from one computer to another, without assistance from the application developer. If tasks are individual data structures, the programmer must typically provide support routines to facilitate their relocation.

1.2 Contributions

This thesis makes a number of practical contributions to the body of work on dynamic load balancing, addressing computational problems and environments which were poorly served by earlier methods. In particular, from the very start, this work targeted complex, irregular applications. Few assumptions were made regarding the nature of the tasks comprising a computation: Tasks might be coarse- or fine-grained, uniform or disparate in their respective loads. They might communicate with one another very intensively or not at all. The algorithms they execute might result in simple, predictable loads or complex loads with multiple components. The environments in which these tasks execute might be comprised of many computers or only a few, be connected by proprietary, high-speed networks or low-speed, commercial technology. The computers might have available resources which are uniform, or which vary tremendously, with differing memory and differing numbers of and speeds of processors. In short, this work addresses load balancing in a more *general* case, where the neat assumptions typical of most load balancing algorithms do not hold.

The specific contributions of this thesis are:

1) A well-defined methodology for load balancing, including algorithms for each step in the load balancing process. [50, 53].
2) Novel methods for balancing multiple types (*vectors*) of load simultaneously—for example, both computation time and memory usage across a set of computers [51, 55].
3) New techniques for handling heterogeneous systems, in which resources available at each computer may vary greatly. Coupling these methods with the vector techniques yields a very powerful tool for managing multiple resources in a heterogeneous environment [52, 56].

4) A method for dynamically repartitioning a problem to make more effective use of computational resources [45, 56].
5) Algorithms for selecting tasks to satisfy the ideal load transfers which take into account the cost of relocating those tasks, in terms of either the size of their data structures or the impact of their relocation on the communication topology of the application [53, 54].
6) A new algorithm for calculating the ideal transfer of load between computers. This algorithm out-performs existing techniques, either in terms of the judiciousness of work transfer or time of execution [53].

These methods are supported with a body of analytical and experimental study and have been applied to non-trivial applications in science and engineering [45, 50, 51, 52, 53, 54, 55, 56]. In combination, the techniques provide a comprehensive load balancing framework capable of serving a wide variety of applications across a broad array of computational resources.

1.2.1 Nomenclature

Because this thesis targets heterogeneous environments, it is necessary to make clear distinctions among certain terms. When dealing with homogeneous computing environments, notions of "load," "work," "utilization," and "runtime" are often used interchangeably. This is perfectly acceptable because an abstract quantity of work, say 100 iterations of a fluid flow solver over a 100,000-cell grid partition, presumably requires the same execution time on every machine. In heterogeneous environments this is definitely not the case. If a problem is characterized by abstract, algorithmic considerations such as the number of operations or the count of data structures, this will translate differently into processor or memory usage depending on the particular machine on which the problem is run. So, a distinction must be made between the quantities which are invariant across a set of computers and those which vary according to the computer in question. We refer to the former, abstract algorithmic quantities as the *load* of a computer. For example, in a particle simulation, a par-

tition of the problem may contain 125,000 particles. The load for that region could be taken to be 125,000. On one machine, processing those particles might require five compute seconds per iteration and on another machine require ten seconds per iteration. We refer to this variant quantity as the *utilization.* The degree to which a particular computer is utilized by a certain load is determined by that computer's *capacity.*

1.2.2 Notation

The following variables and notations are used to denote various quantities in the system described by the problem statement:

- There are P computers in the system.
- The number of processors on computer i is denoted Q_i.
- If the network connecting the computers is a d-dimensional mesh or torus, the sizes of its dimensions are $K_0, K_1, \ldots, K_{d-1}$, respectively.
- The diameter of the network is denoted D and is the length of the longest path between any two computers in the network, according to the routing algorithm used. (E.g., for a d-dimensional mesh, D would be $\sum_{i=0}^{d-1}(K_i - 1)$, assuming messages are routed fully through each dimension before proceeding to the next.)
- The mapping function from tasks to their respective computers is called M. Thus, $M(i)$ is the computer to which task i is mapped.
- T_i is the set of tasks mapped to computer i.
- The set of neighbors of either computer i or task i is denoted N_i, as appropriate to the context in which i is used. The neighbors of a computer are those adjacent to it in the *physical network.* The neighbors of a task are those tasks with which it communicates.

Chapter 2 Basic Methodology and Algorithms

This chapter considers load balancing for concurrent computations running on networks of *homogeneous* computers. The goal is to minimize the run time of a computation by ensuring that no computer is assigned a processing load significantly greater than the average load. The loads of tasks are characterized in a simple manner and are assumed to change in a relatively slow fashion with respect to the time scale on which load balancing is performed. Moreover, it is assumed that resources other than the processor, such as memory, do not factor into the relocation of the tasks comprising a computation. Finally, the effects of task migration on an application's communication structure are ignored.

To make the load balancing problem more tractable, it is useful to specify a step-by-step, high-level methodology to address the problem. One such methodology is the following [49, 50, 53], which is an extension to that in [57]:

1) **Load Evaluation:** The load of each computer is determined, either by having the programmer provide an estimate of resource needs of the tasks or by actually measuring the tasks' resource usage.
2) **Profitability Determination:** Based on the total load measured at each computer, the efficiency of the computation is calculated, and based on the estimated cost of reassigning tasks, a profitability calculation is used to determine if such a remapping is worthwhile.
3) **Load Transfer Calculation:** Using the loads measured in the first step, computers calculate the ideal degree to which they should transfer load to or from other computers. Transfers of tasks with resource needs equal to these pairwise, directed quantities should result in more uniform resource usage.
4) **Task Selection:** Using the load transfer quantities calculated previously, tasks

are selected for transfer or exchange between computers. This phase may be repeated several times until the transfer quantities have been adequately met.

5) **Task Migration:** Once the tasks' new locations are determined, any data structures associated with those tasks are transferred from their old locations to their new locations, and the computation resumes.

By decomposing the load balancing process into distinct phases, one can experiment in a "plug-and-play" fashion with different strategies at each of the above steps, allowing the space of techniques to be more fully and readily explored. It is also possible to customize a load balancing algorithm for a particular application by replacing more general methods with those specifically designed for a certain class of computations.

2.1 Algorithms

There are several algorithms that one could use for each of the phases listed above. Described here are some of the alternatives, along with motivations for particular choices in some instances.

2.1.1 Load Evaluation

The usefulness of any load balancing scheme is directly dependent on the quality of load measurement and prediction. Accurate load evaluation is necessary to determine that a load imbalance exists, to calculate how much load should be transferred to alleviate that imbalance and to determine which tasks best fit the ideal load transfer quantities. Load evaluation can be performed either completely by the application, completely by the load balancing system or with a mixture of application and system facilities.

The primary advantage of an application-based approach is its predictive power. The application developer, having direct knowledge of the algorithms and their inputs, has the best chance of determining the future load of a task. In a finite-element solver, for example, the load may be a function of the number of grid cells. If the

number of cells changes due to grid adaptation, news of that change can be immediately propagated to the load balancing system. For more complex applications, the disadvantage of this approach is in determining how the abstract load of a task translates into actual CPU cycles. If the execution time of an application is a function of several algorithmic variables, determining the relative weighting of those variables can be difficult. System dependent factors such as the compiler optimizations used, the size of the cache(s), etc., can easily skew the execution time for a task by a large factor.

One way to overcome the performance peculiarities of a particular architecture is to measure the utilization of a task by directly timing it. One can use timing facilities to profile each task, providing accurate measurements in the categories of execution time and communication overhead. These timings can easily be provided by a library or runtime system: Such systems would label any execution time between communication operations as computation time and any execution time actually sending or receiving data as communication time. A system-only approach may fall short when it comes to load prediction, however, because past behavior may be a poor predictor of future performance. For applications in which the load evolves in a relatively smooth fashion, data modeling techniques from statistics, such as robust curve fitting, can be used. This is discussed in greater detail in Chapter 5. However, if the load evolves in a highly unpredictable manner, given that the system has no knowledge of the quantities affecting the load, the application developer must provide additional information.

The most robust and flexible approach is perhaps a hybrid of both the application- and system-only methods. By combining application-specific information with system timing facilities, it is much more practical to predict performance in a complex application. In a particle simulation, for example, the time required in one iteration on a partition of the problem may be a function of the number of grid cells as well as the number of particles contained in those grid cells. By using timing routines, the application can determine how to weight each in predicting the execution time for the next iteration.

Given the limitations of application-only and system-only approaches, a general purpose load balancing framework must allow the use of an application-specific load prediction model and provide the profiling routines necessary to make that model accurate. The system can provide a set of generic models that are adequate for broad classes of applications. The system should also provide feedback on the quality of the load prediction model being used. If the load prediction is inaccurate relative to the actual utilization, the system should generate appropriate warnings.

Whatever the load prediction model used, the output of load evaluation is the following: For a given task j, the load of that task is determined to be l_j. The load of a computer i is therefore the sum of the loads of the tasks assigned to it

$$L_i = \sum_{j \in T_i} l_j \tag{2.1}$$

2.1.2 Profitability Determination

For load balancing to be useful, one must determine *when* to load balance. Doing so is comprised of two phases: detecting that a load imbalance exists and determining if the cost of load balancing exceeds its possible benefits.

The load balance (or efficiency) of a computation is the ratio of the average computer load to the maximum computer load, $\mathit{eff} = \frac{L_{\text{avg}}}{L_{\max}}$. A load balancing framework might, therefore, consider initiating load balancing whenever the efficiency of a computation is below some user-specified threshold $\mathit{eff}_{\min}$. In applications where the total load is expected to remain fairly constant, load balancing would be undertaken only in those cases where the load of some computer exceeds $\frac{L_{\text{avg}}}{\mathit{eff}_{\min}}$, where L_{avg} is calculated initially or provided by the application. A similar approach was described in [28, 33, 57] in which load balancing was initiated whenever a computer's load falls outside specified upper and lower limits.

The above method is poorly suited for situations in which the total load is changing. For example, if a system is initially balanced and the load of every computer doubles, the system is still balanced; the above method would cause load balancing

to be initiated if $\mathit{eff}_{\text{min}}$ was greater than 50 percent. Another method that has been suggested is to load balance if the difference between a computer's load and the local load average (i.e., the average load of a computer and its neighbors) exceeds some threshold [57]. The problem with this technique is that it may fail to guarantee global load balance. Consider, for example, the case of a linear array. If computer i has load iL_{const}, then the local load average at any of the non-extremal computers would be $\frac{(i-1)+i+(i+1)}{3}L_{\text{const}} = iL_{\text{const}}$, so load balancing would not be initiated even for a very small threshold, despite the fact that the global efficiency is only 50 percent. (I.e., $L_{\text{max}} = PL_{\text{const}}$ and $L_{\text{avg}} = \frac{P}{2}L_{\text{const}}$.) Load balancing would be initiated by the extremal computers only if the relative threshold was $\mathbf{O}\big((1 - \mathit{eff}_{\text{min}})^{P-1}\big)$, which would be unreasonably small even for moderate values of $\mathit{eff}_{\text{min}}$ on large arrays. The same analysis applies in the case where a computer would initiate load balancing whenever the relative difference between its load and that of one of its neighbors exceeded some threshold. Once again, to guarantee an efficiency of $\mathit{eff}_{\text{min}}$, the relative difference must in general be less than $\mathbf{O}\big((1 - \mathit{eff}_{\text{min}})^{D}\big)$. The problem with such a tight bound is that, in many cases when it is violated, load balancing may actually be unnecessary.

The reason these ad-hoc methods have been suggested is that they are inexpensive and *completely local.* They also introduce no synchronization point into an otherwise asynchronous application. Certainly these are qualities for which to strive. Given the increasing availability of threads and asynchronous communication facilities, global load imbalance detection may be less costly than previously perceived. By using a separate load balancing thread at each computer, the load imbalance detection phase can be overlapped with computation in an application. If the load balancing threads synchronize, this would have no affect on the application. Thus, the simplest way to determine the load balance may be to calculate the maximum and average computer loads using global maximum and sum operations, which will complete in $\mathbf{O}(\log_2 P)$ steps on most architectures. Using these quantities, one can calculate the efficiency directly.

Even if a load imbalance exists, it may be better not to load balance, simply because the cost of load balancing would exceed the benefits of a better load distri-

bution. The time required to load balance can be measured directly using available facilities. The expected reduction in run time due to load balancing can be estimated loosely by assuming efficiency will be increased to $\mathit{eff}_{\text{min}}$ or more precisely by maintaining a history of the improvement in past load balancing steps. If the expected improvement exceeds the cost of load balancing, the next stage in the load balancing process should begin [57]. More precisely, load balancing should be undertaken when the following holds

$$(\mathit{eff}_{\text{cur}} < \mathit{eff}_{\text{min}}) \bigwedge \left(\left(1 - \frac{\mathit{eff}_{\text{cur}}}{\mathit{eff}_{\text{new}}} \right) T_{\text{step}} > T_{\text{bal}} \right) \tag{2.2}$$

where $\mathit{eff}_{\text{cur}}$, $\mathit{eff}_{\text{min}}$, $\mathit{eff}_{\text{new}}$ are the current efficiency, desired minimum efficiency and expected efficiency after load balancing, respectively, T_{step} is the time until the next load balancing opportunity, and T_{bal} is the estimated time for load balancing.

2.1.3 Load Transfer Calculation

After determining that it is advantageous to load balance, one must calculate how much load should *ideally* be transferred from one computer to another. In the interest of preserving communication locality, these transfers should be undertaken between neighboring computers. Of the load transfer algorithms presented in the literature, three in particular stand out: the hierarchical balancing method, the generalized dimensional exchange, and diffusive techniques.

Hierarchical Balancing Method

The hierarchical balancing (HB) method is a global, recursive approach to the load balancing problem [19, 57]. In this algorithm, the set of computers is divided roughly in half, and the total load is calculated for each subset. The load transfer between the subsets is that required to make the load per computer in each equal. I.e., for one subset of P_1 computers with total load L_1 and another subset of P_2 computers having

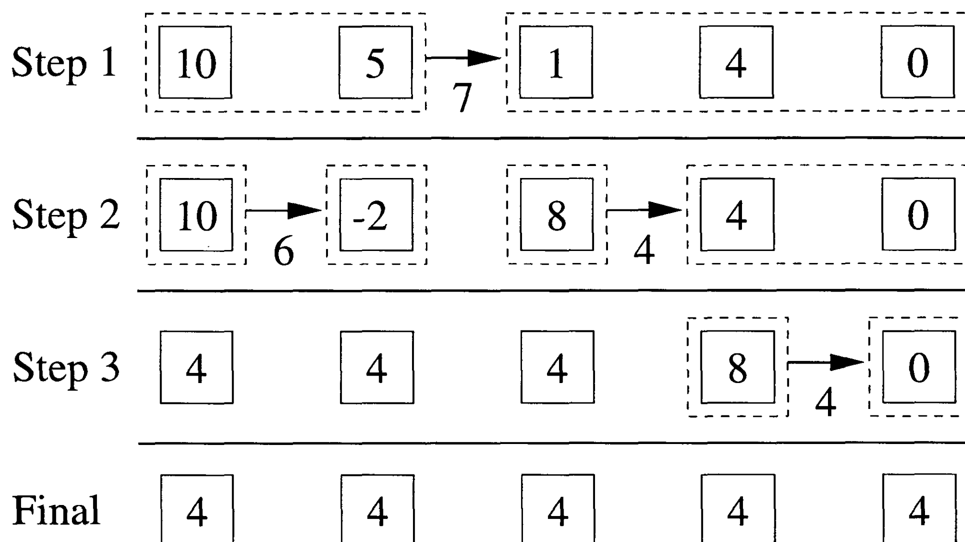


Figure 2.1: Example of hierarchical balancing method on 5-computer array.

an aggregate load of L_2, the transfer from the first subset to the second is given by

$$\Delta L_{(1,2)} = L_1 - \frac{P_1}{P_1 + P_2}(L_1 + L_2) = \frac{P_2 L_1 - P_1 L_2}{P_1 + P_2} \tag{2.3}$$

Once the load transfer has been calculated, each subset is itself divided and balanced recursively, taking into account transfers calculated at higher levels. The HB algorithm calculates the transfers required to achieve "perfect" load balance in $\mathbf{O}(\log_2(P))$ steps. An example of this process is shown in Figure 2.1. In that figure, computers are represented by squares, with their loads in the centers. The dotted boxes indicate the subsets at each stage, and the arrows are the load transfers between subsets. Note that the load of a computer temporarily becomes negative, assuming that the computers at the interface between subsets are responsible for transferring load between the subsets.

One disadvantage of the HB method is that all data transfer between two partitions occurs at a single point. While this may be acceptable on linear array and tree networks, it will fail to fully utilize the bandwidth of more highly connected networks. A simple generalization of the HB method for meshes and tori is to perform the algo-

rithm separately in each dimension. For example, on a 2-D mesh, the computers in each column could perform the HB method (resulting in each row having the same total load), then in each row (resulting in each computer having the same total load). This modification is similar to the row/column broadcast approach used in [5]. For general, d-dimensional meshes and tori, this algorithm requires $\mathbf{O}\left(\sum_{i=0}^{d-1} \log_2(K_i)\right)$ steps. Note that, in the case of hypercubes, this dimensional hierarchical balancing (DHB) method reduces to the dimensional exchange method presented for hypercubes in [11, 57].

Generalized Dimensional Exchange

In the dimensional exchange (DE) method, the computers of a hypercube pair up with their neighbors in each dimension and exchange half the difference in their respective loads. This results in balance in $\log_2(P)$ steps. The authors of [59] present a generalization of this technique for arbitrary connected graphs, which they call the generalized dimensional exchange (GDE). For a network of maximum degree $|N_{\max}|$, the links between neighboring computers are minimally colored so that no computer has two links of the same color. For each edge color, a computer exchanges with its neighbor across that link λ times their load difference. This process is repeated until a balanced state is reached

$$\Delta L_{(i,j)}^{(t+1)} = \Delta L_{(i,j)}^{(t)} + \lambda(L_i^{(t)} - L_j^{(t)}) \text{ for each neighbor } j \in N_i$$

where

$$\Delta L_{(i,j)}^{(0)} = 0 \text{ for each neighbor } j \in N_i$$

For the particular case where λ is 0.5, the GDE algorithm is called the averaging GDE method (AGDE) [59]. (The AGDE method was also presented in [19] but was judged to be inferior to the HB method because of the latter's lower time complexity.) The authors of [59] also present a method for determining the value of λ for which the algorithm converges most rapidly; they call the GDE method using this parameter the optimal GDE method (OGDE). While these methods are very diffusion-like and

have been described as "diffusive" in the literature [19], they are not based on a numerical solution of the diffusion equation, as the authors of [59] rightly point out.

Diffusion

Diffusive methods are based on the solution of the diffusion equation, $\frac{\partial L}{\partial t} = \nabla^2 L$. Diffusion was first presented as a method for load balancing in [11]. Diffusion was also explored in [57] and was found to be superior to other load balancing strategies in terms of its performance, robustness and scalability. A more general diffusive strategy was presented in [16]; it is shown in modified form as Program 2.1. Unlike previous work, this method uses an implicit differencing scheme to solve the heat equation on a multi-dimensional torus to a specified accuracy. The advantage of an implicit scheme is that the timestep size in the diffusion iteration is not limited by the number of neighbors. For explicit schemes, the timestep size is limited to $\frac{1}{2d}$ on a d-dimensional mesh or torus. In [49], an improved, second-order diffusion scheme was derived. That algorithm is shown as Program 2.2. Derivations of both the first- and second-order algorithms are given in Chapter 3, as part of the derivation of a faster, more general diffusion scheme.

2.1.4 Task Selection

Once load transfers between computers have been calculated, it is necessary to determine which tasks should be moved to meet those quantities. The quality of task selection directly impacts the ultimate quality of load balancing.

There are two options in achieving the desired transfer between two computers. One can attempt to move tasks unidirectionally from one computer to another, or one can exchange tasks between the two computers, resulting in a net transfer of work. If tasks are numerous and fine-grained, a simple one-way transfer using first-fit selection may suffice. However, if the average task load is high relative to the magnitude of the transfers, it may be very difficult to find tasks that fit those quantities. By exchanging tasks one can potentially satisfy small transfers by swapping two sets of

```
diffuse(...)
    A_i := 1 + α|N_i|
    T_{i,j} := α
    n := ⌈ ln α / ln[ globalmax( A_i^{-1} Σ_{j∈N_i} T_{i,j} ) ] ⌉
    ΔL_{(i,j)} := 0 for all j ∈ N_i
    while eff < eff_min do
        L_i^{(0)} := L_i
        for m := 1 to n do
            send L_i^{(m-1)} to all j ∈ N_i
            receive L_j^{(m-1)} from all j ∈ N_i
            L_i^{(m)} := A_i^{-1} ( L_i^{(0)} + Σ_{j∈N_i} T_{i,j} L_j^{(m-1)} )
        end for
        ΔL_{(i,j)} := ΔL_{(i,j)} + A_i^{-1} T_{i,j} ( L_i^{(0)} - L_j^{(n-1)} ) for all j ∈ N_i
        L_i := L_i^{(n)}
    end while
end diffuse
```

Program 2.1: First-order implicit diffusion algorithm for computer i.

tasks with roughly the same total load. In cases where there are enough tasks for one-way transfers to be adequate, a cost metric such as that described in Chapter 4 can be used to reduce unnecessary exchanges.

The problem of selecting which tasks to exchange to achieve a particular load transfer is **NP**-complete, since it is simply the subset sum problem. Fortunately, approximation algorithms exist which allow the subset sum problem to be solved to a specified non-zero accuracy in polynomial time [35]. For a given transfer, $\Delta L_{(i,j)}$, the goal is to find the subset of the n total tasks on computers i and j, which, if exchanged, would result in the net transfer of load closest to $\Delta L_{(i,j)}$ without exceeding it. Note that since the tasks on computer j are being considered for transfer from j to i, rather than from i to j, their loads are taken as negative. Also, note that if the loads are measured as real numbers, such as fractions of a second, these values can be converted to integers by expressing them in round units such as milliseconds.

```
diffuse(...)
    $\mathcal{A}_i := 1 + \frac{\alpha}{2}|N_i|$
    $\mathcal{B}_i := 1 - \frac{\alpha}{2}|N_i|$
    $\mathcal{T}_{i,j} := \frac{\alpha}{2}$
    $n := \left\lceil \frac{\ln \alpha}{\ln \left[ \text{globalmax}\left( \mathcal{A}_i^{-1} \sum_{j \in N_i} \mathcal{T}_{i,j} \right) \right]} \right\rceil$
    $\Delta L_{(i,j)} := 0$ for all $j \in N_i$
    while eff < eff_min do
        $L_i^{(0)} := \mathcal{B}_i L_i + \sum_{j \in N_i} \mathcal{T}_{i,j} L_j$
        for $m := 1$ to $n$ do
            send $L_i^{(m-1)}$ to all $j \in N_i$
            receive $L_j^{(m-1)}$ from all $j \in N_i$
            $L_i^{(m)} := \mathcal{A}_i^{-1} \left( L_i^{(0)} + \sum_{j \in N_i} \mathcal{T}_{i,j} L_j^{(m-1)} \right)$
        end for
        send $L_i$ to all $j \in N_i$
        receive $L_j$ from all $j \in N_i$
        $\Delta L_{(i,j)} := \Delta L_{(i,j)} + \mathcal{A}_i^{-1} \mathcal{T}_{i,j} \left( L_i^{(0)} - L_j^{(n-1)} \right) + \mathcal{T}_{i,j} (L_i - L_j)$ for all $j \in N_i$
        $L_i := L_i^{(n)}$
    end while
end diffuse
```

Program 2.2: Second-order implicit diffusion algorithm for computer i.

To solve the selection problem descibed above, specify a function, $F(\Delta L, m)$, which is true if a net exchange of ΔL is possible by the exchange of a subset of the first m of the n tasks. Let $F(0,0)$ be true, and let $F(\Delta L, 0)$ be false for all $\Delta L \neq 0$. For $m > 0$, $F(\Delta L, m)$ can be calculated via dynamic programming

$$F(\Delta L, m+1) = F(\Delta L, m) \vee F(\Delta L - l_{m+1}, m) \tag{2.4}$$

The desired transfer is the one for which $F(\Delta L, n)$ is true and for which ΔL is closest to $\Delta L_{(i,j)}$. This process is shown in Figure 2.2. In that figure, the net load exchanges of subsets are represented by dots on the axis. Initially, there is only the empty set, for 0 total load exchange. At each step i, existing subsets are augmented by adding l_i

to each subset, as shown by the arrows. This appears to double the number of subsets at each step, for a total of 2^n subsets in the end. Since the loads are integers, however, the number of subsets is limited by the fact that the load sums of the subsets can only take $\mathbf{O}(nl_{\max})$ different values, where $l_{\max}$ is the largest absolute value of l_j for all j. The execution time of the algorithm is therefore proportional to the values of l_j, rather than merely the number of values n, making it pseudopolynomial. Specifically, the run time is $\mathbf{O}(n^2 l_{\max,})$. Approximating the values l_j allows one to make the algorithm fully polynomial for a given level of approximation. Specifically, when the lower b bits in the representation of each l_j are truncated, where $b = \left\lceil \log \frac{\epsilon l_{\max}}{n} \right\rceil$, the relative deviation from the optimal transfer, which would be found by the original algorithm, is at most $\epsilon = \frac{n 2^b}{l_{\max}}$. This is shown by the following:

$$\sum_{j \in S} l_j \geq \sum_{j \in S'} \geq \sum_{j \in S'} l'_j \geq \sum_{j \in S} l'_j \geq \sum_{j \in S} (l_j - 2^b) \geq \sum_{j \in S} l_j - n2^b \tag{2.5}$$

where S is the optimal subset under with the original values, l_j, and S' is the optimal subset under the approximated values, l'_j. The approximation algorithm therefore has a run time of $\mathbf{O}\left(\frac{n^2 l_{\max}}{2^b}\right) = \mathbf{O}\left(\frac{n^3}{\epsilon}\right)$, which is a polynomial for any given target accuracy, ϵ. The approximation algorithm is shown in action in Figure 2.3. The effect of the approximation is that no distinction is made among subsets lying between adjacent tick marks on the axis.

The next step is to determine ϵ, the accuracy of the approximation algorithm. In general, it may be unnecessary for a computer to fully realize its ideal load transfers. The load transfers given by the algorithms in the previous section are *eager* algorithms. That is, they specify the transfer of load in instances where it may be unnecessary. In the case of a large point load disturbance, for example, the failure of two computers far from that disturbance to transfer their own load may have little or no effect on the global load balance. One way of determining to what extent a computer must achieve its load transfers is the following. In general, a computer has a set of outgoing (positive) transfers and a set of incoming (negative) transfers. For a particular computer i, denote the sum of the former by ΔL_i^+ and the sum of the

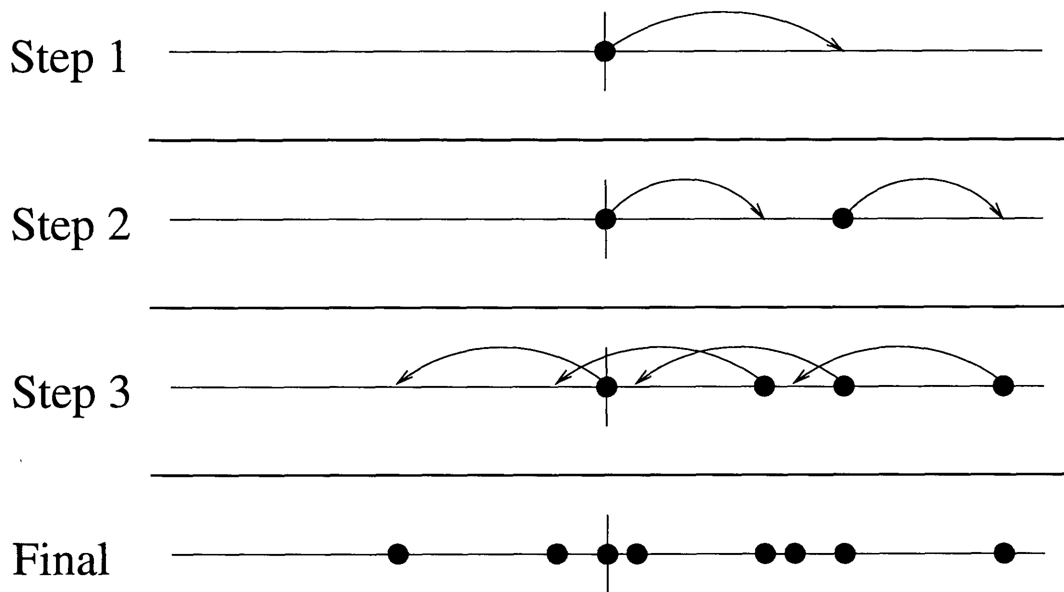


Figure 2.2: Example of pseudopolynomial subset sum algorithm.

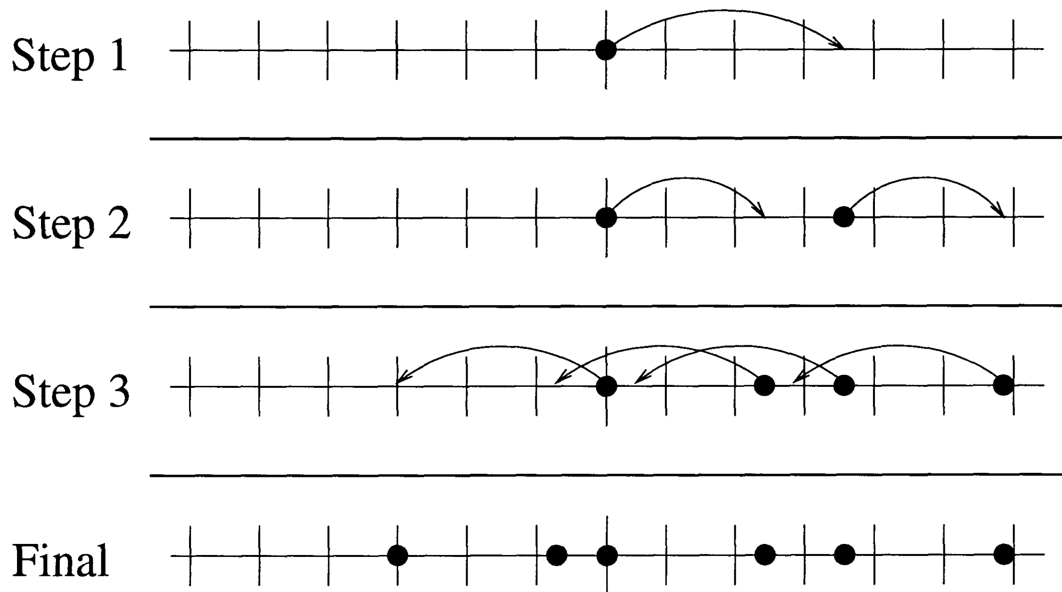


Figure 2.3: Example of fully polynomial subset sum algorithm.

latter by ΔL_i^-. In order to achieve the desired efficiency, a computer must guarantee that its new load is less than $\frac{L_{\text{avg}}}{\mathit{eff}_{\text{min}}}$. Assuming that all of its incoming transfers are achieved, its new load will be at least $L_i - \Delta L_i^-$. Thus, in order to guarantee that its new load is less than $\frac{L_{\text{avg}}}{\mathit{eff}_{\text{min}}}$, a computer must leave at most a fraction ϵ of its outgoing transfers unsatisfied, according to

$$\frac{L_{\text{avg}}}{\mathit{eff}_{\text{min}}} \geq L_i - \Delta L_i^- - (1 - \epsilon)\Delta L_i^+$$

Solving for the maximum such ϵ gives

$$\epsilon_{\text{max}} = 1 - \frac{L_i - \Delta L_i^- - \frac{L_{\text{avg}}}{\mathit{eff}_{\text{min}}}}{\Delta L_i^+}$$

In practice, ϵ_{max} should have a lower limit of 10^{-2} or 10^{-3}, since a value of zero is possible, particularly in the case of the computer with the maximum load. Also, note that using ϵ_{max} in the approximation algorithm does not guarantee that a satisfactory exchange of tasks will be found. No accuracy can guarantee that, since such an exchange may be impossible with a given set of tasks. Instead, it merely provides some guidance as to how hard the approximation algorithm should try to find the best solution.

Since the selection algorithm cannot, in general, satisfy a particular load transfer in a single attempt, it is necessary to make multiple attempts. For example, in the worst-case scenario where all of the tasks are on one computer, only those computers that are neighbors of the overloaded computer can hope to have their incoming load transfers satisfied in the first round of exchanges. In such a case, one would expect that at least $\mathbf{O}(D)$ exchange rounds would be necessary. An algorithm for task selection is thus as follows. The load transfers are colored in the same manner as described for the GDE algorithm. For each color, every computer attempts to satisfy its transfer of that color, adjusting ϵ_{max} to account for the degree to which its transfers have thus far been fulfilled. The algorithm is repeated when the colors have been exhausted. Termination occurs when progress toward further load transfer ceases. Termination

can occur earlier if all of the computers have satisfied the minimum requirement of their outgoing load transfer quantities (i.e., if ϵ_{max} is one at every computer). The first termination condition is guaranteed to be met: For a given configuration of tasks, there is some minimum non-zero exchange. The total outstanding load transfer will be reduced by at least that amount at each step. Since the transfer quantities are finite in size, the algorithm will terminate. This is admittedly a very weak bound. In typical situations, task selection will seldom require more than a few iterations—at most it may require $\mathbf{O}(D)$ steps in the case of severe load imbalance. A safe approach would be to bound the number of steps by some multiple of D.

As the above selection algorithm suggests, a task may move multiple hops in the process of achieving load transfers. Since the data structures for a task may be large, this store-and-forward style of remapping may prove costly. A better method is to instead transfer a *token*, which contains information about a task such as its load and the current location of its data structures. Once task selection is complete and these tokens have arrived at their final destinations, the computers can send the tasks' states directly to their final locations.

2.1.5 Task Migration

In addition to selecting which tasks to move, a load balancing framework must also provide mechanisms for actually moving those tasks from one computer to another. Task movement must preserve the integrity of a task's state and any pending communication. Depending on the granularity of the tasks, different levels of user assistance may be required. If tasks are processes, the runtime system can transfer the entire address space of a task from one computer to another [2, 30]. If tasks are finer-grained entities, such as individual threads or a collection of data structures, transport of a task's state may require assistance from the application, especially when complex data structures such as linked lists or hash tables are involved. For example, the user may be required to provide routines which read and write the state of a task from and to the network, and which free the state once the task has been relocated.

2.2 Results

The techniques described above were implemented in the Scalable Concurrent Programming Library (SCPlib), which is described in Appendix A. The load balancing framework therein provides for both empirical utilization measurement and user-provided load estimates. Load balancing is initiated when the profitability equation (2.2) is true. Load transfers are determined by diffusion and are satisfied using the task selection algorithm described in Section 2.1.4. Finally, tasks are migrated asynchronously, with all communication rerouted automatically and task state communicated with user-supplied functions.

One of the applications implemented with SCPlib was a three-dimensional concurrent particle simulation [38, 45]. That application used direct simulation monte carlo (DSMC), a technique for the simulation of collisional plasmas and rarefied gases. The DSMC method solves the Boltzmann equation by simulating individual particles. Since it is impossible to simulate the actual number of particles in a realistic system, a small number of macroparticles are used, each representing a large number of real particles. The simulation of millions of these macroparticles is made practical by decoupling their interactions. First, the space through which the particles move is divided into a grid. Collisions are considered only for those particles within the same grid cell. Furthermore, collisions themselves are not detected by path intersections but rather are approximated by a stochastic model, for which the parameters are the relative velocities of the particles in question. Statistical methods are used to recover macroscopic quantities such as temperature and pressure. By limiting and simplifying the interactions in this fashion, the order of the computation is drastically reduced.

The concurrent DSMC application has been used to model neutral flow in plasma reactors used in VLSI manufacturing. The DSMC algorithm is executed in parallel by using a partitioned grid. The concurrent DSMC algorithm for a single partition is given as Program 2.3. Each task executes the DSMC algorithm, satisfying data dependencies due to particle transport via communication with its neighbors. Global communication allows the calculation of domain-wide statistics.

```
dsmc_compute(...)
    do
        move particles
        send away particles that exit current partition
        receive particles from neighboring partitions
        collide particles
        gather/scatter to obtain global statistics
        calculate termination condition based on global statistics
    while not converged
end dsmc_compute
```

Program 2.3: Concurrent DSMC algorithm for a single grid partition.

The Gaseous Electronics Conference (GEC) reactor is a standard reactor design that is being studied extensively. In an early version of the DSMC application, which used regular, hexahedral grids, a simulation of the GEC reactor was conducted on a 580,000-cell grid. Of these cells, 330,000 cells represented regions of the reactor through which particles may move; the remaining "dead" (particle-less) cells comprise regions outside the reactor. Simulations of up to 2.8 million particles were conducted using this grid. As this description details, only 57 percent of the grid cells actually contained particles. Even for those cells that did contain particles, the density varied by up to an order of magnitude. Consequently, one would expect that a standard spatial decomposition and mapping of the grid would result in a very inefficient computation. This was indeed the case. The GEC grid was divided into 2,560 partitions and mapped onto 256 processors of an Intel Paragon. Because of the wide variance in particle density for each partition, the overall efficiency of the computation was quite low, at approximately 11 percent. This efficiency was improved to 86 percent by load balancing, including the cost of load balancing. This resulted in an 87 percent reduction in the run time. Figure 2.4 shows the corresponding improvement in load distribution.

On a more recent version of the DSMC code, which uses irregular, tetrahedral grids, a simulation was conducted on a 124,000-cell grid of the GEC reactor. This

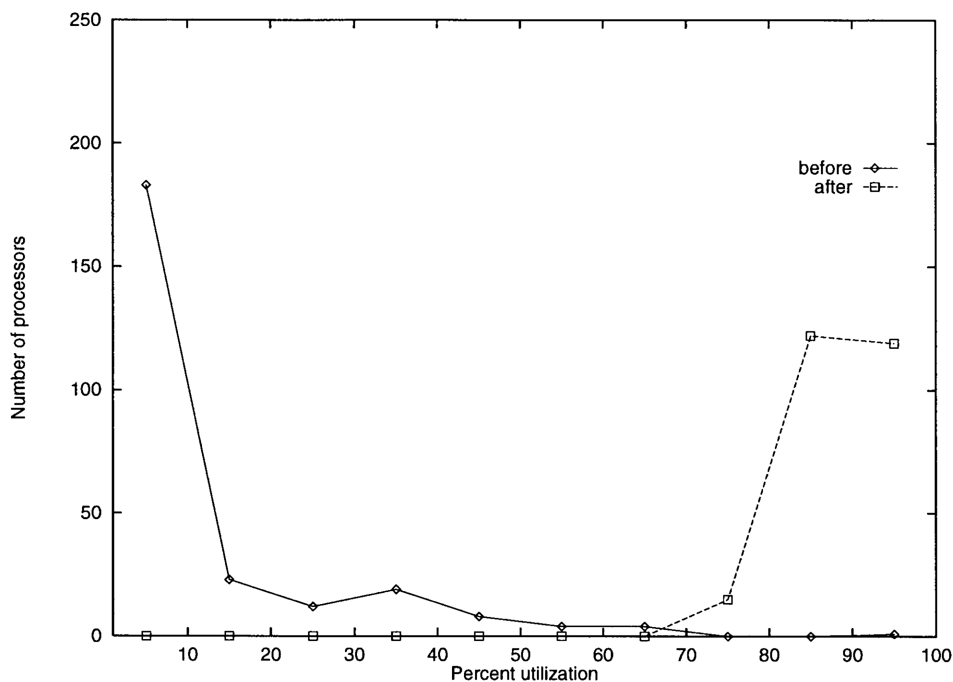


Figure 2.4: Utilization distributions for DSMC application before and after load balancing.

grid is shown in Figure 2.5. The main difference is that this grid is *boundary-fitted* and, thus, does not include the particle-free "dead" regions described above. This problem was run with 1.2 million particles on 128 processors of an Intel Paragon. Each processor had approximately five partitions mapped to it. Load balancing was able to maintain an efficiency of 82 percent, reducing the run time by a factor of 2.6. Load balancing for this problem required, on average, 12 seconds per attempt.

The DSMC application has also been used in the simulation of proprietary reactor designs at the Intel Corporation. These simulations were conducted on networks of between 10 and 25 IBM RS6000 workstations. Without load balancing, the efficiency of these computations was typically between 40 and 60 percent. Load balancing was able to maintain an efficiency of over 80 percent, increasing throughput by as much a factor of two.

Many of the load balancing techniques described in this thesis have also been incorporated into another DSMC code, developed by researchers at the Russian In-

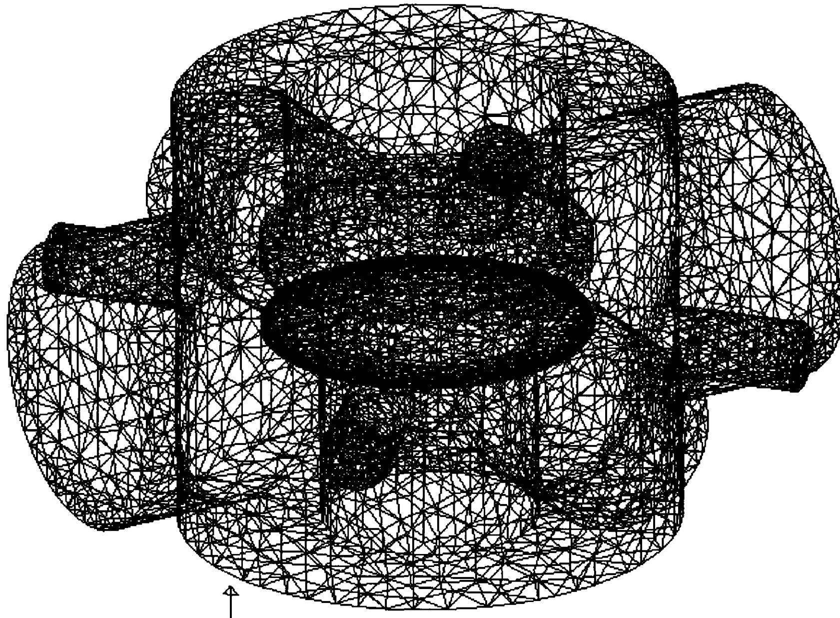


Figure 2.5: 140,000-tetrahedra grid of the GEC reactor.

stitute of Theoretical and Applied Mechanics [21]. In this case, however, load was transferred not by relocating entire partitions from one computer to another, but rather by exchanging small groups of cells along the partition interfaces between adjacent computers. A feature of this approach is that locality is naturally maintained since the algorithm is effectively adjusting partition boundaries. For a problem of space capsule reentry running on up to 256 processors of an Intel Paragon, 80 percent of linear speedup was obtained with dynamic load balancing, versus 55 percent of ideal speedup for a random static mapping, and 10 percent of ideal speedup with no load balancing. It is interesting to note that the random static mapping actually achieved fairly good load balance, but that the communication between the widely distributed cells was very costly, reducing scalability.

2.3 Related Work

The step-by-step load balancing methodology given at the beginning of this chapter is based on that in [57]. The latter framework differs, however, in that it makes no distinction between task selection and task migration. Moreover, the authors consider only uniform sets of tasks, and largely leave issues of task selection to the application developer. The authors do, however, consider other aspects of load balancing that were neglected in this thesis, such as the aging of load information and its effect on the performance of the load balancing system.

Another methodology for load balancing is given in [40]. There, the authors break load balancing into information, transfer and location rules. The information rule is roughly equivalent to load measurement in terms of this thesis; it determines how load information is collected and stored. The transfer rule is similar to the profitability calculation, as it determines when a task should be transferred. Finally, the location rule combines the load transfer calculation and task selection phases, fully determining where a task should be sent.

Other task-based approaches to load balancing include a scalable task pool [18], a heuristic for transferring tasks between computers based on probability vectors [13] and a scalable, iterative bidding model [42]. All of these techniques make assumptions, such as that of complete task independence or task load uniformity, that are not applicable in the context of this thesis, where the goal is to improve the performance of an application divided into communicating, variable-sized tasks.

An alternative to the load transfer algorithms in Section 2.1.3 is the gradient method. Gradient load balancing methods have been explored extensively in the literature [28, 33, 57]. As pointed out in [33, 57], the basic gradient model may result in over- or undertransfers of work to lightly loaded processors. The authors of [33] present a workaround in which computers check that an underloaded processor is still underloaded before committing to the transfer, which is then conducted directly from the overloaded to underloaded processor. While the method does have the scalability of diffusive and GDE strategies, it has been shown to be inferior in its

performance [57].

Recursive bisection methods partition the problem domain to achieve load balance and to reduce communication costs. Most presentations of these techniques appear in the context of static load balancing [4, 58], although formulations appropriate for dynamic domain repartitioning do exist [47, 48]. While many methods exist for repartitioning a computation, including various geometrically based techniques, the most interesting methods utilize the spectral properties of a matrix encapsulating the adjacency in the computation. Unfortunately, these methods have a fairly high computational cost. They also blur the distinct phases of load balancing presented at the beginning of the chapter. The combination of these limitations makes such techniques unsuitable for use in a general purpose load balancing framework.

Another approach for mapping grid-based problems to concurrent hardware involves space-filling curves [34]. These techniques use the curve to generate a mapping that preserves the physical locality inherent in the problem. Load balancing is achieved by assigning different regions of the curve to different computers.

Heuristics for load balancing particle simulations (relevant here because of the application targeted in Section 2.2) include [15], in which partition boundaries are adjusted for a one-dimensional decomposition, and [24], in which a diffusion-like technique is used. While both of these techniques perform well, they are not readily applicable to more general applications or domain decompositions.

2.4 Summary

This chapter has given a basic methodology and algorithms for load balancing. The effectiveness of that framework was demonstrated for a large-scale industrial application. As mentioned at the start, however, these techniques neglect issues of heterogeneity in the computing environment, treat the load of tasks as a simple quantity, and fail to consider the impact of task migration on communication locality. In general, such issues cannot be ignored, and the subsequent chapters take them up, one by one.

Chapter 3 Improved Load Transfer Calculation

This chapter presents the derivation of a general diffusion algorithm for calculating the ideal load transfers between adjacent computers. This technique uses adaptive timestepping to speed convergence once a smooth, low-frequency load distribution is reached. As part of that derivation, a first-order implicit scheme, similar to that in [16], is derived. Then, a second-order accurate technique is presented. These techniques are combined to give the final diffusion algorithm. Finally, the improved diffusion technique is compared to the other load transfer algorithms described in Chapter 2.

3.1 General Diffusion Framework

Although a partial differential equation is the model for diffusive load balancing, the underlying mathematics are actually a system of ordinary differential equations, since there are no spatial derivatives. (A network of computers is inherently spatially discrete.) If the load of computer i is denoted L_i, an equation in a general version of such a system is given by

$$\frac{\mathrm{d}L_i}{\mathrm{d}t} = \sum_{j \in N_i} (\mathcal{D}_{j,i} L_j - \mathcal{D}_{i,j} L_i) \tag{3.1}$$

for all i, where $\mathcal{D}_{i,j} \geq 0$ and $\mathcal{D}_{j,i} \geq 0$ for all i and for $j \in N_i$. I.e., a computer's load changes only in response to the loads of its neighbors. Closer examination of (3.1) reveals that $\mathcal{D}_{i,j}$ and $\mathcal{D}_{j,i}$ have two fundamental features: They seek to establish that the ratio $\frac{L_i}{L_j}$ is equal to the ratio $\frac{\mathcal{D}_{j,i}}{\mathcal{D}_{i,j}}$, and they determine the rate at which those two

ratios are equalized. This is seen more clearly if (3.1) is rewritten as

$$\frac{\mathrm{d}L_i}{\mathrm{d}t} = \sum_{j \in N_i} (\mathcal{D}_{i,j} + \mathcal{D}_{j,i}) \left(\hat{\mathcal{D}}_{j,i} L_j - \hat{\mathcal{D}}_{i,j} L_i\right) \tag{3.2}$$

where

$$\hat{\mathcal{D}}_{i,j} = \frac{\mathcal{D}_{i,j}}{\mathcal{D}_{i,j} + \mathcal{D}_{j,i}} \tag{3.3}$$

and

$$\hat{\mathcal{D}}_{j,i} = \frac{\mathcal{D}_{j,i}}{\mathcal{D}_{i,j} + \mathcal{D}_{j,i}} \tag{3.4}$$

In this case, it is clear that the coefficients $\hat{\mathcal{D}}_{i,j}$ and $\hat{\mathcal{D}}_{j,i}$, which always sum to 1, establish the ratio of L_i to L_j, and that the rate at which that ratio is established is proportional to $\mathcal{D}_{i,j} + \mathcal{D}_{j,i}$. If (3.1) is rewritten in matrix vector form, the resulting system of equations is

$$\frac{\mathrm{d}L}{\mathrm{d}t} = \mathcal{C}L \tag{3.5}$$

where

$$\mathcal{C}_{i,i} = -\sum_{j \in N_i} \mathcal{D}_{i,j} \tag{3.6}$$

and

$$\mathcal{C}_{i,j} = \mathcal{D}_{j,i} \tag{3.7}$$

for $j \in N_i$; otherwise, $\mathcal{C}_{i,j} = 0$.

(3.5) is actually a *compartmental system* [3, 60], as it satisfies the following criteria:

1) Diagonal terms are nonpositive: $C_{i,i} \leq 0$ for all i.
2) Off-diagonal terms are nonnegative: $C_{i,j} \geq 0$ for all i and for all $j \neq i$.
3) Column sums are nonpositive: $\sum_i C_{i,j} \leq 0$ for all j.

In particular, the column sums of C are all zero, so the system is a *conservative* compartmental system. (I.e., total load is preserved.) Compartmental systems have been used extensively in the study of biological systems, such as oxygen transport in the bloodstream. The basic concept is to represent a system as a graph, where vertices

represent biological entities, such as a cell, and the edges represent pathways, such as a cell membrane, through which material can pass. The coefficients determine the rate and gradient of material transmission through the interfaces between compartments.

An interesting property of compartment systems is that they are not, in general, simply diffusive systems which converge towards some steady state in which material ceases to be transferred. In fact, depending on the coefficient matrix, $\mathcal{D}$, permanent flow patterns may exist. This is one reason why compartment systems are useful for studying biological systems. Persistent flow is not what one seeks to have in load balancing, however. To guarantee that no flow patterns exist in (3.5), one must not merely guarantee that $\frac{\mathrm{d}L}{\mathrm{d}t} = \mathcal{C}L = 0$ for some $L \neq 0$, since that implies only that the system has reached equilibrium. Instead, one must guarantee that $\mathcal{D}_{j,i}L_j - \mathcal{D}_{i,j}L_i = 0$ for all i and for all $j \in N_i$. This can only occur if the coefficient matrix D is consistent; that is, it can only happen if the relative loads can actually be achieved. Consider, for example, a network of three computers. If the ratio $\frac{L_0}{L_1}$ is to be 2, and the ratio $\frac{L_1}{L_2}$ is to be 2, and the ratio $\frac{L_2}{L_0}$ is also to be 2, then there is an inconsistency; the ratios cannot be established by any positive L_0, L_1 and L_2. In effect, if each expression $\mathcal{D}_{j,i}L_j - \mathcal{D}_{i,j}L_i$ is represented by separate row of a matrix $\hat{\mathcal{C}}$, one seeks to establish that $\hat{\mathcal{C}}L = 0$ for some $L \neq 0$. $\hat{\mathcal{C}}$ is in general an overspecified system of equations, which is why inconsistencies are possible.

Attempts to analyze $\hat{\mathcal{C}}$ along standard lines of linear algebra obscure the fundamental features that $\mathcal{D}$ must have. An alternative procedure is to use graphical analysis, along the lines of signal flow networks [12, 22], which are used to understand compartmental systems, as well as to study electrical circuits [31]. In signal flow network analysis, a series of simplification rules are applied to the network; these transformations are analogous to operations on determinants to find solutions to a system of equations. A similar analysis can be used to check the consistency of $\hat{\mathcal{C}}$. In particular, let G be a weighted directed graph, which has a vertex for each computer, two edges between each pair of neighboring computers, and a self edge for each computer. Let the edge from computer i to its neighbor j be weighted $\frac{\mathcal{D}_{j,i}}{\mathcal{D}_{i,j}}$, and let the edge from computer i to itself be weighted 1. A graph for a 2×2 mesh of computers

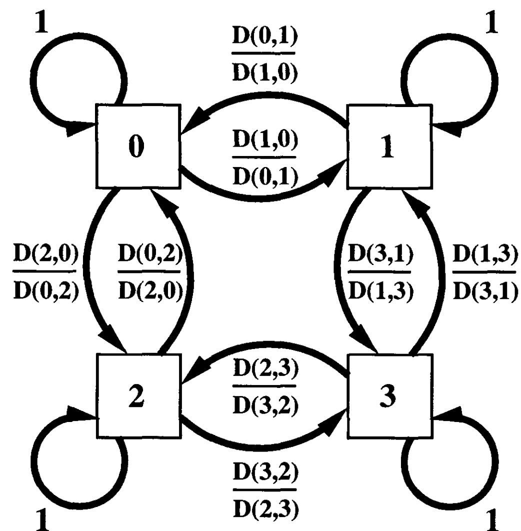


Figure 3.1: Relative load graph for 2×2 mesh of computers.

is shown in Figure 3.1. What this graph in effect represents is the relative loads that (3.5) seeks to establish. Note that in order for $\hat{\mathcal{C}}$ to be consistent, the product of the edge weights along all paths in G from some vertex i to another vertex j must be the same. If two paths from i to j exist which have different products, then the load ratio of computers i and j is inconsistent, since two sets of equations (those along the first and second paths) imply different values for that ratio. Note that equality of path products holds if and only if all cycle products in G have a value of 1. (If the path product from vertex i to vertex j is p, then the product of any path from j to i is $\frac{1}{p}$. Therefore, the product of any such cycle is $\frac{p}{p} = 1$.) To put it another way, a computer's load must be equal to itself. This cycle product property will be used in a subsequent chapter to show that particular choices of $\mathcal{D}$ are valid.

3.2 Algorithms

The diffusion problem described by (3.1) can be approached in a number of ways. In Section 2.1.3, a first-order approach was given that solved (3.1) for the case where $\mathcal{D}_{i,j} = \mathcal{D}_{j,i} = 1$. Below are derivations for first- and second-order algorithms for

general $\mathcal{D}$. These approaches are combined to yield an adaptive-timestepping method that dramatically speeds convergence. The former derivations largely follow those in [16, 49].

3.2.1 First-Order Diffusion Algorithm

The first step to deriving a first-order diffusion scheme it to discretize (3.5), which yields

$$\Delta L = \alpha \mathcal{C} L \tag{3.8}$$

where α is the desired accuracy of the diffusion algorithm and, in this case, is the same as the timestep. (3.8) can be differenced using an explicit (forward-in-time) scheme

$$L^{(t+1)} - L^{(t)} = \alpha \mathcal{C} L^{(t)} \tag{3.9}$$

or using an implicit (backward-in-time) scheme

$$L^{(t+1)} - L^{(t)} = \alpha \mathcal{C} L^{(t+1)} \tag{3.10}$$

Rewriting (3.9) and (3.10) so that the terms are segregated by timestep yields

$$L^{(t+1)} = (I + \alpha \mathcal{C})\, L^{(t)} \tag{3.11}$$

and

$$(I - \alpha \mathcal{C})\, L^{(t+1)} = L^{(t)} \tag{3.12}$$

respectively. Solving (3.12) for $L^{(t+1)}$ yields

$$L^{(t+1)} = (I - \alpha \mathcal{C})^{-1}\, L^{(t)} \tag{3.13}$$

The difficulty with (3.11) is that stability of the iteration is achieved only when α is limited so that the eigenvalues of $I + \alpha\mathcal{C}$ are at most 1 [1, 11]. The Geršgorin disk theorem provides a bound on the eigenvalues of $I + \alpha\mathcal{C}$ [43]: The stability of (3.11)

is guaranteed if

$$|\mu_i - (1 + \alpha\mathcal{C}_{i,i})\,| \leq \alpha \sum_{j \in N_i} |C_{i,j}| \tag{3.14}$$

holds for all i. From the properties of $\mathcal{D}$ given above, (3.14) can be rewritten as

$$\mu_i - \left[1 + \alpha\left(|N_i| - \sum_{j \in N_i}\right)\right] \leq \alpha \sum_{j \in N_i} C_{i,j} \tag{3.15}$$

Eliminating terms in (3.15) and rewriting it gives

$$\mu_i \leq 1 - \alpha|N_i| \tag{3.16}$$

From (3.16), one can see that $\mu_i \leq 1$ for all i implies that

$$\alpha_{\max} = \frac{1}{|N_{\max}|} \tag{3.17}$$

where $N_{\max}$ is the largest set N_i. This is the same result as was given in [11, 59].

Fortunately, (3.13) does not suffer the same restrictions on timestep size as (3.11). Intuitively, this is because $(I - \alpha\mathcal{C})^{-1}$ is a dense matrix, incorporating load information from every computer, whereas $I + \alpha\mathcal{C}$ is a sparse matrix that includes only the load information of a computer's neighbors. However, provided that the inversion of $I - \alpha\mathcal{C}$ is conducted to an accuracy of α, the radius over which load information must actually be gathered in an implicit scheme is bounded by a constant [16].

Using (3.12) as a starting point, note that $I - \alpha\mathcal{C}$ can be rewritten as $\mathcal{A} - \mathcal{T}$, where $\mathcal{A}$ is a matrix with diagonal entries

$$\mathcal{A}_i = 1 - \alpha\mathcal{C}_{i,i} \tag{3.18}$$

and where $\mathcal{T}$ is a matrix with entries

$$\mathcal{T}_{i,j} = \alpha\mathcal{C}_{i,j} \tag{3.19}$$

Thus, (3.12) can be rewritten as

$$(\mathcal{A} - \mathcal{T})L^{(t+1)} = L^{(t)} \tag{3.20}$$

If (3.20) is multiplied through by $\mathcal{A}^{-1}$, the rewritten result is

$$L^{(t+1)} = \mathcal{A}^{-1}\left(\mathcal{T}L^{(t+1)} + L^{(t)}\right) \tag{3.21}$$

(3.21) can be established by a Jacobi iteration

$$\left[L^{(t+1)}\right]^{(m+1)} = \mathcal{A}^{-1}\left(\mathcal{T}\left[L^{(t+1)}\right]^{(m)} + L^{(t)}\right) \tag{3.22}$$

where $\left[L^{(t+1)}\right]^{(0)} = L^{(t)}$. The Geršgorin disk theorem implies that the eigenvalues of $\mathcal{A}^{-1}\mathcal{T}$ are bounded by the row sums of the absolute values of its off-diagonal entries [43]. The accuracy of the Jacobi iteration (3.22) is a function $\alpha = \rho^m$ of the spectral radius. The spectral radius of $\mathcal{A}^{-1}\mathcal{T}$ is the maximum eigenvalue, by the theorem of Perron and Frobenius, since $\mathcal{A}^{-1}\mathcal{T}$ is non-negative [3]

$$\rho\left(\mathcal{A}^{-1}\mathcal{T}\right) = \max_i\left(\mathcal{A}_i^{-1}\sum_{j\in N_i}\mathcal{T}_{i,j}\right) \tag{3.23}$$

This implies that the number of iterations n required is

$$n = \left\lceil \frac{\ln\alpha}{\ln\rho\left(\mathcal{A}^{-1}\mathcal{T}\right)} \right\rceil \tag{3.24}$$

Finally, the load transfer from computer i to computer j is that due to including the load of computer j in the equations. I.e., it is the difference between what the load of computer i is with the actual load of computer j, and what the load of computer i would have been if the loads of computers i and j were the same, relative to the coefficients $\mathcal{D}_{i,j}$ and $\mathcal{D}_{j,i}$

$$\Delta L_{(i,j)}^{(t+1)} = \Delta L_{(i,j)}^{(t)} + \mathcal{A}_i^{-1}\mathcal{T}_{i,j}\left(\frac{\mathcal{D}_{i,j}}{\mathcal{D}_{j,i}}\left[L_i^{(t+1)}\right]^{(0)} - \left[L_j^{(t+1)}\right]^{(n-1)}\right) \tag{3.25}$$

```
diffuse(...)
    A_i := 1 + α Σ_{j∈N_i} D_{i,j}
    T_{i,j} := α D_{j,i}
    n := ⌈ ln α / ln[globalmax(A_i^{-1} Σ_{j∈N_i} T_{i,j})] ⌉
    ΔL_{(i,j)} := 0 for all j ∈ N_i
    while eff < eff_min do
        L_i^{(0)} := L_i
        for m := 1 to n do
            send L_i^{(m-1)} to all j ∈ N_i
            receive L_j^{(m-1)} from all j ∈ N_i
            L_i^{(m)} := A_i^{-1} (L_i^{(0)} + Σ_{j∈N_i} T_{i,j} L_j^{(m-1)})
        end for
        ΔL_{(i,j)} := ΔL_{(i,j)} + A_i^{-1} T_{i,j} ((D_{i,j}/D_{j,i}) L_i^{(0)} - L_j^{(n-1)}) for all j ∈ N_i
        L_i := L_i^{(n)}
    end while
end diffuse
```

Program 3.1: Generalized first-order implicit diffusion algorithm for computer i.

The combination of equations (3.18), (3.19), (3.22), (3.24), and (3.25) results in the diffusion algorithm in Program 3.1. Note that for the case where $\mathcal{D}_{i,j} = \mathcal{D}_{j,i} = 1$ for all i and for all $j \in N_i$, the algorithm is the same as Program 2.1.

3.2.2 Second-Order Diffusion Algorithm

A second-order alternative to the differencing in (3.10) is the midpoint rule

$$L^{(t+1)} - L^{(t)} = \alpha \mathcal{C} \left(\frac{L^{(t+1)} + L^{(t)}}{2} \right) \tag{3.26}$$

(3.26) can be rewritten as

$$(I - \frac{\alpha}{2}\mathcal{C})L^{(t+1)} = (I + \frac{\alpha}{2}\mathcal{C})L^{(t)} \tag{3.27}$$

Substituting $\mathcal{A}-\mathcal{T}$ for $I-\frac{\alpha}{2}\mathcal{C}$ and $\mathcal{B}+\mathcal{T}$ for $I+\frac{\alpha}{2}\mathcal{C}$ in (3.27) yields

$$(\mathcal{A}-\mathcal{T})L^{(t+1)} = (\mathcal{B}+\mathcal{T})L^{(t)} \tag{3.28}$$

where $\mathcal{A}$ and $\mathcal{B}$ are matrices with diagonal entries

$$\mathcal{A}_i = 1 - \frac{\alpha}{2}\mathcal{C}_{i,i} \tag{3.29}$$

and

$$\mathcal{B}_i = 1 + \frac{\alpha}{2}\mathcal{C}_{i,i} \tag{3.30}$$

respectively, and $\mathcal{T}$ is a matrix with off-diagonal entries

$$\mathcal{T}_{i,j} = \frac{\alpha}{2}\mathcal{C}_{i,j} \tag{3.31}$$

Multiplying (3.28) through by $\mathcal{A}^{-1}$ and rewriting gives the equation

$$L^{(t+1)} = \mathcal{A}^{-1}\left[\mathcal{T}L^{(t+1)} + (\mathcal{B}+\mathcal{T})\,L^{(t)}\right] \tag{3.32}$$

A Jacobi iteration establishes (3.32)

$$\left[L^{(t+1)}\right]^{(m+1)} = \mathcal{A}^{-1}\left[\left[\mathcal{T}L^{(t+1)}\right]^{(m)} + (\mathcal{B}+\mathcal{T})\,L^{(t)}\right] \tag{3.33}$$

where $\left[L^{(t+1)}\right]^{(0)} = (\mathcal{B}+\mathcal{T})\,L^{(t)}$. The number of Jacobi iterations required to reach an accuracy of α is the given by (3.24) above.

The load transfer from computer i to computer j is given by

$$\begin{aligned}\Delta L_{(i,j)}^{(t+1)} &= \Delta L_{(i,j)}^{(t)} + \mathcal{A}_i^{-1}\mathcal{T}_{i,j}\left(\frac{\mathcal{D}_{i,j}}{\mathcal{D}_{j,i}}\left[L_i^{(t+1)}\right]^{(0)} - \left[L_j^{(t+1)}\right]^{(n-1)}\right) + \\ &\quad \mathcal{T}_{i,j}\left(L_i^{(t)} - L_j^{(t)}\right)\end{aligned} \tag{3.34}$$

Equations (3.29), (3.31), (3.33), (3.24), and (3.34) together comprise the diffusion algorithm in Program 3.2. When $\mathcal{D}_{i,j} = \mathcal{D}_{j,i} = 1$ for all i and for all $j \in N_i$, this

```
diffuse(...)
    A_i := 1 + (α/2) Σ_{j∈N_i} D_{i,j}
    B_i := 1 − (α/2) Σ_{j∈N_i} D_{i,j}
    T_{i,j} := (α/2) D_{j,i}
    n := ⌈ ln α / ln[ globalmax( A_i^{-1} Σ_{j∈N_i} T_{i,j} ) ] ⌉
    ΔL_{(i,j)} := 0 for all j ∈ N_i
    while eff < eff_min do
        send L_i to all neighbors j ∈ N_i
        receive L_j from all neighbors j ∈ N_i
        L_i^{(0)} := B_i L_i + Σ_{j∈N_i} T_{i,j} L_j
        for m := 1 to n do
            send L_i^{(m−1)} to all j ∈ N_i
            receive L_j^{(m−1)} from all j ∈ N_i
            L_i^{(m)} := A_i^{-1} ( L_i^{(0)} + Σ_{j∈N_i} T_{i,j} L_j^{(m−1)} )
        end for
        ΔL_{(i,j)} := ΔL_{(i,j)} + A_i^{-1} T_{i,j} ( (D_{i,j}/D_{j,i}) L_i^{(0)} − L_j^{(n−1)} ) + T_{i,j} (L_i − L_j)
            for all j ∈ N_i
        L_i := L_i^{(n)}
    end while
end diffuse
```

Program 3.2: Generalized second-order implicit diffusion algorithm for computer i.

algorithm is that in Program 2.2.

3.2.3 Adaptive-Timestepping Diffusion Algorithm

While the algorithms above quickly decimate large load imbalances, they converge slowly once a smooth, low-frequency state is reached. One way to overcome this difficulty is to increase the timestep size, δt, as the load imbalance becomes less severe. A rigorous technique to do this is to apply both of the above methods to calculate ΔL for a particular δt [36]. The first-order method in Program 3.1 produces a local error of $\mathbf{O}(\delta t^2)$. The second-order accurate method in Program 3.2 produces a local error of $\mathbf{O}(\delta t^3)$. Thus, if a timestep is taken with both methods, the difference

between the values produced by each gives an estimate of the error for that δt. Taking the maximum such difference at any computer to be denoted $err_{\max}$, the relative error is $err_{\text{rel}} = \frac{err_{\max}}{L_{\max}}$. Using this error estimate, which is proportional to δt^2, δt is adjusted to be as large as possible to achieve the desired error

$$\delta t_{\text{new}} = \delta t_{\text{old}}(1-\alpha)\sqrt{\frac{\alpha}{err_{\text{rel}}}}$$

where $\alpha = 1 - \mathit{eff}_{\min}$ is the desired accuracy. (The $1-\alpha$ term is a safety factor to avoid having to readjust the timestep size if the previous adjustment was too large.) The resulting adaptive timestepping diffusion algorithm is given as Program 3.3.

3.3 Results

Some of the transfer vector algorithms presented in Section 2.1.3 have been previously compared in terms of their execution times [11, 19, 57, 59]. What has been poorly studied, with the exception of experiments in [57], is the amount of *load transfer* these algorithms require to achieve load balance. The algorithms in Section 2.1.3 were implemented using the Message Passing Interface (MPI) [41] and were run on up to 256 processors of a Cray T3D. The hierarchical balancing (HB) algorithm was mapped to the three-dimensional torus architecture of the T3D by partitioning the network along the largest dimension at each stage and transferring load between the processors at the center of the plane of division. The dimensional hierarchical balancing (DHB) technique treated the network as a three-dimensional mesh. The averaging and optimal generalized dimensional exchanges (AGDE and OGDE) and diffusion algorithms took advantage of the wrap-around connections. Figure 3.2 compares the total load transfer and execution times for the above transfer vector algorithms on varying numbers of processors. (diff-1 denotes the first-order diffusion algorithm, Program 3.1, and diff-2 is the adaptive-timestepping diffusion algorithm, Program 3.3.) In this case, a randomly chosen computer contained all of the load in the system, and the transfer vector algorithms improved the efficiency to at least 99 percent. This

diffuse(...)
  $\delta := \alpha$
  $\Delta L_{i,j} := 0$ for all $j \in N_i$
  while $\mathit{eff} < \mathit{eff}_{\min}$ do
    send L_i to all $j \in N_i$
    receive L_j from all $j \in N_i$
    $L_i^{(0)} := L_i$
    $L_j^{(0)} := L_j$ from all $j \in N_i$
    do
      $\mathcal{A}_i := 1 + \delta \sum_{j \in N_i} \mathcal{D}_{i,j}$
      $\mathcal{T}_{i,j} := \delta \mathcal{D}_{j,i}$
      $\bar{\mathcal{A}}_i := 1 + \frac{\delta}{2} \sum_{j \in N_i} \mathcal{D}_{i,j}$
      $\bar{\mathcal{B}}_i := 1 - \frac{\delta}{2} \sum_{j \in N_i} \mathcal{D}_{i,j}$
      $\bar{\mathcal{T}}_{i,j} := \frac{\delta}{2} \mathcal{D}_{j,i}$
      $n := \left\lceil \frac{\ln \alpha}{\ln \left[\text{globalmax} \left(\mathcal{A}_i^{-1} \sum_{j \in N_i} \mathcal{T}_{i,j} \right) \right]} \right\rceil$
      $\bar{L}_i^{(0)} := \bar{\mathcal{B}}_i L_i + \sum_{j \in N_i} \bar{\mathcal{T}}_{i,j} L_j$
      for $m := 1$ to n do
        send $\langle L_i^{(m-1)}, \bar{L}_i^{(m-1)} \rangle$ to all $j \in N_i$
        receive $\langle L_j^{(m-1)}, \bar{L}_j^{(m-1)} \rangle$ from all $j \in N_i$
        $L_i^{(m)} := \mathcal{A}_i^{-1} \left(L_i^{(0)} + \sum_{j \in N_i} \mathcal{T}_{i,j} L_j^{(m-1)} \right)$
        $\bar{L}_i^{(m)} := \bar{\mathcal{A}}_i^{-1} \left(\bar{L}_i^{(0)} + \sum_{j \in N_i} \bar{\mathcal{T}}_{i,j} \bar{L}_j^{(m-1)} \right)$
      end for
      $err_i := |\bar{L}_i^{(n)} - L_i^{(n)}|$
      $err_{\max} := \text{globalmax}(err_i)$
      if $\frac{err_{\max}}{L_{\max}} > \alpha$ then
        $\delta := \delta(1 - \alpha) \sqrt{\alpha \frac{L_{\max}}{err_{\max}}}$
      end if
    while $\frac{err_{\max}}{L_{\max}} > \alpha$
    $\Delta L_{(i,j)} := \Delta L_{(i,j)} + \bar{\mathcal{A}}_i^{-1} \bar{\mathcal{T}}_{i,j} \left(\frac{\mathcal{D}_{i,j}}{\mathcal{D}_{j,i}} \bar{L}_i^{(0)} - \bar{L}_j^{(n-1)} \right) + \bar{\mathcal{T}}_{i,j} (L_i - L_j)$ for all $j \in N_i$
    $L_i := \bar{L}_i^{(n)}$
    if $\frac{err_{\max}}{L_{\max}} < \alpha$ then
      $\delta := \delta(1 - \alpha) \sqrt{\alpha \frac{L_{\max}}{err_{\max}}}$
    end if
  end while
end diffuse

Program 3.3: Generalized adaptive timestepping diffusion algorithm for computer i.

scenario was intended to illustrate the worst-case behavior of the algorithms and is the case for which much analysis of the algorithms has been done. As that figure shows, with the exception of the HB method, all of the algorithms transferred a fairly judicious quantity of load. The diffusion and AGDE algorithms transferred the least load, with the DHB and OGDE algorithms transferring up to 30 and 12 percent more load, respectively. In this case, the AGDE algorithm seems to be the best bet, transferring the same load quantity as the diffusion algorithms and doing so at least ten times faster. It is on such basis that the GDE algorithm has been considered superior to diffusion [59].

In Figure 3.3, the same quantities are compared, except that the load assignments were uniformly distributed between 0.8 and 1.2. The goal here was to illustrate the algorithms' performance characteristics in a more realistic situation—in particular, that of *balance maintenance.* This experiment tells a somewhat different story than the point disturbance scenario. In this case the diffusion algorithms transferred the least load. Specifically, the other algorithms transferred up to 127 percent more load in the case of the HB method, 80 percent more for the DHB technique, 32 percent more for the AGDE and 60 percent more for the OGDE. As the number of processors grew, however, the speed advantage of the non-diffusive algorithms was much less apparent than in the point disturbance scenario. Given that the transfer of tasks can be quite costly in applications involving gigabytes of data, the small performance advantage (at most 14 milliseconds in this case) offered by the non-diffusive algorithms is of questionable value.

A few other important points to note are these: Although the OGDE algorithm was somewhat faster that the AGDE algorithm, as its proponents in [59] have shown, it transferred around 20 percent more load in the above test cases. Also, despite the speed of the HB algorithm, which was the primary consideration in [19], the algorithm transfers an extraordinary load volume in order to achieve load balance, as was also illustrated in [57]. There thus appears to be little to recommend it, except perhaps in the case of tree or linear array networks.

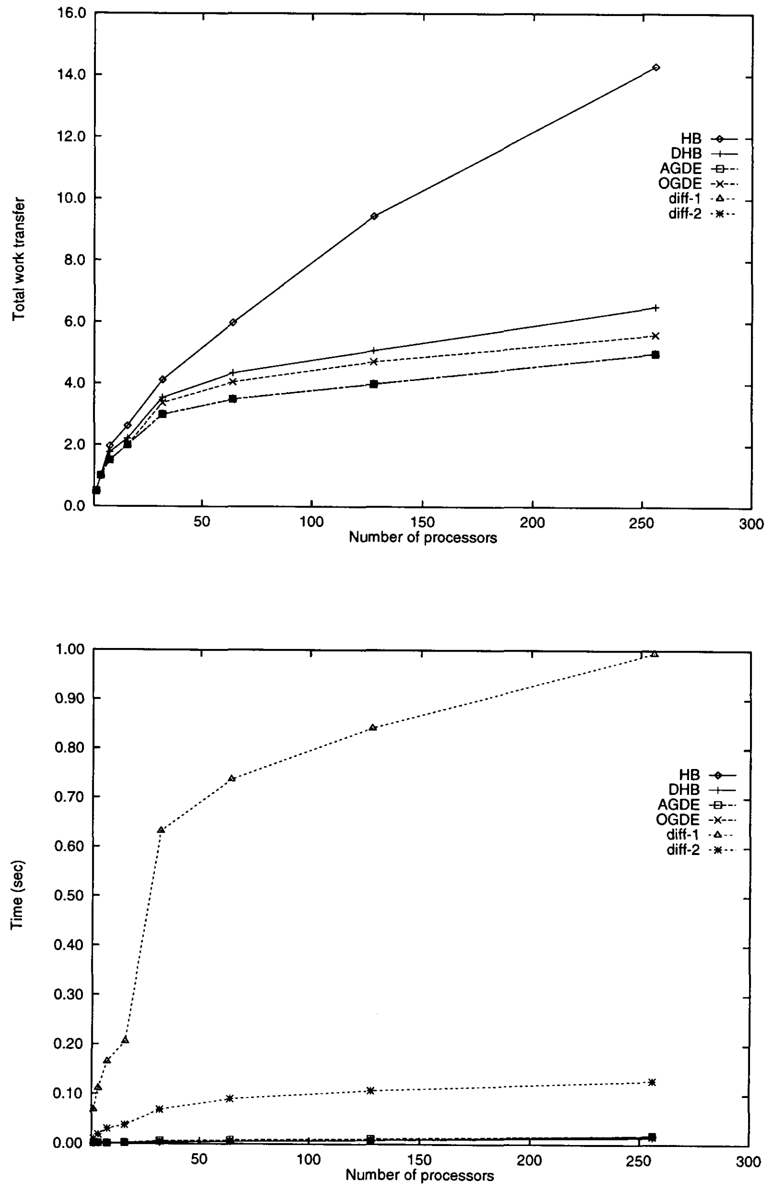


Figure 3.2: Worst-case total load transfer (top) and execution times (bottom) of transfer vector algorithms for varying numbers of processors.

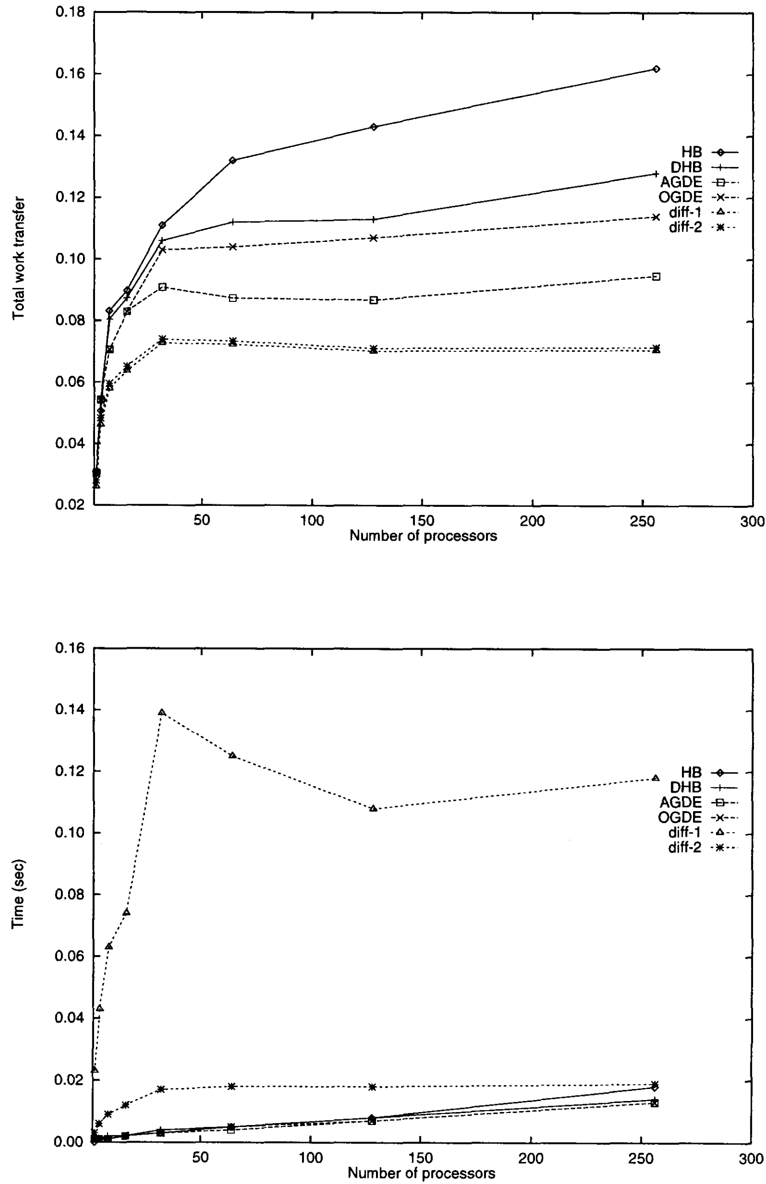


Figure 3.3: Average-case total load transfer (top) and execution times (bottom) of transfer vector algorithms for varying numbers of processors.

3.4 Related Work

An implicit scheme for diffusion was presented in [16]. That algorithm was specifically designed for a three-dimensional torus. With minor modifications, the algorithm there can be adapted to arbitrary graphs, as was shown in Program 3.1, and with more substantial changes, its performance can be significantly improved.

A diffusion model model similar to that in (3.1) was presented in [11]. That work modeled diffusion by the equation

$$\frac{\mathrm{d}L_i}{\mathrm{d}t} = \sum_{j \in N_i} \alpha_{i,j} \left(L_j - L_i \right) \tag{3.35}$$

(Actually, the model was expressed in a completely explicit form

$$L_i^{(t+1)} - L_i^{(t)} = \sum_{j \in N_i} \alpha_{i,j} \left(L_j^{(t)} - L_i^{(t)} \right) \tag{3.36}$$

However, (3.35) is more appropriate, since it does not imply a particular differencing scheme.) Thus, this method has fewer degrees of freedom than (3.1), since each neighbor difference is weighted with a single coefficient. While that limitation is unimportant for homogeneous systems, it presents problems for the heterogeneous case considered in Chapter 7. The diffusion algorithm based on (3.36) was found to be inferior to the dimensional exchange (DE) on hypercubes, due to the latter's faster convergence.

The authors of [59] give a lengthy comparison of the explicit diffusion and GDE algorithms. Their model of diffusion is essentially the same as that in (3.35), except that they make the simplifying assumption that $\alpha_{i,j}$ is the same for all i and for all $j \in N_i$. Based on comparisons of the number of iterations required to reach a globally balanced state and the reduction in load variance as a function of the number of iterations, they conclude that the GDE algorithm is superior to diffusion.

In [57], the authors compare an explicit diffusion-like technique to the HB, DE and gradient methods. They conclude that diffusion is superior since it transfers less work than the HB and gradient methods and is more general than the DE technique,

in that it applies to arbitrary graphs rather than simply hypercubes. They also prefer the locality and potential asynchrony of diffusion to the global approach used the DE and HB methods.

3.5 Summary

This chapter presents an improved diffusion algorithm that converges faster than previous algorithms, is applicable to arbitrary graph networks, and has generalized coefficients for load transfer between pairs of adjacent computers. That algorithm compares very favorably to other techniques, requiring significantly lower transfers of load to achieve a balanced state. The effectiveness of diffusion demonstrated here is contrary to that in other analyses. That discrepancy is due both to improvements in the diffusion technique used here, as well as the more realistic metrics of comparison that were applied.

Chapter 4 Cost-Driven Task Selection

In many applications, task selection cannot be conducted without accounting for the transfer costs of the tasks involved. Dynamic load balancing algorithms that fail to consider costs in their task relocation decisions may have a detrimental effect on an application. For example, if tasks are moved arbitrarily, *communication overhead* may dramatically increase, exceeding the run time improvement due to better load distribution. Also, if tasks are disparate in the sizes of their states, ignoring that fact in task selection may result in significantly higher task migration times than are necessary. To avoid such difficulties, one must consider task transfer costs in the load balancing process.

This chapter presents a method for reducing the cost of task transfers. Whereas the task selection technique in Chapter 2 sought to establish whether a particular transfer was possible, this method also determines the cost at which a transfer can be achieved. The algorithm is then shown to limit the degree to which load balancing adversely affects an application's communication structure. Furthermore, experiments demonstrate that the techniques can even reduce existing communication costs by moving tasks closer to those tasks with which they communicate. Other experiments illustrate the reduction in tasks migration time that results from consideration of the number and size of tasks involved in a transfer.

4.1 Cost-Driven Algorithm

As mentioned in Chapter 2, the task selection problem is weakly **NP**-complete, since it is simply the subset sum problem. An approximation algorithm was presented in Section 2.1.4 which solved the selection problem to a specified non-zero accuracy in polynomial time. That algorithm failed to account for transfer costs, however. In general, one would like to associate a cost with the transfer of a given set of tasks

and find the lowest cost set for a particular desired transfer. This problem can be attacked by considering a problem related to the subset sum problem, namely the 0-1 knapsack problem. In the latter problem, one has a knapsack with a maximum weight capacity W and a set of n items with weights w_i and values v_i, respectively. One seeks to find the maximum-value subset of items whose total weight does not exceed W. In the context of task selection, one has a set of tasks each with loads l_i and transfer costs c_i. It is important to note that l_i can be negative if task i is being considered for transfer in the direction opposite to the ideal transfer quantity. For example, consider a case where ΔL units of load need to be transferred from one computer to another. The task loads of the first computer would be positive, since their inclusion in the transfer set contributes to achieving the ideal transfer quantity. The task loads of the second computer would be negative, since their transfer to the first computer would actually increase the load difference between the computers. Similarly, the cost c_i can be negative if it is actually advantageous to transfer task i between two computers, as it would be, for example, when the transfer of that task improves communication locality.

For a given transfer, $\Delta L_{(i,j)}$, one wishes to find the minimum-cost set of tasks whose exchange achieves that transfer. One can specify a cost function, $C(\Delta L, m)$, which is the minimum cost of a subset of tasks 0 through $m-1$ that achieves a net transfer of ΔL. Letting $C(0,0)$ be zero, and $C(\Delta L, 0)$ be ∞ for $\Delta L \neq 0$, one can find the values of $C(\Delta L, m)$ by computing, in order of increasing m, the following:

$$C(\Delta L, m+1) = \min\{C(\Delta L, m), C(\Delta L - l_{m+1}, m) + c_{m+1}\}$$

Put simply, the iteration establishes whether or not including task i reduces the cost for each possible transfer ΔL. In the end, the lowest cost for a transfer ΔL is given by $C(\Delta L, m)$. As was the case in Section 2.1.4, this algorithm is pseudopolynomial; the run time is $O(n^2 l_{\max})$, where $l_{\max}$ is the largest absolute value of any l_i. This difficulty is circumvented by approximating the values l_i. If the lower b bits of each l_i are truncated, where $b = \left\lceil \log \frac{\epsilon l_{\max}}{n} \right\rceil$, the relative deviation from the optimal load

transfer is at most $\epsilon = \frac{n2^b}{l_{\max}}$. The proof of this follows in same manner as the proof given for the subset sum approximation algorithm in Section 2.1.4. The run time is thereby reduced to $O(\frac{n^2 l_{\max}}{2^b}) = O(\frac{n^3}{\epsilon})$, which is a polynomial for any given non-zero ϵ.

Now that function $C(\Delta L, n)$ has been calculated, the question becomes which transfer to use. The value of $C(\Delta L, n)$ which is finite and for which ΔL is closest to $\Delta L_{(i,j)}$, without exceeding it, is lowest cost of a transfer within ϵ of the transfer actually closest to $\Delta L_{(i,j)}$ (i.e., the transfer that would have been found by an exact search). One might be tempted to take the subset that yielded that value. However, by using a subset that is somewhat further away from $\Delta L_{(i,j)}$, one can potentially achieve a much lower cost. A rigorous approach for this is as follows: Given a target accuracy ϵ, define $\epsilon' = 1 - \sqrt{1-\epsilon}$. If the above approximation algorithm is performed to accuracy ϵ', the result, $\Delta L'$, is lowest cost of the transfer closest, within an accuracy of ϵ', to $\Delta L_{(i,j)}$. Taking the subset with the lowest cost that is within ϵ' of $\Delta L'$ transfer gives the lowest cost subset that is within $\epsilon'^2 = \epsilon$ of the transfer actually nearest to $\Delta L_{(i,j)}$. ϵ itself is determined as specified in Section 2.1.4.

4.2 Results

The cost-driven selection algorithm described above was incorporated into the load balancing framework from Chapter 2 and applied to both real and synthetic applications. In those tests, both the maintenance and improvement of communication locality are demonstrated, as well as the reduction of task transfer costs.

4.2.1 Comparison for Communication Cost Metrics

A primary concern in the transfer of tasks is that such transfers not disrupt the communication locality of an application. If the communication costs of an application are significantly increased by relocating tasks far from the tasks with which they communicate, it may be better not to load balance.

Under random load conditions, several locality-preserving cost metrics were compared. In the first case, a task's transfer cost was taken to be the change in the distance from the actual location of its data structures to its proposed new location. I.e., the transfer for task i was

$$c_i = dist(M_{\text{new}}(i), M_{\text{old}}(i)) - dist(M_{\text{cur}}(i), M_{\text{old}}(i))$$

where *dist* is a function which gives the network distance between any two computers, and M_{old}, M_{cur} and M_{new} are the original task mapping, the current proposed task remapping and the new proposed task remapping, respectively. In short, the cost of a transfer is positive if it increases the distance between the proposed new location of a task and its old location, and the cost is negative if that distance decreases. This cost metric takes nothing into account regarding the location of a task's communicants. So, once a task has moved away from its neighbors, there is no encouragement for it to move back. Thus, one would expect this metric to retard locality degradation but not to prevent it.

Another metric considered was that the cost be the change in a task's distance from its original location when the computation was first started

$$c_i = dist(M_{\text{new}}(i), M_{\text{orig}}(i)) - dist(M_{\text{cur}}(i), M_{\text{orig}}(i))$$

In this case, a task is encouraged to move back to where it began. If the locality was good at the beginning, one would expect this metric to preserve that locality. One would not expect it to improve locality that was poor initially.

The final cost metric used was based on the idea of a *center of communication.* Specifically, for each task, the ideal computer at which to relocate it was determined by finding M_{center} which minimized

$$\sum_{j \in N_i} V_{i,j} dist(M_{\text{center}}(i) - M_{\text{old}}(j))$$

where $V_{i,j}$ is the volume of communication between tasks i and j. In a two-dimensional

mesh, for example, one would calculate the weighted average of the row/column locations of a task's neighbors. The cost of moving a task is then the change in distance from its ideal location.

$$c_i = dist(M_{\text{new}}(i), M_{\text{center}}(i)) - dist(M_{\text{cur}}(i), M_{\text{center}}(i))$$

Of course, a task's neighbors are moving at the same time, so the ideal location is changing somewhat during the selection process. In most cases, however, one would expect the ideal location of a task not to change greatly even if its neighbors move about somewhat. One would expect that this metric would *improve poor locality* as well as maintain existing locality.

The three metrics described above were compared with the zero-cost metric in a synthetic computation. The computation was begun on a 16×16 mesh of Intel Paragon nodes with 10 tasks each. The tasks were connected in a three-dimensional grid, with each task having an average of two neighbors on the local computer and one neighbor on each of the four adjacent computers. Thus, the initial locality was high. After load balancing had brought the efficiency to 90 percent, the task loads were changed so that the efficiency was reduced to around 70 percent, and each task calculated the average distance between itself and its neighbors. Figure 4.1 shows this average distance as a function of the number of load balancing steps. As one can see, locality decays rapidly if no attempt is made to maintain it. The first cost metric slows that decay but does not prevent it. The second and third metrics limit the increase in the average distance metric to factors of 2.1 and 2.6, respectively. Figure 4.1 presents a case in which the locality was poor initially—tasks were assigned to random computers. The third metric was used to improve the locality, and ultimately reduced the average distance between communicating tasks by 79 percent. This is within 23 percent of the locality obtained when the problem was started with high locality. As these figures show, a cost metric can have a tremendous impact on the locality of an application. The metrics used were fairly simple; more complex metrics might yield even better results.

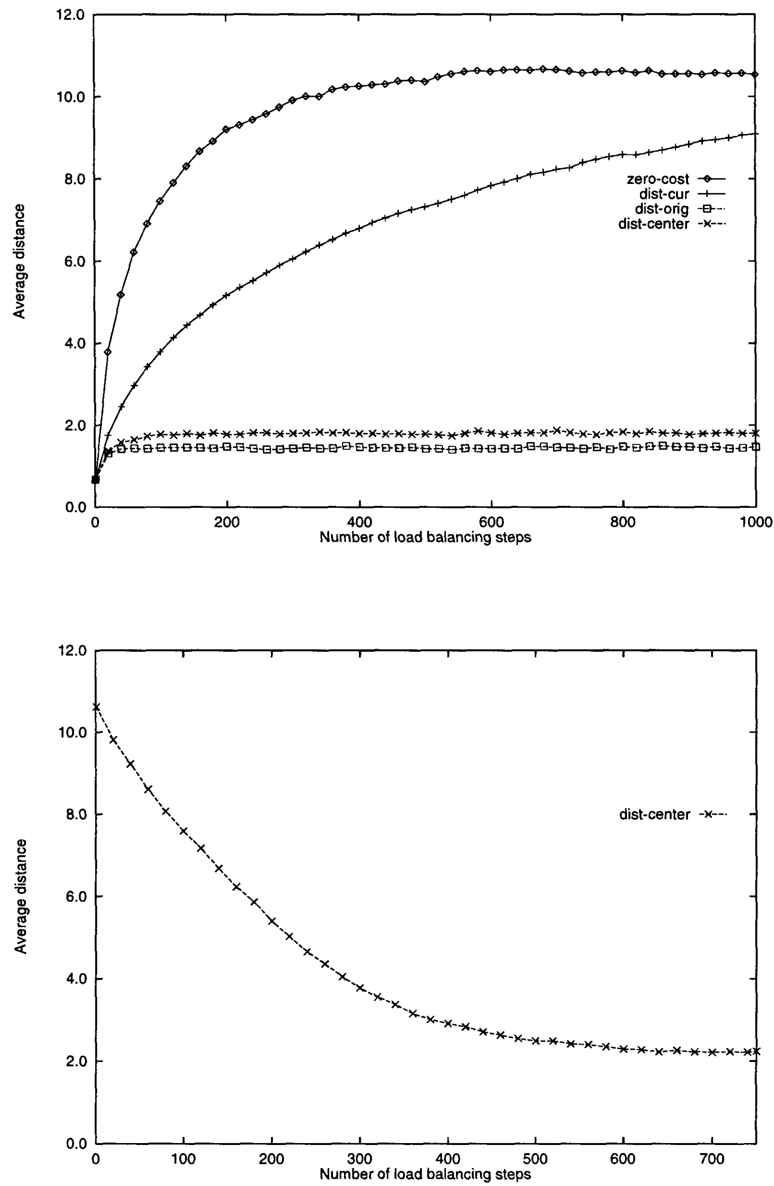


Figure 4.1: Average distance between communicating tasks as a function of load balancing steps for various locality metrics (top) and the improvement of initially poor locality (bottom).

4.2.2 Comparison for Other Cost Metrics

If cost is not used to constrain task movement, a prodigious number of tasks will often be transferred. The following experiments demonstrate that cost functions can dramatically reduce the number or total size of tasks migrated during load balancing.

If task movement is deemed "free," a large number of tasks will often be transferred in order to achieve load balance. For example, in 100 trials of an artificial computation on 256 nodes of an Intel Paragon with 10 tasks per node and a mean efficiency of 70 percent, an average of 638 tasks were transferred to achieve an efficiency of at least 90 percent. Certainly one would not expect that 25 percent of the tasks needed to be transferred for such an improvement. By setting the transfer cost of a task to be one instead of zero, the average number of tasks transferred was reduced to by a factor of four to 160. This is approximately six percent of the tasks in the system.

Reducing the size of the tasks transferred may prove more important than reducing the number of tasks transferred. For example, it may be less expensive to transfer two very small tasks than a single, but much larger one. In an experiment the same as the above where the size of the tasks' data structures were uniformly distributed on the interval between 128 and 512 kilobytes, taking a task's transfer cost to be the size of its data structures reduced the average time to migrate all of the tasks from 8.4 to 3.8 seconds. Similar results were obtained in the simulation of a silicon wafer manufacturing reactor running on a network of 20 workstations. This application was briefly described in Section 2.2. In that case, using unit task transfer cost reduced the transfer time by 50 percent over zero cost, and using the tasks' sizes as the transfer cost reduced the transfer time by 61 percent.

4.3 Related Work

In [44], the authors describe a global optimization approach to task assignment. These techniques consider the costs of placing tasks on the same or different computers. The methods used to optimize this **NP**-hard problem include simulated annealing, tabu

search and a stochastic probe.

Another load balancing technique that addresses communication costs is recursive spectral bisection [4, 47, 48, 58]. In this method, the adjacency of tasks in the communication graph is captured by a matrix. The tasks are partitioned among computers based on the spectral properties of that matrix. While this method performs well, especially for problems involving irregular grids, it is a costly, global approach. The techniques presented here have the advantage of being asynchronous and completely local in nature.

Finally, space-filling curves have been used to map grids to computers in a way that preserves existing spatial locality [34]. In the context of this chapter, the assignment due to the space-filling curve could be used in a manner similar to the M_{center} mapping.

4.4 Summary

This chapter has presented a task selection algorithm that considers the transfer costs of the tasks involved. The method is a variant of an approximation algorithm for the 0-1 knapsack problem. The effectiveness of the technique is demonstrated, both for maintaining and improving the communication locality of an application, as well as for reducing the cost of task migration.

Chapter 5 Vector-based Load Balancing

Dynamic load balancing techniques currently in the literature characterize resource utilization by a single number. While such a representation is adequate for many applications, there are broad classes of computations whose resource needs are not accurately captured by a scalar. Examples of such applications include those that are comprised of multiple, rapidly alternating computational phases, each with different load distribution properties. In such computations, an equal combined load does not necessarily imply equal loads for the individual phases; the efficiency may remain low even though the computation is "balanced." A simple example illustrates the problem: Consider a two-phase computation running on only two computers, as shown in Figure 5.1. The first phase of computation takes 10 seconds on computer 1 and 20 seconds on computer 2. The second phase of computation requires 20 seconds on computer 1 and 10 seconds on computer 2. If resource usage (time) is measured as a scalar, it appears that each computer is being used for 30 total seconds; thus, the computation seems efficient. If a synchronization point exists between the phases, however, then the computation is not actually balanced. Computer 1 must wait 10 seconds for computer 2 during the first phase, and computer 2 must wait 10 seconds for computer 1 during the second phase. Hence, the computation is only 75 percent efficient. Because the phases are brief, tasks cannot necessarily migrate whenever the phases alternate; the cost of doing so would exceed the benefits of a better load distribution. Instead, tasks must be remapped in such a way that both phases are equal.

Another class of application that is poorly served by the scalar load view includes those with disparate computation and memory requirements. Migrating tasks to reduce the variance in the computation requirements at each computer may increase the variance in memory usage, causing problems in memory-constrained environments. Other applications not addressed by the scalar load view are those with rapidly evolv-

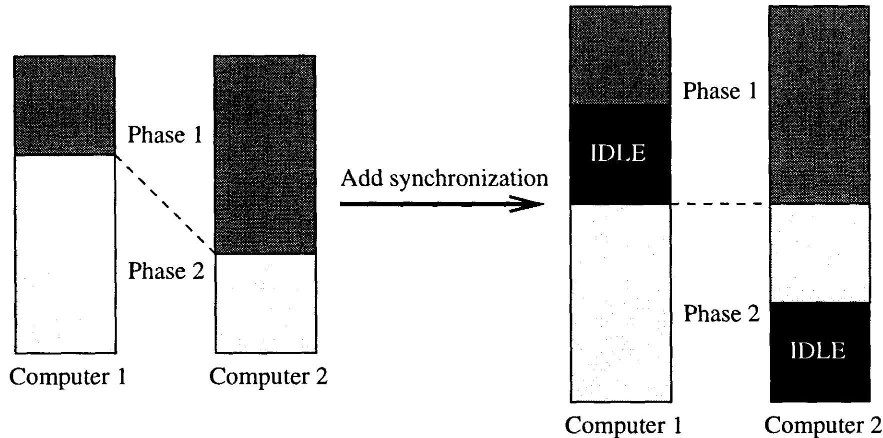


Figure 5.1: Example of low efficiency in a "balanced" system.

ing, but predictable loads. Balancing the current load will not, in general, guarantee a balance in the future (and vice-versa). For these classes of applications, a more comprehensive characterization of resource usage is required.

One solution is to view the load of a computer as a *vector,* rather than a scalar, where each component of the vector captures a different aspect of a computer's load. A load balancing scheme using such an approach would jointly redistribute all of the load components. For example, in the case of a multiphase computation, if the resource usage at a particular computer is characterized by a vector, where each vector component represents the computation time of one of the phases, equality of the vector across the computers would guarantee a uniform load distribution for the individual phases.

This chapter presents the vector dynamic load balancing technique as a refinement to methods presented in previous chapters. The vector-based technique is applied to each of the three classes of applications outlined above, providing performance results on real and synthetic applications that demonstrate the improvements made over traditional techniques. These results confirm that the vector-based dynamic load balancing offers a dramatic advantage for certain types of applications.

5.1 Algorithmic Modifications

Incorporation of the vector methodology primary involves replacing scalar load variables with vector load variables. In some instances, however, more extensive changes are required. Below are the modifications necessary to each of the phases described in Chapter 2.

5.1.1 Load Evaluation

Under the scalar model, the computational requirements of a task could be determined by using standard operating system routines to measure its resource usage, by having the application programmer provide an estimate of a task's load based on algorithmic considerations, or by a combination of the system- and programmer-provided information. While such facilities still prove useful for the vector model, in many situations, the application developer must explicitly determine what the components of the load vector should be. Section 5.2 gives the specifics of load measurement for the three classes of applications presented there.

Whatever the method for determining the loads of the tasks mapped to a computer, in the end, the task loads are summed to yield the computer's total load

$$\vec{L}_i = \sum_{j \in T_i} \vec{l}_j \tag{5.1}$$

where $\vec{L}_i$ is the total load vector at computer i, T_i is the set of tasks mapped to that computer, and $\vec{l}_j$ is the load vector associated with task j.

5.1.2 Profitability Determination

Once the total load has been determined at each computer, the computers must communicate to detect the presence of a load imbalance. In the scalar case, the efficiency is given by

$$eff = \frac{\frac{1}{P}\sum_{i=0}^{P-1} L_i}{\max_{i=0}^{P-1} L_i} = \frac{L_{\text{avg}}}{L_{\max}} \tag{5.2}$$

where P is the number of computers. In the vector case, each component of the efficiency is given by

$$\vec{eff}_k = \frac{\frac{1}{P}\sum_{i=0}^{P-1}\vec{L}_{i,k}}{\max_{i=0}^{P-1}\vec{L}_{i,k}} = \frac{\vec{L}_{\text{avg},k}}{\vec{L}_{\max,k}} \tag{5.3}$$

Once the efficiency has been calculated, the next step is to determine if the efficiency improvement possible by load balancing merits the cost. In cases where the run time is inversely proportional to the efficiency, a useful criterion on which to remap tasks is

$$(eff_{\text{cur}} < eff_{\text{min}}) \bigwedge \left(\left(1 - \frac{eff_{\text{cur}}}{eff_{\text{new}}}\right) T_{\text{step}} > T_{\text{bal}} \right) \tag{5.4}$$

where eff_{cur}, eff_{min}, eff_{new} are the current efficiency, requested efficiency and estimated post-load balancing efficiency, respectively, and T_{step} and T_{bal} are the time between load balancing opportunities and time required by load balancing, respectively [57]. This means that load balancing will be undertaken when the efficiency is below the user threshold, and when the expected reduction in run time, which is proportional to $1 - \frac{eff_{\text{cur}}}{eff_{\text{new}}}$, exceeds the cost of load balancing. (eff_{new} and T_{bal} can initially be taken as equal to eff_{min} and zero, respectively, until they are determined by an actual attempt at task remapping.) The problem with applying this criterion to the vector case is that the run time may not be inversely proportional to the efficiency. For example, in the case where one of the efficiency vector components represents the distribution of memory usage, an improvement in that aspect of the efficiency may or may not improve run time. Moreover, even when run time is inversely proportional to the efficiency, the decrease in run time between a single pair of load balancing opportunities may not merit the cost of task reassignment. A more useful criterion is the following

$$\left(\bigvee_k \vec{eff}_{\text{cur},k} < \vec{eff}_{\text{min},k} \right) \wedge \frac{T_{\text{epoch}}}{T_{\text{epoch}} + T_{\text{bal}}} < eff_{\max} \tag{5.5}$$

where T_{epoch} is the time since the last load balancing attempt (or since the computation began) and $eff_{\max}$ is the maximum *time-based* efficiency desired by the user, taking into account that some time must be spent in relocating tasks. In this case,

load balancing is undertaken if the efficiency is less than the user-specified limit for any of the load components and if the maximum time efficiency would still be possible given the cost of load balancing. For example, if the user wants a time-based efficiency between 90 and 95 percent, the time-based components of $\overrightarrow{eff}_{\min}$ would be 0.90, and $\overrightarrow{eff}_{\max}$ would be 0.95. The remaining components of $\overrightarrow{eff}_{\min}$ could characterize the balance in the use of memory or other resources.

5.1.3 Load Transfer Calculation

If load balancing is deemed profitable, the computers next calculate the ideal load transfers. In Section 2.1.3, three algorithms, the hierarchical balancing (HB) method, the generalized dimensional exchange (GDE) and diffusion, were presented. For each of these algorithms, the vector modifications generally entail replacing all of the scalar load quantities in the calculation with vector values.

Hierarchical Balancing Method

In the HB method, the computers are divided into two subsets, and work is transferred between the subsets so as to make their relative loads equal. These subsets are then recursively subdivided and balanced. Extended to the vector case, the transfer between two subsets of computers becomes

$$\Delta\vec{L}_{(1,2)} = \frac{P_2\vec{L}_1 - P_1\vec{L}_2}{P_1 + P_2} \tag{5.6}$$

where $\vec{L}_1$ and $\vec{L}_2$ are the total loads of the two subsets of computers being balanced, and P_1 and P_2 are the numbers of computers in the respective subsets.

Generalized Dimensional Exchange

The GDE method uses an iterative approach to calculate the load transfers. The network links between adjacent computers are minimally colored so that no computer has two links of the same color. For each color, the computers at the ends of a link

of that color exchange load equal to some fraction, λ, of their load difference. The vector GDE iteration is

$$\Delta\vec{L}_{(i,j)}^{(t+1)} = \Delta\vec{L}_{(i,j)}^{(t)} + \lambda(\vec{L}_i^{(t)} - \vec{L}_j^{(t)}) \text{ for each } j \in N_i$$

where

$$\Delta\vec{L}_{(i,j)}^{(0)} = \vec{0} \text{ for each } j \in N_i$$

Diffusion

Like the GDE method, diffusion is an iterative approach to determining the amount of load to be transferred. Program 5.1 is the vector version of the first-order, implicit diffusion algorithm from Chapter 3. The only subtle change is that the number of Jacobi iterations required may be different for each vector component. Thus, the number of iterations actually performed is the maximum required by any component. Also, as in the scalar case, dynamic variation of the timestep size can dramatically improve the performance of this algorithm. The adaptive timestepping version is given as Programs 5.2 and 5.3. The only additional complications arise from the adjustment of the timestep size. In this case, the timestep becomes a vector, each component of which may be increased or decreased independently.

5.1.4 Task Selection

If load is measured as a scalar, task selection may consider either one-way transfers of tasks or two-way exchanges. (The latter potentially allows satisfaction of small transfer quantities by exchanging two subsets of tasks with roughly the same total load.) The vector model, however, actually necessitates bidirectional exchanges since components of the load transfer may in general occur in different directions. For example, in Figure 5.1, load must be transferred in both directions to achieve load balance.

In Chapters 2 and 4, two techniques for selecting tasks were presented. The former technique simply sought to determine the exchange of tasks closest to the

diffuse(...)
  $\vec{\mathcal{A}}_{i,k} := 1 + \vec{\alpha}_k \sum_{j \in N_i} \vec{\mathcal{D}}_{i,j,k}$ for all k
  $\vec{\mathcal{T}}_{i,j,k} := \vec{\alpha}_k \vec{\mathcal{D}}_{j,i,k}$ for all k
  $\vec{n}_k := \left\lceil \frac{\ln \vec{\alpha}_k}{\ln \left[\text{globalmax}\left(\vec{\mathcal{A}}_{i,k}^{-1} \sum_{j \in N_i} \vec{\mathcal{T}}_{i,j,k}\right)\right]} \right\rceil$ for all k
  $n_{\max} := \max_k \vec{n}_k$
  $\Delta\vec{L}_{(i,j)} := \vec{0}$ for all $j \in N_i$
  while $\bigvee_k \overrightarrow{\mathit{eff}}_k < \overrightarrow{\mathit{eff}}_{\min,k}$ do
    $\vec{L}_i^{(0)} := \vec{L}_i$
    for $m := 1$ to $n_{\max}$ do
      send $\vec{L}_i^{(m-1)}$ to all $j \in N_i$
      receive $\vec{L}_j^{(m-1)}$ from all $j \in N_i$
      $\vec{L}_{i,k}^{(m)} := \vec{\mathcal{A}}_{i,k}^{-1} \left(\vec{L}_{i,k}^{(0)} + \sum_{j \in N_i} \vec{\mathcal{T}}_{i,j,k} \vec{L}_{j,k}^{(m-1)}\right)$ for all k
    end for
    $\Delta\vec{L}_{(i,j),k} := \Delta\vec{L}_{(i,j),k} + \vec{\mathcal{A}}_{i,k}^{-1} \vec{\mathcal{T}}_{i,j,k} \left(\frac{\vec{\mathcal{D}}_{i,j,k}}{\vec{\mathcal{D}}_{j,i,k}} \vec{L}_{i,k}^{(0)} - \vec{L}_{j,k}^{(n_{\max}-1)}\right)$
      for all $j \in N_i$ and for all k
    $\vec{L}_i := \vec{L}_i^{(n_{\max})}$
  end while
end diffuse

Program 5.1: Vector first-order implicit diffusion algorithm for computer i.

desired load transfer; the latter also incorporated the costs of task transfers. Both of those algorithms can be adapted for vector loads.

The task selection algorithm from Section 2.1.4 used dynamic programming. Approximation of the task load values made the algorithm polynomial. In the vector case, for a given transfer, $\Delta\vec{L}_{(i,j)}$, the goal is to find the subset of the n total tasks on computers i and j whose exchange would result in the net transfer of load closest to $\Delta\vec{L}_{(i,j)}$, without exceeding any component of it. As before, a function, $F(\Delta\vec{L}, m)$, is specified, which is true if a net transfer of $\Delta\vec{L}$ is possible by the exchange of a subset of tasks 0 through $m-1$. $F(\vec{0}, 0)$ is true, and $F(\Delta\vec{L}, 0)$ is false for all $\Delta\vec{L} \neq \vec{0}$. The

diffuse(...)
    $\vec{\delta} := \vec{\alpha}$
    $\Delta\vec{L}_{i,j} := \vec{0}$ for all $j \in N_i$
    while $\mathit{eff} < \mathit{eff}_{\min}$ do
        send $\vec{L}_i$ to all $j \in N_i$
        receive $\vec{L}_j$ from all $j \in N_i$
        $\vec{L}_i^{(0)} := \vec{L}_i$
        $\vec{L}_j^{(0)} := \vec{L}_j$ from all $j \in N_i$
        do
            $\vec{\mathcal{A}}_{i,k} := 1 + \vec{\delta}_k \sum_{j \in N_i} \vec{\mathcal{D}}_{i,j,k}$ for all k
            $\vec{\mathcal{T}}_{i,j,k} := \vec{\delta}_k \vec{\mathcal{D}}_{j,i,k}$ for all k
            $\vec{\vec{\mathcal{A}}}_{i,k} := 1 + \frac{\vec{\delta}_k}{2} \sum_{j \in N_i} \vec{\mathcal{D}}_{i,j,k}$ for all k
            $\vec{\vec{\mathcal{B}}}_{i,k} := 1 - \frac{\vec{\delta}_k}{2} \sum_{j \in N_i} \vec{\mathcal{D}}_{i,j,k}$ for all k
            $\vec{\vec{\mathcal{T}}}_{i,j,k} := \frac{\vec{\delta}_k}{2} \vec{\mathcal{D}}_{j,i,k}$ for all k
            $\vec{n}_k := \left\lceil \frac{\ln \vec{\alpha}_k}{\ln \left[\text{globalmax}\left(\vec{\mathcal{A}}_{i,k}^{-1} \sum_{j \in N_i} \vec{\mathcal{T}}_{i,j,k}\right)\right]} \right\rceil$ for all k
            $n_{\max} := \max_k \vec{n}_k$
            $\vec{\vec{L}}_{i,k}^{(0)} := \vec{\vec{\mathcal{B}}}_{i,k} \vec{L}_{i,k} + \sum_{j \in N_i} \vec{\vec{\mathcal{T}}}_{i,j,k} \vec{L}_{j,k}$ for all k
            for $m := 1$ to $n_{\max}$ do
                send $\langle \vec{L}_i^{(m-1)}, \vec{\vec{L}}_i^{(m-1)} \rangle$ to all $j \in N_i$
                receive $\langle \vec{L}_j^{(m-1)}, \vec{\vec{L}}_j^{(m-1)} \rangle$ from all $j \in N_i$
                $\vec{L}_{i,k}^{(m)} := \vec{\mathcal{A}}_{i,k}^{-1} \left(\vec{L}_{i,k}^{(0)} + \sum_{j \in N_i} \vec{\mathcal{T}}_{i,j,k} \vec{L}_{j,k}^{(m-1)} \right)$ for all k
                $\vec{\vec{L}}_{i,k}^{(m)} := \vec{\vec{\mathcal{A}}}_{i,k}^{-1} \left(\vec{\vec{L}}_{i,k}^{(0)} + \sum_{j \in N_i} \vec{\vec{\mathcal{T}}}_{i,j,k} \vec{\vec{L}}_{j,k}^{(m-1)} \right)$ for all k
            end for
            $\overrightarrow{\text{err}}_{i,k} := |\vec{\vec{L}}_{i,k}^{(n_{\max})} - \vec{L}_{i,k}^{(n_{\max})}|$ for all k
            $\overrightarrow{\text{err}}_{\max,k} := \text{globalmax}(\overrightarrow{\text{err}}_{i,k})$ for all k
            for all k do
                if $\frac{\overrightarrow{\text{err}}_{\max,k}}{\vec{L}_{\max,k}} > \vec{\alpha}_k$ then
                    $\vec{\delta}_k := \vec{\delta}_k (1 - \vec{\alpha}_k) \sqrt{\vec{\alpha}_k \frac{\vec{L}_{\max,k}}{\overrightarrow{\text{err}}_{\max,k}}}$
                end if
            end for
        while $\bigvee_k \frac{\overrightarrow{\text{err}}_{\max,k}}{\vec{L}_{\max,k}} > \vec{\alpha}_k$

Program 5.2: Vector adaptive-timestepping diffusion algorithm for computer i (part 1).

$$\Delta\vec{L}_{(i,j),k} := \Delta\vec{L}_{(i,j),k} + \vec{\vec{\mathcal{A}}}_{i,k}^{-1}\vec{\vec{\mathcal{T}}}_{i,j,k}\left(\frac{\vec{\mathcal{D}}_{i,j,k}}{\vec{\mathcal{D}}_{j,i,k}}\vec{\vec{L}}_{i,k}^{(0)} - \vec{\vec{L}}_{j,k}^{(n_{\max}-1)}\right) + \vec{\vec{\mathcal{T}}}_{i,j,k}\left(\vec{L}_{i,k} - \vec{L}_{j,k}\right)$$

for all $j \in N_i$ and for all k

$\vec{L}_i := \vec{\vec{L}}_i^{(n_{\max})}$

for all k do

if $\frac{\overrightarrow{\mathrm{err}}_{\max,k}}{\vec{L}_{\max,k}} < \vec{\alpha}_k$ then

$\vec{\delta}_k := \vec{\delta}_k(1 - \vec{\alpha}_k)\sqrt{\vec{\alpha}_k \frac{\vec{L}_{\max,k}}{\overrightarrow{\mathrm{err}}_{\max,k}}}$

end if

end for

end while

end diffuse

Program 5.3: Vector adaptive-timestepping diffusion algorithm for computer i (part 2).

values of $F(\Delta\vec{L}, m)$ are computed in order of increasing m by the following

$$F(\Delta\vec{L}, m+1) = F(\Delta\vec{L}, m) \vee F(\Delta\vec{L} - \vec{l}_{m+1}, m) \tag{5.7}$$

The best transfer is that for which the value of $\Delta\vec{L}$ is closest to $\Delta\vec{L}_{(i,j)}$ and for which $F(\Delta\vec{L}, n)$ is true. The "closest" transfer is that which minimizes

$$\sqrt{\sum_k \left(\frac{\Delta\vec{L}_{(i,j),k} - \Delta\vec{L}_k}{\Delta\vec{L}_{(i,j),k}}\right)^2} \tag{5.8}$$

(Division by $\Delta\vec{L}_{(i,j),k}$ normalizes the distance with respect to the scales of the transfer components.) An example of the algorithm for a length-2 vector is shown in Figure 5.2. In that figure, sets are represented by dots. Initially, the only set is the empty set, which lies at the origin. New subsets are created at each step by adding a task to all existing subsets. This algorithm is pseudopolynomial, however, with a run time proportional to $\mathbf{O}\left(n \prod_k n\vec{l}_{\max,k}\right) = \mathbf{O}\left(n^{d+1} \prod_k \vec{l}_{\max,k}\right)$, where d is the dimension of the vector, and where $\vec{l}_{\max}$ is the component-by-component absolute maximum of all $\vec{l}_m$

(i.e., $\vec{l}_{\max,k}$ is $\max_{m=0}^{n-1} |\vec{l}_{m,k}|$ for all k). This difficulty is overcome by approximating the values $\vec{l}_m$. The lower $\vec{b}_k$ bits in the representation of each $\vec{l}_{m,k}$ are truncated, where $\vec{b}_k = \left\lceil \log \frac{\vec{\epsilon}_k \vec{l}_{\max,k}}{n} \right\rceil$, the relative deviation from the optimal transfer is at most $\vec{\epsilon}_k = \frac{n2^{\vec{b}_k}}{\vec{l}_{\max,k}}$. This is true since

$$\sum_{j \in S} \vec{l}_{j,k} \geq \sum_{j \in S'} \vec{l}_{j,k} \geq \sum_{j \in S'} \vec{l}'_{j,k} \geq \sum_{j \in S} \vec{l}'_{j,k} \geq \sum_{j \in S} (\vec{l}_{j,k} - 2^{\vec{b}_k}) \geq \sum_{j \in S} \vec{l}_{j,k} - n2^{\vec{b}_k} \tag{5.9}$$

holds for all k, where S is the optimal subset using the original values, $\vec{l}_j$, and S' is the optimal subset with the approximated values, $\vec{l}'_j$. The run time of the approximation algorithm is thus $\mathbf{O}\left(n^{d+1} \prod_k \frac{\vec{l}_{\max,k}}{2^{\vec{b}_k}}\right) = \mathbf{O}\left(\frac{n^{2d+1}}{\prod_k \vec{\epsilon}_k}\right)$, a (potentially large) polynomial for any given d and $\vec{\epsilon}$. An example of the approximation algorithm appears in Figure 5.3. New subsets are created at each step only when they fall within an empty rectangle.

As was shown in Chapter 4, the algorithm above can be modified to incorporate the costs of task transfers, such as the effect of moving a particular task on communication locality. Such an algorithm calculates not only the ability to achieve a particular transfer, but also the minimum cost of achieving it. In this case, a cost function, $C(\Delta\vec{L}, m)$, is defined as the minimum cost of a subset of tasks 0 through $m-1$ that achieves a net transfer of $\Delta\vec{L}$. Let $C(\vec{0}, 0)$ be zero, and let $C(\Delta\vec{L}, 0)$ be ∞ for $\Delta\vec{L} \neq \vec{0}$. The values of $C(\Delta\vec{L}, m)$ are computed via

$$C(\Delta\vec{L}, m+1) = \min\{C(\Delta\vec{L}, m), C(\Delta\vec{L} - \vec{l}_{m+1}, m) + c_{m+1}\}$$

The run time of this algorithm in its exact and approximated versions is the same as the cost-invariant algorithm above.

For vector loads, the selection algorithm accuracy, $\vec{\epsilon}$, is determined by the following considerations. A computer generally has a set of outgoing (positive) transfer quantity components and a set of incoming (negative) transfer quantity components. For a particular computer i, denote the sum of the former by $\Delta\vec{L}_i^+$ and the sum of the latter by $\Delta\vec{L}_i^-$. (I.e., let $\Delta\vec{L}_{i,k}^+$ be the sum of positive $\Delta\vec{L}_{(i,j),k}$ for all $j \in N_i$; similarly, $\Delta\vec{L}_{i,k}^+$ is the sum of negative $\Delta\vec{L}_{(i,j),k}$ for all j.) To achieve the desired vector efficiency,

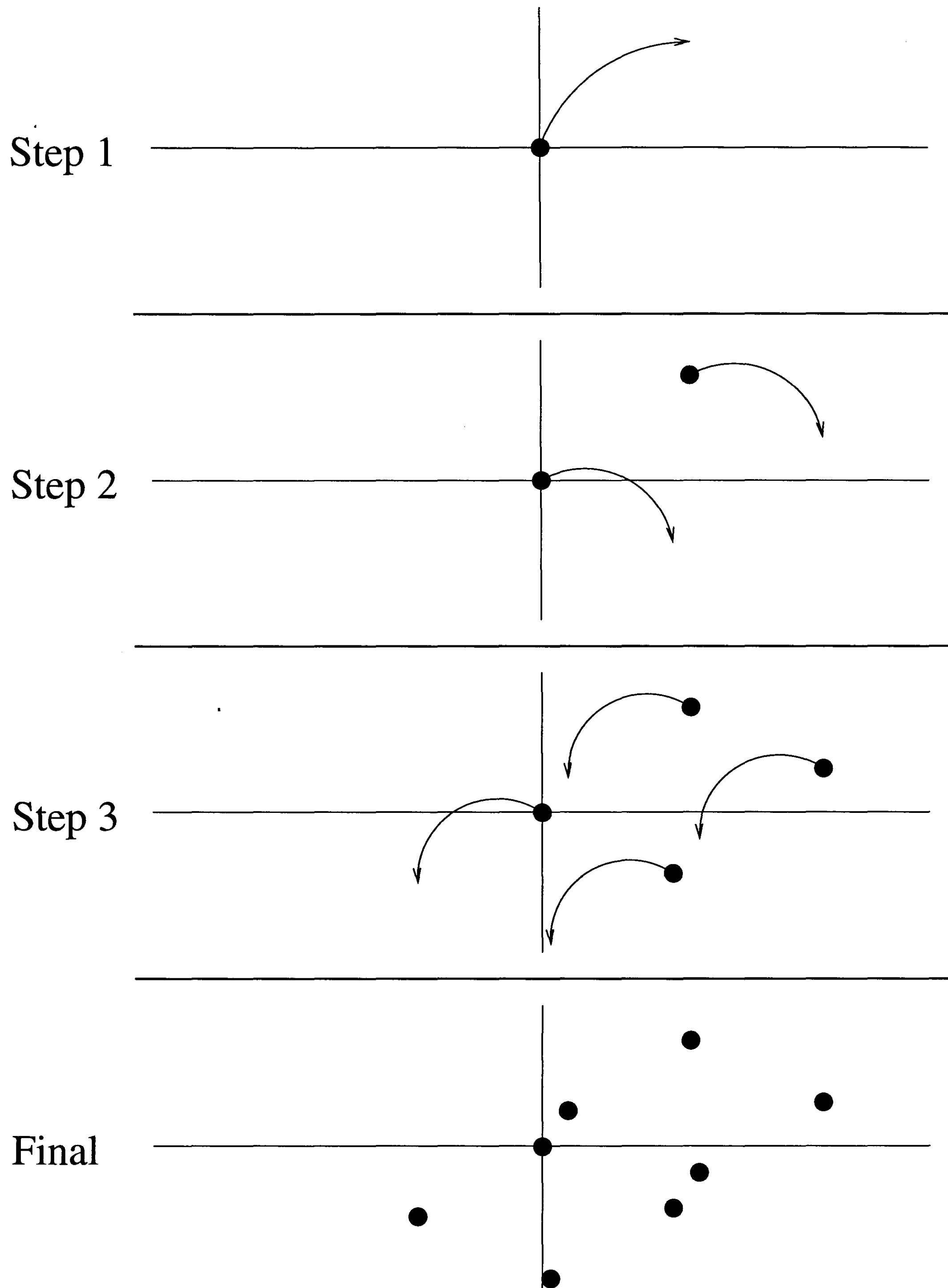


Figure 5.2: Example of pseudopolynomial vector subset sum algorithm.

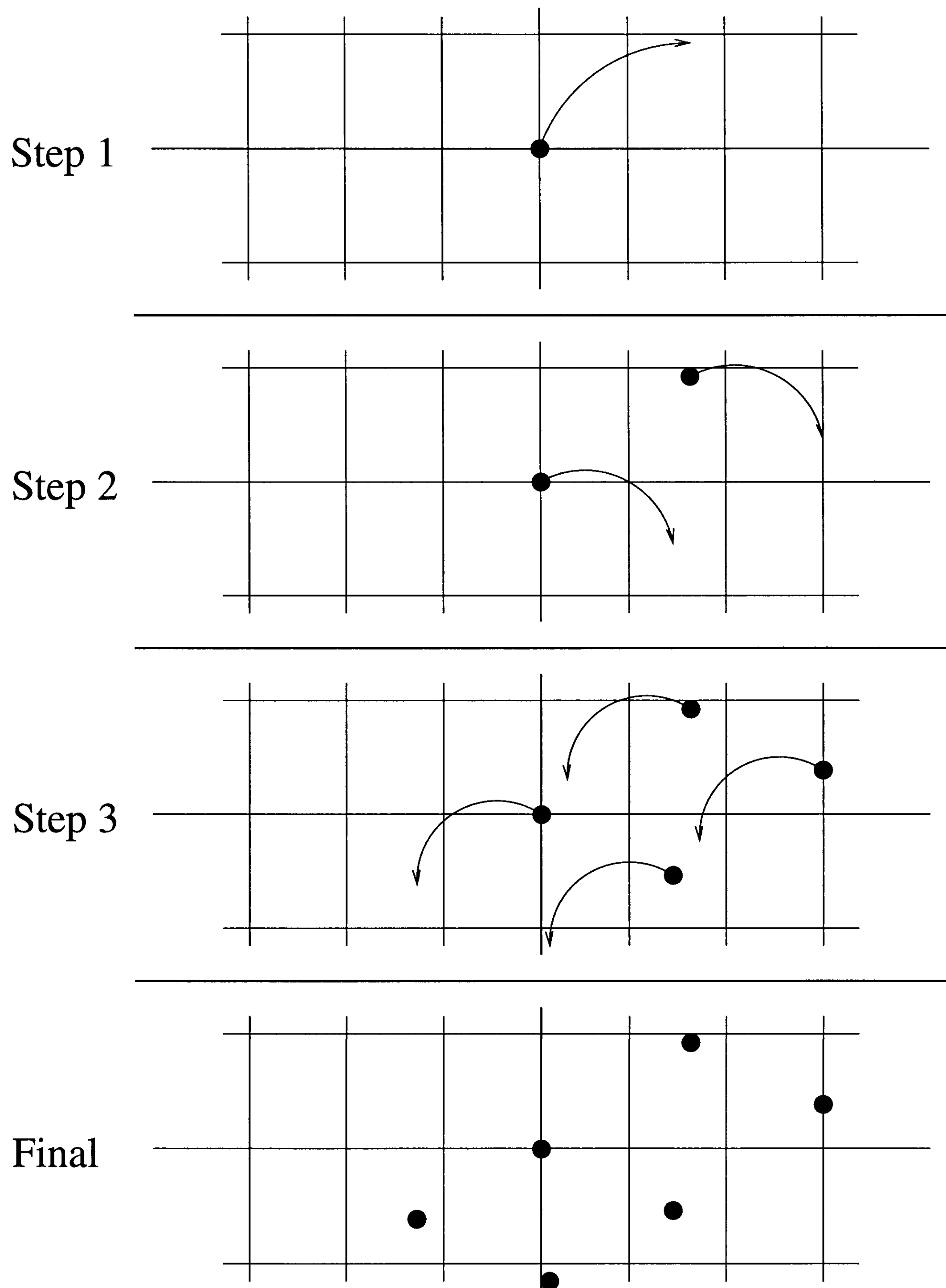


Figure 5.3: Example of fully polynomial vector subset sum algorithm.

$\overrightarrow{eff}_{\min}$, each computer must guarantee $\vec{L}_{i,k} \leq \frac{\vec{L}_{\text{avg},k}}{\overrightarrow{eff}_{\min,k}}$, for all k. Assuming that all of the incoming components of the transfer quantities are satisfied, the new load of computer i will be at least $\vec{L}_i - \Delta\vec{L}_i^+$. Thus, in order to guarantee that its new load components are at most $\frac{\vec{L}_{\text{avg},k}}{\overrightarrow{eff}_{\min,k}}$, for all k, respectively, a computer can leave no more than $\vec{\epsilon}$ of its outgoing transfer vectors unsatisfied, according to

$$\frac{\vec{L}_{\text{avg},k}}{\overrightarrow{eff}_{\min,k}} \geq \vec{L}_{i,k} - \Delta\vec{L}_{i,k}^- - (1 - \vec{\epsilon}_k)\,\Delta\vec{L}_{i,k}^+ \tag{5.10}$$

Solving (5.10) for the maximum such $\vec{\epsilon}$ gives

$$\vec{\epsilon}_{\max,k} = 1 - \frac{1}{\Delta\vec{L}_{i,k}^+}\left(\vec{L}_{i,k} - \Delta\vec{L}_{i,k}^- - \frac{\vec{L}_{\text{avg},k}}{\overrightarrow{eff}_{\min,k}}\right) \tag{5.11}$$

As before, the selection algorithm will not, in general, achieve the desired load transfers in a single attempt. Multiple task selection rounds will typically be required to sufficiently reach the load transfer quantities.

5.1.5 Task Migration

The task migration phase is unaffected by the vector modifications and is conducted as described in Section 2.1.5.

5.2 Results

The vector dynamic load balancing methods described above were implemented in terms of the Scalable Concurrent Programming Library. This library is described in Appendix A. The improved load balancing framework that resulted was in turn applied to the three classes of applications mentioned at the beginning of the chapter.

5.2.1 Applications with Multiple Phases

Dynamic load balancing techniques already in the literature have concentrated entirely on single-phase computations. That is, they work only for applications which are comprised of a single mode of computation between synchronization points. Examples of such applications include Navier-Stokes flow solvers and particle simulations without self-consistent electromagnetic fields. As concurrent simulation techniques become more advanced, however, multiphase computations will appear with increasing frequency. Such applications involve two or more tightly interleaved computational phases separated by synchronization points. Failure to jointly balance each of the phases will potentially result in poor efficiency.

Direct Simulation Monte Carlo Application

The vector dynamic load balancing technique was applied to the direct simulation monte carlo (DSMC) application described in Section 2.2. When the particles being simulated in DSMC are ions, they contribute to and are influenced by an electromagnetic field. Consequentially, an electrostatic field solver has been incorporated into the application. This solver uses a face-based finite element (FEM) technique, taking as its input the charge density in each grid cell and returning the electric field. A preconditioned conjugate gradient method is used to solve the system of equations that results from the FEM. This field solver is described in greater detail in Appendix B.

The spatial decoupling at the core of the DSMC method makes it an ideal candidate for parallelization, since two partitions of grid cells interact only along their boundaries. The same applies to the field solver. The grid for a problem is prepared for parallel simulation by first dividing it into five to ten partitions per computer. Uniform execution time across computers is achieved by dynamically remapping the partitions. For the particle transport phase, dynamic load balancing is necessary because the concentration of particles in a region of the grid changes during the course of the simulation. The grid may be refined in areas of high particle concentration in

```
dsmc_compute(...)
    do
        do
            send field boundary information to neighbors
            receive field boundary information from neighbors
            calculate updated field
            gather/scatter to determine convergence
        while not converged
        move particles
        send away particles that exit current partition
        receive particles from neighboring partitions
        collide particles
        gather/scatter to obtain global statistics
        calculate termination condition based on global statistics
    while not converged
end dsmc_compute
```

Program 5.4: Concurrent DSMC algorithm, with self-consistent fields, for a single partition.

order to preserve the integrity of the DSMC model. This in turn affects the run time of the field solver, which operates on the same grid used in the particle transport phase. The schematic multiphase algorithm is given in Figure 5.4.

The necessity of dynamic load balancing to this application's efficiency was illustrated during a 1.2 million-particle simulation on a 140,000-cell grid of the Gaseous Electronics Conference (GEC) reactor. (See Figure 2.5.) This simulation was run on 128 nodes of an Intel Paragon. Each node had approximately five partitions mapped to it. Without load balancing, the first 150 timesteps required 1,762 seconds, for an efficiency of 31 percent. A scalar view of resource usage, which considered the particle transport and field calculations in aggregate, improved the situation somewhat: real efficiency was improved to 45 percent, and execution time reduced to 1,217 seconds. (83 of those seconds were required by two rounds of task remapping.) Because the load distribution characteristics of the particle transport and field calculation phases were quite different, the scalar approach failed to generate much of an improvement.

load metric	none	scalar	vector
run time (secs)	1,762	1,217	787
efficiency	31	45	70

Table 5.1: Summary of the run time and efficiency for the DSMC application without load balancing and with the scalar and vector views of load.

Although the scalar efficiency as given by Equation 5.2 was high (the library estimated it to be over 75 percent), the actual efficiency, in terms of available computing resources being usefully employed, was much lower. To use the vector load balancing methods described above, timing calls were inserted before and after both the particle transport and field solver phases. The times of these two phases where then passed on to the load balancing routine. As a result of the additional information provided by the time vector, the efficiency improvement was much greater. The run time was reduced to 787 seconds, for an overall efficiency of 70 percent. (98 seconds were required by two instances of load balancing.) These are summarized in Table 5.1.

Particle-in-Cell Application

The advantages offered by the vector load view were also seen in a particle-in-cell (PIC) simulation of ion thruster backflow around a satellite [39, 45]. (See Figure 5.4.) Like the DSMC application, this simulation involved particle transport and field calculation phases. The primary differences between the two applications were that the PIC code simulated collisionless plasmas and used a regular grid on a simpler geometry. (The skeleton algorithm is the same as that in Program 5.4, without the "collide" step.) The satellite grid was divided into 1,575 partitions and mapped onto 256 processors of a Cray T3D. As with the GEC reactor simulation described above, the efficiency was quite low, at 54 percent. Techniques based on scalar load reduced the run time for 100 timesteps of this simulation from 2,374 seconds to 2,014 seconds, for an improved real efficiency of 63 percent. An even greater improvement in efficiency was offered by the vector-based algorithms. The vector approach improved

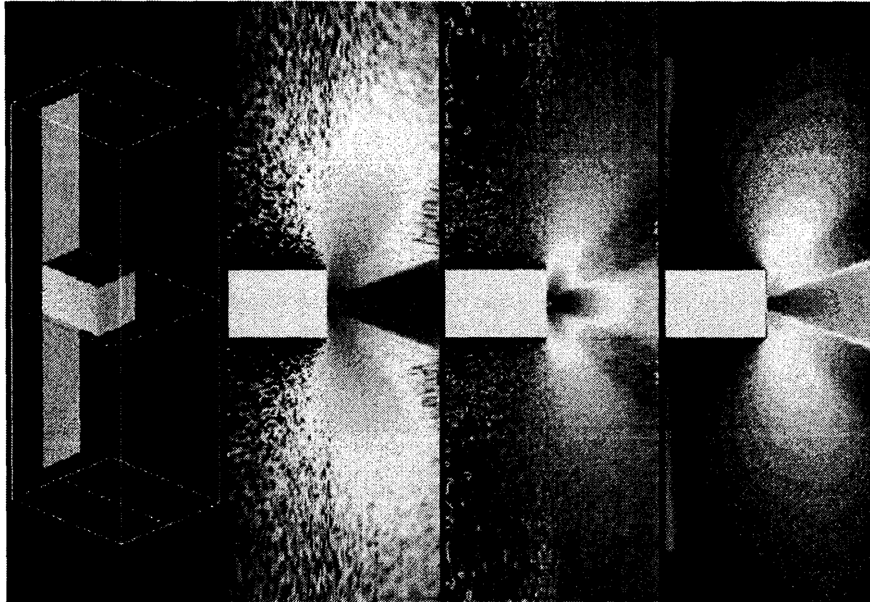


Figure 5.4: ESEX/Argos geometry and cutplanes for ion density, charge-exchange ion density and electric field.

the efficiency to 72 percent, reducing the run time to 1,775 seconds. A detailed breakdown revealed that vector load balancing had improved the field calculation efficiency from 73 percent to 94 percent and the particle transport efficiency from 29 percent to 43 percent. These results are summarized in Table 5.2. A larger improvement in the particle transport efficiency was impossible because the time for that phase in one of the partitions was so high that no matter what computer it was assigned to, that computer was overworked, slowing down the entire computation. This suggests that dynamic granularity control (i.e., the ability to divide and merge tasks during execution) must be coupled with vector load balancing to achieve higher efficiency in such cases.

5.2.2 Applications with Disparate Computation and Memory Requirements

In many applications, computation and memory requirements are highly correlated—run time is proportional to the size of the data structures operated upon. If this is

load metric	none	scalar	vector
run time (secs)	2,374	2,014	1,775
total eff.	54	63	72
field solve eff.	73	78	94
particle trans. eff.	29	41	43

Table 5.2: Summary of the run time and efficiency for the PIC application without dynamic load balancing and with the scalar and vector views of load.

not the case, however, and in fact, the computation and memory needs of tasks are relatively independent of one another, reassigning tasks to reduce the variation in computation time across computers may increase the variation in memory usage. In environments where memory is constrained, this may slow a computation down (due to virtual memory paging) or even cause it to terminate due to memory allocation failure. To circumvent this difficulty one can consider the computational and memory requirements of a task together, as a vector. By balancing the time-memory vectors across computers, one would guarantee that both computation time and memory usage are well-distributed.

A situation in which tasks' computation and memory requirements varied in proportion occurred in a DSMC simulation of neutral flow in the GEC reactor. (I.e., no electromagnetic fields were used.) The grid density throughout the reactor varied according to the complexity of its internal features. However, the particle density was uniform throughout the reactor. Thus, the number of particles in a partition was proportional to the total volume contained by that partition, not the number of cells. Some partitions had a large number of grid cells, but comparatively few particles. As a result, they were memory-intensive, but required little computation time. Other partitions contained more particles but fewer grid cells, making them more compute-intensive and less memory-intensive. When the simulation was run on 12 processors of an Avalon A12, the load balancing algorithm speeded execution as one would expect; specifically, it maintained an average time per step of about 2.0 seconds versus 2.5 seconds per step without load balancing. However, the amount

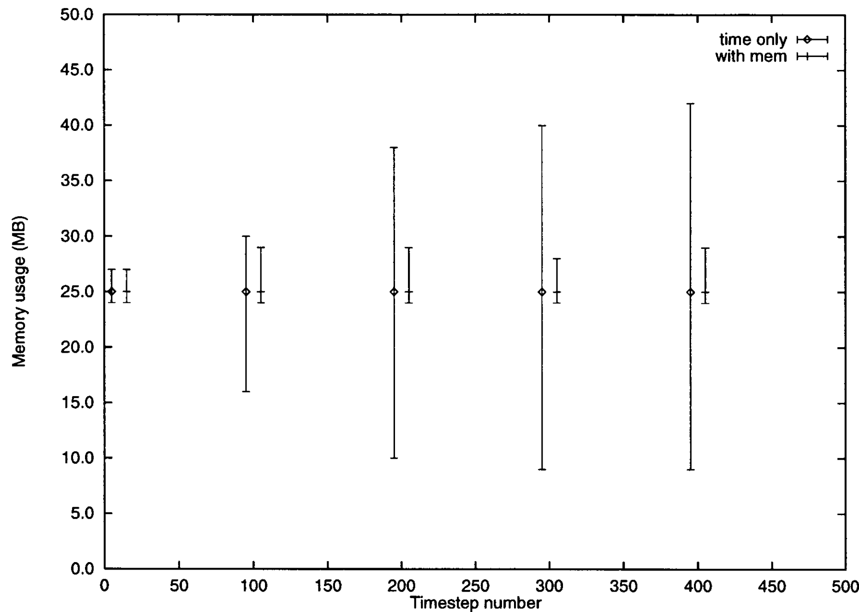


Figure 5.5: Variation in memory usage after several load balancing rounds with a time-only scalar load metric and a time-memory vector load metric.

of memory used on each computer, which was initially quite uniform, varied increasingly with each load balancing attempt. Initially, the memory usage per computer was between 24 and 27 megabytes. After the first round of load balancing, 16 and 30 megabytes were the lower and upper limits. In subsequent rounds, these extrema increased to 10 and 38, 9 and 40, and 9 and 42 megabytes, respectively. While this posed no difficulties on the A12, due to the large amount of memory available at each processor (512 megabytes), it could pose a problem on a network of workstations, where individual memory capacities of 32 to 64 megabytes are more typical. In a second experiment on the A12, a load vector was used that included both computation time and memory. In this case, the time per step was reduced to 2.2 seconds on average. The memory usage remained much more balanced, however, increasing to upper and lower limits of 24 and 29 megabytes, respectively, in the worst case. These memory disparity results are shown in Figure 5.5. So, while the quality of balance in computation achieved with the vector metric was somewhat lower, that balance was achieved with a significantly reduced disparity in memory usage.

5.2.3 Applications with Rapidly Changing, but Predictable Computation Times

In some applications, the computational requirements of tasks may change quickly, but fairly predictably. Under such circumstances, remapping tasks so that the current computation time is nearly equal at each computer will not, in general, guarantee that the computation time will be equal in the future. Similarly, guaranteeing that the computation time will be equal in the future may not result in an equal distribution at present. An initial approach at overcoming this difficulty would be to consider the average future computational requirements of tasks and to reassign them based on that basis. While one would expect this method to offer some advantages, room remains for further improvement. In particular, consider the case where the load of a given task j evolves from step t_0 to step t_1 according to the following linear model

$$l_j(t) = \frac{(t_1 - t)l_j^{(0)} + (t - t_0)l_j^{(1)}}{t_1 - t_0} \tag{5.12}$$

The computation time for a particular computer i at a particular step t is thus

$$L_i(t) = \sum_{j \in T_i} l_j(t) \tag{5.13}$$

Note that since the computational requirements vary linearly, if they are equal everywhere at both steps t_0 and t_1, respectively, then they are equal at all steps in between. To guarantee that both $L_i(t_0)$ and $L_i(t_1)$ are equal everywhere, respectively, one must find an assignment of tasks that jointly makes $\sum_{j \in T_i} l_j^{(0)}$ and $\sum_{j \in T_i} l_j^{(1)}$ equal everywhere, respectively. This is the same situation as was encountered for two-phase computations in Section 5.2.1 above; there are two quantities which must be simultaneously balanced by task relocation. So, let the load vector for a particular task be $\vec{l}_j = \langle l_j^{(0)}, l_j^{(1)} \rangle$.

The vector technique can, in fact, be applied to any situation in which the step times of tasks can be accurately modeled by a polynomial. If the step times vary quadratically, then a 3-vector containing the times at unique steps t_0, $t_{\frac{1}{2}}$, and t_1,

completely characterizes the time at all other steps, since a quadratic function is uniquely specified by values at three different points. In general, if a task's step time is accurately described by an n-degree polynomial, an $(n+1)$-size vector will capture the behavior of that task.

Synthetic Application

The advantages of the vector approach were illustrated in parametric experiment conducted on 256 processors of a Cray T3D. Each computer began with 5 tasks assigned to it. The tasks' initial times per step, $l_j^{(0)}$, were random values uniformly distributed on $(1,4)$. The tasks final step times, $l_j^{(1)}$, were distributed uniformly over the interval $(1,8)$. So, the variance in resource requirements was increasing with each step. As the tasks were originally assigned, the efficiency for 100 steps was 73 percent. Task migration based on the scalars $l_j^{(0)}$ yielded a small improvement to 75 percent. Using instead the scalars $l_j^{(1)}$, for which the times were highest and most varied, improved the efficiency to 81 percent. Using the average time, $\frac{l_j^{(0)}+l_j^{(1)}}{2}$, improved the efficiency even further, to 86 percent. However, by considering the load vector, $\vec{l_j}$, described above, an efficiency of 93 percent was obtained. These results are summarized in Table 5.3.

A second parametric experiment was conducted just as above in which each task had step times at t_0, $t_{\frac{1}{2}}$, and t_1 that were uniformly distributed on $(1,4)$, $(1,6)$ and $(1,8)$, respectively. Between these points, the tasks' loads varied quadratically. Leaving the tasks in place resulted in a base efficiency of 72 percent. Using the starting step times reduced efficiency to 68 percent, and using the ending step times reduced it to 67 percent. Using the average step time, calculated by integrating the polynomial and dividing by $t_1 - t_0$, improved efficiency to 80 percent. Using a 2-vector of only the starting and ending times improved efficiency slightly, to 73 percent. Finally, using the 3-vector that also included the intermediate step time value improved efficiency to 88 percent. These results are also summarized in Figure 5.3.

Of course, in the above situations, it was assumed that a load model existed a priori. In real applications that may not be the case. For applications in which the

load metric	efficiency for linear loads	efficiency for quadratic loads
none	73	72
$l_i^{(0)}$	75	68
$l_i^{(1)}$	81	67
l_i^{avg}	86	80
$\langle l_i^{(0)}, l_i^{(1)} \rangle$	93	73
$\langle l_i^{(0)}, l_i^{(\frac{1}{2})}, l_i^{(1)} \rangle$	n/a	88

Table 5.3: Summary of the efficiency for linear and quadratically varying step times with different load metrics.

time complexity of the algorithms is readily analyzable, one can develop a model using that analysis. For more general applications, a better technique would be to sample the load of a task at each step in the computation. Then, by maintaining a history of past samples, statistical properties such as the mean and variance can be calculated, and more importantly, a technique such as least mean squared deviation or least mean absolute deviation (the latter being more robust for noisy data [36]) can be used to model the load.

Direct Simulation Monte Carlo Application

A DSMC simulation of the GEC reactor demonstrated the usefulness of load modeling. In this problem, neutral particles were injected through one of the small, corner inlets of the reactor. As a result, the load was increasing locally around the inlet at a rapid rate, and increasing globally at a slower rate. Four experiments were conducted on a 12-processor Avalon A12. First, no load balancing was used. In that case, the wall clock time per simulation iteration increased from 1.8 seconds to 3.2 seconds over 500 iterations, as the particle count rose from 1.1 to 1.2 million. In the next three scenarios, load balancing was considered every 100 steps and was initiated if the estimated efficiency was below 90 percent. For the first test, the average CPU time per step over the previous 100 steps was used as a task's load. This resulted in some improvement in run time. Load balancing was conducted at three of four

opportunities and reduced step time by an average of 0.15 seconds per attempt, resulting in a time of 2.6 seconds for step 500. Next, the CPU time for the most recent step was used—equivalent to using the $l_j^{(0)}$ load metric above. In this case, load balancing made a more dramatic difference, reducing step time by an average of 0.28 seconds for a final time per step of 2.4 seconds. The step time reduction quickly dissipated in each case, though, since the load estimate was most accurate immediately after load balancing. In the final case, a least mean absolute deviation fit was calculated, and a load vector of the initial and predicted load at 100 steps in the future was used. This is the same as using the $\langle l_j^{(0)}, l_j^{(1)} \rangle$ vector above. In this final case, load balancing was only attempted once, reducing the time per iteration by 0.6 seconds, for a final time per step of 2.1 seconds. In all three load balancing tests, load balancing required an average of 10.5 seconds per attempt. Table 5.4 and Figure 5.6 give the complete results. In the former, only one number is given if load balancing was not conducted. In the latter, sharp drops in the run time mark the points at which load balancing occurred. In these experiments, the vector load balancing method not only yields a lower time per step, but also required the fewest load balancing attempts, due to the higher accuracy of the load metric. Note that had field calculations also been performed, the load vector should have had three components, assuming that the field solver time remained relatively constant, or four components, assuming that any grid adaption associated with increased particle density increased the field solver's run time as well.

5.3 Related Work

The need for alternative load measurement schemes is also addressed in [14]. The authors point out that load measurement based only on processor utilization neglects the use of other resources, such as memory and disk space. The authors address this problem by combining the time demands on these resources into a single number, rather than considering them separately.

In [25], simultaneous balancing of multiple types of resource utilization is also

timestep	step time before/after lb			
number	no lb	avg lb	last lb	pred lb
0	1.8	1.8	1.8	1.8
100	2.3	2.3/2.1	2.3/1.8	2.3/1.7
200	2.6	2.4/2.3	2.2/1.8	1.8
300	3.0	2.6/2.4	2.2/2.0	1.9
400	3.1	2.5	2.3/2.0	2.0
500	3.2	2.6	2.4	2.1

Table 5.4: Results without load balancing and before and after load balancing for three load metrics.

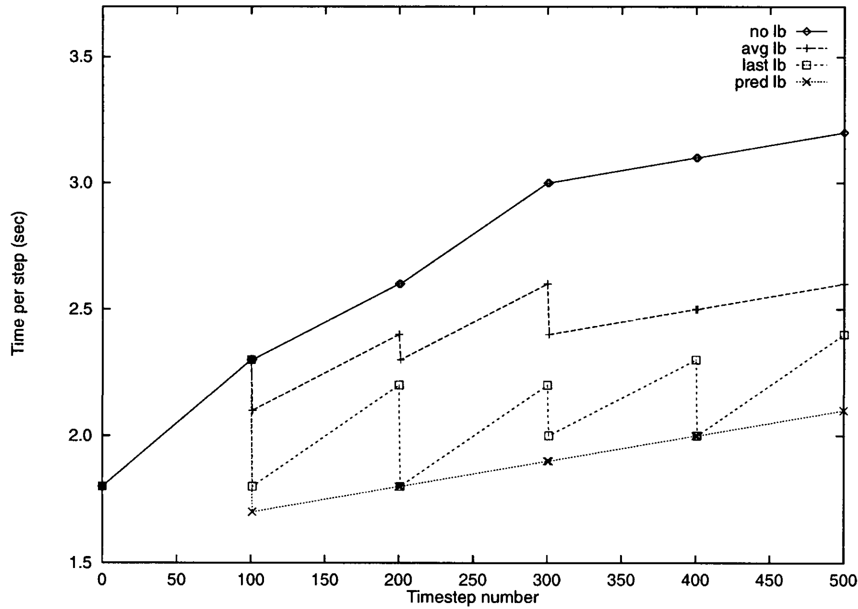


Figure 5.6: Step times without load balancing and with load balancing for three load metrics.

considered. As above, however, the author's technique is to combine separate load values into a single aggregate value via a weighted sum. Ironically, these load values are considered separately for the purpose of profitability determination: The author describes the alternatives of "or" and "and" balance initiation. In the former case, load balancing is undertaken if any category of load exceeds its threshold; in the latter case, it is undertaken only if all categories exceed their respective thresholds. Neither of these aggregate load metrics performs very well, and the author concludes that "although these two experiments do not cover the area of combined workload descriptors comprehensively, we conclude that major performance improvements can hardly be expected when more complex workload descriptors are used." That conclusion is contradicted by the experiments in this chapter.

The authors of [15] note a lack of efficiency improvement for their concurrent PIC application whenever the run time for the field solver phases is roughly the same as the particle transport phase. Their load balancing technique considers only the particle distribution, so one would not expect an improved load balance in the field phases. The authors suggest no solution to the problem, other than noting that, in many scenarios, the particle concentration increases over time, making the particle phase dominant. Thus, as the steady state of such problems is reached, a particle-based scalar approach performs fairly well.

5.4 Summary

Load balancing techniques that attempt to capture the load characteristics of tasks with a single number may perform poorly for certain types of applications. Applications with multiple computational phases, each with a different load distribution, will be balanced either so that only one phase is balanced or so that all phases are semi-balanced. In applications with large variations in the memory and computational requirements, reassignment of tasks to balance the computation may skew memory usage. Finally, applications in which the load changes rapidly may either be balanced in the present or in the future, but not both. A solution to these difficulties is to

consider load to be a vector, where each component of the vector captures a different aspect of the load—one phase versus another, computation versus memory, and present versus future load. The effectiveness of this approach is demonstrated for each of the application categories described above.

Chapter 6 Dynamic Granularity Control

Dynamic load balancing is required when an application developer cannot determine a priori the resource requirements of the tasks comprising a computation. Such situations are especially common in simulation, where resource usage is often a function of the unknown quantities being resolved by the computation. A similar difficulty arises in partitioning a problem into tasks. It may not be readily apparent to what level a given problem should be divided. A partitioning that is initially adequate may not remain so as the computation progresses. In Section 5.2.1, the efficiency achieved for the particle-in-cell application was limited by the fact that one of the partitions contained so many particles that any computer to which it was assigned was overloaded. One workaround would be to uniformly divide the problem into many more tasks so that such situations would be less likely to occur. Unfortunately, communication costs are roughly proportional to the number of tasks in many applications, so such an approach could dramatically reduce scalability. Moreover, in some applications, the tasks represent regions of the problem, between which data dependencies are relaxed. Increasing the number of tasks in such circumstances could reduce accuracy and/or slow convergence.

An alternative to placing the full burden of finding an adequate partitioning on the application developer is to partition the problem dynamically. Instead of relying on a static decomposition to be sufficient for the entire duration of the computation, the problem would be redivided as needed. For example, if a particular task overworks any computer to which it is assigned, that task could be further subdivided so that it could be spread over more than one computer. In general, whenever the granularity of tasks is too coarse to achieve load balance, dynamic task division may allow an improved load distribution.

Another problem addressed by dynamic adjustment of task granularity occurs in *hybrid* computer systems. Such platforms are networks of symmetric multiprocessors,

combining both shared memory and distributed memory paradigms. In this chapter, only hybrid systems in which computers have a uniform number and speed of processors are considered; this restriction is lifted in Chapter 7. In such environments, even if a task does not overwork a computer as a whole, it may overwork an individual processor. For example, if a computer has four processors but only one task assigned to it, it will be only 25 percent utilized, even though its load may be the same as all other computers. In such situations, tasks must be divided to make full use of available processors.

This chapter presents the algorithmic modifications required to incorporate dynamic granularity control, along with results from applying those techniques to real and synthetic applications.

6.1 Algorithmic Modifications

The changes to the load balancing framework necessary for dynamic granularity control involve determining whether tasks should be divided, and, if so, choosing the tasks to divide. These techniques are presented independently of the vector techniques from Chapter 5. They are combined in Section 6.2, below.

6.1.1 Load Evaluation

The loads of tasks are determined as previously described in Section 2.1.1.

6.1.2 Profitability Determination

Load balancing initiates based on the criteria outlined in Sections 2.1.2. If load balancing is unnecessary, it may still be necessary to divide tasks to utilize all available processors on a computer—therefore, the "granularity adjustment" phase described below is conducted in any case.

6.1.3 Load Transfer Calculation

Calculation of load transfers is the same as described in Sections 2.1.3 and 3.2.

6.1.4 Task Selection

As pointed out in Sections 2.1.4 and 4.1, the task selection algorithms therein cannot, in general, achieve the desired load transfers in a single attempt. While multiple rounds of task selection may attain the desired quantities, it is possible that a load balanced state will not be reached. Failure to satisfy load transfers are particularly likely when the tasks are very coarse-grained. In such circumstances, it may be impossible to achieve load balance, simply because there are too few options for task selection. To increase the number of options, one can divide the existing tasks, given some rudimentary support from the application programmer. For example, in a finite element calculation, where a task is represented by a single partition of the problem grid, the partition could be further subdivided to yield two or more additional tasks.

A useful criterion for selecting which tasks to divide is to choose a load threshold. A practical starting point for such a threshold is half the maximum computer load. Any tasks with loads over that threshold should be divided, and the selection process repeated. If the division of tasks significantly improves the load balance achieved, but still does not reach the minimum desired efficiency, divisions can continue using a reduced threshold.

Of course, there is a trade-off between achieving a higher load balance and increasing the communication overhead due to the greater number of tasks. In particular, let the fraction of time spent communicating be C. Assume that the communication overhead increases by a factor of $f_C(\frac{n'}{n})$, where n is the current number of tasks and n' is the number of tasks that would result if another round of divisions were made. If the efficiency with the current set of tasks is *eff* and the efficiency achievable with an increased number of tasks is *eff*$'$, then tasks should actually be divided only when the following holds

$$(1-C)\left(1-\frac{\mathit{eff}}{\mathit{eff}'}\right) > C' - C \tag{6.1}$$

where $C' = f_C(\frac{n'}{n})C$. (Note that C' is the fraction of communication relative to the original execution time.) In other words, the reduction in run time, which is proportional to the efficiency change, $1 - \frac{eff}{eff'}$, multiplied by the percent execution time, $1 - C$, must exceed the change in communication overhead, $C' - C$

The above analysis neglects changes in communication overhead due to the disruption of communication of locality, nor does it consider communication cost reductions that might result from using the locality-improving communication cost functions from Chapter 4. If such effects are noticeable, then an application- or platform-dependent heuristic can be used to improve the selection criterion.

6.1.5 Task Migration

Task migration is the same as described in Section 2.1.5, with the exception that a task's neighbors must not only be notified if it has relocated, but also if it has been divided.

6.1.6 Granularity Adjustment

Assuming that a task can only be executed on one processor at any given time, if the tasks on a multiprocessor computer are too heavily loaded, they will make poor use of the processors. In general, if the load of a task is greater than $\frac{L_{avg}}{Q\,eff_{min}}$, where Q is the number of processors per computer, then some processor on that computer will still be executing that task while the other processors are idle. So, tasks should be divided until their loads are less than $\frac{L_{avg}}{Q\,eff_{min}}$. Once again, when tasks are divided, those tasks with which they communicate must be notified.

6.2 Vector Extensions

The combination of vector techniques and dynamic granularity control primary involves the replacing the scalar criteria for selecting tasks to divide with vector criteria. Below are the details of these changes for each phase of load balancing.

6.2.1 Load Evaluation

Task loads are evaluated as detailed in Section 5.1.1.

6.2.2 Profitability Determination

The criteria from Section 5.1.2 are used to determine when to load balance. As mentioned above, whether or not load balancing is undertaken, the granularity adjustment phase must be conducted.

6.2.3 Load Transfer Calculation

Load transfers are calculated as described in 5.1.3.

6.2.4 Task Selection

If tasks are very coarse, it may be necessary to divide them. In particular, if the desired efficiency is not achieved for one or more of the vector components, then a vector threshold should be chosen. The components of this threshold should be infinite if the desired efficiency was achieved for that component. For example, if memory balance is achieved, but not computation balance, tasks should be divided based only on their computational loads. Tasks should be divided if any of their load components exceed those of the vector load threshold. If the efficiency is improved for one of the components, then that component of the threshold should be lowered. As mentioned in Section 6.1.4, above, there is a trade-off between achieving better load balance and incurring higher communication costs due to having more tasks. This trade-off is less clear in the vector case since the balance of some vector components, such as memory, may not affect run time in a readily analyzable way. A practical approach would be to use the time-based components to justify increased communication costs and to have communication-independent policies for other load components. Tasks might always be split, for example, if memory remains unbalanced, or they might be split only if the memory balance falls outside some range.

6.2.5 Task Migration

Task migration is unchanged from Section 2.1.5.

6.2.6 Granularity Adjustment

If the time-based load components of a task are too high, the task may overwork a processor in a multiprocessor computer. So, a task should be divided if one of its load components exceeds $\frac{L_{\text{avg},k}}{Qeff_{\text{min},k}}$ for some k.

6.3 Results

The load balancing framework from previous chapters was augmented with the dynamic granularity control techniques described above. The resulting framework was applied to both real and synthetic applications to demonstrate the advantage of dynamic partitioning over static partitioning.

6.3.1 Synthetic Application

In a synthetic application, 16 computers were assigned random total loads, uniformly distributed on the interval (0,1). The initial efficiency averaged 52 percent. With a single task per computer, load balancing yielded no improvement. When static sets of 2, 4, 8 and 16 tasks were assigned to each computer, load balancing increased efficiencies to 63, 78, 84 and 93 percent, respectively. In the final experiment, a single task was assigned to each computer, and the task was split on demand by the load balancing algorithm. In this case, an average efficiency of 91 percent was achieved, with an average of only 4.6 tasks per computer. These results are summarized in Figure 6.1. As these experiments show, by dynamically splitting the largest tasks rather than statically dividing all tasks, a higher efficiency was achieved with fewer total tasks. Achieving load balance with the fewest tasks possible is important because communication costs typically increase with the number of tasks, as the next experiment shows.

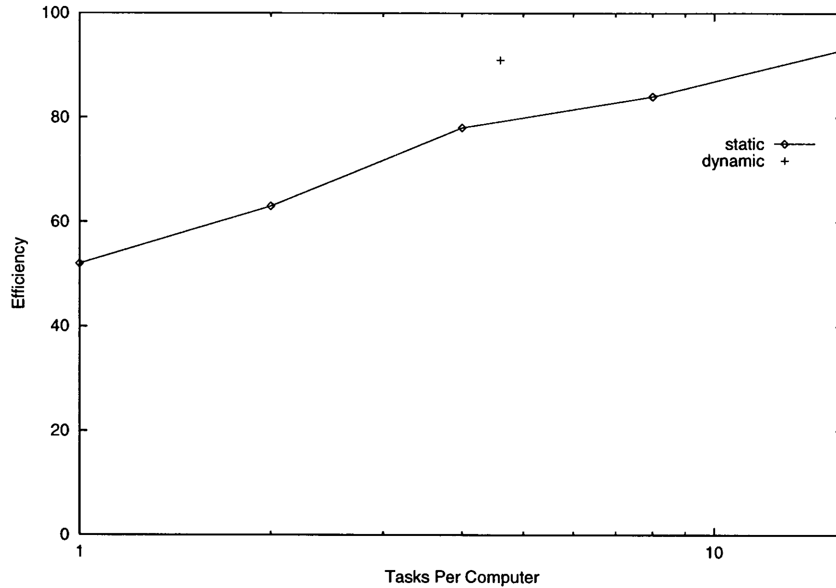


Figure 6.1: Efficiency versus the number of partitions per computer for static and dynamic partitionings.

6.3.2 Direct Simulation Monte Carlo Application

The direct simulation monte carlo (DSMC) application described in Section 2.2 was found to benefit from dynamic granularity control [37]. The problem addressed was once again the Gaseous Electronics Conference (GEC) reactor. Using a 140,000-cell grid of the reactor, simulations of 3.2 million particles were conducted on 128 processors of a Cray T3D. In those simulations, the number of partitions per processor was varied from 1 to 16 in multiples of two, and the performance was measured before and after load balancing in each case. Finally, a case was run in which a single partition was assigned to each computer. Any further partitioning was guided by the algorithms described in Section 6.1 above. The results of that experiment are shown in Figure 6.2. As that graph shows, the highest performance was achieved with dynamic granularity control. In particular, dynamic granularity control required only 5.84 partitions per computer, and achieved 10 percent better performance than the best-case static partitioning (8 partitions per computer) with 27 percent fewer partitions. Although the performance gain was not particularly large, the fact that

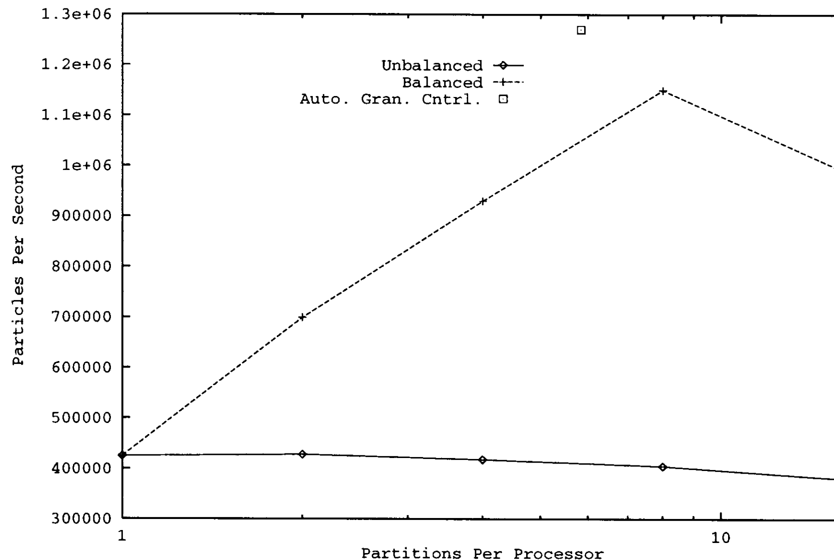


Figure 6.2: Performance versus the number of partitions per computer for static and dynamic partitionings.

it was achieved with no additional effort on the part of the user makes it quite significant. This is important, because as Figure 6.2 shows, there is a significant performance penalty for under- or overestimating the number of partitions needed. In particular, beyond 8 partitions per computer, the additional communication costs begin to outweigh the improvements provided by load balancing.

6.4 Related Work

In [32], the authors examine demand-driven task creation/division for tree-based computations such as branch-and-bound algorithms. They find that lazy task creation significantly outperforms eager task creation, reducing the total number of tasks by a very large factor. The task division criterion used is similar to that described here.

Recursive spectral bisection also provides a method for dynamic repartitioning [4, 47, 48, 58]. An advantage of this approach is that it creates no additional tasks; tasks are rebalanced by redistributing data structures among the tasks rather than

redistributing the tasks themselves. Unfortunately, these methods are designed for grid-based problems and are not readily adaptable to other types of computations. Moreover, the techniques here have the advantage, as they did in previous chapters, of being asynchronous and local in nature. A computer can decide to divide its tasks independently of other computers and need only notify those computers on which neighbors of its tasks reside.

6.5 Summary

Dynamic granularity control lessens the burden on the application developer by allowing a problem decomposition to be improved at runtime. Tasks which are too large are divided into smaller tasks to provide more options for task relocation or to make more effective use of multiple processors within a computer. In practice, dynamic task partitioning outperforms static partitionings, because the division of tasks is demand-driven and is taken only to the necessary level. This is important since communication costs typically increase with the number of tasks.

Chapter 7 Heterogeneous Systems

The use of networks of personal computers, workstations and symmetric multiprocessors as a computing platform requires improved dynamic load balancing techniques. Unlike traditional multicomputers, such as the Cray T3D/E or the Intel Paragon, the computers in a typical local area network are often not of the same processing performance nor do they have the same available memory. As a result, the techniques from previous chapters of this thesis, which consider computing resources to be homogeneous, are insufficient. This chapter extends those algorithms through a relatively simple set of modifications. The resulting load balancing framework allows users to leverage a wider variety of machines for a computation than previously possible. Experiments conducted on a network composed of personal computers running Windows NT, together with workstations and multiprocessor servers running various versions of Unix, demonstrate the effectiveness of these techniques. These results also motivate further work in the area, by exposing deficiencies in current algorithms with respect to simultaneously balancing both processing and memory requirements.

7.1 Algorithmic Modifications

Presented here are the changes to the scalar load framework described in Section 2.1, including improvements from Sections 3.2, 4.1 and 6.1. Merger with the vector techniques from Chapter 5 is described subsequently.

7.1.1 Load Evaluation

Before load balancing can begin, each computer must determine how much load has been assigned to it and at what rate it can process that load. In determining the load of a task, there are two options. One method is to use abstract quantities such as

the number of operations in an algorithm or the count of data structures. Whatever the quantity used, the number must accurately reflect the utilization that will result from executing the task. For example, if the run time for a step of a multibody gravitational simulation is $\mathbf{O}(n^2)$, where n is the number of bodies, then the load should be n^2, since that is the quantity to which the utilization (execution time) is proportional. The memory load, on the other hand, would be n, assuming that no n^2 intermediate data structures are used in the calculations. Note, however, that this section considers balancing either processor time or memory usage, but not both; the details of balancing multiple resources simultaneously are described below.

If the load of task j is taken to be l_j, then the load of computer i is

$$L_i = \sum_{j \in T_i} l_j \tag{7.1}$$

where T_i is the set of tasks mapped to that computer. In that case, the utilization of computer i is given by

$$U_i = \frac{L_i}{C_i} \tag{7.2}$$

where C_i is the computer's capacity.

An alternative to using algorithmic quantities is to measure the utilization directly using system facilities. For example, one might use system calls to obtain the CPU time or amount of memory used by a task. In that case, the task utilizations are summed to give the computer utilization

$$U_i = \sum_{j \in T_i} u_j \tag{7.3}$$

and (7.2) is rearranged to give the computer's load

$$L_i = C_i U_i \tag{7.4}$$

Of course, in both of the above cases, it is assumed that one knows the resource capacity of a given computer. There are a number of ways to determine a computer's

capacity. If the capacity measured is processing speed, a benchmarking program—possibly the target application with a smaller test problem—can be used to determine the relative speeds of a set of computers. These offline performance numbers, along with other statistics, such as the computers' individual memory capacities, can be put into a file which is read at the start of the computation. Such machine description files, without the capacity data, are already commonly used to list the names or addresses of the machines on which the computation is to be run along with the locations of the binaries to load on each machine.

Another approach to capacity determination is to measure *both* the loads *and* utilizations of the tasks. For example, in a particle simulation, one would count the number of particles as well as measure the CPU time required to process those particles. By dividing the former quantity by the latter, one could dynamically calculate the particle processing rate (capacity). This approach combines well with load prediction using sampling and extrapolation, as described in Section 5.2.3.

7.1.2 Profitability Determination

Once load measurement is complete, the computers must collectively determine if a load imbalance exists and whether the cost of remedying that imbalance is exceeded by the cost of allowing the imbalance to persist. The degree of load balance (efficiency) is given by

$$eff = \frac{U_{\text{bal}}}{U_{\max}} \tag{7.5}$$

where U_{bal} is the utilization with perfect load distribution

$$U_{\text{bal}} = \frac{\sum_i U_i}{\sum_i C_i} \tag{7.6}$$

and where $U_{\max}$ is the maximum computer utilization. Note that the U_{bal} is in general different from U_{avg}, the average computer utilization, which would have been the numerator in the efficiency calculation for the homogeneous case. Once the efficiency is calculated, the criteria from Section 2.1.2 can be applied to determine whether or

not to load balance.

7.1.3 Load Transfer Calculation

If a commitment to load balance is made, the next step is to calculate the amount of load that must be transferred between computers to achieve a balanced computation. A number of algorithms have been proposed for this task, including the hierarchical balancing method, the generalized dimensional exchange and heat diffusion methods. Described here are the modifications required to each of those algorithms to support heterogeneous systems.

Hierarchical Balancing Method

The hierarchical balancing (HB) method is a global, recursive approach to the load balancing problem [19, 57]. The set of computers is partitioned into two subsets, and the total load is calculated for each subset. The load transfer from the first subset to the second is that required to make their loads per computer equal

$$\Delta L_{(1,2)} = \frac{P_2 L_1 - P_1 L_2}{P_1 + P_2} \tag{7.7}$$

where L_1 and L_2 are the loads of the two subsets, and P_1 and P_2 are the numbers of computers in each subset. The two subsets are then recursively subdivided and balanced, taking into account the load transfers that occurred at higher levels.

In the case of a heterogeneous system, the goal is to establish an equal load *per capacity* (i.e., an equal utilization) in each subset of computers. Thus, the load transfer becomes

$$\Delta L_{(1,2)} = \frac{C_2 L_1 - C_1 L_2}{C_1 + C_2} \tag{7.8}$$

where C_1 and C_2 are the total capacities of two subsets.

Generalized Dimensional Exchange

The generalized dimensional exchange (GDE) is a simple, iterative scheme for calculating load transfers [59]. In this algorithm, the "links" between adjacent computers are colored so that no computer has two links of the same color. For each color, computers transfer load to or from their neighbors along the links of that color until an adequately balanced state is reached. The load transfer accumulated from computer i to computer j at step t of the algorithm is

$$\Delta L_{(i,j)}^{(t)} = \Delta L_{(i,j)}^{(t-1)} + \lambda(L_i^{(t-1)} - L_j^{(t-1)}) \tag{7.9}$$

where $L_{(i,j)}^{(0)}$ is 0, and λ is a constant between 0 and 1.

For the heterogeneous case, the transfer must be weighted to account for the relative capacities of the two computers. The resulting transfer iteration becomes

$$\Delta L_{(i,j)}^{(t)} = \Delta L_{(i,j)}^{(t-1)} + 2\lambda \left(\frac{C_j L_i^{(t-1)} - C_i L_j^{(t-1)}}{C_i + C_j} \right) \tag{7.10}$$

The factor of 2 before λ is required so that λ has the same meaning as in the homogeneous case. I.e., if all of the capacities are equal, the factor of two will be canceled out.

Diffusion

Like the GDE, heat diffusion is an iterative technique for determining how much load to transfer between computers. For the heterogeneous case, the goal is once again to transfer load between computers so as to make their respective utilizations equal. Thus, the change in load for computer i is that which moves its load closer to being in balance with respect to its neighbors' loads and their respective capacities

$$\frac{\mathrm{d}L_i}{\mathrm{d}t} = \alpha \sum_{j \in N_i} \frac{C_i L_j - C_j L_i}{C_i + C_j} \tag{7.11}$$

This equation bears a substantial resemblance to the general diffusion equation (3.1) in Chapter 3. In fact, the particular choices of

$$\mathcal{D}_{i,j} = \frac{C_j}{C_i + C_j} \tag{7.12}$$

and

$$\mathcal{D}_{j,i} = \frac{C_i}{C_i + C_j} \tag{7.13}$$

make (3.1) equal to (7.11). This choice of $\mathcal{D}$ is valid by the arguments in Section 3.1, since the presence of a non-unit product cycle in the derived graph described there would imply that $C_i < C_j < ... < C_{j'} < C_i$, for some i, j and j', which is a contradiction.

7.1.4 Task Selection

The algorithms from Sections 2.1.4 and 4.1 can be used unmodified to select tasks to satisfy the load transfers.

7.1.5 Task Migration

Once the computers have resolved the new locations of their tasks, the actual data structures of those tasks are transferred from their old locations to their new locations. If tasks are taken to be whole processes and the architecture and operating system of the machines is the same, task migration can be accomplished by directly transferring the address space of a process from one machine to another [2, 30]. If a task is a smaller unit of work, say a thread within a process or a collection of data structures to be acted upon, or if the machines in the system are of mixed architecture, then additional support must be provided in the task migration phase. For example, the user could provide code to write the state of a task to the network on the sending computer and another routine to read the state of a task from the network and recreate it on the receiving computer. These routines must be *typed* so that basic data types (integer, character, floating point) can be converted between the respective machines. In some

environments, these state transport routines can be made to look very much like the checkpointing routines that are normally required for long-running concurrent computations—an example of this is given in Appendix A.

7.1.6 Granularity Adjustment

Once tasks have arrived at their new locations, it may be beneficial to increase or decrease the number of tasks on a given computer. For example, on a computer with more than one processor, there may be too few tasks to fully utilize all of the processors. In that case, tasks whose utilizations exceed the balanced computer utilization will prevent the desired efficiency from being achieved. By dividing any task with a utilization higher than $\frac{U_{\text{bal}}}{Q_i eff_{\text{min}}}$, where Q_i is the number of processors on computer i, this problem is eliminated.

7.2 Vector Extensions

The load balancing methods described above redistribute a single type of load over a set of heterogeneous computers. When multiple types of load, such as computation and memory, are to be jointly reassigned, a different approach must be used. In particular, by considering the loads, utilizations and capacities of computers as vectors, simultaneous management of multiple resources becomes possible. Presented here is the incorporation of the vector techniques from Chapter 5 into the heterogeneous framework described above.

7.2.1 Load Evaluation

In the vector case, the load of a task contains multiple components for different load categories. For example, consider a particle simulation with an integrated electromagnetic field field solver, in which the particles move through a grid over which the field is solved. The computational load of such an application is characterized by both the number of particles, on which the particle movement phase most directly

depends, and the number of grid cells, which determines the time for the field solution phase. A third component could be the memory used—a weighted sum of the particle and cell counts. Determination of the capacities for each of these components is essentially the same as in the scalar case. For the computational phases, either an instrumented test problem can be used to calculate the relative phase-specific speeds of various machines or these quantities can be calculated by the instrumented application at run time. ("Instrumented" means that timing calls are introduced into the application to determine the breakdown of execution time for the various phases.) So, given that the load of a particular task j is $\vec{l}_j$, the load of a computer is

$$\vec{L}_i = \sum_{j \in T_i} \vec{l}_j \tag{7.14}$$

The utilization of a computer is given by

$$\vec{U}_{i,k} = \frac{\vec{L}_{i,k}}{\vec{C}_{i,k}} \tag{7.15}$$

for all k.

7.2.2 Profitability Determination

In the vector case, the degree of imbalance becomes a vector as well

$$\overrightarrow{\mathit{eff}}_k = \frac{\vec{U}_{\mathrm{bal},k}}{\vec{U}_{\mathrm{max},k}} \tag{7.16}$$

for all k, where the components of $\vec{U}_{\mathrm{bal}}$ and $\vec{U}_{\mathrm{max}}$ are given by

$$\vec{U}_{\mathrm{bal},k} = \frac{\sum_i \vec{L}_{i,k}}{\sum_i \vec{C}_{i,k}} \tag{7.17}$$

and

$$\vec{U}_{\mathrm{max},k} = \max_i \vec{U}_{i,k} \tag{7.18}$$

respectively. The decision to load balance is based on the criteria in Section 5.1.2.

7.2.3 Load Transfer Calculation

To calculate the ideal load transfer, the scalar loads and capacities in the transfer algorithms must be replaced by vectors.

Hierarchical Balance Method

For the HB method, the vector transfer between two partitions of computers becomes

$$\Delta\vec{L}_{(1,2),k} = \frac{\vec{C}_{2,k}\vec{L}_{1,k} - \vec{C}_{1,k}\vec{L}_{2,k}}{\vec{C}_{1,k} + \vec{C}_{2,k}} \tag{7.19}$$

for all k.

Generalized Dimensional Exchange

For the GDE algorithm the transfer is now

$$\Delta\vec{L}^{(t)}_{(i,j),k} = \Delta\vec{L}^{(t-1)}_{(i,j),k} + 2\lambda\left(\frac{\vec{C}_{j,k}\vec{L}^{(t-1)}_{i,k} - \vec{C}_{i,k}\vec{L}^{(t-1)}_{j,k}}{\vec{C}_{i,k} + \vec{C}_{j,k}}\right) \tag{7.20}$$

for all k.

Diffusion

For diffusion, the values of $\mathcal{D}$ for the general diffusion algorithms in Section 3.2 are

$$\vec{\mathcal{D}}_{i,j,k} = \frac{\vec{C}_{j,k}}{\vec{C}_{i,k} + \vec{C}_{j,k}} \tag{7.21}$$

and

$$\vec{\mathcal{D}}_{j,i,k} = \frac{\vec{C}_{i,k}}{\vec{C}_{i,k} + \vec{C}_{j,k}} \tag{7.22}$$

for all k.

7.2.4 Task Selection

The cost-indifferent and cost-driven task selection algorithms from Section 5.1.4 apply to the heterogeneous case without modification.

7.2.5 Task Migration

Task migration is the same as described in Section 7.1.5.

7.2.6 Granularity Adjustment

If a processor-related utilization component of a task is too high, the processor to which it is assigned may be overworked. Task division should occur whenever a task's processing utilization component exceeds $\frac{\vec{U}_{\text{bal},k}}{Q_i \vec{eff}_{\min,k}}$ for some k.

7.3 Results

The methods described above were implemented in the Scalable Concurrent Programming Library, which is described in Appendix A. Using that framework, a number of experiments were conducted, involving both real and artificial applications. The results of those experiments are presented here along with a description of the environment in which the experiments were performed.

7.3.1 Heterogeneous Testbed

The testbed for the experiments described below was a network of 10 personal computers, workstations and multiprocessor servers. This network included single- and dual-processor Dell PCs, a two-processor Silicon Graphics Origin 200 server, two Indigo 2 and three Indy workstations, a Sun SparcServer and a Digital Equipment AlphaStation. Included in that list are machines with both 32- and 64-bit words as well as big- and little-endian byte orderings (i.e., most significant bit first and least significant bit first, respectively). These machines are described in greater detail in

Num	Processor Type	Operating System	Memory (MB)	Relative Speed	Memory to Speed
1	30 MHz Sparc	SunOS 4.1.3	128	1.0	128.0
2	150 MHz Alpha	Digital UNIX 3.2	64	4.4	14.5
3	133 MHz R4600	IRIX 5.3	64	6.0	10.7
4	133 MHz R4600	IRIX 6.2	64	6.0	10.7
5	133 MHz R4600	IRIX 6.2	64	6.0	10.7
6	150 MHz R4400	IRIX 6.2	288	6.8	42.4
7	200 MHz R4400	IRIX 5.3	128	8.6	14.9
8	200 MHz Pentium	Windows NT 4.0	64	13.0	4.0
9	180 MHz R10000 (x2)	IRIX 6.4	128	38.0	3.4
10	266 MHz Pentium II (x2)	Windows NT 4.0	256	39.0	6.6

Table 7.1: Descriptions of computers in the heterogeneous testbed.

Table 7.1. Note that the performance of the machines for a benchmark problem varied by a factor of almost 40, and the available memory varied by over a factor of four. The benchmark problem used to determine the relative speeds was a small test case for the particle simulation application described in Chapter 2.

7.3.2 Parametric Experiments

To demonstrate the effectiveness of the heterogeneous load balancing framework, tests were conducted using synthetic computations on the heterogeneous testbed described above. In particular, four scenarios were covered, including computations with single phases, computations with multiple phases, computations with rapidly changing loads, and computations with high memory requirements. In each of these experiments, a simple tetrahedral grid was partitioned among the computers in the testbed. A "task" in the context of the load balancing framework was thus a grid partition and the operations performed on it.

Single-Phase Computations

In many types of computations, such as electromagnetic and fluid flow finite element solvers, the processing time for a grid partition is proportional to the number of fun-

damental grid structures, such as cells, faces, edges or points, that it contains. So, for the first test case, the load for a partition was taken to be the number of cells within it. A 12,540-tetrahedra box grid was partitioned for the 10 machines. The number of cells per computer initially varied from 1,006 to 1,633, due to the fact that the grid was more dense in the corners of the box. (A simple geometric partitioning was used.) The utilization, determined by dividing the number of cells on a computer by its capacity from Table 7.1, varied from 41 to 1,553, with an efficiency of 6 percent. After homogeneous load balancing, the variance in the loads was significantly reduced, ranging from 1,205 to 1,428, which would have yielded an efficiency of 88 percent had the machines actually been homogeneous. However, because the machines were heterogeneous, the efficiency remained very low, at 7 percent, with the utilization varying from 32 to 1,417. With heterogeneous load balancing, the resulting efficiency was 93 percent, with utilizations ranging from 88 to 105. The computer load assignments are summarized in Figure 7.1. As the graph shows, the loads under heterogeneous balancing were very close to the scaled processing rates of the computers, whereas the initial loads and those that resulted from homogeneous load balancing exceeded several computers' load capacities.

Multi-Phase Computations

The time complexity of different phases in a computation may depend on different variables. Consider for example, a uniform-density particle simulation with coupled electromagnetic fields. In phases such as particle transport and collision, the particle count in a partition would dominate its computation time. Since the particle density is assumed to be uniform, the number of particles would be proportional to the volume of the partition. For the field solver phases, the computation would depend most heavily on the number of grid elements, such as cells. Assuming that all cells are of roughly equal size, the ratio of these two quantities should be equal in each partition. They might not be proportional, however, if the grid cells are smaller in some regions than others. Since small cells may be needed to capture complex geometric features, a disproportion of partition volume to cell count is possible. The resulting disparity

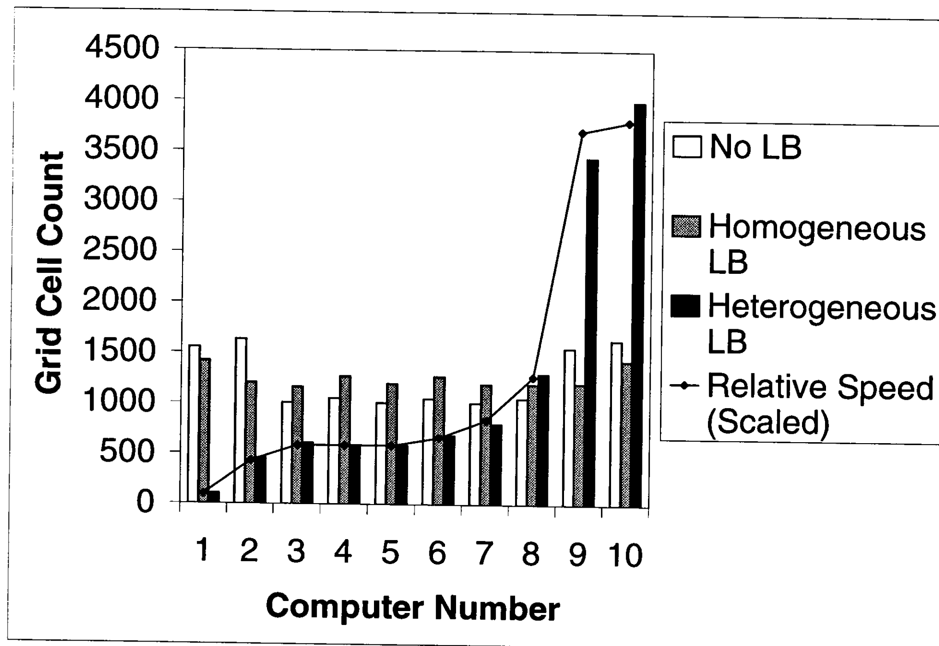


Figure 7.1: Computer load assignments for single-phase box grid problem on heterogeneous testbed.

would carry over to the loads for the particle- and field-related phases.

The computers in the testbed were assigned partitions from the 12,540-cell box grid used above. Although the geometry is uniform, the cells are more dense in the corners of the grid, as was described above; thus, one would expect that cell count and volume would not be in equal proportion for each partition. The problem was then balanced, assuming that the processing rates of the computers were the same with respect to both the cell- and volume-driven phases. When the problem was balanced using only cell counts as the load metric, an efficiency of 90 percent was achieved for the cell-driven phase. However, the distribution of volume was worse, with an efficiency of 52 percent. Conversely, when volume was used as the load metric, 90 percent utilization was achieved for the volume phases, versus only 69 percent for the phases depending on cell counts. To remedy this problem, one can consider the loads

Load Balancing Scenario	Cell Phase Efficiency	Volume Phase Efficiency
None	6	8
Scalar (Cell)	90	52
Scalar (Volume)	69	90
Vector (Cell/Volume)	85	83

Table 7.2: Results of load balancing two-phase box grid problem on heterogeneous testbed.

of the tasks to be a vector, where the components are the cell counts and total volume of the grid partition for that task, as was described in Chapter 5 and Section 7.2, above. When the problem was balanced using these vectors, efficiencies of 85 and 83 percent were reached for the cell- and volume-dependent parts of the computation, respectively. These results are summarized in Table 7.2.

Rapidly Evolving Computations

In some applications, the loads of tasks may change too rapidly for load balancing to keep up, due to the prohibitive cost of load balancing as often as needed. In such situations, balancing based on a vector of both the current, actual load and the future, predicted load can result in a superposition of loads that varies in the same way at each computer, allowing a better load balance to be maintained for a longer period of time, as was seen in Section 5.2.3. In a heterogeneous environment, the problem of tasks with rapidly changing loads is magnified, since the utilization distribution would change greatly if such tasks were assigned to slow computers.

An experiment was conducted in which the 12,540-cell grid used above was partitioned for the 10 computers in the testbed. Each task was assigned a random number between 1 and 2, which represented the factor by which its load would grow during the rest of the computation. In a real application, such load changes might be due to grid adaption, for example. If tasks were reassigned based only on their initial loads, the initial efficiency was 90 percent, and the final efficiency was 74 percent. If they were reassigned based on their final loads, the efficiency was 73 percent at the

Load Balancing Scenario	Initial Efficiency	Final Efficiency
None	6	6
Scalar (Initial)	90	74
Scalar (Final)	73	94
Vector (Initial/Final)	87	91

Table 7.3: Results of load balancing rapidly evolving box grid problem on heterogeneous testbed.

beginning but increased to 94 percent by the end of the computation. If the initial and final loads of the tasks were combined into a vector, and the computation was balanced based on that, the initial and final efficiencies were 87 and 91 percent. Table 7.3 summarizes these results.

Memory-Intensive Computations

In a homogeneous computing environment, as long as the memory and processing requirements of tasks are highly correlated, there is no need to balance these two requirements separately. In a heterogeneous environment, however, one has to contend with the fact that the relative memory and processing capacities of the machines may not be proportional. As is shown in Table 7.1, the memory to processing speed ratio of the machines varies from 3.4 to 128.0, with a median of 10.7. This variation is a result of the fact that the processing speeds of computers have been increasing much more rapidly than their memory capacities. Thus, the newest machines in the testbed (the last three) have the least memory relative to processing performance. This presents a problem for memory-intensive applications. For example, while computers 9 and 10 in Table 7.1 comprise 60 percent of the total processing capacity, they contain only 31 percent of the total memory. In fact, for any problem requiring over 434 megabytes of memory, the memory capacity of the computer 9 would be exceeded, since roughly 30 percent of the work would be assigned to it. So, the practical problem size would be limited to 434 megabytes, even though there is a total of 1,248 megabytes of memory available. To circumvent this difficulty, one can reassign work to balance processor

utilization, *subject to constraints on memory use.*

To illustrate the benefits of trading off memory versus computation, an experiment was conducted in which a 200,000-cell box grid was partitioned for the 10 computers described above. Assuming that there are 4 kilobytes of data structures associated with each cell, this grid would require approximately 800 megabytes of memory. Thus, if the problem were rebalanced only on the basis of computational costs, one would expect that the memory capacities of some of the computers would be exceeded. That was, in fact, the case: When the grid was balanced based only on the computers' speeds, the memory capacities of computers 8, 9 and 10 were all exceeded. The situation was particularly bad for computer 9; roughly 58,000 cells were relocated to it, occupying 227 megabytes of memory, almost twice that computer's actual memory capacity. This problem was balanced again, accounting for the available memory capacities of the machines, which were assumed to be 75 percent of the values in Table 7.1. In this case, none of the memory capacities were exceeded. However, the computational efficiency was poor, at 6 percent. Balancing on both memory and processor loads also met all of the memory capacities, but improved the processing efficiency only slightly, to 7 percent. As one can see from the summary of these results, in Figure 7.2, the fastest computers were not assigned as many cells as they could have been. This was due to the fact that balancing memory and processor utilization was contradictory. For example, when balancing memory, computer 1 would be assigned approximately 10 percent of the load, whereas it would be assigned less than 1 percent when balancing processor utilization. So, the load transfer quantities for computer 1 would generally contain components that transferred load both away from it and onto it. Since the unit of both processing and memory load is a cell, such transfers cannot be met entirely.

An algorithm for determining the best assignment of load with constrained memory is as follows: Distribute load among computers according to their processing capacities *only.* If the memory capacities of any computers are exceeded as a result, assign loads to them equal to their memory capacities and remove them from consideration. Then, recursively reassign the excess load among the computers with

available memory, removing "full" computers at each step until all work has been assigned. This algorithm could only be incorrect if it were possible to reassign load to a non-full computer so that the maximum processor utilization would be lowered (i.e., the efficiency would be raised). That would be possible only if a full computer had a processor utilization less than some non-full computer, or if the processor utilizations of any two non-full computers were not equal. That contradicts the invariant that the algorithm maintains at each step: The initial assignment on processing capacities makes the utilizations equal; reassignment of excess load from the full computers to the non-full computers can only make the latters' utilizations higher. If any computers are not full, their utilizations would be made equal in the last load assignment made by the algorithm.

The algorithm described above can actually be implemented using the HB method from Section 7.2.3. The procedure is as follows: Assign load to computers using the HB algorithm with the full processing capacities of the computers. If the memory capacities of any computers are exceeded, set their capacities to zero, and reassign the excess work using the HB algorithm. Repeat the reassignment of excess work until none remains. For this to work, the HB algorithm must be modified slightly: The recursion should terminate not only when a one-computer partition is reached, but also when a zero-capacity partition is reached. Since at minimum one computer could become "full" at each step, the HB algorithm must be repeated at most $\mathbf{O}(P)$ times. The above algorithm was implemented and gave the results shown by the rightmost bar for each computer in Figure 7.2. As that graph shows, the load assignments are very close to the ideal assignments, which were calculated by hand. The achieved processor efficiency with the memory-packing HB algorithm was 35 percent versus an ideal efficiency of 38 percent.

The modified HB algorithm assumes, of course, that the total memory capacity of the machines is higher than that required by the problem. If the problem exceeds the total memory capacity, the best approach is probably to balance based on memory only, so that each computer's capacity is exceeded by the same fraction. The algorithm also assumes that processing and memory requirements for tasks are highly

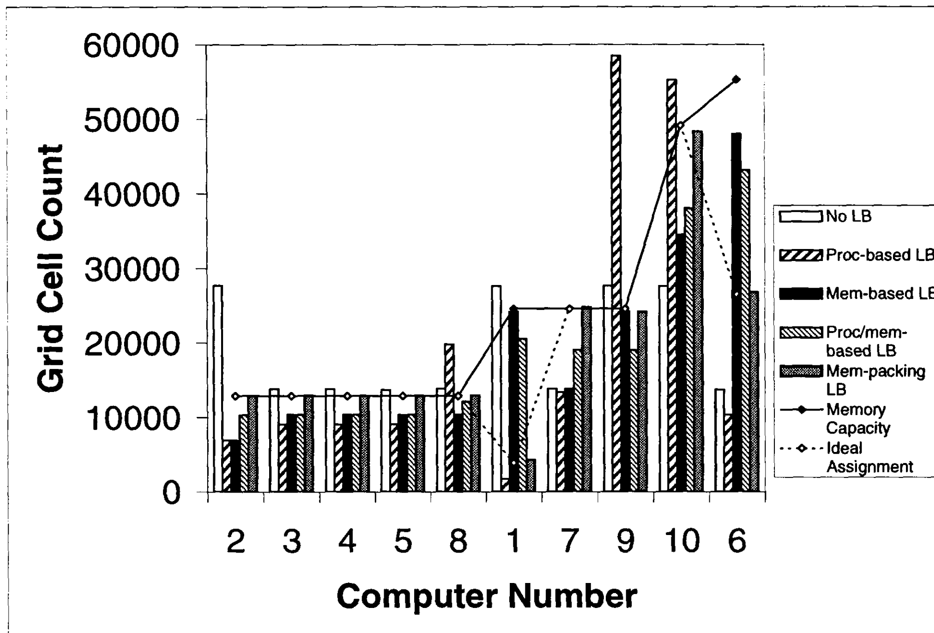


Figure 7.2: Computer load assignments for memory-intensive box grid problem on heterogeneous testbed. (Computers are sorted by memory capacity instead of processing rate.)

correlated. If this is not the case, vector methods might be able to perform well, as they did in Section 5.2.2 for homogeneous environments.

Note that the simple trick of setting the full computers' capacities to zero and reassigning the excess load will not work with the GDE and diffusion algorithms from Section 7.2.3. Because load transfer decisions are made only using local information, a zero-capacity computer can prevent the transfer of load between two non-full computers. (In terms of heat diffusion, a zero-capacity computer is a perfect insulator.) For example, consider a linear array of computers in which the middle computer has zero capacity. Load cannot be transferred from computers on its left to those on its right, or vice-versa, since the middle computer will not accept additional load, even if that load is to be immediately transferred away. Another practical problem occurs when

zero-capacity computers are adjacent: The algorithms produce divide-by-zero errors. It is possible that these algorithms can be modified to handle zero-capacity computers correctly so that load can be redistributed in memory-constrained environments.

7.3.3 Application Experiments

The direct simulation monte carlo (DSMC) application described in Section 2.2 was applied to a small test problem, as well as a large reactor simulation. Presented here are the results of those tests, which were conducted on the heterogeneous testbed described above.

Small Box Problem

The DSMC application was used for a simulation of a 54,000-cell box grid containing 432,000 particles. Four experiments were conducted using this problem. First, the problem was run without load balancing. In this case, time steps required an average of 14.1 seconds each. Next, a homogeneous load balancing strategy was used with CPU time as the task loads. Although CPU time is actually a utilization metric and not a load metric, it can be used as the load if the computers are considered to be homogeneous. If a capacity-invariant quantity such as the particle count were used, no tasks would have been moved, since all of the computers initially had the same number of particles. After homogeneous load balancing, simulation steps required 6.5 seconds each. Also, in subsequent load balancing steps, computers continued to transfer large numbers of tasks, without improving the step time. This was due to the absence of computer capacity estimates; the utilizations of the computers did not vary as the load balancing algorithms expected. For example, transferring 10 seconds of work from one computer to another might change the utilization of the latter computer by much more or much less that 10 seconds. In the third test case, the capacities from Table 7.1 were used, reducing the time per step to 2.5 seconds. Unlike the homogeneous case, the number of tasks transferred dropped off rapidly after the first two load balancing rounds. A few tasks continued to be transferred

Load Balancing Scenario	Step Time (sec)	Improvement Factor
None	14.1	None
Homogeneous	6.5	2.2×
Heterogeneous (static capacities)	2.5	5.6×
Heterogeneous (dynamic capacities)	2.0	7.1×

Table 7.4: Results of load balancing box DSMC problem on entire heterogeneous testbed.

Load Balancing Scenario	Step Time (sec)	Improvement Factor
None	4.3	None
Homogeneous	5.2	None
Heterogeneous (static capacities)	2.4	1.8×
Heterogeneous (dynamic capacities)	2.0	2.2×

Table 7.5: Results of load balancing box DSMC problem on heterogeneous testbed without computer 1.

in subsequent load balancing rounds, however, due to the differences between the capacity estimates and the actual capacities of the computers. In the final test, the computers' capacities were calculated dynamically by dividing the total number of particles on each computer by the CPU time required to process them. This improved performance even more, reducing the average step time to 2.0 seconds. No further task transfers took place after the third load balancing round, as the capacity estimates were quite exact. These results are summarized in Table 7.4. Because the improvement numbers in Table 7.4 were skewed by the presence of computer 1, which made the unbalanced case extremely slow, another round of tests were conducted in which that machine was omitted. The results from those tests are given in Table 7.5.

Large Reactor Problem

The Gaseous Electronics Conference (GEC) reactor also provided a test case for heterogeneous load balancing. Using a 140,000-tetrahedra grid of this reactor, a 1.1 million particle simulation was conducted on the machines in the heterogeneous testbed. When the grid was partitioned with a naive partitioner, which treated all machines as equal, the problem failed to load, because it immediately exceeded the available memory on some of the machines. A more sophisticated partitioner was devised which divided the grid according to the machines' memory capacities. Each computer was assigned a partition with a cell count proportional to its available memory. When the problem was run using this partitioning, a time step of the simulation took an average of 44.1 seconds. Dynamic load balancing based on processor speeds alone failed, as the memory capacities of computers 8 and 9 were exceeded. Dynamic load balancing using both processor speeds and memory capacities improved over the initial partitioning somewhat, reducing step time to 32.5 seconds. A greater improvement was achieved in this instance than in the parametric memory-balancing case above; this was due to the fact that, because of the variance in particle and grid density, the memory and processing requirements of tasks were not perfectly correlated. So, balancing memory and processor utilization were not necessarily contradictory, as they were when processing and memory were in the equal proportions. Finally, balancing the problem with the memory-packing HB algorithm described above gave an even lower average step time—7.8 seconds using static processor capacities and 6.1 seconds using dynamic processor capacities, respectively. These results are summarized in Table 7.6.

7.4 Related Work

The authors of [27] investigate the static assignment of a series of tasks to a collection of heterogeneous machines using list scheduling. They analyze the worst-case behavior of their algorithm, assuming that the capacities of the computers and computational

Load Balancing Scenario	Step Time (sec)	Improvement Factor
Homogeneous Partitioning	Failed	N/A
Heterogeneous Partitioning	44.1	None
Proc-based LB	Failed	N/A
Proc/mem-based LB	32.5	1.4×
Mem-packing LB (static capacities)	7.8	5.7×
Mem-packing LB (dynamic capacities)	6.1	7.2×

Table 7.6: Results of load balancing GEC reactor DSMC problem on heterogeneous testbed.

requirements of the tasks are known beforehand.

The dynamic assignment of tasks in a heterogeneous environment is considered in [8]. The authors address the problem of communication costs, in that their approach tends to assign processes which communicate heavily with one another to nearby computers. The basic idea of their approach is to cluster tightly coupled processes and computers and to map clusters of the former to clusters of the latter.

In [29] a manager-worker approach is expounded, in which a central agent dispatches jobs to a collection of heterogeneous machines. The load balancing methodology used there is the same as that in [40]. Just as the manager-worker scheme works fairly well for scheduling independent tasks on small homogeneous networks, it also works well on heterogeneous networks. The uncertainty in the processing rates of heterogeneous computers is not fundamentally different from the existing uncertainty of the tasks' execution times on a homogeneous network.

Presented in [61] is a sophisticated manager-worker scheme for very large networks of heterogeneous machines. Task assignment is static, but takes into account the resource needs of the task as well as the resources available at the target computer.

As the authors of [26] rightly point out, simple load balancing, with the goal of maximizing processor utilization, may not be the best approach for all applications

running in heterogeneous environments. In particular, if the application is itself heterogeneous, in that different tasks do different kinds of work, it is better to assign tasks to the computers that do their particular type of work fastest. The authors' solution is make tasks have an affinity for which they are best suited. This affinity increases the likelihood that tasks will be mapped to the appropriate computers. Note that a similar affect can be achieved with the vector approach from Section 7.2. If each type of computation is a separate vector component, then appropriate capacities can be assigned to computers so that tasks migrate towards the computers with the highest relative capacity.

In [10], an explicit finite difference calculation is executed on a network of heterogeneous machines. The authors compare the performance of homogeneous and heterogeneous static partitionings of the problem grid, as well as for heterogeneous dynamic load balancing. In the dynamic load balancing scheme, processes on different computers periodically exchange grid elements to balance their execution times. The authors do not specify how they determined the number of grid cells to transfer.

The authors of [17] use a manager-worker algorithm for the simulation of viscoelastic fluid flow. Unlike the approach used here, the entire grid is replicated at each computer, and tasks are given work assignments for particular parts of the grid, for which they report results on completion of their calculations.

An alternative to the datatype translation scheme for task migration in Section 7.1.5 is given in [46]. There the authors describe a technique based on recompilation that would allow the state of a running process to be migrated between computers of different architecture. Their methods are preliminary, however, and many details, such as the handling of complex, dynamically allocated data structures are not fully worked out.

7.5 Summary

This chapter has presented new methods for load balancing in heterogeneous environments. In particular, it introduced new versions of the hierarchical balancing

method, the generalized dimensional exchange, and heat diffusion, all of which take into account the load capacities of the target computers. These modified algorithms are presented in the context of a larger framework, which includes determination of when to load balance and selection of tasks to transfer or divide to achieve a better load distribution. Finally, another set of modifications allows multiple types of load to be simultaneously reassigned. These vector techniques perform well in one of two circumstances: They work when the load vector components are uncorrelated, or if they are correlated, when they relate to the same capacity on each computer. The methodology fails in instances where load components are tightly coupled, but depend on uncorrelated capacities. In such situations, there is essentially a "conflict of interest" between meeting one set of capacities versus meeting another. A preliminary algorithm is given which addresses the case of balancing processing requirements in the presence of memory constraints.

Chapter 8 Conclusions

This thesis has considerably extended the scope of dynamic load balancing techniques. Applications and environments which were poorly served by previous efforts are now addressed. In particular, techniques are introduced which allow the simultaneous reassignment of multiple types of load. These techniques provide load balancing for applications with multiple phases, with rapidly changing loads, and with disparate computation and memory requirements. Also addressed are applications with insufficient task decompositions. In such circumstances tasks are dynamically repartitioned to make more effective use of available resources. The above methods are in turn extended to provide load redistribution in heterogeneous systems. Included in that effort is a preliminary algorithm for memory-constrained environments. Together, these techniques allow concurrent computations to run efficiently on a wider variety of platforms than previously possible.

The thesis also considers the effect of load balancing on the communication structure of applications by proposing mechanisms that take into account the costs of relocating tasks to different computers. This technique not only maintains existing communication locality; it also improves locality that was initially poor. Finally, an improved load transfer algorithm is developed which either transfers less work or executes more rapidly than earlier algorithms.

Further work remains to be done, however. While the memory-packing hierarchical balancing method in Section 7.3 works well enough, in Section 3.3, the hierarchical balancing method was shown to transfer more work than necessary in highly connected networks, such as meshes and tori. As a result, a modified version of the generalized dimensional exchange or diffusion algorithm would probably perform better in such cases. Moreover, a general framework in which zero-capacity computers are allowable would be useful for networks of workstations that become idle. When a user is logged on, the workstation would have a capacity of zero; when the user logs

off, the capacity of the workstation would be raised to allow tasks to migrate onto it. A potential method for handling such cases can be found in Chapter 3. The coefficients $\mathcal{D}$ can be chosen so that permanent load flow patterns *do* exist. In particular, load could be routed away from zero-capacity computers using such flow patterns. In addition, other capacity adjustments could be made to account for competition with other applications. For example, one might divide the baseline processing capacity for a given computer by its "load average," which is a measure of the average number of processes competing for CPU time over some recent history.

Another area of investigation relates to dynamic granularity control. This thesis considered the division of tasks to better satisfy the ideal load transfer quantities or to make more effective use of multiple processors within a computer. Another alternative is to merge tasks. This would eliminate extraneous options in task selection and reduce scheduling and communication overhead. The criteria for task agglomeration would look much like those for task division; a utilization threshold would be chosen below which tasks should be merged, and specific tasks would be selected for merger based on the expected benefits that their amalgamation would provide. Note, too, that tasks could be logically merged rather than physically merged. In other words, a collection of tasks could be treated as a single *metatask* for the purpose of task selection and migration, similar to [8]. This would reduce selection overhead as well as improve communication performance if the tasks within that collection communicate with one another.

Appendix A Scalable Concurrent Programming Library

The Scalable Concurrent Programming Library (SCPlib) provides basic programming technology to support concurrent, irregular applications. Like its predecessor, the Concurrent Graph Library [45], SCPlib has been applied to a variety of large-scale industrial simulations and is portable to a wide range of platforms. Applications implemented using SCPlib include particle simulations for non-continuum gas and plasma flows [37, 38], an electrostatic field solver (see Appendix B), a continuum fluid flow solver, a tensor math package, and proprietary applications at the Intel Corporation. Platforms on which SCPlib runs include distributed-memory multicomputers such as the Cray T3D/E, Intel Paragon and Avalon A12, shared-memory systems such as the SGI PowerChallenge and Origin 2000, as well as networks of workstations running Unix and PCs running Windows NT. On each of these platforms, the library provides an optimized, portable set of low-level functionality, including message-passing, thread management, synchronization, I/O, and performance monitoring. The library also provides a higher level of functionality, which includes heterogeneous communication and file I/O, load balancing, and dynamic granularity control. This appendix briefly describes the SCPlib programming model and summarizes the implementation of dynamic load balancing and granularity control.

A.1 Programming Model

The SCPlib programming model is based on the concept of a concurrent graph of communicating tasks, called nodes. This model is designed to abstract the mapping of work away from particular computers by encapsulating computation within these nodes. Each node is comprised of a thread of execution, a set of named user state

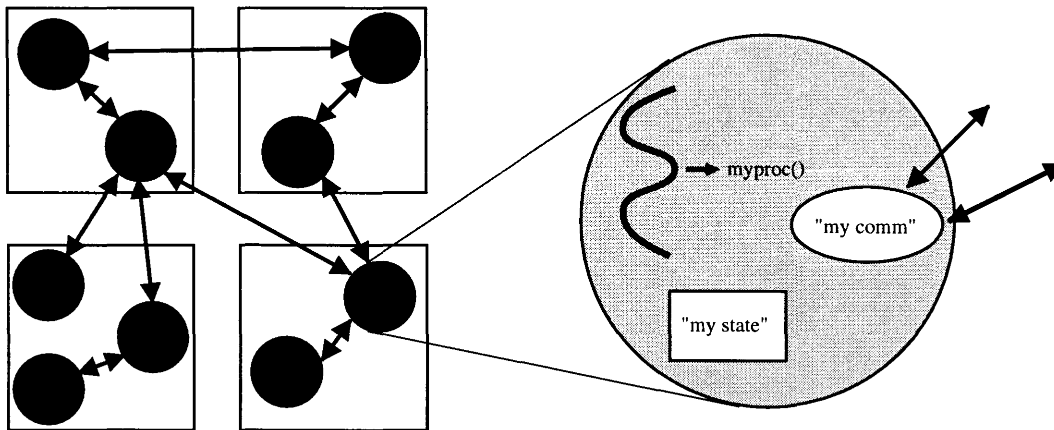


Figure A.1: A concurrent graph and the internal structure of one of the nodes in that graph.

data, and set of named communicators, the latter of which each contain one or more communication ports. This structure is shown schematically in Figure A.1. “Named” states and communicators are those which have been specifically bound to a node. The reason that a node might have multiple states and communicators is due to the use of layered libraries or the presence of multiple computational phases. For example, when the DSMC application described in Section 2.2 is coupled with the field solver from Appendix B, it is useful to have a separate set of communication ports for each, so that their messages do not interfere with one another.

In the process of binding states, the user specifies the names of routines to write and read that state to and from a communication port, to free the state, and to split the state when the node is divided. (Like states, functions are bound to names, since function pointers are not valid across heterogeneous architectures.) Together, these user-supplied routines allow a node's state to be transported from one computer to another and to be divided during granularity adjustment. Likewise, when a communicator is bound to a node, the user provides routines to partition the communicator in conjunction with the associated state. For example, in a grid-based computation, once the grid state has been divided, the communicator must be split among the child nodes and new ports created for communication along new partition interfaces.

In addition to providing communication facilities, ports also provide other functionality. SCPlib ports are similar to Unix descriptors in that the same routines can be used to write to a communication port or to a file port. This allows considerable reuse of application code, since the same routines used to read and write a data structure can be used for both communication and file I/O. A particularly beneficial reuse of application code involves checkpointing routines—procedures used to write the state of an application to a file for later resumption. These same routines can be reused for load balancing; node movement is essentially checkpointing a node through the network rather than to a file. Furthermore, communication through ports is typed; when a port is created, the writer first inputs a header describing its data type sizes, etc. The reader can then transparently perform the appropriate data transformations, if necessary. This allows communication between tasks running on heterogeneous architectures, as well as the ability to read checkpoints written on different platforms.

Another feature ports provide is global communication. A global sum, for example, is implemented by two ports at each node—an input port, into which a node's value is written, and an output port, from which the sum is read. Note that split-phase nature of global communication makes it asynchronous. A node can do useful work between inputting a value and reading the result.

Since all communication between nodes occurs through the port abstraction, the mapping of nodes to computers is transparent to the user. The library can thus move a node dynamically, using the user-supplied routines to transport the state. Similarly, the library can divide a node by calling the user state and communicator split routines. Node movement and splitting are used in conjunction to provide portable dynamic load balancing and granularity control, both of which are discussed in greater detail below.

A.2 Implementation of Dynamic Load Balancing and Granularity Control

SCPlib implements the methodology described in Chapter 2, along with the extensions from Chapters 3 through 7. A summary of that implementation is as follows:

1) **Load Evaluation:** A node's load is determined either by a user-supplied load function, or in the homogeneous case, by measuring the node's utilization (CPU time). In the heterogeneous case, the user may also provide a utilization routine to allow the capacity to be calculated dynamically.
2) **Profitability Determination:** Based on the computers' loads, utilizations, and capacities, load balancing is undertaken if the minimum desired efficiency is not met, and if the estimated time for load balancing will not exceed some fraction of the total wallclock time for the computation.
3) **Load Transfer Calculation:** Ideal load transfers are calculated using diffusion on highly-connected networks and the hierarchical balancing method on single bus and linear array networks. In both cases, vector heterogeneous versions are used.
4) **Task Selection:** Task selection either uses the cost-free, subset sum approximation algorithm, or, if a user-supplied node move cost function is given, the cost-driven, 0-1 knapsack-based algorithm. If the nodes are too coarse-grained, they are divided based on their utilizations, assuming that the user has supplied the necessary support routines in the binding of the node's states and communicators.
5) **Task Migration:** Once the new locations of nodes are determined, the nodes' states are relocated using the user-supplied routines to read/write the state from/to a port. Also, the nodes' communication ports are transparently reconfigured.
6) **Granularity Adjustment:** If any node's utilization is too high, it is divided so that the processors within its computer will be more fully utilized. (This

assumes, once again, that the user has provided the required routines.) The computation then resumes.

Load balancing is undertaken when all nodes call one of four load balancing functions. Those variants are:

1) **Simple homogeneous/static heterogeneous:** The user provides a vector load function, a minimum time efficiency (the load components are assumed to all be time-based and are subject to the same minimum efficiency), a maximum time efficiency, and the name of the function to continue the node's execution if it is relocated. Capacity is set statically outside of load balancing.
2) **Complex homogeneous/static heterogeneous:** The user provides a vector load function, the load components' units (i.e., whether the load components are computation, memory, etc.), a minimum vector efficiency, a maximum time efficiency, a node transfer cost function, and the name of the node continuation function. Capacity is static.
3) **Simple dynamic heterogeneous:** The user provides a vector load function, a vector utilization function, a minimum time efficiency (the utilization components are assumed to be time-based and are subject to the same minimum efficiency), a maximum time efficiency, and the name of the node continuation function. Capacity is determined dynamically by dividing load by utilization. If the cither the load or utilization functions are not passed in, then a static capacity is used to calculate the other. (Note that in the latter case, this is the same as the first instance above.)
4) **Complex dynamic heterogeneous:** The user provides a vector load function, the load components' units, a vector utilization function, the utilization components' units (i.e., whether the utilization is seconds, percent of memory, etc.), a minimum vector efficiency, a maximum time efficiency, a node transfer cost function, and the name of the node continuation function. Capacity is calculated dynamically, unless the load or utilization function is not passed, in which case a static capacity is used. (Without the utilization function, this case

is the same as the second case above).

In addition to the load balancing routines above, SCPlib provides a set of utilization profiling functions. These functions allow the user to measure CPU time and memory used, whether by the nodes entire execution or by a particular part of its execution. For example, the user could segregate computation time into several vector components by inserting profiling calls at appropriate points in the code. SCPlib also provides mechanisms to track changes in utilization quantities, and based on those utilization samples, to model future utilization. Finally, the library provides a set of communication cost functions based on the original location and "center of communication" concepts described in Section 4.2.1.

A.3 Related Work

The Concurrent Graph Library is the predecessor to SCPlib. While the high-level abstraction is the same as in SCPlib, the programming details are quite different. Instead of FIFO streams of data, the Graph Library uses low-latency remote procedure calls (RPCs). Similar to SCPlib, the destination of these RPCs is hidden from the user. However, the RPCs are conducted using function pointers and arguments are passed in native data format, so the library cannot operate in heterogeneous environments.

Two other libraries that provide a similar level of support for concurrent computing are CHAOS and Cilk. CHAOS provides a framework for data and control decomposition of irregular, adaptive array-based codes via index translation and communication schedules [20]. This library differs from SCPlib in that it is only appropriate for regular data structures and in that the communication structure is determined by the data reference patterns in the code. In fact, CHAOS is designed to work in conjunction with High-Performance Fortran.

Cilk provides a multithreaded programming environment with integrated load balancing [6]. In many respects, the programming model is very similar to the Con-

current Graph Library. It is designed for tree-structured computations, however, and does not fit the single-program, multiple-data style of most scientific applications.

A.4 Summary

SCPlib provides a high-level concurrent programming abstraction and implements the load balancing and granularity control mechanisms described in this thesis. The library is based on the concept of a concurrent graph of tasks that can dynamically relocate and divide themselves. Communication between tasks is implemented by FIFO streams that automatically convert data types in heterogeneous environments. Together these facilities provide dynamic load balancing and granularity control for irregular applications on a wide variety of platforms.

Appendix B Face-based Finite Element Field Solver

An important technique used in particle dynamics is that of direct simulation monte carlo (DSMC) [38]. The technique involves the direct simulation of particles, as opposed to techniques such as Navier-Stokes flow solvers, which consider gases and plasmas to be fluids. The simulation of individual particles is made feasible by the fact that the domain in which the particles move is divided into a grid. At any given point in the simulation, interactions between particles are only considered for those particles that are in the same grid cell. This spatial decoupling drastically reduces the complexity of the problem. Furthermore, within a single grid cell, collisions are calculated using a stochastic model based on the relative velocities of the particles rather than by calculating actual path intersections. Once again, the necessary computation is reduced.

For an important class of particle dynamics problems ranging from plasma reactors used in silicon wafer fabrication to satellites using ion thrusters, the incorporation of a self-consistent electric field model is essential for accurate simulation. When DSMC simulations involve ions, these particles are affected by and contribute to the electric field. Nearby conducting surfaces also affect the field. If the electric field is changing on a time scale that is much lower than that of particle motion, this field calculation can typically be performed by alternately calculating the field based on the charge density of the particles and moving the particles under the updated field. Since the field is considered to be fixed during particle movement, the electrostatic field, described by the Poisson equation, is used. This appendix describes a concurrent Poisson solver and the incorporation of that solver into the concurrent DSMC application described in Section 2.2. The solver itself is based on a novel, face-based finite element method that avoids problems associated with

traditional vertex-based methods. This solver has been validated on several test problems, which are also described herein.

B.1 Derivation

The equation for the electrostatic potential ϕ on a three-dimensional domain V with surface S is the Poisson equation:

$$-\nabla \cdot (\epsilon \nabla \phi) = \rho \tag{B.1}$$

where ρ and ϵ are the charge density and permittivity functions, respectively. Assuming ϵ is constant in the domain of interest, equation (B.1) can be rewritten as

$$-\nabla^2 \phi = \frac{\rho}{\epsilon} \tag{B.2}$$

where

$$\nabla^2 \phi = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2} \tag{B.3}$$

in Cartesian coordinates. Having solved for ϕ, the electric field $\vec{E}$ is simply

$$\vec{E} = -\nabla \phi \tag{B.4}$$

B.1.1 Overview of the Finite Element Method

In the finite element method (FEM), the partial differential equation (PDE) of interest is reformulated as a *variational problem* [23]. For boundary-value problems with the form

$$\mathcal{L}\phi = f \tag{B.5}$$

where $\mathcal{L}$ is a differential operator and f is some function, the problem is one of minimization. In particular, one describes the "energy" (i.e., error) of a solution by a "functional" F and attempts to find ϕ for which $F(\phi)$ is minimal. Such a ϕ is a

solution to the PDE for which F was designed. A specific procedure for doing this is the Rayleigh-Ritz method. First, define the bracketed inner product for real-valued problems as

$$\langle \phi, \psi \rangle = \iiint_V \phi\psi \, \mathrm{d}V \tag{B.6}$$

For an operator $\mathcal{L}$ that is self-adjoint

$$\langle \mathcal{L}\phi, \psi \rangle = \langle \phi, \mathcal{L}\psi \rangle \tag{B.7}$$

and positive definite

$$\langle \mathcal{L}\phi, \psi \rangle \begin{cases} > 0 & \text{if } \phi \neq 0 \\ = 0 & \text{if } \phi = 0 \end{cases} \tag{B.8}$$

an appropriate functional is

$$F(\phi) = \frac{1}{2}\langle \mathcal{L}\phi, \psi \rangle - \langle f, \phi \rangle \tag{B.9}$$

To make the above minimization problem tractable, one must restrict the dimensionality of the space from which candidate solutions are chosen. This can be done by approximating ϕ

$$\phi \approx \sum_{e \in V_h} \phi^e N^e \tag{B.10}$$

where ϕ^e is the value of ϕ at element e, and N^e is the element's value for N. N itself is a set of piecewise polynomial functions defined over V_h, the discretized version of V. In a three-dimensional domain, this discretization often takes the form of tetrahedral elements, which are chosen because they can capture complex geometries and because they can be readily coarsened or refined through a process called *grid adaption.*

In traditional FEM, values of ϕ in V_h are associated with vertices of these tetrahedra. Unfortunately, there are several problems with vertex-based function values. In particular, associating values with the vertices of a grid has negative implications for concurrent implementation and for the elegant handling of heterogeneous boundary conditions. The former is a result of the fact that the number of tetrahedra sharing

a vertex may be quite large. Within a partition of the grid, this is no great difficulty, but for vertices that lie on a partition boundary, it means that a large number of messages may need to be sent between adjoining partitions. Furthermore, this connectivity information may not be provided by automatic grid generation tools. An even less tractable problem is that of appropriately handling boundary conditions. In particular, for boundary-fitted grids, the boundary conditions associated with a spacecraft surface, for example, are associated with the faces of tetrahedra. However, since multiple faces share a vertex, the boundary conditions must be interpolated or local continuity requirements must be relaxed. This is nontrivial. One way to resolve these difficulties is to associate values with the centers of the faces of tetrahedra rather than the vertices thereof.

B.1.2 Face-based Finite Element Method

First, consider the choice of basis functions for a face-based scheme. Specifically, define N by a set of functions

$$N_i = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{B.11}$$

for each i and j in V_h. In other words, N_i takes the value 1 at face center i of the grid and the value 0 at all other face centers. The support of N_i consists of the two tetrahedra sharing face i. This gives the representation for $\phi(x, y, z)$

$$\phi(x, y, z) = \sum_{e \in V_h} \left(\sum_{i=1}^{4} \phi_i^e N_i^e(x, y, z) \right) \tag{B.12}$$

That is, $\phi(x, y, z)$ for an arbitrary point in V_h is a linear interpolation/extrapolation of the values of ϕ_h at the face centers of the tetrahedron containing the point (x, y, z). These interpolants/extrapolants are given by

$$N_i^e(x, y, z) = \frac{1}{6V_c^e}(d_{c,i}^e + d_{x,i}^e x + d_{y,i}^e y + d_{z,i}^e z) \tag{B.13}$$

where the coefficients $d^e_{c,i}$, $d^e_{x,i}$, $d^e_{y,i}$ and $d^e_{z,i}$ are given by the formulae

$$d^e_{c,1}\phi^e_1 + d^e_{c,2}\phi^e_2 + d^e_{c,3}\phi^e_3 + d^e_{c,4}\phi^e_4 = \begin{vmatrix} \phi^e_1 & \phi^e_2 & \phi^e_3 & \phi^e_4 \\ x^e_1 & x^e_2 & x^e_3 & x^e_4 \\ y^e_1 & y^e_2 & y^e_3 & y^e_4 \\ z^e_1 & z^e_2 & z^e_3 & z^e_4 \end{vmatrix} \tag{B.14}$$

$$d^e_{x,1}\phi^e_1 + d^e_{x,2}\phi^e_2 + d^e_{x,3}\phi^e_3 + d^e_{x,4}\phi^e_4 = \begin{vmatrix} 1 & 1 & 1 & 1 \\ \phi^e_1 & \phi^e_2 & \phi^e_3 & \phi^e_4 \\ y^e_1 & y^e_2 & y^e_3 & y^e_4 \\ z^e_1 & z^e_2 & z^e_3 & z^e_4 \end{vmatrix} \tag{B.15}$$

$$d^e_{y,1}\phi^e_1 + d^e_{y,2}\phi^e_2 + d^e_{y,3}\phi^e_3 + d^e_{y,4}\phi^e_4 = \begin{vmatrix} 1 & 1 & 1 & 1 \\ x^e_1 & x^e_2 & x^e_3 & x^e_4 \\ \phi^e_1 & \phi^e_2 & \phi^e_3 & \phi^e_4 \\ z^e_1 & z^e_2 & z^e_3 & z^e_4 \end{vmatrix} \tag{B.16}$$

$$d^e_{z,1}\phi^e_1 + d^e_{z,2}\phi^e_2 + d^e_{z,3}\phi^e_3 + d^e_{z,4}\phi^e_4 = \begin{vmatrix} 1 & 1 & 1 & 1 \\ x^e_1 & x^e_2 & x^e_3 & x^e_4 \\ y^e_1 & y^e_2 & y^e_3 & y^e_4 \\ \phi^e_1 & \phi^e_2 & \phi^e_3 & \phi^e_4 \end{vmatrix} \tag{B.17}$$

and where

$$V^e_c = \frac{1}{6}\begin{vmatrix} 1 & 1 & 1 & 1 \\ x^e_1 & x^e_2 & x^e_3 & x^e_4 \\ y^e_1 & y^e_2 & y^e_3 & y^e_4 \\ z^e_1 & z^e_2 & z^e_3 & z^e_4 \end{vmatrix} \tag{B.18}$$

and x^e_i, y^e_i and z^e_i are the coordinates of the center of face i. Note, however, that because the potentials of adjacent cells are made to be the same only at the face centers, discontinuities in the potential can exist across face boundaries. Resolution of this difficulty is beyond the scope of this work.

Because the differential operator in (B.2) is self-adjoint and positive definite, the

solution to that equation can be obtained by minimizing the functional

$$F(\phi) = \frac{1}{2} \iiint_V \left[\left(\frac{\partial \phi}{\partial x} \right)^2 + \left(\frac{\partial \phi}{\partial y} \right)^2 + \left(\frac{\partial \phi}{\partial z} \right)^2 \right] \mathrm{d}V - \iiint_V \frac{\rho}{\epsilon} \phi \, \mathrm{d}V \tag{B.19}$$

On a discretized domain V_h, this functional is approximated by

$$F(\phi) \approx \sum_{e \in V_h} F^e(\phi^e) \tag{B.20}$$

where the functional on a particular tetrahedral element e is

$$F^e(\phi^e) = \frac{1}{2} \iiint_{V^e} \left[\left(\frac{\partial \phi^e}{\partial x} \right)^2 + \left(\frac{\partial \phi^e}{\partial y} \right)^2 + \left(\frac{\partial \phi^e}{\partial z} \right)^2 \right] \mathrm{d}V - \iiint_{V^e} \frac{\rho}{\epsilon} \phi^e \, \mathrm{d}V \tag{B.21}$$

and where the potential within that tetrahedron is

$$\phi^e(x, y, z) = \sum_{i=1}^{4} \phi_i^e N_i^e(x, y, z) \tag{B.22}$$

Equation (B.21) is minimized when its derivative with respect to each of its face centers ϕ_i^e is zero

$$\frac{\partial F^e(\phi^e)}{\partial \phi_i^e} = \left[\sum_{j=1}^{4} \iiint_{V^e} \nabla \phi^e \cdot \nabla \phi^e, \mathrm{d}V \right] - \iiint_{V^e} \frac{\rho}{\epsilon} \frac{\partial \phi^e}{\partial \phi_i^e} \, \mathrm{d}V \tag{B.23}$$

Using equation (B.22), rewrite (B.23) as

$$\begin{aligned} \frac{\partial F^e(\phi^e)}{\partial \phi_i^e} &= \left[\sum_{j=1}^{4} \phi_i^e \iiint_{V^e} \nabla N_i^e \cdot \nabla N_j^e \, \mathrm{d}V \right] - \iiint_{V^e} \frac{\rho}{\epsilon} N_i^e \, \mathrm{d}V \\ &= \left[\sum_{j=1}^{4} \phi_i^e \iiint_{V^e} \left(\frac{\partial N_i^e}{\partial x} \frac{\partial N_j^e}{\partial x} + \frac{\partial N_i^e}{\partial y} \frac{\partial N_j^e}{\partial y} + \frac{\partial N_i^e}{\partial z} \frac{\partial N_j^e}{\partial z} \right) \mathrm{d}V \right] - \\ &\quad \iiint_{V^e} \frac{\rho}{\epsilon} N_i^e \, \mathrm{d}V \end{aligned} \tag{B.24}$$

Equation (B.24) can be viewed as a matrix problem

$$\frac{\partial F^e(\phi^e)}{\partial \phi^e} = A^e \phi^e - b^e \tag{B.25}$$

where

$$A^e_{ij} = \iiint_{V^e} \left(\frac{\partial N^e_i}{\partial x}\frac{\partial N^e_j}{\partial x} + \frac{\partial N^e_i}{\partial y}\frac{\partial N^e_j}{\partial y} + \frac{\partial N^e_i}{\partial z}\frac{\partial N^e_j}{\partial z} \right) \mathrm{d}V \tag{B.26}$$

and

$$b^e_i = \iiint_{V^e} \frac{\rho}{\epsilon} N^e_i \,\mathrm{d}V \tag{B.27}$$

For the basis function (B.13) defined above, (B.26) and (B.27) are

$$A^e_{ij} = \frac{V^e}{36(V^e_c)^2}(d^e_{x,i}d^e_{x,j} + d^e_{y,i}d^e_{y,j} + d^e_{z,i}d^e_{z,j}) \tag{B.28}$$

and

$$b^e_i = \frac{V^e}{4}\frac{\rho^e}{\epsilon} \tag{B.29}$$

(ρ is taken to be constant within a grid cell.)

The above definition of b^e_i only applies in the interior of the mesh, however. On the outer elements, additional complexities arise due to the incorporation of boundary conditions. The two boundary conditions that are of interest are Dirichlet (fixed potential) and Neumman (fixed normal component of the field). One can specify the latter by

$$-\vec{E} \cdot \vec{n} = q \tag{B.30}$$

where $\vec{n}$ is the unit normal to the face in question. The resulting functional is somewhat more complex

$$F(\phi) = \frac{1}{2}\iiint_V \left[\left(\frac{\partial \phi}{\partial x}\right)^2 + \left(\frac{\partial \phi}{\partial y}\right)^2 + \left(\frac{\partial \phi}{\partial z}\right)^2 \right] \mathrm{d}V - \iiint_V \frac{\rho}{\epsilon}\phi \,\mathrm{d}V - \iint_S q\phi \,\mathrm{d}S \tag{B.31}$$

As was done with equation (B.19), (B.31) can be defined for a particular tetrahedra. When this restricted functional is differentiated with respect to a surface element ϕ^e_i,

the result is

$$\frac{\partial F^e(\phi^e)}{\partial \phi_i^e} = \left[\sum_{j=1}^{4} \phi_i^e \iiint_{V^e} \left(\frac{\partial N_i^e}{\partial x}\frac{\partial N_j^e}{\partial x} + \frac{\partial N_i^e}{\partial y}\frac{\partial N_j^e}{\partial y} + \frac{\partial N_i^e}{\partial z}\frac{\partial N_j^e}{\partial z}\right) \mathrm{d}V\right] - \iiint_{V^e} \frac{\rho}{\epsilon} N_i^e \,\mathrm{d}V - \iint_{S^s} q N_i^s \,\mathrm{d}S \tag{B.32}$$

where s is the surface triangle centered around ϕ_i^e. Once again, equation (B.32) can be rewritten as a system of equations. In this case, A remains as in (B.26), and b is given by

$$b_i^e = \iiint_{V^e} \frac{\rho}{\epsilon} N_i^e \,\mathrm{d}V + \iint_{S^s} q N_i^s \,\mathrm{d}S \tag{B.33}$$

Note that for one of the most common Neumman boundary conditions, the plane of symmetry, q is zero. (I.e., there is no gradient in ϕ with respect to the normal.) Using (B.13), (B.33) can be expressed explicitly as

$$b_i^e = \frac{V^e}{4}\frac{\rho^e}{\epsilon} + \frac{\Delta^s}{\sqrt{3}} q^s \tag{B.34}$$

(q is taken to be constant within a face.)

The full system of equations to be solved is obtained by assembling all the small systems of equations for individual tetrahedra

$$A = \sum_{e \in V_h} \bar{A}^e \tag{B.35}$$

and

$$b = \sum_{e \in V_h} \bar{b}^e \tag{B.36}$$

where $\bar{A}^e$ and $\bar{b}^e$ denote the expanded matrices and vectors, respectively, that result from converting from local to global element numbering. Finally, it is necessary to adjust for Dirichlet boundary conditions. If the potential is fixed at p on face i, one can eliminate that face's equation from A and b

$$A_{ij} \leftarrow 0, \text{ for } j \in \{1..M\} \tag{B.37}$$

and

$$b_i \leftarrow 0 \tag{B.38}$$

and incorporate it into the other equations, $j \neq i$

$$A_{ji} \leftarrow 0 \tag{B.39}$$

and

$$b_j \leftarrow b_j - A_{ji}p \tag{B.40}$$

B.1.3 Conjugate Gradient Method

The conjugate gradient (CG) method is an iterative method for solving symmetric, positive definite systems of equations [3]. It has important properties of robustness and rapid convergence. The condition number of a matrix—the ratio of the largest to smallest eigenvalue—directly impacts the convergence of the CG algorithm. For the matrix resulting from FEM applied to the Poisson equation, the condition number is $\mathbf{O}(h^{-2})$, where h is the length of the shortest edge connecting two vertices of a tetrahedron. By finding a preconditioner (in this case, a matrix M^{-1} approximating A^{-1}), the equivalent system becomes

$$M^{-1}Ax = M^{-1}b \tag{B.41}$$

This system potentially has much more favorable spectral properties, and the CG algorithm will converge faster as a result. The exact choice of M^{-1}, however, is a matter for future investigation. The preconditioned CG algorithm is given as Program B.1.

B.2 Concurrent Implementation

Solving the Poisson equation on a grid involves the following:

(i) Calculating the values of ρ at the face centers.

```
i = 0
r^(0) = b − Ax^(0)
while ||r^(i)|| >= EPS · (||A|| · ||x|| + ||b||) do
        i = i + 1
        solve Mz^(i−1) = r^(i−1)
        γ_{i−1} = r^(i−1)T · z^(i−1)
        if i = 1 then
                p^(1) = z^(0)
        else
                β_{i−1} = γ_{i−1}/γ_{i−2}
                p^(i) = z^(i−1) + β_i p^(i−1)
        end if
        q^(i) = Ap^(i)
        α_i = γ_{i−1}/(p^(i)T · q^(i))
        x^(i) = x^(i−1) + α_i p^(i)
        r^(i) = r^(i−1) − α_i q^(i)
end while
```

Program B.1: Sequential preconditioned conjugate gradient algorithm.

(ii) Calculating the potential at the face centers by solving the system of equations $Ax = b$ described in Section B.1.2.

Note that the entries of A need not be calculated explicitly, but can rather be calculated "on the fly" during grid traversal. This has the advantage of interacting nicely with grid adaptation or other deformations.

Once the charge density at each face center has been calculated, a concurrent CG algorithm is used to calculate the potential. The sequential CG algorithm, Program B.1, has four points of communication in its parallel implementation. These are

(i) Calculating the matrix-vector products $Ax^{(0)}$ and $Ap^{(i)}$. This involves nearest neighbor communication to satisfy data dependencies in the vector dot products between rows of A and the $x^{(0)}$ and $p^{(i)}$ vectors. Here, "nearest neighbors" are the partitions sharing a grid face.

(ii) Preconditioning the CG iteration. This may involve no communication in the case of a simple D^{-1} Jacobi preconditioner , or considerable communication for a no-fill incomplete Cholesky preconditioner.

(iii) Calculating the vector dot product $r^{(i-1)^T} \cdot z^{(i)}$. Since r and z are mapped conformally, this requires only a global sum for a single value.

(iv) Calculating the the vector dot product $p^{(i)^T} \cdot q^{(i)}$. Once again, a simple global sum is all that is needed.

B.3 Validation

The following test problems were used to validate the Poisson solver. The simulations were run in parallel on a small network of workstations and in each case the analytic potential was calculated to the highest accuracy possible given the level of discretization in the underlying grid.

B.3.1 Infinite Conducting Plates

This problem involves two infinite conducting plates located at $z = 0$ and $z = 1$ with potentials $-\phi_0$ and $+\phi_0$, respectively. The problem was validated on a $1 \times 1 \times 1$ cube with the faces at $z = 0$ and $z = 1$ set as described and with the remaining faces set to the symmetric boundary condition. Analytically, the potential between the plates is given by

$$\phi(x, y, z) = \phi_0(2z - 1) \tag{B.42}$$

B.3.2 Conducting Box

This problem involves a $1 \times 1 \times 1$ conducting box in which the face at $z = 0$ has potential ϕ_0, and the remaining faces have zero potential. This problem was validated on a box grid with boundary conditions set exactly as described. Within the box, the

analytical function for the potential is

$$\phi(x,y,z) = \frac{16\phi_0}{\pi^2} \sum_{r,s \text{ odd}} \frac{\sinh \gamma_{rs}(1-z)}{rs \sinh \gamma_{rs}} \sin r\pi x \sin s\pi y \tag{B.43}$$

where

$$\gamma_{rs} = \pi\sqrt{r^2+s^2} \tag{B.44}$$

B.3.3 Infinite Conducting Plates with Intermediate Charge

This problem involves two infinite conducting plates located at $z = 0$ and $z = 1$ with potentials $-\phi_0$ and $+\phi_0$, respectively. The charge density between the plates is given by

$$\rho(x,y,z) = \rho_0(1-z^2) \tag{B.45}$$

This problem was validated on a $1 \times 1 \times 1$ box grid with the $z = 0$ and $z = 1$ faces set as described above, the remaining faces set to the symmetric boundary condition and the interior charge density set accordingly. Between the plates, the analytical function for the potential is

$$\phi(x,y,z) = \phi_0(2z-1) + \frac{\rho_0}{12\epsilon_0}(z^4 - 6z^2 + 5z) \tag{B.46}$$

B.3.4 Conducting Box with Interior Charge

This problem involves a $1 \times 1 \times 1$ conducting box in which the faces are have potential ϕ_0. Within the box, the charge density is given by

$$\rho(x,y,z) = 3\rho_0\pi^2 \sin(\pi x)\sin(\pi y)\sin(\pi z) \tag{B.47}$$

This problem was validated on a $1 \times 1 \times 1$ box grid with the potential on the faces and with the interior charge density set as described above. Within the box, the

analytical function for the potential is

$$\phi(x, y, z) = \phi_0 + \frac{\rho_0}{\epsilon_0} \sin(\pi x) \sin(\pi y) \sin(\pi z) \tag{B.48}$$

B.4 Integration into the DSMC Algorithm

The final parallel DSMC algorithm, with the face-based field solver integrated, is shown as Program B.2. As described in the introduction, the electrostatic field is calculated given the charge densities within each cell. Particles are then moved and collided under the influence of that field. Finally, the new charge densities are calculated, and the process repeats.

B.5 Related Work

The derivation of the face-based finite element scheme above borrows the notation from [23] and substantially follows the derivation given there for a traditional, vertex-based approach. An alternative to a face-based scheme is also presented in [23]; that scheme uses a vector, edge-based approach. While this approach works well and does not have the same inter-cell continuity problems as the face-based approach above, the degree of information sharing around an edge is not fixed, complicating concurrent implementation.

In [9], the authors consider a face-based approach as well as an edge-based method. Their face-based approach differs, however, in that the normal components are specified on each face.

B.6 Summary

This appendix has given the derivation of a face-based finite element method for calculating the electrostatic potential and electric field within a tetrahedral grid given the charge densities within that grid and any boundary conditions. The system of

equations that results is solved using the conjugate gradient method, the concurrent implementation of which is described in the context of a particle simulation application.

Work remaining to be done on the field solver includes enforcing continuity of the potential along face boundaries. Also needed is an effective preconditioner to speed convergence of the CG iteration.

```
dsmc_compute(...)
    while time not exhausted do
        calculate charge density b_i at each face i

        begin field solver

        send and recv charge densities for faces of tetrahedra
            along partition interfaces
        calculate local residual r = b - Au
        while ||r|| < EPS · (||A|| · ||x|| + ||b||) do
            solve Mz = r
            calculate local vector dot product r^T · z
            scatter/gather to get global vector dot product r^T · z
            update local vector p
            send and recv p values for faces of tetrahedra
                along partition interfaces
            calculate local matrix-vector product Ap
            calculate local vector dot product p^T · q
            scatter/gather to get global vector dot product p^T · q
            update local vectors u and r
        end while
        calculate E = ∇u for each cell

        end field solver

        while particles still moving do
            calculate acceleration g + (q/m)E for each particle
            move particles
            send and recv particles
        end while
        collide particles
    end while
end partition
```

Program B.2: Concurrent DSMC algorithm, with integrated field solver, for a single partition.

Bibliography

[1] W. Ames, *Numerical Methods for Partial Differential Equations,* New York, NY: Academic Press, 1992.

[2] E. Alard and G. Bernard, "Preemptive process migration in networks of Unix worstations," *Proceedings of the 7th International Symposium on Computer and Information Sciences,* 1992.

[3] O. Axelsson, *Iterative Solution Methods,* New York, NY: Cambridge University Press, 1996.

[4] S. Barnard and H. Simon, "A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems," *Concurrency: Practice and Experience,* vol. 6, pp. 101–117, 1994.

[5] M. Barnett, D. Payne, R. van de Geijn and J. Watts, "Broadcasting on meshes with wormhole routing," *Journal of Parallel and Distributed Computing,* vol. 35, pp. 111-122, 1996.

[6] R. Blufome, et al., "Cilk: an efficient multithreaded runtime system," *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* pp. 207–216, ACM Press, 1995.

[7] J. Boillat, "Load balancing and Poisson equation in a graph," *Concurrency: Practice and Experience,* vol. 2, pp. 289–313, 1990.

[8] N. Bowen, C. Nikolaou and A. Ghafoor, "On the assignment problem of arbitrary process systems to heterogeneous computer systems," *IEEE Transactions on Computers,* vol. 41, pp. 257–273, 1992.

[9] F. Brezzi and D. Marini, "A survey of mixed finite element approximiations," *IEEE Transactions on Magnetics,* vol. 30, pp. 3547–3551.

[10] C. Cap and V. Strumpen. "Efficient parallel computing in distributed workstation environments," *Parallel Computing,* vol. 19, pp. 1221–1234, 1993.

[11] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. Parallel and Distributed Computing,* vol. 7, pp. 279–301, 1989.

[12] M. Eisen, *Mathematical Methods and Models in the Biological Sciences: Linear and One-dimensional Theory*, Englewood Cliffs, NJ: Prentice-Hall, 1988.

[13] D. Evans and W. Butt, "Dynamic load balancing using task-transfer probabilities," *Parallel Computing,* vol. 19, pp. 897–916, 1993.

[14] D. Ferrari and S. Zhou, "An emperical investigation of load indices for load balancing applications," *Proceedings of Performance '87: the 12th International Symposium on Computer Performance Modeling, Measurement and Evaluation,* North Holland, pp. 515–528.

[15] R. Ferraro, P. Liewer and V. Decyk, "Dynamic load balancing for a 2D concurrent plasma PIC code," Center for Research on Parallel Computing Technical Report CRPC-91-6, 1991.

[16] A. Heirich and S. Taylor, "A parabolic load balancing algorithm," *Proc. 24th Int'l Conf. on Parallel Programming,* vol. 3, CRC Press, pp. 192–202, 1995.

[17] P. Henriksen and R. Keunings, "Parallel computation of the flow of integral viscoelastic fluids on a network of heterogeneous workstations," *International Journal for Numerical Methods in Fluids,* vol. 18, pp. 1167–1183, 1994.

[18] H. Hofstee, J. Lukkien and J. van de Snepscheut, "A distributed implementation of a task pool," *Research Directions in High Level Parallel Progamming Languages,* J. Banatre and D. Le Metayer, eds.. New York, NY: Springer-Verlag, 1992.

[19] G. Horton, "A multi-level diffusion method for dynamic load balancing," *Parallel Computing,* vol. 19, pp. 209–218, 1993.

[20] Y.-S. Hwang, et al., "Runtime and language support for compiling adaptive irregular problems on distributed-memory machines," *Software: Practice and Experience,* vol. 25, pp. 597–621, 1995.

[21] M. Ivanov, G. Markelov, S. Taylor and J. Watts. "Parallel DSMC Strategies for 3D Computations," *Proceedings of Parallel CFD '96*, pp. 485-492, 1996.

[22] J. Jacquez, *Compartmental Analysis in Biology and Medicine,* Ann Arbor, MI: University of Michigan Press, 1985.

[23] J. Jin, *The Finite Element Method in Electromagnetics,* New York, NY: John Wiley and Sons, 1993.

[24] G. Kohring, "Dynamic load balancing for parallelized particle simulations on MIMD computers," *Parallel Computing,* vol. 21, pp. 683–693, 1995.

[25] T. Kunz, "The influence of different workload descriptions on a heuristic load balancing scheme," *IEEE Transactions on Software Engineering,* vol. 17, pp. 725–730, 1991.

[26] C. Leangsuksun, J. Potter and S. Scott, "Data placement analysis for a distributed heterogeneous high performance computing environment," *Proceedings of the High Performance Computing Symposium,* 1995.

[27] K. Li and J. Dorband, "A task scheduling algorithm for heterogeneous processing," *Proceedings of High Performance Computing '97*, SCS, pp. 183–188, 1997.

[28] F. Lin and R. Keller, "The gradient model load balancing method," *IEEE Transactions on Software Engineering,* vol. 1, pp. 32–38, 1987.

[29] H.-C. Lin and C. Raghavendra, "A dynamic load-balancing policy with a central job dispatcher (LBC)," *IEEE Transactions on Software Engineering,* vol. 18, pp. 148–158, 1992.

[30] M. Litzkow, M. Livny and M. Mutka, "Condor: a hunter of idle workstations," *Proceedings of the 8th International Conference on Distributed Computing Systems,* 1988.

[31] S. Mason and H. Zimmerman, *Electronic Circuits, Signals and Systems*, New York, NY: John Wiley & Sons, 1960.

[32] E. Mohr, D. Kranz and R. Halstead, "Lazy task creation: a technique for increasing the granularity of parallel programs," *IEEE Transactions on Parallel and Distributed Systems,* vol. 2, pp. 264–280, 1991.

[33] F. Muniz and E. Zaluska, "Parallel load-balancing: an extension to the gradient model," *Parallel Computing,* vol. 21, pp. 287–301, 1995.

[34] J. Oden, A. Patra and Y. Feng, "Parallel domain decomposition solvers for adaptive *hp* finite element methods," *SIAM Journal for Numerical Analysis,* vol. 34, pp. 2090–2118, 1997.

[35] C. Papadimitriou, *Computational Complexity.* New York, NY: Addison-Wesley, 1994.

[36] W. Press, S. Teukolsky, W. Vetterling and B. Flannery, *Numerical Recipes in C.* New York, NY: Cambridge, 1992.

[37] M. Rieffel, S. Taylor and J. Watts. "Automatic granularity control for load balancing of concurrent particle simulations," *Proceedings of High Performance Computing '98,* pp. 115–120, Society for Computer Simulation, 1998.

[38] M. Rieffel, S. Taylor, J. Watts and S. Shankar. "Concurrent simulation of plasma reactors," *Proceedings of High Performance Computing '97,* pp. 163–168, Society for Computer Simulation, 1997.

[39] R. Samanta Roy, D. Hastings and S. Taylor, "Three-dimensional plasma paricle-in-cell calculations of ion thruster backflow contamination," *Journal of Computational Physics*, vol. 128, pp. 6–18, 1996.

[40] N. Shivaratri, P. Krueger and M. Singhai, "Load distributing for locally distributed systems," *IEEE Computer*, vol. 25, pp. 33–44, 1992.

[41] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, *MPI: The Complete Reference*, Cambridge, MA: MIT Press, 1995.

[42] J. Song, "A partially asynchronous and iterative algorithm for distributed load balancing," *Parallel Computing,* vol. 20, pp. 853–868, 1994.

[43] J. Stoer and R. Bulrisch, *Introduction to Numerical Analysis,* New York, NY: Springer-Verlag, 1993.

[44] L. Tao, B. Narahari, and Y. Zhao, "Assigning task modules to processors in a distributed system," *Journal of Combinatorial Mathematics and Combinatorial Computing,* vol. 14, pp. 97–135, 1993.

[45] S. Taylor, J. Watts, M. Rieffel and M. Palmer, "The concurrent graph: basic technology for irregular problems," *IEEE Parallel and Distributed Technology,* vol. 4, pp. 15–25, Summer 1995.

[46] M. Theimer and B. Hayes, "Heterogeneous process migration by compilation," *Proceedings of the 11th IEEE Conference on Distributed Computing Systems,* IEEE Press, pp. 18–25, 1991.

[47] R. Van Driessche and D. Roose, "An improved spectral bisection algorithm and its application to dynamic load balancing," *Parallel Computing,* vol. 21, pp. 29–48, 1995.

[48] C. Walshaw and M. Berzins, "Dynamic load-balancing for PDE solvers on adaptive unstructured meshes," *Concurrency: Practice and Experience,* vol. 7, pp. 17–28, 1995.

[49] J. Watts. "A practical approach to dynamic load balancing," Caltech master's thesis, CIT-CS-TR-95-16, 1995.

[50] J. Watts, M. Rieffel and S. Taylor. "Practical dynamic load balancing for irregular problems," *Parallel Algorithms for Irregularly Structured Problems: IRREGULAR '96 Proceedings,* Springer-Verlag LNCS, vol. 1117, pp. 299–306, 1996.

[51] J. Watts, M. Rieffel and S. Taylor. "A load balancing technique for multiphase computations," *Proceedings of High Performance Computing '97,* pp. 15–20, Society for Computer Simulation, 1997.

[52] J. Watts, M. Rieffel and S. Taylor. "Dynamic management of heterogeneous resources," *Proceedings of High Performance Computing '98,* pp. 151–156, Society for Computer Simulation, 1998.

[53] J. Watts and S. Taylor. "A practical approach to dynamic load balancing," *IEEE Transactions on Parallel and Distributed Computing,* vol. 9, pp. 235–248, 1998.

[54] J. Watts and S. Taylor. "Communication locality preservation in dynamic load balancing," *Proceedings of High Performance Computing '98,* pp. 186–190, Society for Computer Simulation, 1998.

[55] J. Watts and S. Taylor. "A vector-based strategy for dynamic resource allocation," submitted to *Concurrency: Practice and Experience,* 1998.

[56] J. Watts and S. Taylor. "Dynamic management of heterogeneous resources," submitted to *Journal of Parallel and Distributed Computing,* 1998.

[57] M. Willebeek-LeMair and A. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Trans. on Parallel and Distributed Systems,* vol. 4, pp. 979–993, 1993.

[58] R. Williams, "Performance of dynamic load balancing algorithms for unstructured mesh calculations." *Concurrency: Practice and Experience,* vol. 3, pp. 457–481, 1991.

[59] C. Xu and F. Lau, *Load Balancing in Parallel Computers,* Boston, MA: Kluwer Academic Publishers, 1997.

[60] E. Yeargers, R. Shonkwiler and J. Herod, *An Introduction to the Mathematics of Biology,* Boston, MA: Birkhauser, 1996.

[61] S. Zhou, X. Zheng, J. Wang and P. Delisle. "Utopia: a load sharing facility for large, heterogeneous distributed computer systems," *Software: Practice and Experience,* vol. 23, pp. 1305–1336, 1993.