A Practical Approach to Dynamic Load Balancing

Jerrell Watts
Scalable Concurrent Programming Laboratory
California Institute of Technology
Pasadena, California 91125

In Partial Fulfillment of the Requirements
for the Degree of Master of Science

October 4, 1995

Acknowledgments

This work was facilitated by a number of people inside and outside the California Institute of Technology. Heading the list is my thesis advisor, Stephen Taylor, whose encouragement, comments and criticisms were essential to the quality and timely completion of this work. Andy Fyfe largely implemented the Concurrent Graph Library, the applications framework into which this work was integrated. Marc Rieffel designed and implemented the initial versions of the node adaption and movement routines which were used by the load balancing code; he also wrote the necessary support routines to load balance the DSMC application. Robie Samanta Roy at the Massachusetts Institute of Technology provided insight into the PIC application, which was also targeted by this work. Alan Stagg of Cray Research, Inc., assisted in porting the Concurrent Graph Library to the Cray T3D, on which performance results were collected. Access to a 256-processor T3D was provided by the NASA Jet Propulsion Laboratory and was facilitated by Caltech. Access to a 512-node Intel Paragon was provided by the Caltech Center for Advanced Computing Research. The author was also partially supported by an NSF Graduate Research Fellowship.

Contents

Chapter 1

Introduction

1.1 Motivation

A number of trends in high performance computing have increased the need for effective dynamic load balancing techniques. In particular, particle/plasma simulations, which have recently become common, generally have much less favorable load distribution characteristics than continuum calculations, such as Navier-Stokes flow solvers. Even for continuum problems, the use of dynamically adapted grids for moving boundaries and solution resolution necessitates run-time load balancing to maintain efficiency. In the past ten years, researchers have proposed a number of strategies for load balancing [5, 7, 8, 9, 13, 14, 15, 16, 19, 20, 22, 27, 30, 31, 32, 33]. Unfortunately, the majority of these techniques have (at least) one of seven deficiencies:

1) **They are unscalable.** Many load balancing schemes rely on the availability of *global knowledge* of the load distribution in a system. The techniques in this thesis require only *local knowledge.* (I.e., a computer need only know the loads of neighboring computers in the physical network.)
2) **Their effectiveness is not theoretically analyzable.** Although many of the techniques in the literature have a strong intuitive appeal, they may result in clearly or even pathologically suboptimal load distributions. The underlying load balancing mechanism in this thesis has been subjected to rigorous mathematical analysis and has provable convergence properties.
3) **They are application-specific.** Techniques that apply to specific data decompositions and problem domains are inherently limited in their usefulness and poorly support the evolution of an application. The framework presented here is very generic, both in terms of its programming interface and its methodology, and has been applied to several problems in concurrent computing.
4) **They have only been applied to small, "toy" problems on small numbers of processors.** By not considering real applications running on large-scale machines, the proponents of other methods have failed to demonstrate the practical utility of the techniques they advocate. The work presented here is applied to two large-scale applications involving complex data structures with realistic physics and geometries.

5) **They are too complex to reasonably implement.** The complexity of many load balancing algorithms in the literature makes errors in their implementation highly likely. Effective implementations of these methods typically involve many details omitted from the description of the algorithms. The methods presented in this thesis are quite simple, and any subtleties are illuminated.
6) **They destroy communication locality and are unable to incorporate communication costs into load movement decisions.** If communicating tasks are moved arbitrarily within a machine, the benefits of resolving a load imbalance may be exceeded by higher communication costs. The techniques presented here intrinsically preserve existing communication locality. Furthermore, the decomposition of the load balancing process provides control over the degree to which this locality is maintained.
7) **They are inherently synchronous in their operation.** Many methods require that all of the computers involved go through the load balancing phase simultaneously. The strategies used in this thesis can be applied to everything from highly synchronous physics simulations, where load evolves in a relatively smooth manner, to highly asynchronous transaction processing systems and operating systems, where load injection occurs randomly[13, 14].

1.2 Problem Statement

The abstract goal of load balancing can be stated as follows:

Given a collection of tasks comprising a computation and a set of computers on which these tasks may be executed, find the mapping of tasks to computers that results in each computer having an approximately equal amount of work.

A mapping that balances the workload of the processors will *typically* increase the overall efficiency of a computation. Increasing the overall efficiency will *typically* reduce the run time of the computation—*that* is the ultimate, practical goal. (As will be shown in Chapter 4, naive balancing of the load does not necessary result in faster computation.)

In considering the load balancing problem it is important to distinguish between *problem decomposition* and *task mapping*. Problem decomposition involves the exploitation of parallelism in the control and data access of an algorithm. The result of this decomposition is a set of communicating tasks that solve the problem in parallel. These tasks can then be mapped to computers in a way that best fits the problem. One concern in task mapping is that each computer have a roughly equal workload. This is the load balancing problem, as stated above. In some cases the computation time associated with a given task can be determined a priori. In such circumstances one can perform the task mapping before beginning the computation; this is called *static* load balancing. For an important and increasingly common class of applications, the workload for a particular task may change over the course of a computation and cannot be estimated beforehand. For these applications the mapping of tasks to computers must change *dynamically*, at runtime.

1.3 Approach

A practical solution of the dynamic load balancing problem involves five distinct phases [32]:[1]

1) **Load Evaluation:** Some estimate of a computer's load must be provided to first determine that a load imbalance exists. Estimates of the work loads associated with individual tasks must also be maintained to determine which tasks should be transferred to best balance the computation.
2) **Profitability Determination:** Once the loads of the computers have been measured, the presence of a load imbalance can be detected. If the cost of the imbalance exceeds the cost of load balancing, then load balancing should be initiated.
3) **Work Transfer Vector Calculation:** Based on the measurements taken in the first phase, the ideal work transfers necessary to balance the computation are calculated.
4) **Task Selection:** Tasks are selected for transfer or *exchange* to best fulfill the vectors provided by the previous step. Task selection is typically constrained by communication locality and storage requirement considerations.
5) **Task Migration:** Once selected, tasks are transferred from one computer to another; state and communications channel integrity must be maintained to ensure algorithmic correctness.

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. This capability is severely lacking in many load balancing strategies in the literature. Indeed, many of the current load balancing algorithms fail to address several of the above concerns altogether. Most provide only the work transfer vectors in step 3 above. While this may certainly be an important contribution, it does not comprise a complete solution to the load balancing problem.

1.4 Contributions

This thesis presents a cohesive, practical load balancing framework that addresses all of the concerns in Section 1.1 and provides all of the mechanisms in Section 1.2. As part of the work, an improved work transfer vector calculation algorithm is presented, based on heat diffusion, which better scales with the degree of accuracy required. Unlike previous efforts in this arena, the techniques have been applied to two large-scale simulations. In the process, the work exposes a deficiency in *all* current load balancing strategies, motivating further work in this area.

1.5 Assumptions and Notational Conventions

This thesis makes the following assumptions regarding the architecture to which the techniques herein are applied:

[1] Actually, the authors of [32] divided the problem into four phases, merging "task selection" and "task migration" into a single step.

1) The interconnect topology is a d-dimensional mesh with $M_0, M_1, \ldots, M_{d-1}$ nodes in each dimension, respectively. The total number of nodes P is thus $M_0 \times M_1 \times \ldots \times M_{d-1}$. Within this mesh, a computer may be identified either by a Cartesian d-tuple that represents its position in each dimension or by a unique scalar identifier. The mesh may or may not have wraparound connections to form a torus.
2) The software system provides a basic message-passing library with simple point-to-point communication (send and receive operations) and basic global operations (global sum, for example).
3) Access to an accurate (milli- to microsecond level) clock is provided.

Chapter 2

Methodology

This chapter presents a design methodology for a complete load balancing solution. As outlined in Chapter 1, there are five steps comprising a practical approach to this problem. Combining the phases gives the complete abstract load balancing algorithm, which is presented in Figure 2.1.

```
load_balance(...)
    evaluate load for each task and sum task loads at each computer
    if profitable to load balance then
        calculate transfer vectors between computers
        select tasks to meet those transfer vectors
        migrate selected tasks to their new computers
    end if
end load_balance
```

Figure 2.1: Abstract algorithm for load balancing.

The following sections elaborate on each step in the above algorithm, presenting various design decisions that one encounters.

2.1 Load Evaluation

The efficacy of any load balancing scheme is directly dependent on the quality of load evaluation. Good load measurement is necessary both to determine that a load imbalance exists and to calculate how much work should be transferred to alleviate that imbalance. One can determine the load associated with a given task analytically, empirically or by a combination of those two methods.

2.1.1 Analytic Load Evaluation

The load for a task is estimated based on knowledge of the time complexity of the algorithm(s) that task is executing along with the data structures on which it is operating. For example, if one knew that a task involved merge sorting a list of 64 elements, one might estimate the load to be 384, since merge sort is an O$(N \log_2 N)$ sorting algorithm, and since $64 \log_2(64) = 64(6) = 384$. This method has the advantage that it is *potentially* very responsive to a task for which the relative workload is changing rapidly over time. In particular, knowing that some parameter which has a tremendous impact on a task's load has changed would allow the load balancing algorithm to anticipate that change ahead of time rather than responding to it after the fact. If the number of particles in a grid cell has doubled in the last time step, for example, knowledge of that fact would lead to different decisions than the assumption that the load for that cell will remain relatively constant. The disadvantage of this method is that it requires a great deal of work on the part of the programmer, and it may be quite inaccurate in any case. Specifically, in analyzing the relative running times for an algorithm, neglecting the constants hidden by the big-O notation may result in very inaccurate load estimates. Cache and paging anomalies as well as other system dependent factors can easily skew the run time for a task by a large factor.

2.1.2 Empirical Load Evaluation

One way to easily overcome the performance peculiarities of a particular architecture is to measure the load of a task directly. Typical machines provide clocks with milli- to microsecond level accuracy. One can use these timing facilities to time each task, providing accurate measurements in the categories of execution time, idle time and communication overhead. In fact, the user need not manually time the code at all. These timings can be easily taken at the library level. A message passing library could certainly be instrumented to accumulate time into various categories: any time between communication operations would be labeled as runtime, any time actually sending or receiving data would be tagged as communication time and any time waiting to receive a message would be accumulated as idle time. Thus, empirical load measurement has the advantage of being very accurate and simple. Its accuracy is limited, however, to situations in which the past load of a task is a good predictor of its future load. While the library could certainly attempt to predict changes in a tasks load based on a load history, universally good prediction requires some analysis on the part of the programmer.

2.1.3 Hybrid Load Evaluation

Coupling the accuracy of empirical load measurement with the predictive power of analytical load estimation, hybrid techniques provide the best load evaluation method. Essentially this involves using timing facilities to estimate the relevant constants in the timing calculations. One can then more accurately predict the future load for a task. For example, if the previous time step in a computation involved 1000 operations and took 15 seconds, the cost per operation is 0.015 seconds. If the next time step will involve 1500 or so operations, it will take approximately 22.5 seconds. (Of course, this is a simple linear example in which purely analytic estimation would have performed just as well. The benefit occurs when a time step involves operations of varying types, whose individual times cannot be determined a priori.)

2.2 Load Balance Initiation

For load balancing to be useful, one must first 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.

2.2.1 Load Imbalance Detection

Load imbalances can be detected in a synchronous fashion by comparing individual computer workloads to the global workload average or in an asynchronous fashion in which computers "notice" when their percent idle time exceeds a certain threshold.

Synchronous Load Imbalance Detection. Most scientific codes have inherent synchronization points. In particular, global norm calculations and other determination detection mechanisms typically involve a global sum or some other reduction operation, the results of which are checked by each task involved. These barrier operations provide an natural, clean point at which to initiate load balancing. When a barrier is initiated, the average load of all of the computers is determined. If the aggregate efficiency is below some user-specified limit, the workload is considered to be imbalanced.

Asynchronous Load Imbalance Detection. A load imbalance can also be detected in an asynchronous fashion. A task can keep track of a window of load history. If its percent utilization during this period drops below a critical threshold, it can issue a global request for load balancing. If the number of pending requests for load balancing exceeds some limit, then the computation would be deemed imbalanced. (Another, simpler criteria for designating an individual computer as underloaded would be to request load balancing whenever a computer has been idle continuously for longer than some specified time.)

2.2.2 Profitability Determination

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 work distribution. 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 100 percent 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.

2.3 Work Transfer Vector Calculation

After determining that it is advantageous to load balance, one must calculate how much work should *ideally* be transferred from one computer to another. (Deciding which tasks to move is a separate issue.) Algorithms for calculating these inter-computer transfer vectors should exhibit provable properties of correctness and termination. This section presents various algorithms for calculating work transfer vectors, along with some discussion of the properties of each.

2.3.1 Heat Diffusion

Heat diffusion provides an intuitive, correct and scalable mechanism for determining where work should be migrated in unbalanced computation. Diffusion was first presented as a method for load balancing in [7]. This work had certain limitations, as pointed out in [13]. Diffusion was also explored in [32] and was found to be superior to other load balancing strategies. A more general diffusive strategy is given in [13, 14]. This method uses a fully implicit differencing scheme to solve the heat equation on a multi-dimensional mesh to a specified accuracy.

Algorithm. The basic diffusion algorithm presented in [13, 14] is given in Figure 2.2. This method has a number of weaknesses in terms of its compatibility with the methodology of this thesis as well as its performance relative to the desired accuracy. To remedy these shortcomings,

```
diffuse(...)
    $\nu := \left\lceil \frac{\ln \alpha}{\ln \frac{6\alpha}{1+6\alpha}} \right\rceil$
    while not converged do
        $u_i^{(0)} := u_i$
        for $k := 1$ to $\nu$ do
            send $u_i^{(k-1)}$ to all neighbors $j \in \psi_i$
            receive $u_j^{(k-1)}$ from all neighbors $j \in \psi_i$
            $u_i^{(k)} := \frac{u_i^{(0)}}{1+6\alpha} + \frac{\alpha}{1+6\alpha} \sum_{j \in \psi_i} u_j^{(k-1)}$
        end for
        $u_i := u_i^{(\nu)}$
        send $u_i$ to all neighbors $j \in \psi_i$
        receive $u_j$ from all neighbors $j \in \psi_i$
        exchange $\alpha(u_i - u_j)$ units of work with each neighbor $j \in \psi_i$
    end while
end diffuse
```

Figure 2.2: The basic, first-order accurate diffusion algorithm. u_i and ψ_i denote the work load and set of neighbors, respectively, of computer i. α is desired maximum imbalance and accuracy of the diffusion algorithm.

```
diffuse(...)
    $\bar{\alpha} := \sqrt{\alpha}$
    $\nu := \left\lceil \frac{\ln \alpha}{\ln \frac{3\bar{\alpha}}{1+3\bar{\alpha}}} \right\rceil$
    $t_{i,j} := 0$ for each neighbor $j \in \psi_i$
    send $u_i$ to all neighbors $j \in \psi_i$
    receive $u_j$ from all neighbors $j \in \psi_i$
    while $u_{\max} > (1+\alpha)u_{\text{avg}}$ do
        $t_{i,j} := t_{i,j} + \frac{\bar{\alpha}}{2}(u_i - u_j)$ for each neighbor $j \in \psi_i$
        $u_i^{(0)} := u_i + \frac{\bar{\alpha}}{2}[(\sum_{j\in\psi_i} u_j) - 6u_i]$
        for $k := 1$ to $\nu$ do
            send $u_i^{(k-1)}$ to all neighbors $j \in \psi_i$
            receive $u_j^{(k-1)}$ from all neighbors $j \in \psi_i$
            $u_i^{(k)} = \frac{u_i^{(0)}}{1+3\bar{\alpha}} + \frac{\bar{\alpha}}{2(1+3\bar{\alpha})} \sum_{j\in\psi_i} u_j^{(k-1)}$
        end for
        $u_i := u_i^{(\nu)}$
        send $u_i$ to all neighbors $j \in \psi_i$
        receive $u_j$ from all neighbors $j \in \psi_i$
        $t_{i,j} := t_{i,j} + \frac{\bar{\alpha}}{2}(u_i - u_j)$ for each neighbor $j \in \psi_i$
    end while
end diffuse
```

Figure 2.3: The improved, second-order accurate diffusion algorithm. u_i and ψ_i denote the work load and set of neighbors, respectively, of computer i. α is desired maximum imbalance and accuracy of the diffusion algorithm. $t_{i,j}$ is the work transfer vector from computer i to computer j.

a few modifications were made. In particular, the actual movement of work is moved outside of the loop, which now simply accumulates transfer vectors. A variety of experiments established that satisfying many small transfer vectors was costly and ineffective versus satisfying large transfer vectors once. A more specific convergence criteria was also utilized. Finally, a second-order accurate, unconditionally stable differencing scheme was used to improve the convergence rate by allowing larger time steps to be taken without adding substantial complexity. The resulting algorithm is presented in Figure 2.3.

Both algorithms assume the mesh is a three-dimensional torus (i.e., periodic boundary conditions). To adapt the routines for non-torus meshes, simply substitute the load of the computer on the mesh's boundary for the load(s) of its nonexistent wrap-around neighbor(s). (This is equivalent to using the Neumann boundary condition $u_x = u_y = u_z = 0$.) To modify the algorithms for arbitrary, d-dimensional meshes, substitute factors of $\frac{d}{2}$ and d for each factor of 3 or 6, respectively, in the numerical calculations.

Derivation. The heat equation in three dimensions is:

$$u_t = \nabla^2 u = u_{xx} + u_{yy} + u_{zz} \tag{2.1}$$

A common method of solving (2.1) is by using finite differencing schemes [1, 4, 23, 34]. Let δ_x^2 denote the discrete Laplacian operator in the x dimension:

$$\delta_x^2 u_{i,j,k} = u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}$$

Define δ_y^2 and δ_z^2 similarly. Since the spatial discretization is arbitrary, take it to be one. Let the temporal discretization α vary so that $0 < \alpha < 1$. (2.1) can be differenced in either an *explicit* manner (sometimes referred to as a *forward in time* scheme):

$$u_{i,j,k}^{(n+1)} - u_{i,j,k}^{(n)} = \alpha(\delta_x^2 + \delta_y^2 + \delta_z^2)u_{i,j,k}^{(n)} \tag{2.2}$$

or an *implicit* manner (sometimes referred to as a *backward in time* scheme):

$$u_{i,j,k}^{(n+1)} - u_{i,j,k}^{(n)} = \alpha(\delta_x^2 + \delta_y^2 + \delta_z^2)u_{i,j,k}^{(n+1)} \tag{2.3}$$

The advantage of (2.2) is that it is very simple computationally. However, one must restrict the value of α so that $0 < \alpha \le \frac{1}{8}$. [1] (2.3) is preferable to (2.2) because it is unconditionally stable, irrespective of the value of α. The disadvantage is that (2.3) involves the solution of a system of equations. This task can be accomplished via a Jacobi iteration, as was done in Algorithm 1. As shown in [13, 14], the number of iterations required is bounded by a small constant.

Unfortunately, the temporal discretization error in (2.3) is O(α) (i.e., the scheme is *first order* accurate in time). Reducing the error to O(α^2) would allow larger time steps to be taken for the same accuracy, reducing the number of steps required for convergence. One method that does this is the Crank-Nicholson differencing scheme:

$$u_{i,j,k}^{(n+1)} - u_{i,j,k}^{(n)} = \frac{\alpha}{2}(\delta_x^2 + \delta_y^2 + \delta_z^2)(u_{i,j,k}^{(n+1)} + u_{i,j,k}^{(n)}) \tag{2.4}$$

This scheme is unconditionally stable in the same manner as (2.3), but since the scheme is centered at $n+1/2$, its truncation error is O(α^2) (i.e., it is *second order* accurate in time) [18]. Thus, we can increase our time step to $\bar{\alpha} = \sqrt{\alpha}$, for the same accuracy as a first-order scheme.

Examination of (2.4) reveals the following system of equations:

$$A^{(n+1)} = Bu^{(n)} \tag{2.5}$$

where A is a matrix with $1 + 3\bar{\alpha}$ terms along the diagonal and six $-\frac{\bar{\alpha}}{2}$ off-diagonal terms in each row/ column, and where B is a matrix with $1-3\bar{\alpha}$ terms along the diagonal and six $\frac{\bar{\alpha}}{2}$ off-diagonal terms in each row/column. Inverting A and rewriting (2.5) results in:

$$u^{(n+1)} = A^{-1}Bu^{(n)} \tag{2.6}$$

[1] Specifically, for meshes of arbitrary dimensionality d, α must satisfy $0 < \alpha \le 2^{-d}$.

(2.6) can be satisfied via a Jacobi iteration by letting $A = D - T$, where D is the diagonal of A. Therefore, (2.6) becomes:

$$(D - T)u^{(n+1)} = Bu^{(n)}$$

which, by multiplying through by D^{-1}, can be rewritten as:

$$u^{(n+1)} = D^{-1}Tu^{(n+1)} + D^{-1}Bu^{(n)}$$

This equation can be established by the following Jacobi iteration:

$$\left[u^{(n+1)}\right]^{(m+1)} = D^{-1}T\left[u^{(n+1)}\right]^{(m)} + D^{-1}Bu^{(n)} \tag{2.7}$$

where $[u^{(n+1)}]^{(0)} = Bu^{(n)}$. Note that D^{-1} has diagonal entries $\frac{1}{1+3\bar{\alpha}}$, and $D^{-1}T$ has six $\frac{\bar{\alpha}}{2(1+3\bar{\alpha})}$ off-diagonal terms in each row/column. The sum of the entries in each row/column of $D^{-1}T$ is thus $\frac{3\bar{\alpha}}{1+3\bar{\alpha}}$. Given that, the Geršgorin disk theorem implies that the eigenvalues of $D^{-1}T$ are bounded by $\frac{3\bar{\alpha}}{1+3\bar{\alpha}}$, and a theorem due to Hirsch gives the spectral radius [28]:

$$\rho\left(D^{-1}T\right) = \frac{3\bar{\alpha}}{1+3\bar{\alpha}} \tag{2.8}$$

The accuracy of the Jacobi iteration is a function of the spectral radius [28]:

$$\rho^{\nu} = \alpha \tag{2.9}$$

(2.8) and (2.9) imply that an accuracy of α will be achieved in a number of iterations specified by:

$$\nu = \left\lceil \frac{\ln \alpha}{\ln \frac{3\bar{\alpha}}{1+3\bar{\alpha}}} \right\rceil \tag{2.10}$$

Finally, to calculate the change in the transfer vectors to/from each neighboring computer, simply substitute $u_{i,j,k}$ for the load $u_{i',j',k'}$ of the neighbor in question in (2.4) and subtract (2.4) from the resulting equation. This leaves:

$$\frac{\bar{\alpha}}{2}\left(u_{i,j,k}^{(n+1)} - u_{i',j',k'}^{(n+1)}\right) + \frac{\bar{\alpha}}{2}\left(u_{i,j,k}^{(n)} - u_{i',j',k'}^{(n)}\right)$$

Proof of Consistency, Stability and Convergence. The consistency of the Crank-Nicholson differencing scheme is demonstrated in the following derivation, modified from that of von Neumann [1]:

$$\begin{aligned}
(u_{xx} + u_{yy} + u_{zz})\,|_{n+1/2} &= \frac{1}{2}\left[(u_{xx} + u_{yy} + u_{zz})\,|_{n+1} + (u_{xx} + u_{yy} + u_{zz})\,|_{n}\right] \\
&= \frac{1}{2}(\delta_x^2 + \delta_y^2 + \delta_z^2)(u_{i,j,k}^{(n+1)} + u_{i,j,k}^{(n)}) \\
&= u_t\,|_{n+1/2} \\
&= \frac{1}{\bar{\alpha}}(u_{i,j,k}^{(n+1)} + u_{i,j,k}^{(n)}) + \mathrm{O}(\bar{\alpha}^2)
\end{aligned} \tag{2.11}$$

Rearranging (2.11) gives us the familiar form in (2.4) above. Notice that as α goes to zero, the discretization error in (2.11) also goes to zero. Therefore, (2.4) is said to be *consistent* with the diffusion equation given in (2.1).

As shown in equation (2.6), the Crank-Nicholson scheme is equivalent to the repeated multiplication of the original vector $u^{(0)}$ by the matrix $A^{-1}B$. If the eigenvalues of $A^{-1}B$ are all less than one in absolute value, then the iteration (2.6) is stable [1, 11]. The proof of upper bound on these eigenvalues for the three-dimensional case follows in a manner similar to that for the one-dimensional case given in [1].

First, note that A and B can be rewritten as follows:

$$A = I + \bar{\alpha}C \tag{2.12}$$

$$B = I - \bar{\alpha}C \tag{2.13}$$

where C is a matrix with 3's along the diagonal and six off-diagonal $-\frac{1}{2}$'s in each row/column. Let λ represent an eigenvalue of $A^{-1}B$, and x the corresponding eigenvector. Each must satisfy:

$$(B - A\lambda)x = 0$$

Substitution of A and B into the above according to (2.12) and (2.13) yields:

$$\left(\frac{1-\lambda}{1+\lambda}I - \bar{\alpha}C\right)x = 0$$

Examination of this reveals that the eigenvalues μ of C can be expressed by:

$$\mu = \bar{\alpha}^{-1}\frac{1-\lambda}{1+\lambda} \tag{2.14}$$

The eigenvectors of C are the same as those for $A^{-1}B$. Solving (2.14) for λ produces:

$$\lambda = \frac{1-\bar{\alpha}\mu}{1+\bar{\alpha}\mu}$$

From the definition of C and the fact that $|C - \mu I| = 0$, one gets:

$$\mu_{i,j,k} = 2\left(\sin^2\frac{i\pi}{M_0} + \sin^2\frac{j\pi}{M_1} + \sin^2\frac{k\pi}{M_2}\right) \tag{2.15}$$

for $i \in \{0..M_0 - 1\}$, $j \in \{0..M_1 - 1\}$ and $k \in \{0..M_2 - 1\}$. By substituting (2.15) into (2.14), the eigenvalues of $A^{-1}B$ are finally obtained:

$$\lambda_{i,j,k} = \frac{1 - 2\bar{\alpha}\left(\sin^2\frac{i\pi}{M_0} + \sin^2\frac{j\pi}{M_1} + \sin^2\frac{k\pi}{M_2}\right)}{1 + 2\bar{\alpha}\left(\sin^2\frac{i\pi}{M_0} + \sin^2\frac{j\pi}{M_1} + \sin^2\frac{k\pi}{M_2}\right)} \tag{2.16}$$

(Note that this result is similar to that obtained by Fourier analysis of the two-dimensional case in [34].) From (2.16), one can see that for any $\lambda_{i,j,k}$ and all $\bar{\alpha} = \sqrt{\alpha} > 0$, $|\lambda_{i,j,k}| \leq 1$. Hence, the method is stable.

Given that the Crank-Nicholson differencing scheme (2.4) is consistent and stable, then it is also convergent, by the Lax equivalence theorem [1].

2.3.2 Gradient Methods

Gradient load balancing methods have been explored extensively in the literature [20, 22, 32]. The basic idea is that each computer classifies itself to be either lightly loaded, properly loaded or heavily loaded by comparing its load to predetermined thresholds, called “low” and “high water marks.” Lightly loaded computers inform their neighbors of their status. This information is propagated to any overloaded computers within a fixed radius (typically the dimensions of the network). Once propagated, a gradient map is constructed to route work from overloaded to underloaded computers. As pointed out in [22, 32], this model may result in over- or undertransfers of work to lightly loaded processors. Transferring too much work is a very serious problem. For example, if a computer has twice the average workload of the other computers, the computation can have a maximum efficiency of fifty percent. On the other hand, if a computer has half as much work as the average, the efficiency can still be as high as $(P - \frac{1}{2})/P$, where P is the number of processors. The authors of [22] 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. This has the advantage of eliminating much of the cost of transferring load via intermediate computers. Despite these improvements, the gradient model is still fundamentally flawed. At best, it is little more than a heuristic. At worst, it can lead to very undesirable behavior. And while it does have the scalability of diffusive strategies, it has been shown to be inferior in its performance [32].

2.3.3 Hierarchical Methods

In hierarchical or multi-level techniques, computers are initially organized into (two) large groups which are balanced between one another. These groups are then recursively divided and load balanced. (The loads of the groups at each level are first determined by having the computers group themselves recursively, summing the total loads of the subgroups to get the load of the new group they comprise.) Hierarchical techniques are explored in [16, 32].[2] The hierarchical method does achieve effective load balance, and it does so in a number of steps logarithmic in the number of computers. However, the algorithm inherently neglects to attempt to minimize the distance and volume of work transferred to achieve load balance. For communication intensive applications the resulting disruption of existing locality in task mapping may have a severe impact. In such situations, the diffusive strategy better preserves existing communication locality. Even when communication is not an issue, the diffusive strategy has been shown to perform as well, with less work transfer [32].

2.3.4 Domain-Specific Methods

The literature contains a number of load balancing methods for specific problem areas. While these methods certainly lack the generality of the techniques above, they *may* offer better per-

[2]The algorithm presented in [16] is referred to as a “new,” “diffusive” method, when in fact it is neither. The same algorithm is presented in [32], and neither presentation bears any resemblance to a diffusive technique such as that given above.

formance under certain circumstances.

Recursive Bisection Methods. Recursive bisection methods operate by recursively partitioning 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 [2, 33], although formulations appropriate for dynamic domain repartitioning do exist [30, 31]. 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 in Chapter 1. The combination of these limitations makes such techniques unsuitable for use in a general purpose load balancing framework.

"Strip" Methods. Heuristics for load balancing particle simulations (relevant to this thesis because of the two applications targeted in Chapter 4) are presented in [9, 19]. Both of these methods partition the physical problem domain in one dimension, then dynamically adjust the partition boundaries to track changes in particle density. The algorithm in [9] uses a global particle count to determine how many particles each computer should have for good load balance. In some sense, the algorithm is little different from the hierarchical methods presented above. In [19], a method which very coarsely approximates diffusion is used. Basically, computers repeatedly transfer fixed, small amounts of work to underloaded neighbors until a balanced state is reached. It should be noted that the author makes dubious claims regarding the superiority of this technique to diffusive methods based on improper comparisons. In short, the author compares the worst case convergence of the diffusive strategy to an average case problem for the strip method. In any case, due to the lack of rigorously analyzable convergence properties, this algorithm was deemed inappropriate for general use.

2.4 Task Selection

Once work transfer vectors between computers have been calculated, it is necessary to determine which tasks should be moved to meet those vectors. The quality of task selection directly impacts the ultimate quality of the load balancing. Hence, it is worthwhile to investigate the many options that are available.

2.4.1 Task Transfer Options

There are two options in satisfying a transfer vector 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. The former is cheaper computationally, because there many fewer options to consider. However, if the average task's workload is high relative to the magnitude of the transfer vectors, it may be very difficult to achieve a good load balance. On the other hand, by exchanging tasks one can potentially satisfy small transfer vectors by swapping two tasks of roughly the same load. The relative advantages of one method over the other may depend to a large extent on the granularity at which a computation has been decomposed. If many tasks (say twenty or more) are mapped to each computer, the transfer-only option may perform well given the relatively small workload of each task, and it will certainly have a much lower cost than considering all the tasks on any pair of computers between which an exchange could be made. If only a few tasks are located at each computer, exchanges will likely be necessary when the transfer vectors are small.

2.4.2 Task Selection Algorithms

The problem of selecting which tasks to move is **NP**-complete, since it is simply the subset sum problem [6]. Thus, exhaustive searches are necessary to decide the optimal solution, but much cheaper approximation algorithms can often be used to great efficacy. In general, exhaustive searches offer a large benefit only when the number of tasks is relatively small (less than 10 per computer). Under such circumstances, the cost of exhaustive search is fully acceptable (2^{10} or even 2^{20} possibilities can be considered in less than one second on most modern computers). Note that for large numbers of tasks (more than 20 per computer), each task comprises an average of five percent of a computer's total workload. Thus, one would expect to be able to find a fairly good set of tasks to move at relatively little cost with an approximation algorithm.

Best Fit/Exhaustive Searches. The best and most costly way of deciding which tasks to transfer or exchange is by exhaustively considering all possible subsets of tasks that one could move. For a set of N elements, there are 2^N subsets, so the cost of this search grows exponentially in N. However, the search can be implemented very efficiently by using bit vectors to represent the subsets and tight loops to calculate the subsets' costs, resulting in a relatively small time per iteration. Even so, as the number of tasks per computer becomes large, the cost of an exhaustive search quickly becomes prohibitive. Fortunately, since the relative load of each task is decreasing, it is much easier to satisfy a given transfer vector using partial searches or approximation methods.

First Fit. In the transfer-only case, the first fit algorithm simply traverses a list of tasks, choosing a particular task for transfer if its load does not exceed the unsatisfied portion of the transfer vector. This is an O(N) algorithm, where N is the number of task from which work is being transferred. The technique may work well when the number of tasks per computer is large but may be arbitrarily poor in general.

There is also a first fit algorithm for the exchange case. The list of tasks on the computer from which work is being transferred is traversed. Tasks that "fit" in the remaining transfer vector are marked for transfer as above. If a task is too large, the algorithm attempts to offset it by using a first fit transfer of tasks in the opposite direction. This algorithm requires O(MN) time, where M is the number of tasks on the computer from which work is being transferred, and N is the number of tasks on the computer to which work is being transferred.

Subset "Trimming." The authors of [6] present a polynomial time approximation algorithm for the subset sum problem. As given, the method can only be applied to one-way transfers and not exchanges. Essentially what the algorithm does is build up a list of possible candidates for the optimal subset, removing those subsets whose sums are "close" to those of other subsets. (The specification of "close" is up to the user.) By restricting the maximum number of subsets under consideration to be proportional to the number of tasks under consideration, the algorithm achieves polynomial running time. Ultimately, the algorithm will find a subset that has a sum "close" to that of the optimal subset.

Simulated Annealing. Simulated annealing is loosely based on a physical situation that occurs in state transitions such as crystallization. For example, if certain liquids are allowed to cool slowly, the molecules comprising them will form a crystal representing the minimum possible energy configuration. If the same liquid is cooled rapidly, the end product will be something other than the minimum-energy crystal. In this context, greedy strategies such as first fit "cool" the problem too quickly. By more slowly and thoroughly exploring the available options, a better solution can often be found.

The fundamental algorithm in simulated annealing is the Metropolis algorithm [23]. Applying this technique to the subset sum problem requires that the following be provided:

- **Current Configuration:** Let C be the set of tasks which have been selected for transfer/exchange. Initially, this set may be empty or may be provided by another algorithm such as first fit.
- **Rearrangement Procedure:** Let R be a routine which permutes the current configuration in some manner in an attempt to improve the solution. For example, one might randomly replace a given task or subset of tasks with another task or subset of tasks. The magnitude of the change should also be a function of the value of the current "temperature" T.
- **Energy Function:** The configuration energy E is simply the difference between the transfer vector and the transfer that would result from that configuration.
- **Annealing Schedule:** A procedure S must be provided to adjust T when progress ceases to be made at its current value.

The Metropolis algorithm repeatedly creates new configurations using R. The new configuration $R(C)$ is made the current configuration with a probability $\exp\{-[E(R(C))-E(C)]/kT\}$, where k is Boltzmann's constant. (Note that if $E(R(C)) < E(C)$, the current configuration is always changed. The important thing is that it is possible to change to a higher energy configuration.) When the current configuration has failed to change for some time, T is updated via S. When T reaches some minimum value, the algorithm stops.

2.4.3 Other Constraints

Other concerns may constrain task transfer options. In particular, transferring certain tasks may exceed the available memory on the computers involved. Similarly, movement of certain tasks may increase communication costs more than it improves the load balance. In such circumstances, it may be necessary to consider only a subset of the tasks on the computers in question. While this may diminish the quality of transfer vector fulfillment, it will also reduce the cost of task selection since there are fewer options to consider.

Memory Requirements. The memory consumed by the state of a task must necessarily constrain its movement. Many current parallel architectures provide no support for virtual memory, so there is a hard, fixed limit on the amount of available memory at each computer. Even if virtual memory is provided, the cost of exceeding the amount of physical memory may well surpass the cost of the load imbalance. The best way to avoid this problem is for the user to allocate and free memory via intermediate routines such as **task_malloc()** and **task_free()**. This would allow one to keep track of the total amount of memory allocated to a particular task, so that it would only be moved to computers on which its state would fit.

Communication Locality. While the diffusion algorithm naturally preserves the existing communication locality of an application, it may be the case that over time, a task will migrate a considerable distance from the computers on which its neighbors in the communication graph reside. Under such circumstances, the delays caused by additional network contention may offset the benefits of being able to freely move the task about. By keeping track of the frequency and volume of communication over each of the channels, one can estimate the cost of having the tasks at each end of the channel reside on the same computers, neighboring computers or computers further apart in the network. Depending on the change in communications cost resulting from a move, one may wish to restrict the computers to which a task can be moved or even eliminate a task from consideration altogether.

2.5 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, including any incoming messages in communication channels. Transportation of a task's state typically requires assistance from the application, especially when complex data structures such as linked lists or hash tables are involved. The migration protocol may also need to handle errors such as memory allocation failures on the destination computer and provide a fall-back protocol for such circumstances.

Like the earlier phases of load balancing, task migration can be implemented in a synchronous or asynchronous fashion. This issue is primarily determined by maintenance of state and channel integrity. Certainly a synchronous implementation simplifies the need for handling messages in transit. In any case, a task must only be moved when its state is in a form consistent with any user-provided routines to transport data structures. Many applications have transient state information in certain phases of the computation that may be difficult and/or costly to transport. Allowing the user to specify when a task selected for movement should actually be moved is certainly an easy way to avoid these difficulties. The issue then becomes whether all tasks must declare their readiness before migration begins, or whether tasks are moved asynchronously on an individual basis.

2.6 Summary

Building a load balancing algorithm requires that one instantiate a mechanism for each phase detailed above. In doing so, one can custom tailor an implementation for the particular requirements of the target application(s). At each stage, one has the option of preserving or destroying existing asynchrony and/or communications locality. For example, if one has a highly asynchronous application which needs to be load balanced very infrequently, one might choose to implement the load imbalance detection phase asynchronously, to avoid unnecessary overhead. However, one may decide that the low cost of synchronizing at the few times when load balancing is necessary is more than offset by the simpler implementation of a synchronous approach at the transfer vector calculation and task selection/migration phases.

Subsequent chapters in this thesis present a simple load balancing implementation based on the methodology explained here. Because of the nature of the applications to which the algorithm was applied, a synchronous, diffusive strategy was used. Details of this implementation are given in Chapter 3, and the results of its application to two large-scale physics codes follows in Chapter 4.

Chapter 3

Implementation

This chapter presents an implementation of load balancing based on the methodology given in Chapter 2. The techniques were implemented in the context of the Concurrent Graph Library, an applications framework which has been successfully applied to a number of large-scale irregular problems [29]. The chapter first presents an overview of the functionality of the Concurrent Graph Library, then gives details on the specific instantiations of the load balancing phases outlined in the previous chapter.

3.1 The Concurrent Graph Library

The Concurrent Graph Library (hereafter referred to simply as the Graph Library) provides an ideal framework in which to implement a practical load balancing algorithm. The Graph Library eliminates the explicit mapping of work to processors found in prevalent concurrent programming systems such as PVM and MPI [10, 12].[1] Under the Graph Library, the computational entities are called "nodes," which can be thought of as contexts of execution. Low-latency remote procedure calls (RPC's) are made over unidirectional channels from one node to another. The mapping of nodes to computers is controlled by the Graph Library and is hidden from the user by these communication channels. Thus, because the mapping of work to computers is not explicit, it is possible to dynamically change this mapping, so long as the user provides some mechanism for packing/unpacking the context of a node (i.e., the node's state) into/from a contiguous buffer. Figure 3.1 shows an example computational graph and its mapping to a set of computers. Figure 3.2 gives a schematic representation of the software structure of an individual node.

The above functionality is implemented on top of a computer-to-computer RPC mechanism. Thus, once the low-level RPC has been implemented on a particular machine, the rest of the library is immediately portable. As a result, the time to port an application under the Graph Library is typically a matter of hours instead of weeks.

Basic Functionality. The Graph Library provides two basic functions for communication and synchronization of nodes:

[1] Actually, both of these systems support the some level of abstraction in the mapping of tasks to computers. This capability is ignored in many implementations, however.

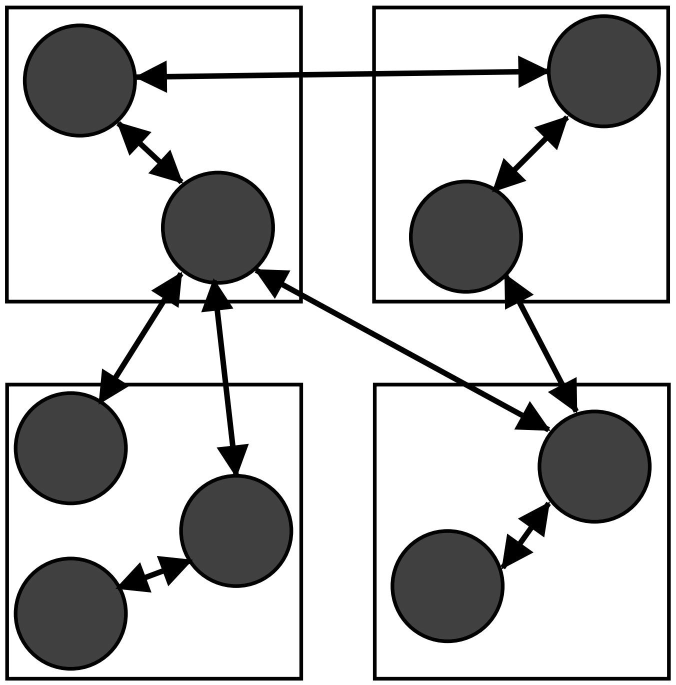

Figure 3.1: A computational graph of nine nodes (represented by shaded discs) mapped onto four computers (represented by squares).

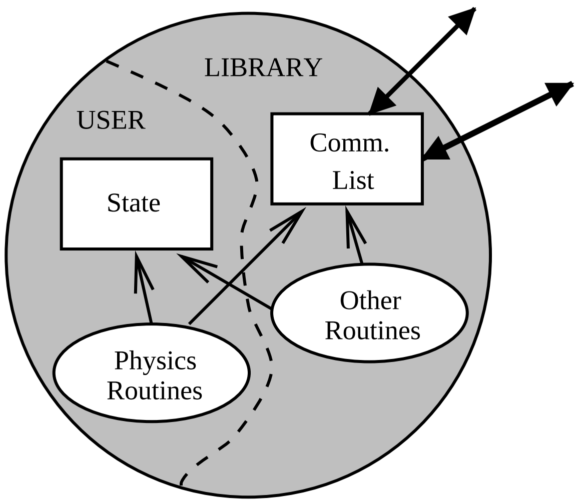

Figure 3.2: The software structure of a node. The user portion is comprised of the node's state and routines that act upon it. The library portion is comprised of the communication list and auxiliary routines such as load balancing and visualization functions.

- *f(len, args)@chnl*: This statement executes the function *f* with arguments *args* of length *len* under the context of the node to which the channel *chnl* leads. (A sketch of the implementation of this is given in Figure 3.3.)
- **barrier(***redfun, globfun, barfun, contfun, len, args***)**: When all nodes have called this routine, their input messages *args*, which are of length *len*, are combined using provided reduction operator *redfun* (if any). The global function *globfun* is called on computer zero with the result of the reduction (this provides an easy way to output a termination condition or a "beginning of time step" message, for example). After this function completes, the barrier function *barfun* (if any) is called at each node with the result of the reduction operation. Finally, after the barrier function has completed at each node, the continuation function *contfun* (if any) is called on each node. (A schematic of the execution of **barrier()** is shown in Figure 3.4.)

The function executed using the node-to-node RPC executes to completion without preemption or suspension. This eliminates the need for locks to ensure exclusive access to data structures, for example. However, due to the lack of implicit sequencing in the model, additional effort is required to ensure that such accesses happen in the proper order. To provide stronger sequencing, such as guaranteeing that data structures have been initialized at each node before the computation functions are begun, the library provides the **barrier()** function. The peculiar form of the **barrier()** function is a consequence of its typical use in scientific applications. A typical scientific application is of the form given in Figure 3.5. Such an algorithm would be implemented under the Graph Library as shown in Figure 3.6.

Node Adaptation and Movement Facilities. The Graph Library provides basic facilities to support load balancing and granularity control. This functionality includes:

- **node_move()**: The specified node is moved from one computer to another.
- **node_split()**: The specified node is split into two nodes.
- **node_merge()**: The specified pair of nodes are merged into a single node.

```
func(arglen, arg)@chnl
    msglen := sizeof({chnl.node,func,arglen,arg})
    msg := {chnl.node,func,arglen,arg}
    node_dispatch(msglen, msg)@chnl.comp

node_dispatch(msglen, msg)
    {node,func,arglen,arg} := msg
    set_node_context(node)
    func(arglen, arg)
end node_dispatch
```

Figure 3.3: Schematic implementation of the node-to-node RPC.

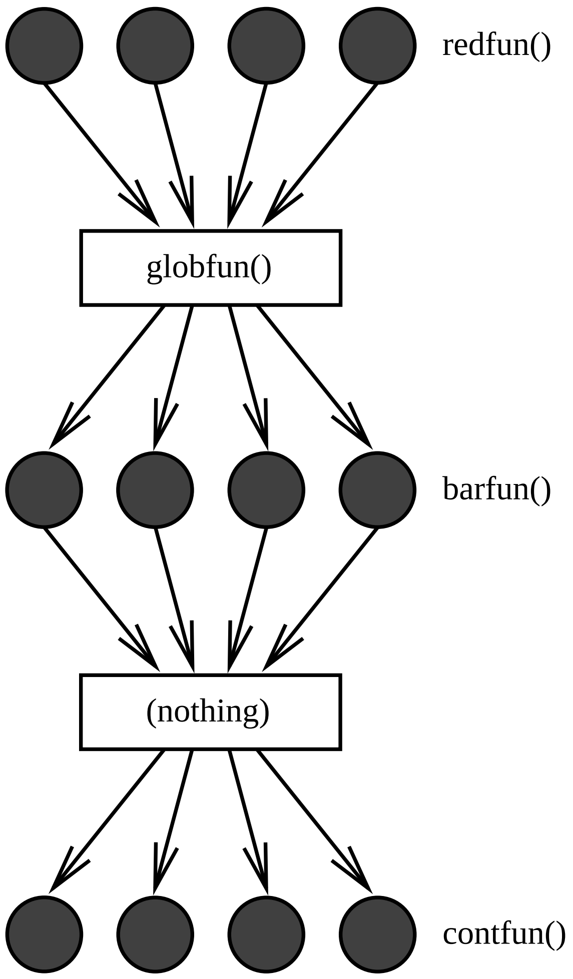

Figure 3.4: Schematic representation of **barrier()** execution.

These functions were designed to be completely local in nature. That is, the movement of a node from one computer to another affects only those two computers plus any computers containing nodes which communicate with the node that was moved. Similarly, the node division and combination routines affect only the computer on which the node(s) reside plus the computers containing neighboring nodes in the graph. For simplicity of implementation, all of these routines are currently executed within a **barrier()** call for correctness; there is no semantic restriction on when the routines are executed, however.

The **node_move()** function moves the given node from its current computer to the specified computer, as shown schematically in Figure 3.7. The user must provide the following support functions:

- **state pack:** This function packs the node's state into a contiguous buffer for transport to the destination computer.
- **state unpack:** Once the node's state has arrived at its new location, this function is used

```
partition(...)
    **initialize** data structures
    do
        **communicate** with neighbors
        **compute** next iteration
        **gather** termination information
    while not all done
    **finalize** computation
end partition
```

Figure 3.5: Abstract algorithm for typical scientific application.

to restore the state by unpacking data structures from a buffer.

- **state free:** This function frees the data structures comprising the node's state on the original computer.

The node's communication list is handled automatically. The original node becomes a "ghost" node, forwarding any incoming function invocations to the node's new location and notifying the originators of those messages to re-target their channels. When the ghost node has received acknowledgments of this change from each of its neighbors in the graph, it disappears. In this way, a node can be moved in the middle of a computation, without fear of message loss, and updates are made on a "need-to-know" basis only.

The **node_split()** function divides a given node into two nodes—a process illustrated in Figure 3.8. This function requires that the user provide the following support routines:

- **state split:** This function takes the original node's state and produces two separate states, which are incorporated into the new nodes.
- **channel split:** This routine re-maps the communication channels of the original node, determining which are inherited by each of the nodes into which it was split.
- **state free:** This function frees the state of the original node.

Messages destined for the original node are handled automatically. Upon dividing, the original node becomes a ghost node, forwarding each RPC that arrives to the appropriate child node. As each message is forwarded, the ghost node tells the sender to re-route its communication channels to point to the appropriate child node. When the re-mapping of all the channels has been acknowledged, the ghost node disappears.

The **node_merge()** function combines two given nodes into a single node. Figure 3.9 illustrates this process. The user must provide the following functions for this routine:

- **state combine:** This function combines the states of the two nodes into a single state, which is incorporated into the final node.
- **channel combine:** Given the channels of the original two nodes, this routine produces the set of channels for the new node.

```
partition(...)
    sent := false
    num_recvd := 0
    barrier(___, ___, **initialize**, send_msgs, 0, NULL)
end partition

send_msgs(...)
    for each channel c to a neighbor do
        recv_msg(___, ___)@c
    sent := true
    if num_recvd = num_neighbors then
        sent := false
        num_recvd := 0
        do_compute(...)
    end if
end send_msgs

recv_msg(...)
    num_recvd := num_recvd + 1
    if sent and num_recvd = num_neighbors then
        sent := false
        num_recvd := 0
        do_compute(...)
    end if
end recv_msg

do_compute(...)
    **compute**(...)
    barrier(and_op, ___, check_termination, send_msgs, sizeof(is_done), is_done)
end do_compute

check_termination(is_done)
    if is_done then
        **finalize**(...)
        exit()
    end if
end check_termination
```

Figure 3.6: Graph Library implementation of a typical scientific application.

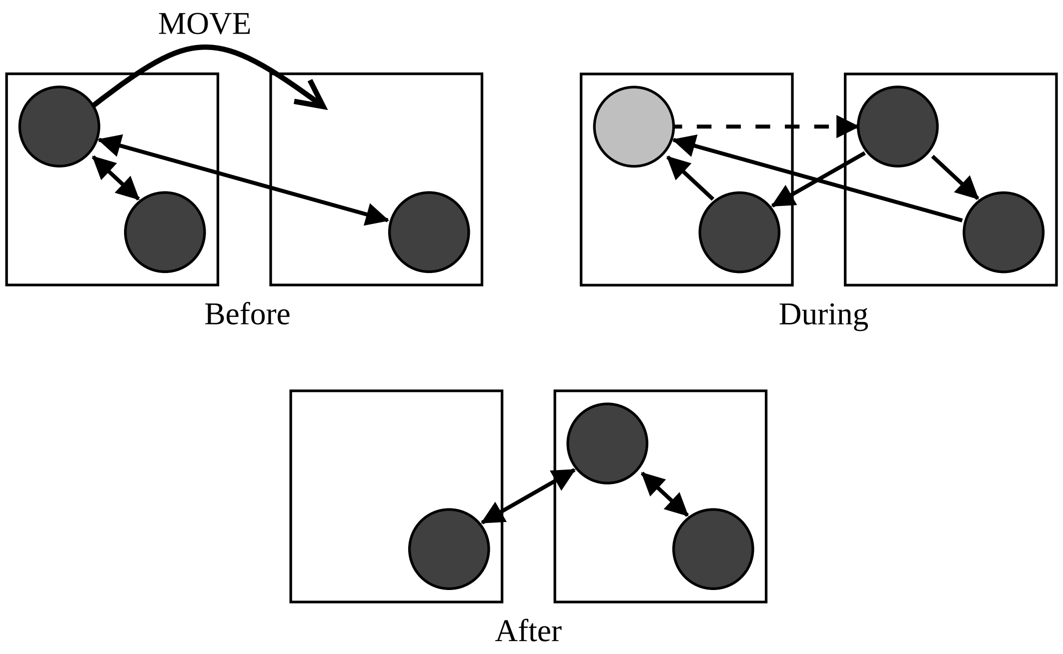

Figure 3.7: Phases of **node_move()**.

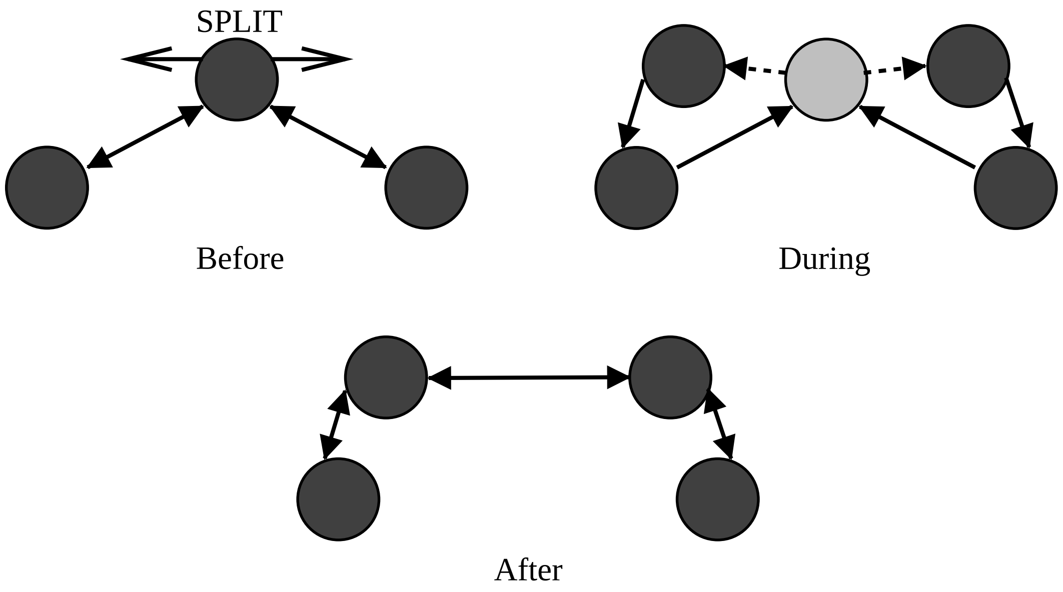

Figure 3.8: Phases of **node_split()**.

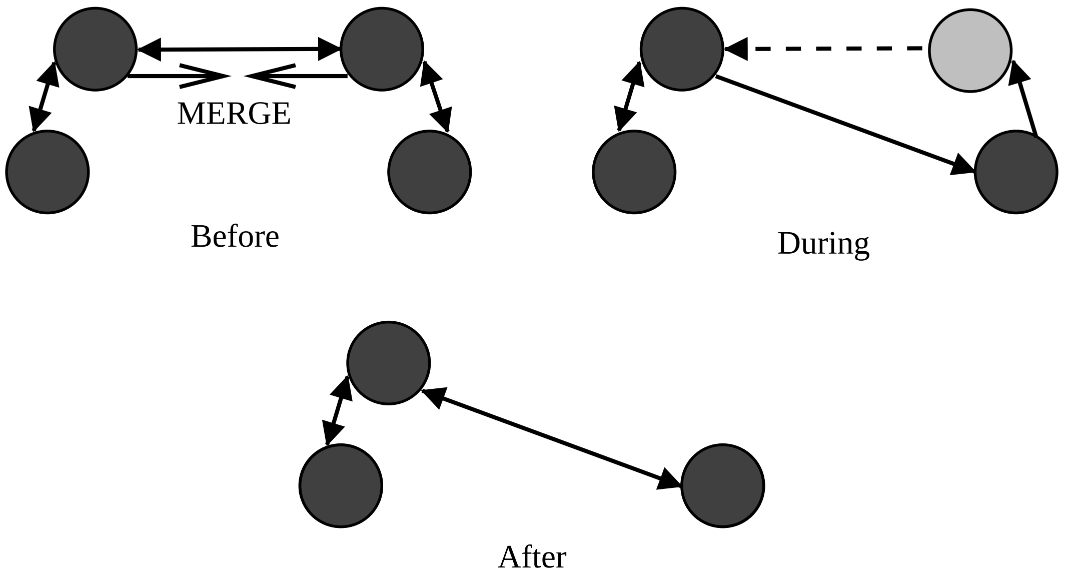

Figure 3.9: Phases of **node_merge()**.

- **state free:** This function frees the states of the original pair of nodes.

When the two nodes merge, one of them becomes a ghost node. The other node remains as-is, incorporating the other node's state information into its own. When a procedure invocation arrives at the old, now-ghost node, it forwards the message to the new, now-combined node, after which it tells the sender of the message to change its communication channels appropriately. When these channel modifications have been acknowledged, the ghost node disappears.

Timing Routines. The Graph Library also provides a timing function at the computer level:

- **computer_time()**: Returns the most accurate measurement of the time since execution began. The value reported is in seconds and fractions thereof.

This function is the basis for load measurement as presented in the next section.

3.2 Implementation Specifics

This section presents instantiations of the load balancing components presented in the previous chapter as well as implementation specifics for the Graph Library that motivate the associated design decisions.

3.2.1 Low-Latency RPC Mechanism

As discussed in Section 3.1, the Graph Library itself is an abstraction of a low-latency, RPC-based programming model found in programming systems such Message-Driven C (MDC) [21].

The RPC model used in the Graph Library allows the user to invoke a function with arguments on a remote computer:

- *f(len, args)@comp*: This statement causes the remote invocation of the function *f* with arguments *args* of length *len* on computer *c*.

As with the node-to-node RPC, a function invoked remotely on a computer executes to completion without yielding flow of control. Once again, the user must guarantee the order but not exclusivity of data structure access by such functions.

On architectures such as the J-Machine, this remote invocation can be implemented at the hardware or microkernel level. On more traditional multicomputers, the computer-to-computer RPC can easily be implemented using message passing routines, as shown in Figure 3.10.

3.2.2 Load Evaluation

The Graph Library and the low-latency RPC mechanism upon which it is layered provide excellent frameworks for accurate load measurement. The measurement of load at the computer and node level is described below.

Computer Load Evaluation. Measuring the load of a computer is a fairly simple task. As shown above, a computer simply receives function pointers and arguments, calling the former with the latter. When a computer is waiting for a RPC, it is idle. When it is executing a function, it is busy. Since remotely-invoked functions have only a single point of entry and exit, timing is easily accomplished by bracketing the call to the function in the dispatch loop with calls to **computer_time()**. A computer's load information can be accessed by the load balancing routines to determine when and how to balance the load.

The Graph Library also provides the functionality to time individual functions. This is accomplished by storing a function pointer and its associated timer in a hash table. (Note that several functions may share a single time accumulator; thus it is possible to time *classes* of functions easily.) Whenever a function is initiated via an RPC, it is looked up in the hash table, and if found, the time for its execution is added to the appropriate accumulator. In this manner, it is possible

```
func(arglen, arg)@comp
    send {func,arglen,arg} to comp

dispatch_loop()
    while true do
        recv {func,arglen,arg}
        func(arglen, arg)
    end while
end dispatch_loop
```

Figure 3.10: Schematic implementation of the computer-to-computer RPC.

```
dispatch_loop()
begin
    while true do
        idle_start := computer_time()
        recv {func,arglen,arg}
        idle_count := idle_count + computer_time() - idle_start
        busy_start := computer_time()
        func(arglen, arg)
        busy_end := computer_time()
        busy_count := busy_count + busy_end - busy_start
        lookup(comp_func_counters, func) := lookup(...) + busy_end - busy_start
    end while
end
```

Figure 3.11: Schematic implementation of computer load measurement.

to extract non-user execution time (such as that due to load balancing) from the busy time for a computer. Another way to extract non-user execution time is to sum the execution times for the nodes on a computer and subtract that total from the execution time for the computer as a whole. Once again, the difference is non-user time.

A schematic of the instrumented computer-to-computer RPC dispatch routine is given in Figure 3.11.

Node Load Evaluation. As with measuring the load of a computer, measuring a node's load is easy to do. Once again, one simply has to accumulate the time that a node spends executing functions, since there is no preemption or suspension of a function called remotely. This load information is individually associated with each node and will be used to determine which nodes should be moved to alleviate a work imbalance.

As with computer-to-computer RPC's, the Graph Library provides the capability for the user to time individual RPC's to nodes. (The function measurement routines for computers cannot be used, because an RPC to a node is implemented by a node dispatch function, which is itself an RPC to the appropriate computer.) Once again, this is done by storing function pointers and time counters in a hash table, adding to the appropriate counter whenever one of its associated functions is called. Depending on whether the hash table used is unique to a particular node or is global for all of the nodes within a computer, functions (or classes thereof) can be timed on a node-by-node basis or across all of the nodes within a computer, respectively. The Graph Library provides both options.

The resulting instrumented dispatch routine for node-to-node RPC is given in Figure 3.12.

The Graph Library also provides a basic set of node load functions which may be supplied to the load balancing routine. One returns the total run time for the given node since the computation began. The other returns the total run time for the node over a finite window of load balancing steps. A larger window dampens out spurious changes in the load of a node, prevent unnecessary work movement. A smaller window allows a more rapid response to a rapidly changing work load. Users may choose to use one of these two routines or provide one of their own as outlined in the following section.

The above load measurement techniques form a good basis for load balancing decisions only when past load is a good predictor of future load (i.e., the loads of nodes and the computers to which they are mapped are not changing drastically relative to the frequency of load balancing). While such an assumption holds for broad classes of applications, there certainly exist applications for which the load may vary greatly over a short time scale. In such situations, the user can provide a function that returns the predicted load for a node, allowing the load balancing routines to make better decisions. Note that the user may wish to use the node load information gathered by the Graph Library to estimate relevant timing constants.

3.2.3 Load Balance Initiation

Since all of the applications currently using the Graph Library already contain synchronization points (via **barrier()** calls), load balancing is initiated by the function call **balance_barrier()**, which is simply a special form of the **barrier()** operation:

- **balance_barrier(**..., *loadfun*, *mineff*, *packfun*, *unpackfun*, *freefun***)**: When all nodes have called this routine, their input messages *args*, which are of length *len*, are combined using *redfun()*. *globfun()* is called on computer zero with the result of the reduction. After *globfun()* completes, the barrier function *barfun()* is called at each node with the result of the reduction operation. After *barfun()* has completed everywhere, load balancing is conducted if the efficiency is less than *mineff*. In this phase, the loads of each node are calculated using the provided *loadfun()* routine. If nodes are moved, the *packfun()*, *unpackfun()* and *freefun()* provide the necessary support for **node_move()** to function properly. After load balancing is complete, *contfun()* is called on each node, which may now reside on a different computer. (Figure 3.13 gives a schematic representation of the execution of **balance_barrier()**.)

```
node_dispatch(msglen, msg)
    {node,func,arglen,arg} := msg
    set_node_context(node)
    busy_start := computer_time()
    func(arglen, arg)
    busy_end := computer_time()
    node_busy_timer[node] := node_busy_timer[node] + busy_end - busy_start
    lookup(global_func_counters, func) := lookup(...) + busy_end - busy_start
    lookup(local_func_counters[node], func) := lookup(...) + busy_end - busy_start
end node_dispatch
```

Figure 3.12: Schematic implementation of node load measurement.

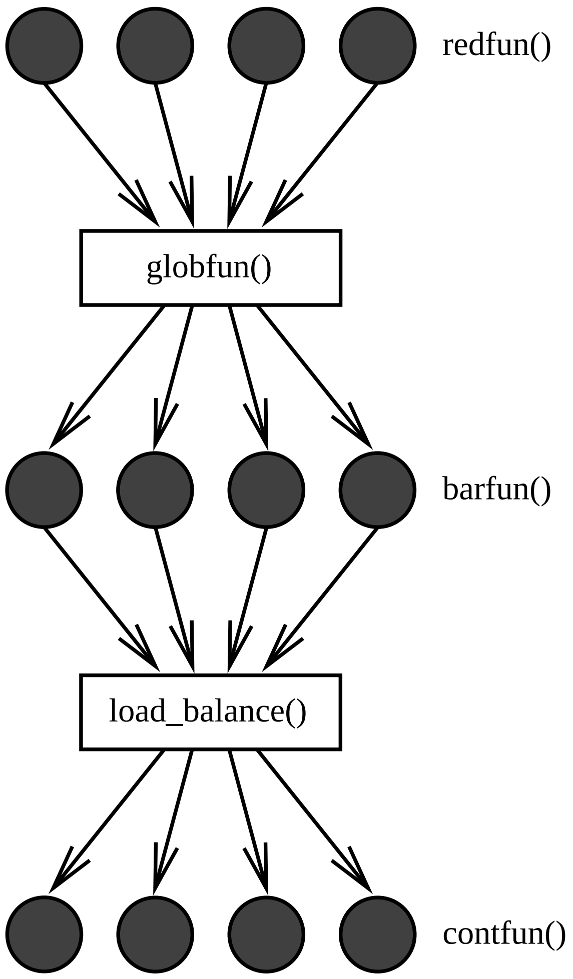

Figure 3.13: Schematic representation of **balance_barrier()** execution.

As mentioned in the description above, each computer sums the loads of its nodes as calculated using *loadfun()*, and if the average load for a computer is less than the *mineff* times the maximum load for any computer, then the next stage of load balancing is begun. At this point, the user determines the frequency at which **balance_barrier()** is called.

3.2.4 Work Transfer Vector Calculation

Transfer vectors are calculated using the second-order accurate diffusive scheme presented in Chapter 2. The initial value of u_i is the load of the computer calculated in the previous step, and the value of α is $1 - mineff$. Once the algorithm has converged to a state where the efficiency meets the user's specifications, nodes are selected for movement.

3.2.5 Node Selection

Nodes are selected to satisfy a transfer vector using a combination of the methods presented in Chapter 2. It was found early on that selecting nodes for exchange gave much better results than one-way transfer. Furthermore, when the number of nodes was small, approximation methods faired poorly in comparison to exhaustive searches. The Graph Library therefore uses the following strategy for deciding which nodes to transfer. If the total number of nodes on the computers involved in a transfer is less than 20, the optimal exchange is found via exhaustive search (on target architectures, this search takes only a fraction of a second). For larger numbers of nodes, the first-fit exchange strategy is used.

After nodes have been selected for transfer, a marker for each node to be transfered is sent to the node's new computer. This marker contains basic information such as the node's load and the number of its current computer. The node selection process is repeated using these markers to satisfy whatever portion of the transfer vectors remains unfulfilled from previous selection steps. The process of selecting/exchanging node markers is repeated until no markers are transferred.

The fact that this algorithm terminates is consequence of the following: All transfer vectors are of some finite size. Each exchange of work reduces the transfer vector in question by at least some minimum amount. I.e., for any given set of nodes, there are two subsets which would result in the minimum positive exchange of work. (Any transfer results in a net exchange of zero work is not performed.) When all of the transfer vectors are less than this minimum possible exchange, no more node markers will move, and the algorithm will terminate. (This is, of course, a very weak bound on time to termination—in practice, this phase requires a number of exchanges roughly proportional to the severity of the imbalance and the diameter of the mesh.)

Note that it is possible for a marker to move an arbitrary distance from its original computer (i.e., the movement is completely unconstrained). The applications targeted in Chapter 4 have a very low communication cost, and the lack of locality preservation had little discernible effect as a result.

3.2.6 Node Migration

Once the node markers have moved to their final destinations, a function is called remotely back on the original computer to fetch the node itself. In this way, costly transfers of state information occur only once, directly between a node's original and final location, rather than between each computer along the path the node's marker took. The node migration protocol provides failsafe mechanisms to handle cases in which a node transfer fails due to the memory constraints of packing/unpacking a node. If a node fails to move for this reason, it is marked as "immobile," and the load balancing process is restarted from the load balancing initiation phase. I.e., if the failure to transfer that node results in a less-than-desirable efficiency, the library attempts to remedy the situation.

Once again, this algorithm can be proven to terminate: Since there are a finite number of nodes in the system, and each node move failure results in at least one node being marked "immobile" and excluded from future consideration, the load balancing process can repeat due to failure a finite number of times. In practice, the process typically repeats at most twice.

3.3 Final Algorithm

The above instantiations of the phases of load balancing result in the final load balancing algorithm for the Graph Library, which is presented in Figure 3.14. Results of using this algorithm to load balance two applications are given in the next chapter.

```
load_balance(...)

    evaluate load for each task and sum task loads at each computer
    for each node n do
        node_load[n] := loadfun(n)
        computer_load := computer_load + node_load[n]
    end for

    if profitable to load balance then
    max_computer_load := global_max(computer_load)
    avg_computer_load := global_sum(computer_load) / num_computers
    if avg_computer_load/max_computer_load < mineff then
        do

            calculate transfer vectors between computers
            tv[] := diffuse()

            select tasks to meet those transfer vectors
            do
                which_nodes := select_nodes(tv[])
                move_markers(which_nodes)
            while nodes_selected > 0

            migrate selected tasks to their new computers
            for each marker m do
                move_node(m)

        while failed_moves > 0
    end if

end load_balance
```

Figure 3.14: Implemented algorithm for load balancing.

Chapter 4

Experiments

The load balancing algorithm presented in Chapter 3 was applied to two large-scale applications running under the Graph Library. Both of these applications exhibit very poor load distribution properties on relevant problems. This chapter gives a brief overview of these applications, including the algorithms and the specific problems to which they are applied. It also provides performance numbers before and after load balancing, demonstrating the practical efficacy of the algorithm given in the previous chapter.

4.1 Direct Simulation Monte Carlo Application

Direct Simulation Monte Carlo (DSMC) is a technique for the simulation of collisional plasmas and rarefied gases [24]. It is applied to particle flows where the Knudsen number[1] is too high for continuum methods such as the Euler or Navier-Stokes equations and too low for collisionless methods such as Particle-in-Cell. Like other gas and fluid modeling techniques, the DSMC method is based on a spatial gridding of the physical problem domain. At each time step particles may interact (collide) only with other particles in the same grid cell. DSMC techniques, as the name implies, simulate the collision of particles using a stochastic model. I.e, collisions are not detected directly by path intersection, but are chosen to occur by sampling a probability distribution function that is based on parameters such as the density, relative velocities and collisional cross-sections (or "diameters") of the particles involved. Both by virtue of limiting iterations to occur between particles within the same grid cell and by using a probabilistic model to simulate the occurrence of collisions, the higher-order computation of all possible path intersections of all particles is avoided. To maintain the physical validity of this model, the grid cell size must be such that the mean free path of a particle spans several cells. When a collision occurs, depending on the species of particles involved, they may simply rebound off of one another, combine chemically or cause one or the other to split (if either particle is a molecule). Once again, the actual results are determined by probabilistic chemistry models which may have both theoreti-

[1] The Knudsen number is a measure of rarefaction. Specifically, it is the ratio of length the mean free (or collisionless) path of a particle to the characteristic dimension (the ratio of particle density to the first spatial derivative of the density).

```
partition(...)
    **load** geometry data into partition
    **initialize** state
    **calculate** local statistics
    **gather/scatter** to obtain global statistics
    while time not exhausted do
        **move** and collide particles for time step $\Delta t$
        **send** away particles that exit current partition
        **receive** particles from neighboring partitions
        **update** cells with arriving particles
        **gather/scatter** to obtain global statistics
        **calculate** termination condition based on global statistics
    end while
end partition
```

Figure 4.1: Concurrent DSMC Algorithm

cal and empirical components. Based on particle distribution functions, macroscopic properties such as pressure, temperature and species concentration can be calculated.

4.1.1 Description

Hawk is a three-dimensional concurrent DSMC application based on techniques developed by Bird [24, 29]. It was written by Marc Rieffel of the Scalable Concurrent Programming Laboratory in conjunction with researchers from the Intel Corporation and the Philips Laboratory at Edwards Air Force Base. Hawk was developed with great care for software engineering. The physics and chemistry modules are easily interchangeable, allowing the rapid incorporation of proprietary modeling by end-users. Hawk is currently being applied to plasma reactor simulations for the Intel Corporation.

Algorithm. The DSMC algorithm that executes at each partition of the problem is given in Figure 4.1. Each node in the concurrent graph represents a partition of physical space and executes this algorithm. The *state* of a node is in essence the collection of particles contained in a region. As stated in the description of the DSMC technique, collisions within each partition (and grid cell therein) are calculated independently. The *physics* routines incorporate associated collision, chemistry and surface models. Once collisions have occurred, any particles that exited from a grid cell are migrated to their new cells, which may reside in different partitions. The *communication list* is used to implement these inter-partition transfers resulting from particle motion.

Problem of Interest. Plasma reactors are prominent in many stages of microprocessor fabrication. Specifically, they are involved in the etching of and deposition onto a wafer substrate. Improvements in reactor design could greatly impact the cost, quality and efficiency of fabrication. Thus far, empirical studies have been the primary component of the reactor design process. Such experiments are unfortunately both money- and time-consuming. Simulation provides a cheaper development path. Before simulating unproven designs, however, one must first verify simulation capabilities using available experimental data.

The Gaseous Electronics Conference (GEC) reactor is a standard reactor design that is being studied extensively. As such, it is a perfect target for parametric studies. The Hawk code described above is currently being validated on the GEC reactor. A 580,000-cell grid for the GEC reactor is shown in Figure 4.2. Of these cells, 330,000 cells represent 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 have been conducted using this grid. In these simulations, ambient conditions such as the port inflow and surface temperatures are specified. From the movement of particles inside the reactor, important macroscopic values such as particle density, temperature and velocity are calculated for each "live" grid cell. Ultimately, when incorporated into surface chemistry models, these parameters will be used to estimate rates of surface etching and material deposition.

4.1.2 Results

As the description of the GEC grid above details, only slightly more than 50 percent of the grid cells actually contain particles. Even for those cells that do contain particles, the density can vary by up to an order of magnitude, as shown in Figure 4.3. Consequently, one would expect that a standard spatial decomposition and mapping of the grid would result in a very inefficient computation. This is 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. As shown in Figure 4.4, this efficiency was improved to 86 percent by load balancing. This resulted in an 88 percent reduction in the run time. Figure 4.5 shows the corresponding improvement in workload distribution.

4.2 Particle-in-Cell Application

Particle-in-Cell (PIC) is a computational technique used for simulating highly rarefied particle flows in the presence of an electromagnetic field. The Knudson number is so high in this regime that the particles are considered to be collisionless. Particles are influenced by and contribute to an ambient electromagnetic field. The fundamental feature of PIC is the order-reducing method of calculating this interaction. A grid is superimposed on the computational domain. The electromagnetic effect of each particle with respect to the vertices of the grid cell containing it is calculated. Then, the governing field equations are solved over grid points, typically using an iterative solver. Once the field solver has converged, the effects of the new field are propagated back to the particles by adjusting their trajectories accordingly. This reciprocal interaction is calculated repeatedly throughout the computation until some termination criteria (such as particle concentration) is reached.

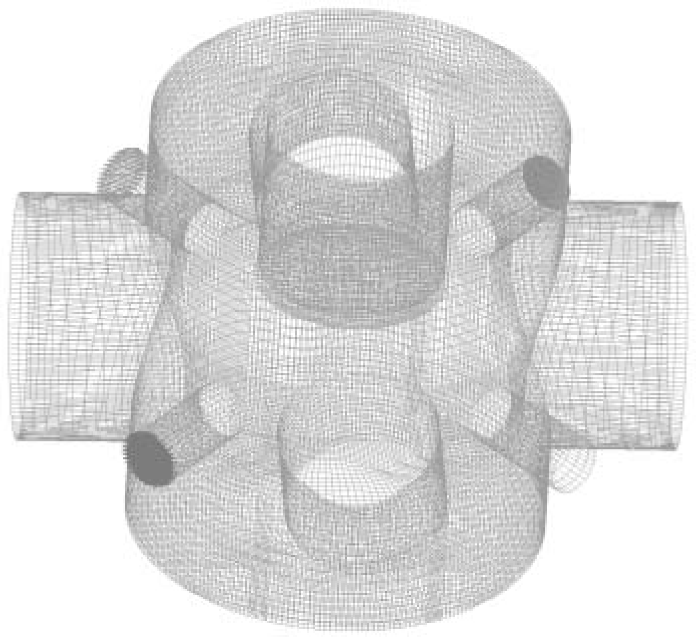

Figure 4.2: The 3-D grid for the GEC reactor.

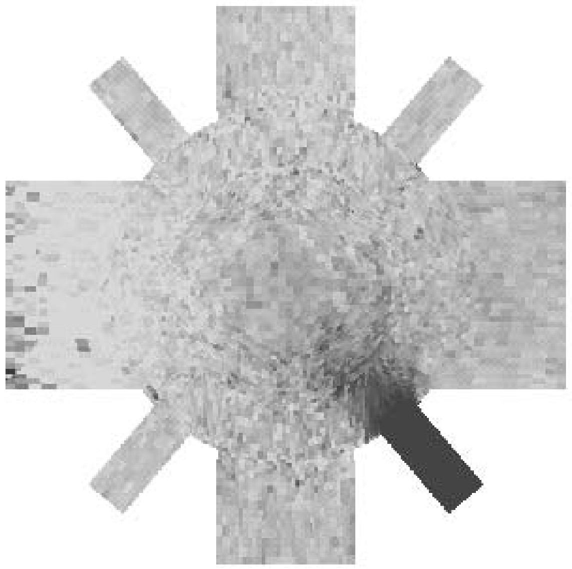

Figure 4.3: Particle density cutplane for the GEC reactor.

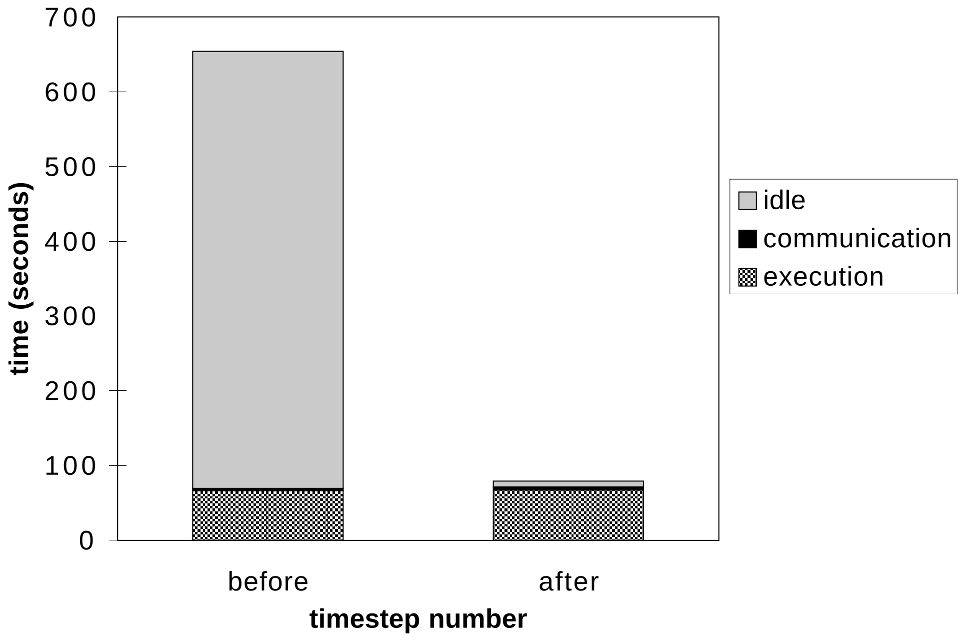

Figure 4.4: Run time breakdowns for 100 time steps of the DSMC code before and after load balancing, respectively.

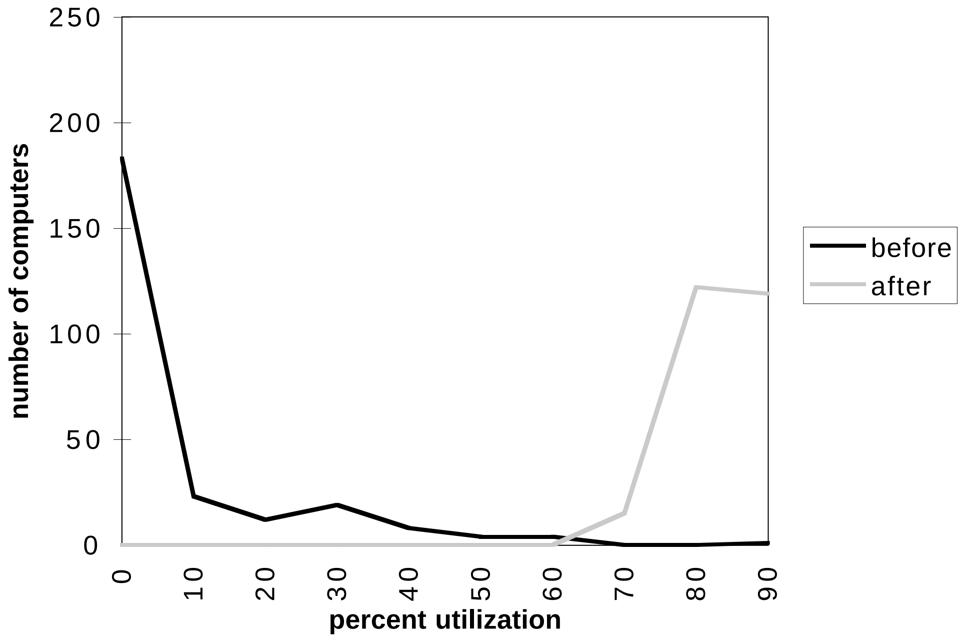

Figure 4.5: Utilization distributions for the DSMC code before and after load balancing, respectively.

```
partition(...)
    load geometry data into partition
    initialize state
    calculate local $\Delta t$ and norm
    gather/scatter to obtain global $\Delta t$ and norm
    while time not exhausted do
        move particles by virtue of velocity and $\Delta t$
        extract particles that exit partition
        send particles that exit to appropriate partition
        receive new particles from neighboring partitions
        update new particle positions within current partition
        gather/scatter to obtain global norm
        calculate termination condition based on global norm
        while global norm $<$ termination condition do
            extract field at boundary of partition
            send boundaries to neighbors
            receive adjacent boundaries from neighbors
            compute single iteration of the field solver
            calculate new local norm
            gather/scatter to obtain new global norm
        end while
    end while
end partition
```

Figure 4.6: Concurrent PIC Algorithm

4.2.1 Description

The Scalable Concurrent Programming Laboratory, in collaboration with the Space Power and Propulsion Laboratory of the MIT Department of Aeronautics and Astronautics, has developed a 3-D concurrent simulation capability called PlumePIC.

Algorithm. The PIC algorithm for a partition of the problem is presented in Figure 4.6. The *state* associated with a node is comprised of a portion of the grid and the particles contained within the corresponding portion of physical space. Each partition is solved independently and appropriate boundary conditions are used to signify what should happen at the interface between partitions. In addition, there is one non-physics boundary condition representing a *cut* in the domain. This boundary condition represents the fact that communication must be used to solve an area of the field or transport particles. At each time step, the algorithm has two parts: Based on some initial field, particles are injected into the domain and moved according to their velocities. If a particle exits a partition, it is communicated to an appropriate neighboring partition. After all particle movement has been conducted, the field is solved using communication to obtain information related to the field in adjacent partitions. The *communication list* associated with each node of the graph describes possible destinations for particles that move outside a partition and data dependencies required to implement the field solver. The *physics* routines used in Figure 4.6 describe the dynamics of particle movement and the solution of the field.

Problem of Interest. Electric propulsion devices are under consideration for a number of space missions, as well as station keeping applications for communications satellites. The issue of spacecraft contamination resulting from this type of propulsion system is receiving increased attention. Of particular interest are ion thrusters, which operate by electromagnetically ejecting a stream of ions at high velocity. A problem arises because complete ionization cannot be achieved at reasonable power levels. Hence, neutral gas is emitted at thermal speeds. The accelerated ions collide with these slow neutrals and charge-exchange, producing fast neutrals and slow ions. The latter can be influenced by local electric fields in the plume. The electric field structure in the plume, as seen in experiments and in computational models, is radial. Consequently, slow ions are pushed out of the beam and move back towards the spacecraft.

This ion backflow can cause a number of problems. It can contaminate spacecraft surfaces (particularly, solar panel arrays or sensors), possibly leading to a current drain if the surface is charged. It can also attenuate and otherwise interfere with electromagnetic waves sent to/from the satellite. Accurate predictions of the structure and return flux levels of this backflow will allow spacecraft designers to place the thrusters at locations where the backflow will be minimized, yielding substantial improvements in the longevity of military and commercial satellites.

This phenomenon was studied in a simulation of the ESEX/Argos satellite. The grid used was a regular grid with 9.4 million cells. The grid was partitioned into 1,575 blocks, which were mapped onto 256-processor Cray T3D. At the end of the simulation, 34 million particles were moving though the domain. Figure 4.7 shows the 3-D geometry as well as cutplanes of the ion density, charge-exchange ion density and electric field at the end of the simulation.

4.2.2 Results

As Figure 4.7 shows, the distribution of particles throughout the domain is very irregular. Moreover, this distribution changes dramatically over time. As a result, any static mapping of grid partitions to computers will result in large inefficiencies at some point in the computation. This fact is illustrated in Figure 4.8, which shows that each computer spends a large percentage of its time idle. Even after load balancing, the idle time for each computer, while often better, is still very high. Certainly, the load balancing algorithm has not improved the work distribution to the same extent that it did with the DSMC code. Closer examination reveals that this shortcoming is due to the two-phase nature of the PIC code. The DSMC application is a single-phase computation, so load balancing it is fairly straight-forward. The PIC code has *two* phases, particle push and field solve, each with very different load distribution characteristics. As a result, balancing the *total* load of these two phases on any given computer does not balance the *individual* phases of the computation. This fact is graphically illustrated in Figures 4.9 and 4.10. As one can see, while the load distribution for the total load at each computer improves dramatically (at least in the sense that the variance is greatly reduced), the load distributions for the two compo-

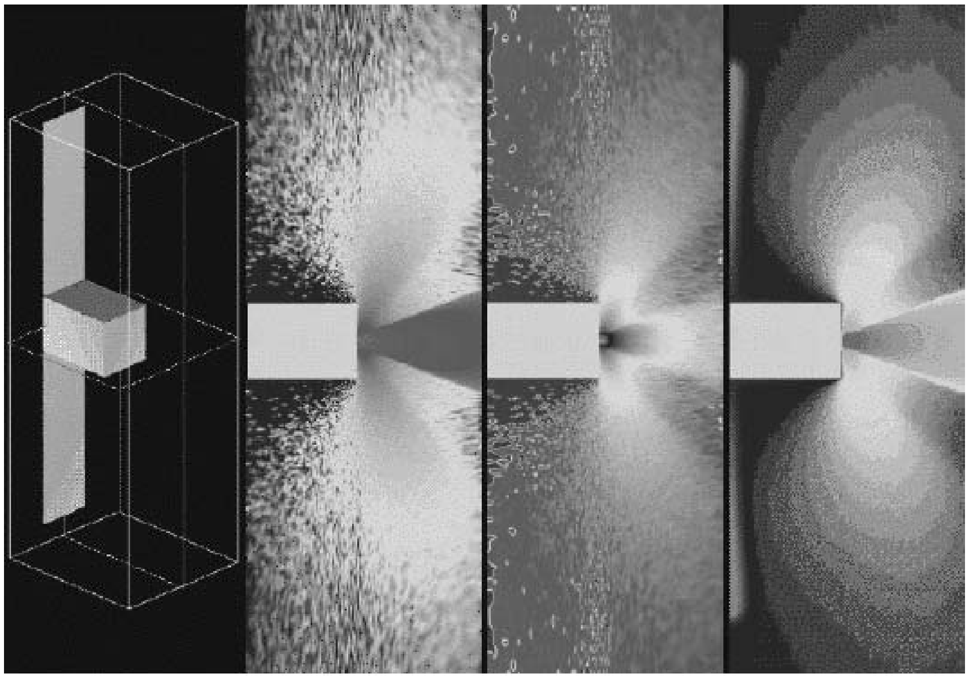

Figure 4.7: 3-D ESEX/Argos geometry and cutplanes ion density, charge-exchange ion density and electric field.

nent phases remain very poor. Consequently, the overall efficiency is low. To see this effect on a smaller scale, consider the case of two computers as shown in Figure 4.11: One has 50 units of phase one work and 100 units of phase two work. The other computer has 100 phase one and 50 phase two units. Obviously, both computers have the same total amount of work. However, because there is synchronization between the completion of phase one by both computers before phase two can begin, the computation is inefficient: The first computer must wait for the second before both can start phase two, and the second computer must wait for the first before the computation can complete. The above examples suggest that what one needs is a load balancing strategy that jointly balances each phase of the computation. (One cannot alternate between two distributions, for example, because the phases may be finely interleaved, making the cost of frequent redistribution of work prohibitive.) One way of doing this is to consider a computer's or node's load to be a *vector*, instead of a scalar, where each vector component is the load of a phase of the computation for that computer. If each component is balanced separately, then the problems encountered above would be circumvented: Each computer would have a roughly equal amount of work for each phase (implying that the total amount of work is also equal). Hence, little or no idle time would occur at synchronization points between phases. Notice that the characteristics of the PIC code also imply that, in general, one must assign multiple partitions to each computer. Some regions of the grid will have a high particle-to-cell ratio. A partition in such a region must be paired with a partition with a low particle-to-cell ratio to achieve effective load balancing of both phases. A similar situation is illustrated in Figure 4.12.

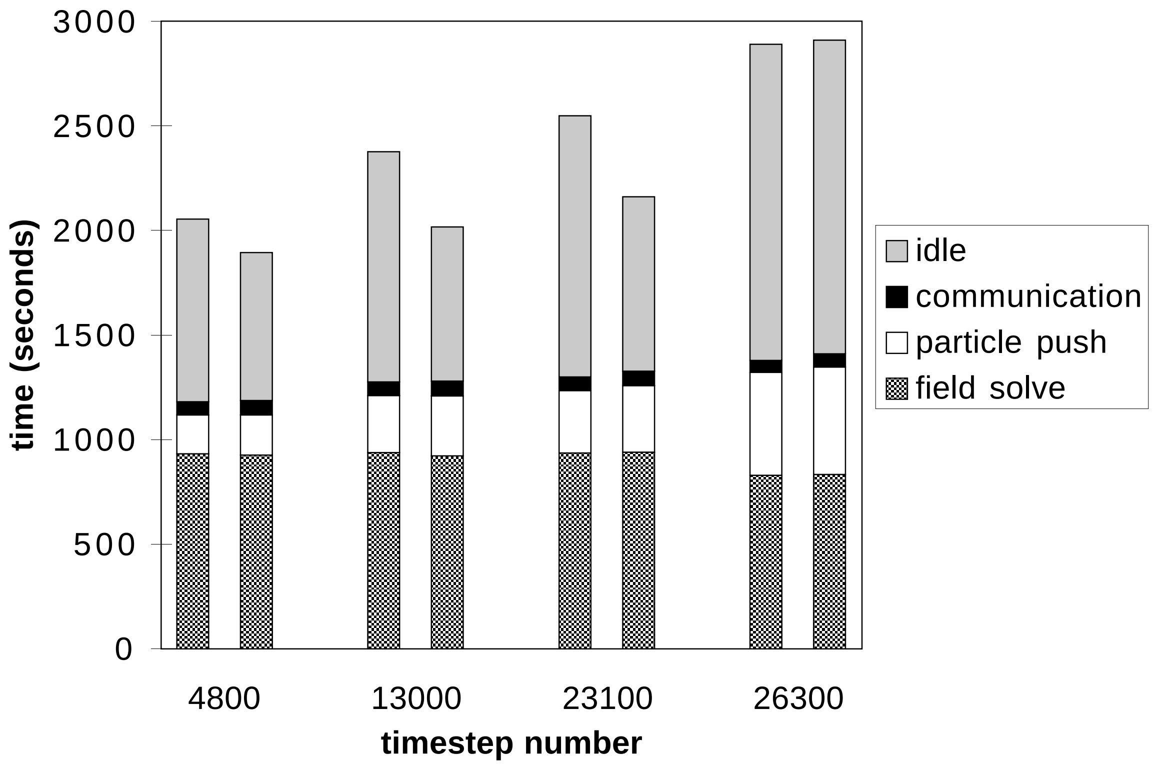

Figure 4.8: Run time breakdowns for 100 time steps of the PIC code, starting at several different time steps. Each pair of adjacent bars show the average time components for each computer before and after load balancing, respectively.

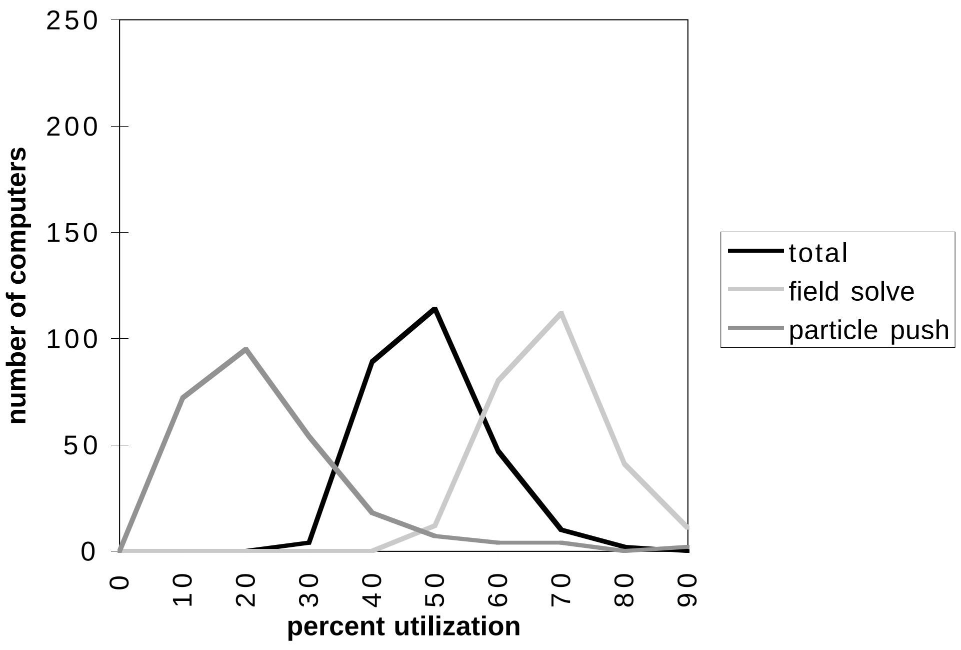

Figure 4.9: Pre-load balancing utilization distributions for computers based on total work, field solver work and particle push work.

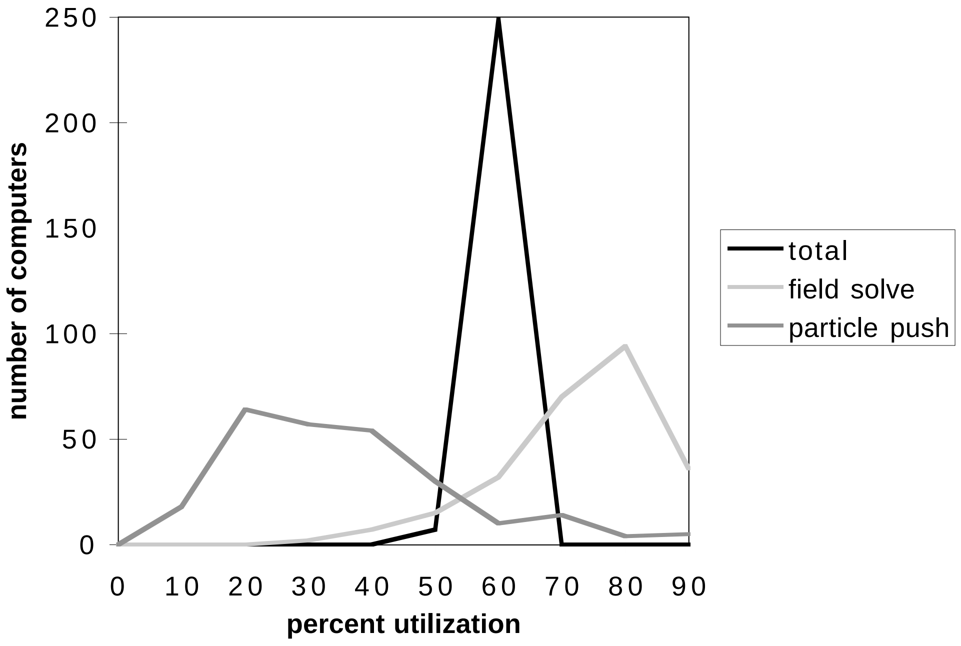

Figure 4.10: Post-load balancing utilization distributions for computers based on total work, field solver work and particle push work.

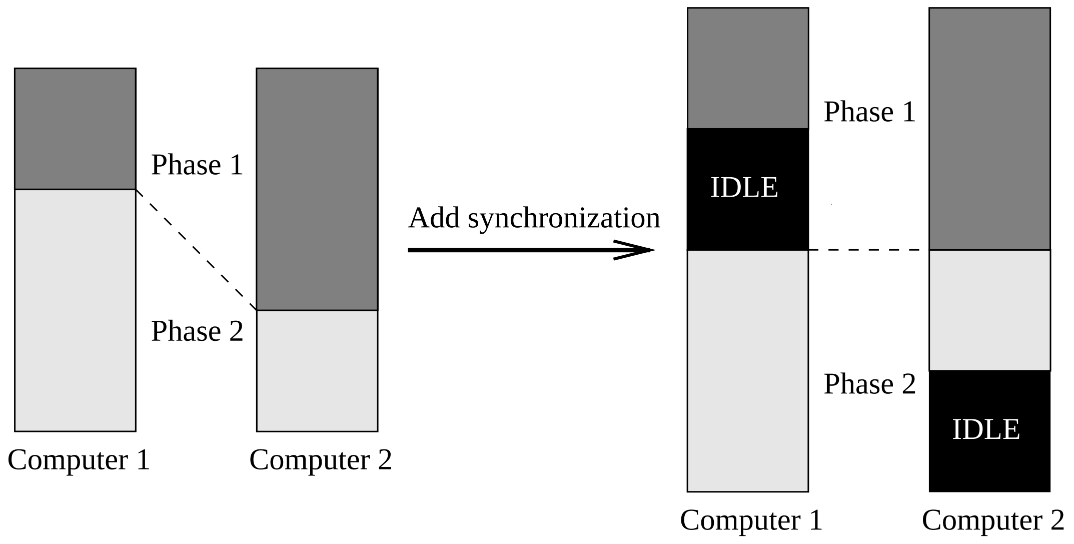

Figure 4.11: Demonstration of low efficiency in a "balanced" system. In the above example, both computers have the same total amount of work (i.e., they are load balanced in some sense). However, because of synchronization interposed between the unbalanced phases, idle time is introduced.

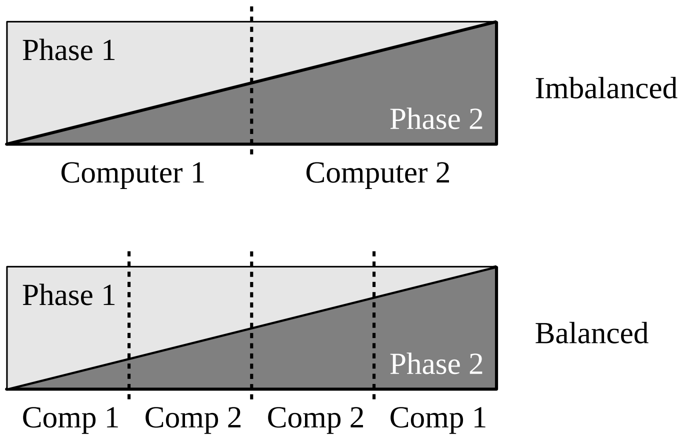

Figure 4.12: The above bars represent a one-dimensional space in which phase one dominates at one end and phase two dominates at the other. This domain cannot be divided evenly between two computers by a single cut. A cut down the middle would balance the total load, but neither of the component phases would be balanced. A cut anywhere else might either balance the first or second phase but not both. The only way to achieve a balance is to assign multiple partitions to each computer.

Chapter 5

Alternative Methodologies

Presented here is a summary of the primary alternatives to the overall methodology espoused in this thesis. Recall that alternatives to the components of the final algorithm have already been discussed in detail in Chapter 2.

Two message passing libraries that provide levels of portability similar to the Graph Library are the Parallel Virtual Machine (PVM) and the Message Passing Interface (MPI) [10, 12]. Unfortunately, the programming models supported by both of these libraries have severe limitations with regard to providing a reasonable framework for adaption and load balancing. PVM provides facilities for task creation and destruction and can be augmented to support task movement and load leveling [5]. These capabilities are implemented via a *task identifier* which abstracts the mapping of tasks to computers. Because PVM does not incorporate the idea of a communication graph, the library cannot easily determine the extent to which information regarding task creation, destruction and movement must be propagated. Consequently, it seems that updates to the task mapping would be inefficient and/or unscalable. Furthermore, because of its message buffering mechanisms, PVM is not thread-safe. While MPI does provide a better environment for multithreaded applications, it precludes task adaption. The set of communicating tasks under MPI is fixed at the beginning of program execution and cannot be changed during a computation. Of course, the limitations of both of these libraries can be circumvented by introducing an intermediate software layer that further abstracts task mapping and inter-task communication and better supports task adaption and movement. This is precisely what the Graph Library does. In fact, the Graph Library has already been implemented on top of both PVM and MPI.

Two libraries that provide support for parallel programming similar to that of the Graph Library are particularly interesting. CHAOS provides a framework for data and control decomposition of irregular, adaptive array-based codes via index translation and communication scheduling [17]. It differs from our approach in that it is appropriate only for FORTRAN-style regular data structures and in that the communication structure is determined implicitly by the reference patterns in the code. (In fact, the CHAOS system is designed to work in conjunction with High-Performance Fortran (HPF).) While the methods worked well for the two applications presented, they appear to be ill-suited for applications with irregular data structures such as linked lists and trees. Cilk provides a multithreaded environment with integrated load balancing [3]. In many respects, the programming model is similar to that in MDC and the Graph Library. Because of certain design decisions, it is best applied to tree-structured computations, however, and does not fit the SPMD style typical of scientific applications.

Most of the load balancing work in the literature considers only the subproblem that is defined as “work transfer vector calculation” in this thesis, ignoring the other issues such as load measurement and task selection. Such work includes gradient methods [20, 22], hierarchical algorithms [16] and early diffusive techniques [7]. Techniques appropriate to specific problem regimes include recursive bisection algorithms [30, 31, 33] and particle simulation methods [9, 19].

Other task-based approaches to load balancing include a scalable task pool [15], a heuristic for transferring tasks between computers based on probability vectors [8] and a scalable, iterative bidding model [27]. 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.

One paper which does address the entire load balancing problem and explores a broad range of options to its solution is [32]. The authors of this paper also conclude that diffusive techniques are superior on the basis of performance, scalability and efficiency. This paper also considers 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 load balancing algorithms.

Chapter 6

Conclusion

This thesis demonstrates that a practical, comprehensive approach to load balancing is possible and effective, but it also shows that substantial work remains to be done. In particular, while a simple scalar approach to load balancing is useful for applications involving a single phase of computation, the method fails to achieve high efficiencies for computations comprised of multiple phases, each with different load distribution properties. In addition, algorithmic improvements such as faster-converging diffusion schemes and better approximation algorithms for task selection need to be incorporated. The option of an asynchronous implementation begs exploration. Dynamic granularity management via task adaption and improved software interfaces would lessen the burden of the application developer.

The following six areas of improvement could dramatically increase the effectiveness, efficiency and utility of the load balancing strategy presented in this thesis:

1) **Consider load as a vector rather than a scalar quantity.** The experiments with the PIC code in Chapter 4 clearly demonstrate the limitations of the scalar view of load. While the load balancing algorithm clearly achieved a good balance for the total load on each computer, it failed to balance the components of the load. As a result, the overall efficiency was low. Only by jointly balancing the phases comprising a computation can one hope to achieve good overall load balance; viewing load as a vector is one way to accomplish this. It is also interesting to note that this is only possible through multiprocessing approaches which map multiple tasks/nodes to a single computer. Thus, single-block approaches (which include the majority of the techniques in the literature) are doomed to failure.
2) **Use load balancing to drive task/node adaption.** Task-based load balancing strategies fail whenever the load of a single task exceeds the average load over all computers. No matter where such a task is moved, the computer to which it is mapped will be overloaded. By dividing the task through routines such as **node_split()**, one can alleviate this problem by providing viable work movement options. In general, adaption can be used to dynamically manage the granularity of a computation so as to maintain the best number of tasks—increasing or decreasing the available options as necessary.
3) **Accelerate convergence of diffusion algorithm.** Both the first- and second-order accurate schemes converge slowly as a function of the size of the mesh network. The conver-

gence becomes especially slow as high frequency components of the load distribution are dampened, exposing low frequency components. Techniques such as multigrid dramatically accelerate the convergence in such circumstances.

4) **Incorporate better approximation algorithms for task selection.** Because the user is already specifying an accuracy tolerance for load balancing, it is completely acceptable to use this specification to guide the exhaustiveness of task selection. The subset trimming algorithm presented in Chapter 2 provides a way to do this for one-way transfers. Generalizing the algorithm would provide a mechanism for exchanges of tasks as well.
5) **Allow load balancing to be done asynchronously.** Load balancing is currently implemented in a barrier. This is not a necessity. Introducing asynchrony into the load balancing process would allow its cost to be overlapped with idle time on underloaded computers and would not disrupt applications that do not have algorithmic synchronization points.
6) **Improve the software interface to load balancing.** While the software interface to load balancing is quite simple—the user need only provide routines to pack and unpack a task's state—further improvements could be made. In particular, it would be possible to use checkpointing routines to move a task's state: The computer from which a task is being moved would write a checkpoint, and the computer to which a task is being moved would read that checkpoint. Of course, the checkpoint would not actually be written to disk but rather transferred via message passing. In this way, the user can reuse software that is already necessary for any large-scale application. In addition, data structure libraries should be modified to support automatic unpacking and packing of hte structures therein, better supporting checkpointing, load balancing and routine message passing.

A practical solution to the dynamic load balancing problem is certainly within reach. This thesis takes important steps toward that solution, both by performing well on a certain class of applications and by exposing the limitations of current approaches through its failure on a more general problem. This demonstration is particularly important since challenges more daunting than those posed by the PIC code loom ahead. The DSMC code, in particular, will become considerably more complex over the next few years. Like the PIC code, it will soon incorporate a self-consistent field, as well as surface chemistry models, grid adaption and visualization capabilities. Consequently, a vector with several components will undoubtedly be needed to adequately characterize the load. In that case and in others, this work thus prepares the way for strategies that better support current and future applications.

Bibliography

[1] W. Ames. *Numerical Methods for Partial Differential Equations.* 3rd ed., Academic Press, 1992.

[2] S. Barnard and H. Simon. “A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems.” *Concurrency: Pract. and Exp.,* 6:101–117, 1994.

[3] R. Blufome, et al.. “Cilk: An Efficient Multithreaded Runtime System.” *Proc. Fifth ACM SIGPLAN Sym. on Princ. & Pract. of Parallel Prog.,* pp. 207–216, ACM Press, 1995.

[4] G. Carrier and C. Pearson. *Partial Differential Equations: Theory and Technique.* 2nd ed., Academic Press, 1988.

[5] J. Casas, et. al.. “Adaptive Load Migration Systems for PVM.” *Proc. of Supercomputing ‘94*, pp. 390-399, IEEE Computer Society Press, 1994.

[6] T. Cormen, C. Leiserson and R. Rivest. *Introduction to Algorithms.* MIT Press/MacGraw-Hill, 1990.

[7] G. Cybenko. “Dynamic Load Balancing for Distributed Memory Multiprocessors.” *J. Parallel and Distributed Comp.,* 7:279–301, 1989.

[8] D. Evans and W. Butt. “Dynamic Load Balancing Using Task-Transfer Probabilities.” *Parallel Comp.,* 19:897–916, 1993.

[9] R. Ferraro, P. Liewer and V. Decyk. “Dynamic Load Balancing for a 2D Concurrent Plasma PIC Code.” Technical Report, Center for Research on Parallel Computing, CRPC-91-6, 1991.

[10] A. Geist, et al.. *PVM: Parallel Virtual Machine.* MIT Press, 1994.

[11] G. Golub and C. Van Loan. *Matrix Computations.* 2nd ed., The Johns Hopkins University Press, 1989.

[12] W. Gropp, E. Lusk and A. Skjellum. *Using MPI.* MIT Press, 1994.

[13] A. Heirich. “Scalable Load Balancing by Diffusion.” Masters Thesis, Caltech Computer Science Department, Caltech-CS-TR-94-04, 1994.

[14] A. Heirich and S. Taylor. “A Parabolic Load Balancing Algorithm.” Technical Report, Caltech Computer Science Department, Caltech-CS-TR-94-13, 1994.

[15] 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., Springer-Verlag, 1992.

[16] G. Horton. “A Multi-Level Diffusion Method for Dynamic Load Balancing.” *Parallel Comp.,* 19:209–218, 1993.

[17] Y.-S. Hwang, et al.. “Runtime and Language Support for Compiling Adaptive Irregular Problems on Distributed-Memory Machines.” *Software: Pract. and Exp.,* 25:597–621, 1995.

[18] E. Isaacson and H. Keller. *Analysis of Numerical Methods.* John Wiley and Sons, 1966.

[19] G. Kohring. “Dynamic Load Balancing for Parallelized Particle Simulations on MIMD Computers.” *Parallel Comp.,* 21:683-693, 1995.

[20] F. Lin and R. Keller. “The Gradient Model Load Balancing Method.” *IEEE Trans. Soft. Eng.,* 1:32–38, 1987.

[21] D. Maskit and S. Taylor. “A Message-Driven Programming System for Fine-Grain Multicomputers.” *Software: Pract. and Exp.,* 24:953-980, 1994.

[22] F. Muniz and E. Zaluska. “Parallel Load-Balancing: An Extension to the Gradient Model.” *Parallel Comp.,* 21:287–301, 1995.

[23] W. Press, S. Teukolsky, W. Vetterling and B. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* 2nd ed., Cambridge Univ. Press, 1992.

[24] M. Rieffel. “Concurrent Simulations of Plasma Reactors for VLSI Manufacturing.” MS Thesis, Caltech Computer Science Department, 1995.

[25] R. Samanta Roy. “Numerical Simulation of Ion Thruster Plume Backflow for Spacecraft Contamination Assessment.” PhD Dissertation, MIT Aeronautics and Astronautics Department, 1995.

[26] R. Samanta Roy, D. Hastings and S. Taylor. “Three-Dimensional Plasma Paricle-in-Cell Calculations of Ion Thruster Backflow Contamination.” Submitted to the *Journal of Computational Physics*, 1995.

[27] J. Song. “A Partially Asynchronous and Iterative Algorithm for Distributed Load Balancing.” *Parallel Computing,* 20:853–868, 1994.

[28] J. Stoer and R. Bulrisch. *Introduction to Numerical Analysis.* 2nd ed., Springer-Verlag, 1993.

[29] S. Taylor, J. Watts, M. Rieffel and M. Palmer. ‘The Concurrent Graph: Basic Technology for Irregular Problems.” Submitted to *IEEE Journal of Computational Science*, 1995.

[30] R. Van Driessche and D. Roose. “An Improved Spectral Bisection Algorithm and Its Application to Dynamic Load Balancing.” *Parallel Comp.*, 21:29–48, 1995.

[31] C. Walshaw and M. Berzins. “Dynamic Load-Balancing for PDE Solvers on Adaptive Unstructured Meshes,”. *Concurrency: Practice and Experience*, 7:17–28, 1995.

[32] M. Willebeek-LeMair and A. Reeves. “Strategies for Dynamic Load Balancing on Highly Parallel Computers.” *IEEE Transactions on Parallel and Distributed Systems*, 4:979–993, 1993.

[33] R. Williams. “Performance of Dynamic Load balancing Algorithms for Unstructured Mesh Calculations.” *Concurrency: Pract. and Exp.*, 3:457–481, 1991.

[34] D. Young and R. Gregory. *A Survery of Numerical Mathematics.* Vol. 2, Addison-Wesley, 1973 (republished by Dover, 1988).