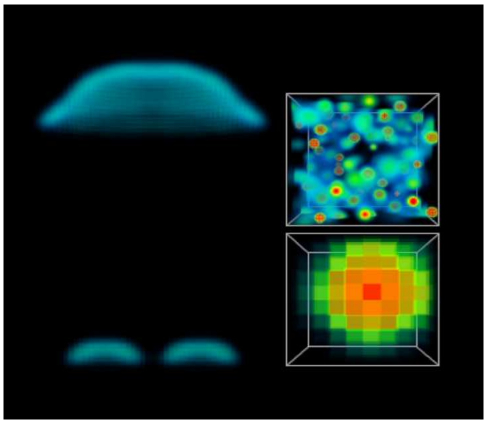
Scalable Load Balancing by Diffusion Alan Heirich

California Institute of Technology

10 February 1994/Revised 18 October 1994



Data aequatione quotcunque fluentes quantitae involvente fluxiones invenire et vice versa.

Sir Isaac Newton, 1687.

(It is useful to solve differential equations.)[1]

The research described in this report is sponsored by the Advanced Research Projects Agency, ARPA Order number 8176, and monitored by the Office of Naval Research under contract number N00014-91-J-1986.

[1] Principia Mathematica, 1st edition, lieber 2, proposition VII. A literal translation is "It is useful, having any given equation involving never so many flowing quantities, to find the fluxions and vice versa."

Contents

Chapter 1

Introduction

This introduction summarizes the motivation, problem, approach, and contributions of this thesis.

1.1 Motivation

High performance computing is in a transition between vector supercomputing and scalable multicomputing. The software strategies which are effective under these two paradigms are different. Scalable multicomputers require scalable algorithms. These are algorithms whose elapsed time complexities do not grow "unreasonably fast" as the problem size scales with the computer system.

1.2 Problem

The efficiency of grid based computations depends on the load balance among processors. Scalable grid computations require scalable load balancing methods. These methods must compute a balanced load distribution while preserving adjacency relationships of a computational domain.

1.3 Approach

Treat the computers in a scalable multicomputer as nodes in a computational grid. Use finite difference techniques to solve the heat equation on this grid. Adjust the workloads among computers to equal the evolving solution. Use only transfers between pairs of adjacent computers for this adjustment.

1.4 Contributions

A simple, correct and scalable load balancing method for grid computations. Correctness of the balance and adjacency properties are demonstrated. Elapsed execution time diminishes with increasing problem size. This demonstrates the algorithm is scalable in N without upper bound and load balance can be maintained at negligible cost for large grid computations.

The algorithm is efficient. Simulations of generic problems from computational fluid dynamics predict that a fraction of a second of elapsed time is sufficient to rebalance after grid adaptations. Even less time is required to solve a static load balancing problem for a million point grid computation on 512 computers.

An analysis of convergence is provided for the static load balancing problem. The time to converge the dynamic problem is usually different than predicted by this static analysis. For this reason simulations are recommended to provide bounds for specific problems instances. The fourth chapter presents simulations of generic problem instances which are of interest in computational fluid dynamics.

Chapter 2

The load balancing problem

The "load balancing" problem has been often discussed but rarely given formal definition. Load balancing is necessary in order to achieve effective use of a multicomputer. In the context of an operating system it is necessary to keep all computers busy in order to achieve maximal throughput. A mechanism must exist to ensure that tasks are distributed evenly among the computers. Otherwise some computers will be underutilized and overall throughput decreased. In the context of algorithms which require synchronization poorly balanced loads will result in some computers sitting idle while they wait for more heavily loaded computers to reach synchronization points. This problem affects most scientific calculations because most numerical algorithms require frequent synchronization.

Problem 2.1 (Load Balancing): Let Ω represent a set of elements of a computational domain and ψ a set of computers. Let $\phi : \Omega \to \psi$ and $\phi' : \Omega \to \psi$ represent assignments of elements to computers and $\vec{u}, \vec{u}'$ the measured workloads on the computers under the assignments ϕ, ϕ'. The *load balancing problem* is to compute, given an initial assignment ϕ, a new assignment ϕ' and vector $\Delta\vec{u}$ such that

(2.1.a) $\vec{u} + \Delta\vec{u} = \vec{u}'$.

(2.1.b) $u_i{}' = u_j{}' \quad \forall i, j \in \psi$.

(2.1.c) ϕ' preserves adjacencies of Ω.

The load balancing problem is often described as two problems: the "static" problem, in which the goal is to compute an initial assignment from a set of elements to a set of computers, such that every computer has the same workload; and the "dynamic" problem, in which the goal is to maintain a balanced workload under changing conditions. This distinction is unecessary since the static problem is a special case of the dynamic problem in which the initial assignment is trivial. The simplest trivial assignment locates all elements on a single computer. Further discussion of the static problem will assume this assignment.

Call conditions (2.1.a), (2.1.b) the "balance" requirement and (2.1.c) the "adjacency" requirement. The adjacency requirement holds that two elements ω_1, ω_2 which are adjacent in the computational domain Ω must reside on either the same computer or on a pair of adjacent computers. The workload u_i on a computer ψ_i is determined from the loads of the individual elements ω which are assigned to ψ_i as

$$u_i = \sum_{\{\omega \in \Omega | \phi(\omega) = \psi_i\}} \text{size}(\omega) \tag{2.1}$$

where size(ω) measures the workload imposed by ω on the computer ψ_i. Taken in isolation this requirement permits pathological solutions, such as one where all elements are assigned to a single computer. When this requirement is combined with the balance requirement pathological solutions are no longer permitted.

The adjacency requirement is necessary to minimize the cost of communication in domain decomposed calculations. Under the assumption that most communication occurs among adjacent elements of a computational domain an assignment ϕ which preserves the adjacencies in Ω minimizes the distances messages have to traverse in the multicomputer. This assumption applies to many problems in computational fluid dynamics and finite elements. As a result it is possible for these calculations to execute on multicomputers without contention for communication channels during a complete simultaneous exchange of data among adjacent elements. This type of exchange is the dominant form of communication for many scalable algorithms.

Bhokari [10] noted that the load balancing problem in full generality is NP-complete by transformation from the graph isomorphism problem. The graph isomorphism problem decides whether two arbitrary graphs are the same. An algorithm which computes solutions to problem 2.1 can solve the graph isomorphism problem if it considers the adjacencies of the problem domain to define one graph and the interconnection structure of the computer system another. If the interconnection structure is assumed to be a mesh then algorithms which solve problem 2.1 cannot solve the graph isomorphism problem. In this case NP-completeness of 2.1 still follows from the balance requirement. The proof is by transformation from the partition problem [26]:

> Given a finite set Ω of elements ω each with positive integer size(ω). Does there exist a subset $\Omega' \subset \Omega$ such that
>
> $$\sum_{\omega \in \Omega'} \text{size}(\omega) = \sum_{\omega \in \{\Omega - \Omega'\}} \text{size}(\omega)$$

This proof can be invalidated by making a further assumption that elements ω have equal weight. Then without loss of generality size(ω) $= 1$ for all ω. Under this assumption polynomial and even logarithmic methods exist to satisfy the balance requirement (2.1.a), (2.1.b). The problem of computing $\Delta\vec{u}$ or $\vec{u}'$ can be solved by a simple tree structured algorithm. The algorithm executes in a sequential number of steps which is logarithmic in the size of the computer system. In order to be useful any load balancing method should be as simple to implement as this algorithm and should require no more elapsed time.

This simple algorithm executes on every computer ψ_i. The goal of the algorithm is to identify the average s_0 of the workloads u_i under an initial assignment ϕ. Computers are identified with nodes of a tree with the understanding that leaf nodes are their own children and the root it's own parent.

Algorithm SIMPLE:

- Compute local workload u_i from expression (2.1).
- Receive sum s_j and count c_j from every child ψ_j.
- Compute $c_i = 1 + \sum_j c_j$.
- Compute $s_i = (u_i + \sum_j s_j c_j)/c_i$.
- Send s_i and c_i to parent ψ_h.

Algorithm **SIMPLE** begins execution at the leaf nodes and terminates at the root. At each node of the tree a computer receives the average workload in each subtree below it. The computer uses these subtree averages to compute a new average for the subtree of which it is the root. It passes this new average to it's parent.

By induction it is easy to see that this algorithm terminates in a state where s_0 for computer 0 at the root of the tree is equal to the average workload among all of the computers. This termination condition does not satisfy the balance requirement. In order to satisfy (2.1.a) it is necessary to compute a vector $\Delta\vec{u}$. This is accomplished by communicating s_0 through the tree in the opposite order, from the root to the leaves.

Algorithm SIMPLE2:

- Receive s_0 from the parent computer.
- Send s_0 to every child computer.
- Compute $\Delta u_i = s_0 - u_i$.

Algorithms **SIMPLE** and **SIMPLE2** contain serial dependencies. These serial dependencies make the algorithms inefficient. They are inefficient because each computer is idle in all but one of the sequential steps of execution. It would be nice to find a concurrent algorithm to compute s_0. A concurrent algorithm like the following one is potentially more efficient because no computer is idle in any step. It is not difficult to imagine this algorithm converges to something like s_0. It has even been proposed informally as a load balancing method. Unfortunately it would not a make a very good one.

Algorithm AVERAGE:

- Compute local workload u_i from expression (2.1).
- Let $u'^{(0)}_i = u_i$.
- For $m = 1$ to τ
  - Send $u'^{(m-1)}_i$ to each of J neighbors ψ_j.
  - Receive $u'^{(m-1)}_j$ from every neighbor ψ_j.
  - Compute $u'^{(m)}_i = (1/J)\sum_j u'^{(m-1)}_j$.
- EndFor m.
- Compute $\Delta u_i = u'^{(\tau)}_i - u_i$.

Algorithm **AVERAGE** is a concurrent iteration in which every computer receives the workloads $u'^{(m-1)}_j$ at it's neighbors, computes their average value, and then adjusts it's own workload $u'^{(m)}_i$ to equal that average. It is easy to be persuaded that this converges to a steady state in which $\vec{u}'^{(m)} = \vec{u}'^{(m-1)}$ and this is the truth. Unfortunately this steady state rarely satisfies the balance requirement of problem 2.1 [1]. For any one dimensional problem the steady state reduces to

$$u'^{(m)}_{i} = \frac{1}{2}\left(u'^{(m-1)}_{i+di} + u'^{(m-1)}_{i-di}\right)$$

This solution can be expanded in Taylor series to demonstrate that

$$\begin{aligned} 2u'_i &= u'_{i+di} + u'_{i-di} \\ &= \left[u'_i + \frac{\partial u'_i}{\partial x}di + \frac{\partial u'^2_i}{\partial x^2}(di^2/2) + \frac{\partial u'^3_i}{\partial x^3}(di^3/6) + \mathcal{O}(di^4)\right] \\ &+ \left[u'_i - \frac{\partial u'_i}{\partial x}di + \frac{\partial u'^2_i}{\partial x^2}(di^2/2) - \frac{\partial u'^3_i}{\partial x^3}(di^3/6) + \mathcal{O}(di^4)\right] \\ &= 2u'_i + \frac{\partial u'^2_i}{\partial x^2}di^2 + \mathcal{O}(di^4) \\ \frac{\partial u'^2_i}{\partial i^2} &= 0 + \mathcal{O}(di^2) \qquad (2.2) \end{aligned}$$

In one dimension the steady solutions of algorithm **AVERAGE** are second order accurate solutions of the Laplace equation (2.2). These are straight lines of arbitrary slope. Only the line of slope zero satisfies (2.1.b).

An extension of this argument to higher dimensions follows immediately from the theory of finite difference equations.

Although algorithm **AVERAGE** is unlikely to converge to solutions of problem 2.1 there are reasons to like it. It is concurrent, and scalable in the sense that it can execute on a multicomputer without contention for communication channels. The converged solutions can be described by Taylor expansions since all transfers occur between pairs of adjacent computers. The algorithm also holds the promise of computing solutions which satisfy the adjacency requirement. Since all exchanges occur between adjacent computers it should be possible to perform these exchanges without destroying existing adjacencies. Although **AVERAGE** is not a correct algorithm for load balancing it represents an approach which will be used in the next chapter to derive a correct and scalable load balancing method.

[1] Specifically, this steady state satisfies the balance requirement exactly when the boundary conditions are periodic.

Chapter 3

A scalable diffusion method

Algorithm DIFFUSION:

- Compute u_i from expression (2.1).
- $u'_i = u_i$
- For $l = 1$ to τ
  - $v_i^{(0)} = u'_i$
  - For $m = 1$ to ν
    - Send $v_i^{(m-1)}$ to all neighbors ψ_j
    - Receive $v_j^{(m-1)}$ from neighbors ψ_j
    - $v_i^{(m)} = \left(\frac{\Lambda}{1+J\Lambda}\right) \sum_j v_j^{(m-1)} + \frac{v_i^{(0)}}{1+J\Lambda}$
  - EndFor m
  - $u'_i = v_i^{(\nu)}$
  - Send u'_i to all neighbors ψ_j
  - Receive u'_j from neighbors ψ_j
  - Transfer $\Lambda\,(u'_i - u'_j)$ units of work to each neighbor ψ_j
- EndFor l

$\epsilon = 0.1$	N (total processors)						
	64	512	4,096	8,000	32K	256K	10^6
Λ	.36	.21	.16	.16	.16	.16	.16
τ	4	4	4	4	4	4	4
ν	7	5	4	4	3	4	4
TIME	21	14	13	12	12	12	12

$\epsilon = 0.01$	N (total processors)						
	64	512	4,096	8,000	32K	256K	10^6
Λ	.36	.21	.16	.16	.16	.16	.16
τ	7	8	8	8	8	8	7
ν	13	8	7	7	7	7	7
TIME	84	56	49	48	47	46	45

Table 3.1: Values for constants of algorithm **DIFFUSION** for given N. TIME is defined by equation (3.26) and is proportional to the elapsed time.

Heat diffusion is a process in nature in which thermal energy diffuses from hot regions into cold ones. The heat equation $\frac{\partial \vec{u}}{\partial t} - \nabla^2 \vec{u} = 0$ describes this process. When taken literally the terms of this equation read that the rate of change $\frac{\partial u_i}{\partial t}$ in any element u_i of a domain $\vec{u}$ depends on the curvature $\nabla^2 \vec{u}$ in the vicinity of u_i. This locality makes the heat equation a good model for a scalable load balancing method. Because the dependency is local an algorithm based on the heat equation requires only local exchanges of information between computers. Because heat diffusion is a concurrent process it is a good model for a concurrent algorithm.

Algorithm **DIFFUSION** is a concurrent iteration in which every computer derives the local curvature $\nabla^2 \vec{u}$ and then adjusts it's workload according to a finite difference approximation to the heat equation. J is the number of computers which are immediately adjacent to ψ_i. The constants τ, ν and Λ are obtained from table 3.1 for instances of the static problem. For instances of the dynamic problem the value of τ presented in this table is a lower bound. In these instances simulations should be used to predict an exact value for τ.

3.1 Derivation

Derive the algorithm for a domain corresponding to a three dimensional mesh connected multicomputer. Consider the heat equation in three dimensions

$$\begin{aligned} u_t &= \nabla^2 u \\ &= u_{xx} + u_{yy} + u_{zz} \end{aligned} \tag{3.1}$$

Taylor expand u to obtain each successive term of equation (3.1). The first term u_t is obtained by expanding in t with all derivatives evaluated at (x, y, z, t).

$$\begin{aligned} u(x,y,z,t+dt) &= u(x,y,z,t) + u_t dt + \mathrm{O}(dt^2) \\ u_t &= \left(\frac{u(x,y,z,t+dt) - u(x,y,z,t)}{dt} \right) + \mathrm{O}(dt) \end{aligned}$$

Expanding u in the spatial variables reveals the curvature terms (where omitted coordinates are interpreted as (x, y, z, t)) to be

$$\begin{aligned} u(x+dx,\cdot,\cdot,\cdot) &= u(\cdot,\cdot,\cdot,\cdot) + u_x dx + \\ & \quad u_{xx}\frac{dx^2}{2} + u_{xxx}\frac{dx^3}{6} + \mathrm{O}(dx^4) \\ u(x-dx,\cdot,\cdot,\cdot) &= u(\cdot,\cdot,\cdot,\cdot) - u_x dx + \\ & \quad u_{xx}\frac{dx^2}{2} - u_{xxx}\frac{dx^3}{6} + \mathrm{O}(dx^4) \\ u(x+dx,\cdot,\cdot,\cdot) + u(x-dx,\cdot,\cdot,\cdot) &= 2u(\cdot,\cdot,\cdot,\cdot) + u_{xx}dx^2 + \mathrm{O}(dx^4) \end{aligned}$$

$$u_{xx} = \left(\frac{u(x+dx,\cdot,\cdot,\cdot) + u(x-dx,\cdot,\cdot,\cdot) - 2u(\cdot,\cdot,\cdot,\cdot)}{dx^2} \right) + \mathrm{O}(dx^2)$$

Similar expansions in y, z show that the heat equation can be rewritten

$$\frac{u(x,y,z,t+dt) - u(x,y,z,t)}{dt} = \frac{1}{dx^2}\left(u(x+dx,y,z,t) + u(x-dx,y,z,t) + \right.$$
$$u(x,y+dy,z,t) + u(x,y-dy,z,t) + u(x,y,z+dz,t) +$$
$$\left. u(x,y,z-dz,t) - 6u(x,y,z,t)\right)$$

Since the spatial dimension is arbitrary take $dx = 1$. Identifying Λ with the time step dt and taking the spatial terms on the right at time $t + \Lambda$ yields an implicit time stepping scheme with unconditional stability.

$$u(x,y,z,t) = (1+6\Lambda)u(x,y,z,t+\Lambda) - \Lambda\left[u(x+dx,y,z,t+\Lambda)\right. \tag{3.2}$$
$$+u(x-dx,y,z,t+\Lambda) + u(x,y+dy,z,t+\Lambda) + u(x,y-dy,z,t+\Lambda)$$
$$\left.+u(x,y,z+dz,t+\Lambda) + u(x,y,z-dz,t+\Lambda)\right]$$

In order to compute solutions at successive time intervals Λ it is necessary to invert the relation $\vec{u}^{(t)} = A\vec{u}^{(t+\Lambda)}$ by solving

$$\vec{u}^{(t+\Lambda)} = A^{-1}\vec{u}^{(t)} \tag{3.3}$$

From (3.2) it is apparent that each row of A contains a diagonal term $(1+6\Lambda)$ and six offdiagonals $-\Lambda$. Let $A = (D-T)$ where D is diagonal. Then the relation $\vec{u}^{(t)} = A\vec{u}^{(t+\Lambda)}$ is equivalent to $\vec{u}^{(t+\Lambda)} = D^{-1}T\vec{u}^{(t+\Lambda)} + D^{-1}\vec{u}^{(t)}$. This relation is satisfied by fixed points of the Jacobi iteration

$$\left[\vec{u}^{(t+\Lambda)}\right]^{(m)} = D^{-1}T\left[\vec{u}^{(t+\Lambda)}\right]^{(m-1)} + D^{-1}\vec{u}^{(t)} \tag{3.4}$$

The matrix $D^{-1}T$ has a zero diagonal and six offdiagonal terms $\frac{\Lambda}{1+6\Lambda}$. D^{-1} is a diagonal matrix with terms $\frac{1}{1+6\Lambda}$. Iteration (3.4) is the central loop of algorithm **DIFFUSION**. This iteration is convergent from all initial conditions and has spectral radius defined by (3.23). Since the iteration is concurrent in all of the unknowns it converges at the same rate as a Gauss-Seidel iteration.

3.2 Correctness

The correctness of algorithm **DIFFUSION** is implied if all solutions of **DIFFUSION** satisfy the conditions of problem 2.1. Recall these conditions:

(2.1.a) $\vec{u} + \Delta\vec{u} = \vec{u}'$.

(2.1.b) $u_i{}' = u_j{}' \quad \forall i,j \in \psi$.

(2.1.c) ϕ' preserves adjacencies of Ω.

3.2.1 Balance

Recall that the balance requirement is defined by (2.1.a),(2.1.b). Consider the finite difference equation (3.2) rearranged to express the change in load with each iteration

$$\begin{aligned} u(x,y,z,t+\Lambda) - u(x,y,z,t) = \Lambda\,[&u(x+1,y,z,t+\Lambda) + u(x-1,y,z,t+\Lambda) \\ &+u(x,y+1,z,t+\Lambda) + u(x,y-1,z,t+\Lambda) \\ &+u(x,y,z+1,t+\Lambda) + u(x,y,z-1,t+\Lambda) \\ &-6u(x,y,z,t+\Lambda)] \end{aligned}$$

or as a vector equation with matrix operator $\mathcal{L}$

$$\vec{u}(t+\Lambda) - \vec{u}(t) = \Lambda\mathcal{L}\vec{u}(t+\Lambda) \tag{3.5}$$

Expand the solution $\vec{u}(t)$in an eigenvector basis $\vec{x}$ of $\mathcal{L}$

$$\vec{u}(t) = \sum_{i,j,k} a_{i,j,k}(t)\vec{x}_{i,j,k}$$

and rewrite the vector equation (3.5) in this expansion

$$\sum_{i,j,k} a_{i,j,k}(t+\Lambda)\vec{x}_{i,j,k} - \sum_{i,j,k} a_{i,j,k}(t)\vec{x}_{i,j,k} = \Lambda \sum_{i,j,k} \mathcal{L}a_{i,j,k}(t+\Lambda)\vec{x}_{i,j,k} \tag{3.6}$$

For the purpose of analysis assume a periodic domain. The eigenvalues $\lambda_{i,j,k}$ of $\mathcal{L}$ are

$$\lambda_{i,j,k} = 2\left(3 - \cos 2\pi\frac{i}{n} - \cos 2\pi\frac{j}{n} - \cos 2\pi\frac{k}{n}\right) \tag{3.7}$$

where i, j and k range from 0 to $n/2-1$. Use the knowledge of these eigenvalues and

$$\mathcal{L}\vec{x}_{i,j,k} + \lambda_{i,j,k}\vec{x}_{i,j,k} = 0$$

to further simplify (3.6)

$$\sum_{i,j,k} \left(a_{i,j,k}(t+\Lambda)\vec{x}_{i,j,k}\left[1+\Lambda\lambda_{i,j,k}\right] - a_{i,j,k}(t)\vec{x}_{i,j,k}\right) = 0$$

and by the completeness and orthonormality of the eigenvectors

$$a_{i,j,k}(t+\Lambda)\left[1+\Lambda\lambda_{i,j,k}\right] - a_{i,j,k}(t) = 0$$

showing that the time dependent decay of each component i, j, k is

$$a_{i,j,k}(\Lambda) = \frac{a_{i,j,k}(0)}{1 + \Lambda\lambda_{i,j,k}} \tag{3.8}$$

An arbitrary component i, j, k decays in amplitude by a factor ϵ in τ steps if

$$[1 + \Lambda\lambda_{i,j,k}]^{-\tau} \leq \epsilon \tag{3.9}$$

Equation (3.8) shows that all components of any initial disequilibrium decay in amplitude to zero. Further this decay occurs at exponential rates. It is this exponential property that underlies the $\mathcal{O}(\log N)$ serial time complexity of the algorithm. The elapsed time complexity $\mathcal{O}(\frac{1}{N}\log N)$ is discussed later in this chapter.

Because all components decay to zero amplitude the calculation of $\vec{u}'$ satisfies the balance requirement (2.1.a),(2.1.b). In order for algorithm **DIFFUSION** to be correct it is also necessary that the transfer of work cause the resulting workloads to equal this calculated $\vec{u}'$. The transfer step of algorithm **DIFFUSION** sends $\Lambda\,(u'_i - u'_j)$ to each neighbor j. On a three dimensional multicomputer the net change at each computer from this transfer is

$$\begin{aligned}
u'_i \;= u'_i \; & -\Lambda\,(u'_i - u'_{x-1,y,z}) \\
& -\Lambda\,(u'_i - u'_{x+1,y,z}) \\
& -\Lambda\,(u'_i - u'_{x,y-1,z}) \\
& -\Lambda\,(u'_i - u'_{x,y+1,z}) \\
& -\Lambda\,(u'_i - u'_{x,y,z-1}) \\
& -\Lambda\,(u'_i - u'_{x,y,z+1}) \\
= \quad & (1 + 6\Lambda)u(x,y,z,t+\Lambda) - \\
& \Lambda\,[u(x+dx,y,z,t+\Lambda) + u(x-dx,y,z,t+\Lambda) \\
& +u(x,y+dy,z,t+\Lambda) + u(x,y-dy,z,t+\Lambda) \\
& +\, u(x,y,z+dz,t+\Lambda) + u(x,y,z-dz,t+\Lambda)]
\end{aligned}$$

This is equation (3.2).

3.2.2 Adjacency

Recall that condition (2.1.c) is the adjacency requirement. If an initial assignment ϕ preserves adjacencies of a domain Ω then the new assignment ϕ' must preserve the same adjacencies. This means that every pair of elements ω_1, ω_2 which are adjacent in Ω reside on either the same computer or on adjacent computers.

In order to demonstrate that solutions of algorithm **DIFFUSION** satisfy the adjacency requirement it suffices to show that algorithm **DIFFUSION** preserves adjacencies which are present in an initial assignment ϕ. This can be demonstrated by an informal argument.

- ϕ' preserves adjacencies of Ω if ϕ preserves adjacencies of Ω and algorithm **DIFFUSION** does not destroy adjacencies.
- Algorithm **DIFFUSION** does not destroy adjacencies if it does not destroy them in any transfer.
- Algorithm **DIFFUSION** need not destroy adjacencies in any transfer because all transfers occur between adjacent computers.

This argument does not suggest that adjacencies are preserved by all possible transfer mechanisms. Instead it suggests that transfer mechanisms exist which preserve adjacencies. To make this point concrete consider the example of a domain decomposed grid computation on a three dimensional multicomputer. Each computer solves a portion of a three dimensional domain. A transfer mechanism which preserves adjacencies selects for transfer those elements ω which are nearest in Ω to elements on adjacent computers. Such a mechanism can be implemented with indexing to have a fixed cost per transaction. Coarse grained versions of this approach have been shown effective in molecular dynamics and vortex calculations [3, 7, 9]. In some applications it may not be practical to transfer work in small quantities. In these cases it may be more efficient to postpone transfers until the algorithm has converged on a value for $\Delta\vec{u}$.

3.3 Static problem analysis

Recall that the static load balancing problem is a special case of problem 2.1. In this special case the initial assignment ϕ maps the entire domain Ω to a single computer ψ_i. Algorithm **DIFFUSION** solves instances of the static problem by diffusing domain elements from ψ_i until the balance requirement is met.

It is possible to analyze the convergence to equilibrium of instances of the static problem. The initial assignment ϕ corresponds to an instance of a unit impulse. A unit impulse is a summation of equally weighted sinusoids. Recall from equation (3.9) that to reduce the amplitude of an arbitrary component i, j, k by ϵ in τ steps of the method requires $[1 + \Lambda\lambda_{i,j,k}]^{-\tau} \leq \epsilon$. The fastest case occurs for the smallest positive eigenvalue $\lambda_{0,0,1} = (2 - 2\cos(2\pi\frac{1}{n}))$ which corresponds to a high frequency sinusoid. To reduce such a disturbance requires

$$\tau = \left\lceil \frac{\ln \epsilon^{-1}}{\ln\left(1 + \Lambda\left(2 - 2\cos 2\pi\frac{1}{n}\right)\right)} \right\rceil \tag{3.10}$$

Convergence of this component approaches $\ln \epsilon^{-1}$ for large n since

$$\lim_{n\to\infty} \ln\left(1 + \Lambda\left(2 - 2\cos 2\pi\frac{1}{n}\right)\right) = 1$$

Convergence of lowest wavenumber component $\lambda_{n/2-1,n/2-1,n/2-1}$ is slow because

$$\tau = \left\lceil \frac{\ln \epsilon^{-1}}{\ln\left(1 + \Lambda(6 - 6\cos\pi\frac{n/2-2}{n/2-1})\right)} \right\rceil \tag{3.11}$$

This converges to ∞ for large n because

$$\lim_{n\to\infty} \ln\left(1 + \Lambda\left(6 - 6\cos\pi\frac{n/2-2}{n/2-1}\right)\right) = 0$$

By understanding the decay of individual components it is possible to analyze the decay of a summation of equally weighted components over time. The following text uses the Poisson bracket $\langle\cdot,\cdot\rangle$ to represent the inner product operator. When discussing loads or eigenvectors it uses $u[x, y, z]$ or $x_{i,j,k}[x, y, z]$ to denote the vector element which corresponds to location x, y, z of the computational grid with the convention that $[0, 0, 0]$ is the first element of the vector. Then the initial disturbance confined to a particular processor x, y, z can be written as an eigenvector expansion

$$\vec{u}(0) = \sum_{l,m,n} a_{l,m,n}(0)\vec{x}_{l,m,n} \tag{3.12}$$

Since by assumption the instance is a unit impulse the initial disturbance $u(0)$ is zero except at element $[x, y, z]$. Then

$$\langle \vec{x}_{i,j,k}, \vec{u}(0)\rangle = x_{i,j,k}[x, y, z] \tag{3.13}$$

This is equal to the initial amplitude $a_{i,j,k}(0)$ of each eigenvector $\vec{x}_{i,j,k}$

$$\begin{aligned}
\langle \vec{x}_{i,j,k}, \vec{u}(0) \rangle &= \left\langle \vec{x}_{i,j,k}, \sum_{l,m,n} a_{l,m,n}(0)\vec{x}_{l,m,n} \right\rangle \\
&= \sum_{l,m,n} \langle \vec{x}_{i,j,k}, \vec{x}_{l,m,n} \rangle \, a_{l,m,n}(0) \\
&= \sum_{l,m,n} a_{l,m,n}(0)\delta_{il}\delta_{jm}\delta_{kn} \\
&= a_{i,j,k}(0) \qquad (3.14)
\end{aligned}$$

In order to facilitate analysis it has been assumed that the computational domain has periodic boundary conditions. Because of this assumption the origin of the coordinate system is arbitrary. Without loss of generality place the origin at the source of the disturbance and take $x = y = z = 0$. This has no effect on the eigenvectors $\vec{x}_{i,j,k}$ and from (3.13), (3.14)

$$a_{i,j,k}(0) = x_{i,j,k}[0, 0, 0] \qquad (3.15)$$

Placing the origin at the source of the disturbance is convenient in considering the first element of the eigenvectors $x_{i,j,k}[0, 0, 0]$. $\mathcal{L}$ has $n/2$ distinct eigenvalues $\lambda_{i,j,k}$ each of algebraic multiplicity two. Each of these eigenvalues has geometric multiplicity eight, ie. has eight linearly independent associated eigenvectors of unit length

$$x_{i,j,k}[x, y, z] = c_{i,j,k} F_1 (2\pi xi/n)\, F_2 (2\pi yj/n)\, F_3 (2\pi zk/n) \qquad (3.16)$$

where each F_i is either sin or cos. By choosing $x = y = z = 0$ this expression is zero except for the single eigenvector for which $F_1(x) = F_2(x) = F_3(x) = \cos(x)$. Without loss of generality restrict consideration to initial disturbances of the form

$$u[0, 0, 0](0) = \sum_{i,j,k} c_{i,j,k} x_{i,j,k}[0, 0, 0] = \sum_{i,j,k} c^2_{i,j,k} \qquad (3.17)$$

From (3.8) define the time dependent disturbance at any location x', y', z'

$$\begin{aligned}
u[x', y', z'](\tau\Lambda) &= \sum_{i,j,k} c_{i,j,k} \left[1 + \Lambda\lambda_{i,j,k}\right]^{-\tau} x_{i,j,k}[x', y', z'] \\
&= \sum_{i,j,k} c^2_{i,j,k} \left[1 + \Lambda\lambda_{i,j,k}\right]^{-\tau} \cos 2\pi \frac{x'i}{n} \\
&\quad \cos 2\pi \frac{y'j}{n} \cos 2\pi \frac{z'k}{n} \qquad (3.18)
\end{aligned}$$

From (3.16) a necessary condition for a normalized eigenvector is

$$
\begin{aligned}
1 &= c_{i,j,k}^2 \sum_{x,y,z} \cos^2\left(2\pi\frac{xi}{n}\right)\cos^2\left(2\pi\frac{yj}{n}\right)\cos^2\left(2\pi\frac{zk}{n}\right) \\
&= c_{i,j,k}^2 \frac{1}{8}\sum_{x,y,z}\left(1+\cos 4\pi\frac{xi}{n}\right)\left(1+\cos 4\pi\frac{yj}{n}\right)\left(1+\cos 4\pi\frac{zk}{n}\right) \\
&= c_{i,j,k}^2 \frac{1}{8}\sum_{x,y,z}\left\{\left(1+\cos 4\pi\frac{yj}{n}\right)\left(1+\cos 4\pi\frac{zk}{n}\right)\right. \\
&\quad \left. +\sum_x \cos\left(4\pi\frac{xi}{n}\right)\sum_{y,z}\cos\left(4\pi\frac{yj}{n}\right)\left(1+\cos 4\pi\frac{zk}{n}\right)\right\} \qquad (3.19)
\end{aligned}
$$

Simplify the preceding expression by the following

Lemma 1

$$\sum_x \cos 4\pi\frac{xi}{n} = 0$$

Proof:

$$
\begin{aligned}
\sum_x \cos 4\pi\frac{xi}{n} &= \sum_x Re\left(e^{\imath 4\pi xi/n}\right) \\
&= Re\sum_x e^{\imath 4\pi xi/n} \\
&= Re\sum_x \left(e^{\imath 4\pi i/n}\right)^x \\
&= Re\left[\frac{e^{\imath 4\pi i/n}\left(1-\left(e^{\imath 4\pi i/n}\right)^n\right)}{1-e^{\imath 4\pi i/n}}\right] \\
&= 0
\end{aligned}
$$

Q.E.D.

Repeated application of lemma (1) to equation (3.19) yields

$$
\begin{aligned}
1 &= c_{i,j,k}^2 \frac{1}{8}\sum_{x,y,z}\left(1+\cos 4\pi\frac{yj}{n}\right)\left(1+\cos 4\pi\frac{zk}{n}\right) \\
&= c_{i,j,k}^2 \frac{1}{8}\left[\sum_{x,y,z}\left(1+\cos 4\pi\frac{zk}{n}\right)+\sum_{x,y,z}\left(\cos 4\pi\frac{jy}{n}\right)\left(1+\cos 4\pi\frac{zk}{n}\right)\right]
\end{aligned}
$$

$$
\begin{aligned}
&= c_{i,j,k}^2 \frac{1}{8}\left[\sum_{x,y,z} 1 + \sum_{x,y,z}\left(\cos 4\pi \frac{zk}{n}\right)\right] \\
&= c_{i,j,k}^2 \frac{1}{8} n
\end{aligned} \tag{3.20}
$$

From which one concludes

$$
c_{i,j,k} = \left(\frac{8}{n}\right)^{1/2} \qquad \forall\, i,j,k
$$

From (3.17),(3.18) and (3.3) the time dependent disturbance at $0,0,0$ is therefore

$$
u[0,0,0](\tau\Lambda) = \frac{8}{n}\sum_{i,j,k}\left[1 + \Lambda 2\left(3 - \cos 2\pi\frac{i}{n} - \cos 2\pi\frac{j}{n} - \cos 2\pi\frac{k}{n}\right)\right]^{-\tau} \tag{3.21}
$$

From (3.21) it follows that $u[0,0,0](\tau\Lambda) \le \epsilon$ when

$$
\frac{8}{n}\sum_{i,j,k}\left[1 + \Lambda 2\left(3 - \cos 2\pi\frac{i}{n} - \cos 2\pi\frac{j}{n} - \cos 2\pi\frac{k}{n}\right)\right]^{-\tau} \le \epsilon
$$

and therefore

$$
\tau = \left\lceil \frac{\ln \epsilon^{-1}}{\ln(8/n)\sum_{i,j,k}\left[1 + \Lambda 2\left(3 - \cos 2\pi\frac{i}{n} - \cos 2\pi\frac{j}{n} - \cos 2\pi\frac{k}{n}\right)\right]} \right\rceil \tag{3.22}
$$

3.4 Scaling

Equation (3.22) is an expression for τ. A similar expression for ν can be obtained by considering the convergence of the Jacobi iteration. From the Geršgorin disc theorem [24] the eigenvalues λ of the Jacobi iteration are bounded $|\lambda| \le \frac{6\Lambda}{1+6\Lambda}$. Since the row and column sums are constant and the iteration matrix is nonnegative ([24], theorem 8.1.22) the spectral radius equals the row sum

$$
\rho\left(D^{-1}T\right) = \frac{6\Lambda}{1+6\Lambda} \tag{3.23}
$$

Define the error in a current value $\vec{u}^{(m)}$ under the iteration as $e(\vec{u}^{(m)}) = (\vec{u}^{(m)} - \vec{u}^*)$ where $\vec{u}^*$ is the fixed point of the iteration. Then for $\nu > 0$

$$
e(\vec{u}^{(\nu)}) = e((D^{-1}T)^{\nu}\vec{u}^{(0)}) = (D^{-1}T)^{\nu} e(\vec{u}^{(0)}) \tag{3.24}
$$

which converges when $\rho(D^{-1}T) < 1$ since $\rho((D^{-1}T)^{\nu}) = (\rho(D^{-1}T))^{\nu}$. In order to quantify the error define the infinity norm

$$\|e(\vec{u}^{(m)})\|_\infty = \max_{x,y,z} \left| e(u^{(m)}_{x,y,z}) \right| = \max_{x,y,z} \left| u^{(m)}_{x,y,z} - u^*_{x,y,z} \right|$$

Using this norm define a necessary condition to improve the accuracy of the solution $\vec{u}$ by a factor ϵ in ν steps to be $\|e(\vec{u}^{(\nu)})\|_\infty \leq \epsilon \|e(\vec{u}^{(0)})\|_\infty$. This is satisfied when $(\rho(D^{-1}T))^{\nu} \leq \epsilon$ and thus

$$\nu = \left\lceil \frac{\ln \epsilon}{\ln \frac{6\Lambda}{1+6\Lambda}} \right\rceil \tag{3.25}$$

The elapsed time complexity of algorithm **DIFFUSION** is proportional to the product $\tau\nu$. Table 3.1 provides values for the parameters τ, ν, Λ and ϵ which minimize this quantity. Equation (3.26) demonstrates that $\tau\nu$ is logarithmic in N.

$$\tau\nu = \left\lceil \frac{\ln \epsilon \ln \epsilon^{-1}}{\ln \frac{6\Lambda}{1+6\Lambda} \ln(8/n) \sum_{i,j,k} \left[1 + \Lambda 2 \left(3 - \cos 2\pi \frac{i}{n} - \cos 2\pi \frac{j}{n} - \cos 2\pi \frac{k}{n} \right) \right]} \right\rceil \tag{3.26}$$

3.5 Remarks

This chapter has presented a simple algorithm which satisfies the balance and adjacency requirements and which executes in decreasing elapsed times for increasing problem sizes. The algorithm causes all components of an initial disturbance to decay in amplitude to zero at exponential rates. Convergence of any disturbance is bounded by the decay of the slowest component (3.11). Although it is possible to analyze the convergence of the static problem exactly it is more difficult to predict convergence of multiple point disturbances. For this reason the next chapter presents simulations of multiple point disturbances.

Algorithm **DIFFUSION** is formulated in such a way that it reduces the worst case imbalance between any two processors ψ_i, ψ_j by a factor ϵ. In a practical context it can be useful to reduce the imbalance below a threshold which is a percentage of the total load. For example, it can be useful to assert that the worst case imbalance will be within 10% of the load average for a given problem.

To use the algorithm in this way formulate ϵ as a function of N and of $|\Omega|$, the total number of domain elements. Define a threshold α of eg. 10% and assert that the solution will have a worst case imbalance of no more than 10% of the load average.

If the initial imbalance is $|\Omega|$ (as in the static problem) then $\epsilon = \frac{\alpha}{N}$. An optimal Λ can be found for any ϵ by minimizing $\tau\nu$.

Consider the example of a computation with 10^7 grid points on 100 computers. The average load is $10^7/10^2 = 10^5$. If $\alpha = 10\%$ then the desired worst case imbalance is 10^4. Solving $\epsilon|\Omega| = \alpha|\Omega|/N$ gives $\epsilon 10^7 = 10^{-1}10^7 10^{-2} \Rightarrow \epsilon = 10^{-3}$.

Chapter 4

Simulations

This chapter presents simulations of three generic problems of interest in computational fluid dynamics and of a problem with active source terms. The simulations which follow were executed with a fixed time step $\Lambda = 0.1, \nu = 3$ and $\epsilon = 0.1$. The results of these simulations demonstrate that τ is small for realistic problems of interest and elapsed time is a fraction of a second for all instances. As table 3.1 suggests lower times can be achieved by permitting Λ, ν and ϵ to vary with N. All timings are for a J-machine[1] [34] with 33 MHz processors.

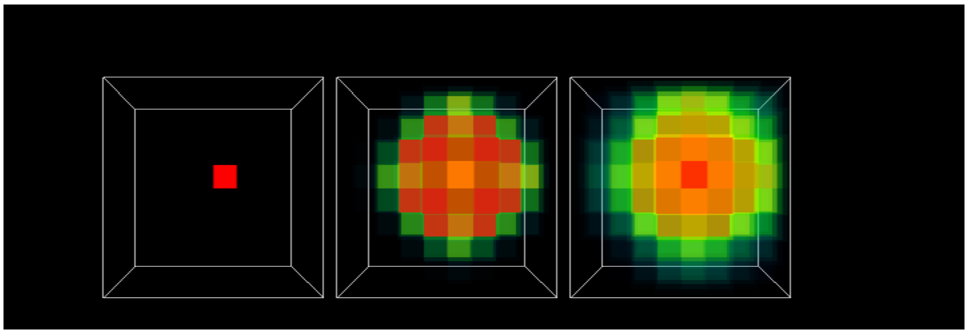
Spectral bisection [38] is a popular method to solve instances of the static problem. This method can require considerable cpu time for large problems. Algorithm **DIFFUSION** solved an instance with one million unknowns and 512 computers [41] in a few hundred milliseconds.

Solution methods for problems in fluid dynamics and structural analysis often increase the density of a computational grid in response to properties of the solution. A bow shock resulting from the preceding calculation and a generic launch sequence with a moving boundary were simulated for a problem with one billion unknowns on one million computers. The grid density was increased 100% in regions of the shock and 400% at the moving boundary. Both of these disturbances were removed by algorithm **DIFFUSION** in a few hundred milliseconds.

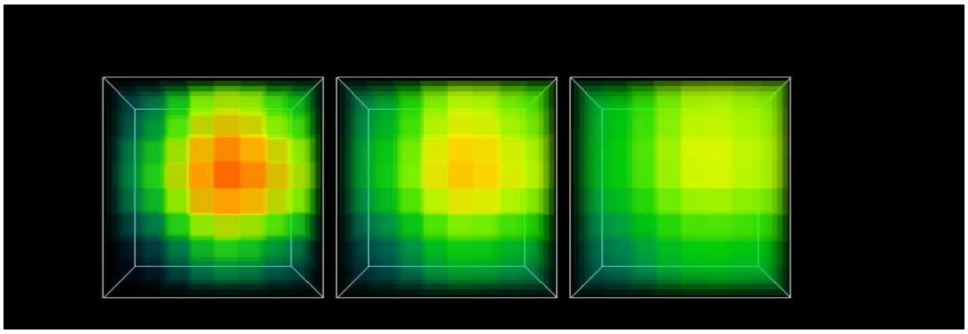
The algorithm is formulated under the assumption that no new work is created during the time the algorithm is executing. A final simulation shows that the algorithm is robust even in the presence of large and frequent injections of new work. Algorithm **DIFFUSION** kept the disturbance below the magnitude of the average injection for 1000 iterations. The disturbance quickly dissipated after the injection

[1]The J-machine is an experimental design built at MIT.

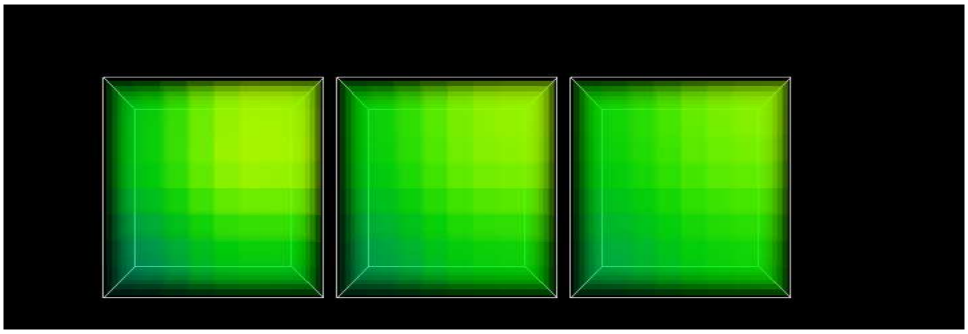
ceased.



0 ms 34.375 ms 68.75 ms

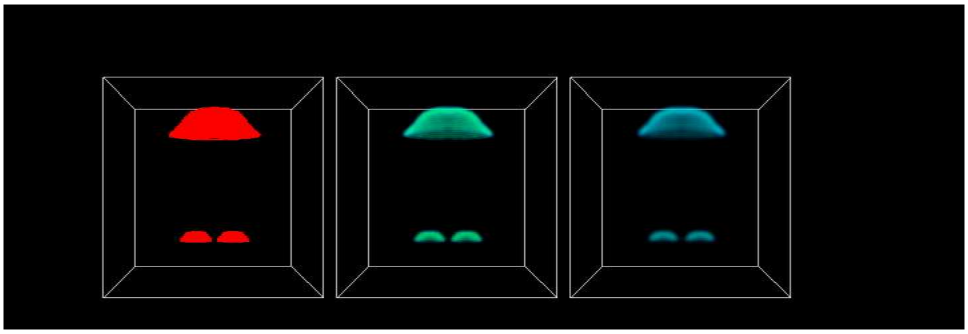


103.125 ms 137.5 ms 171.875 ms

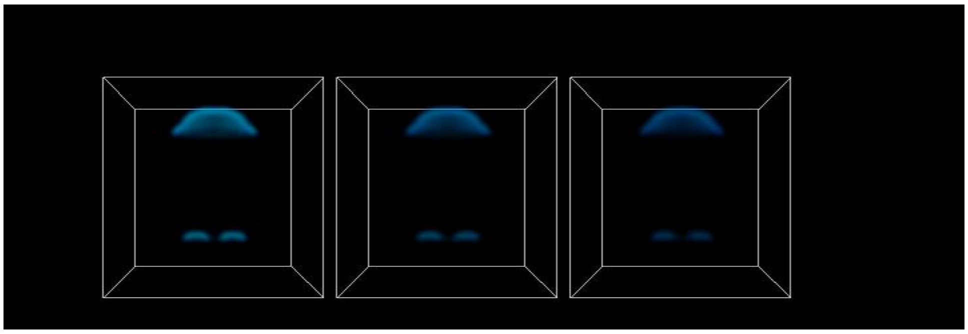


206.25 ms 240.625 ms 275 ms

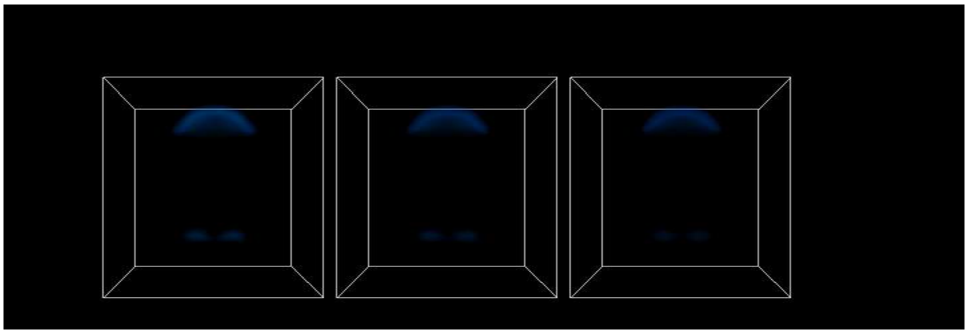
Figure 4.1: Static load balancing problem for a grid of one million points on a system of 512 computers. The initial imbalance was reduced by 90% in 6 steps. Condition (2.1.b) was satisfied to within 10% of the load average in 162 steps.



0 ms 34.375 ms 68.75 ms



103.125 ms 137.5 ms 171.875 ms



206.25 ms 240.625 ms 275 ms

Figure 4.2: Dynamic load balancing problem following h-refinement adaptation for a grid of one billion points on a system of one million computers. Condition (2.1.b) was satisfied to within 10% of the load average in 170 steps.

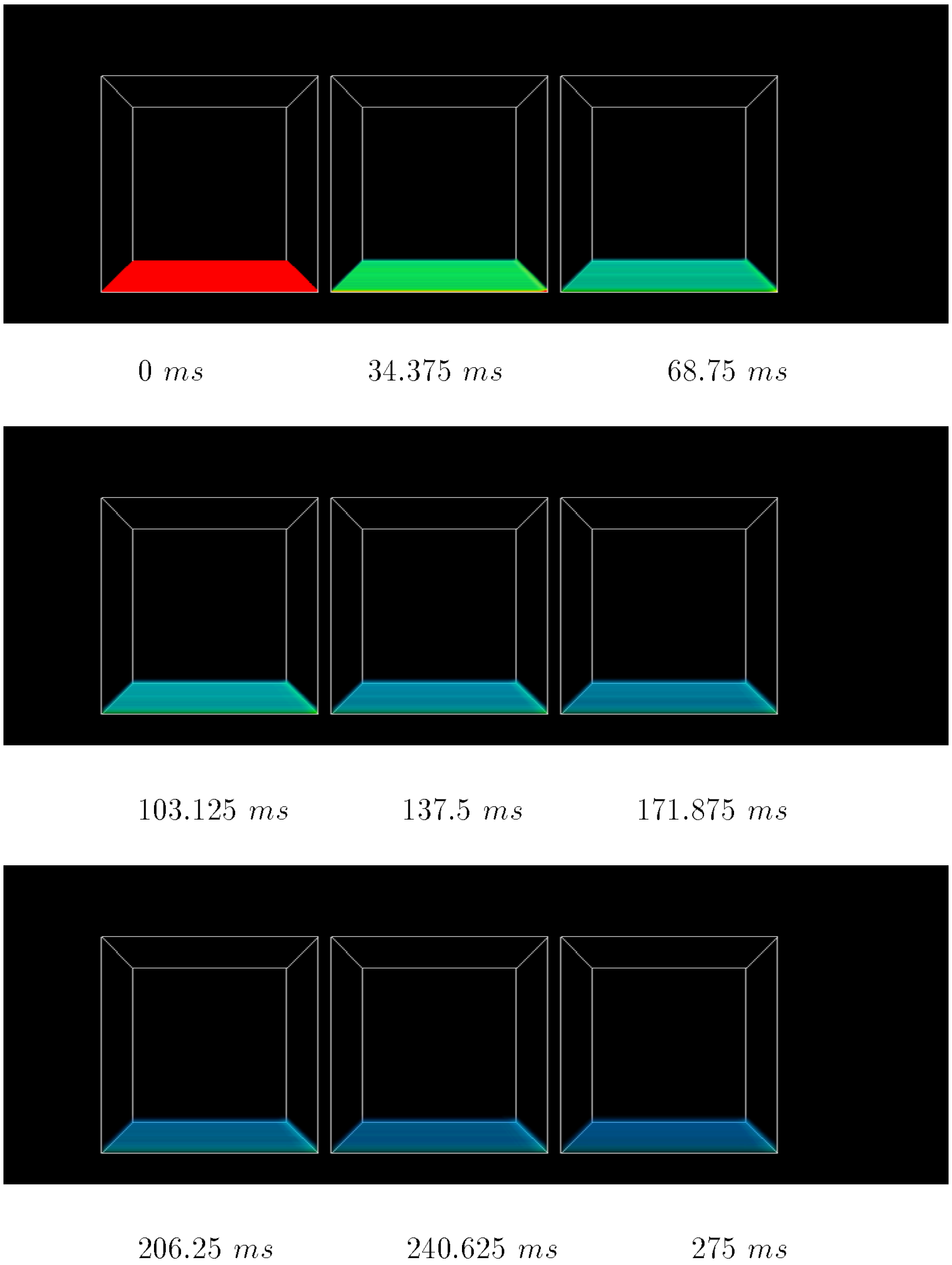


Figure 4.3: Dynamic load balancing problem following moving boundary adaptation for a grid of one billion points on a system of one million computers. Condition (2.1.b) was satisfied to within 10% of the load average in 50 steps.

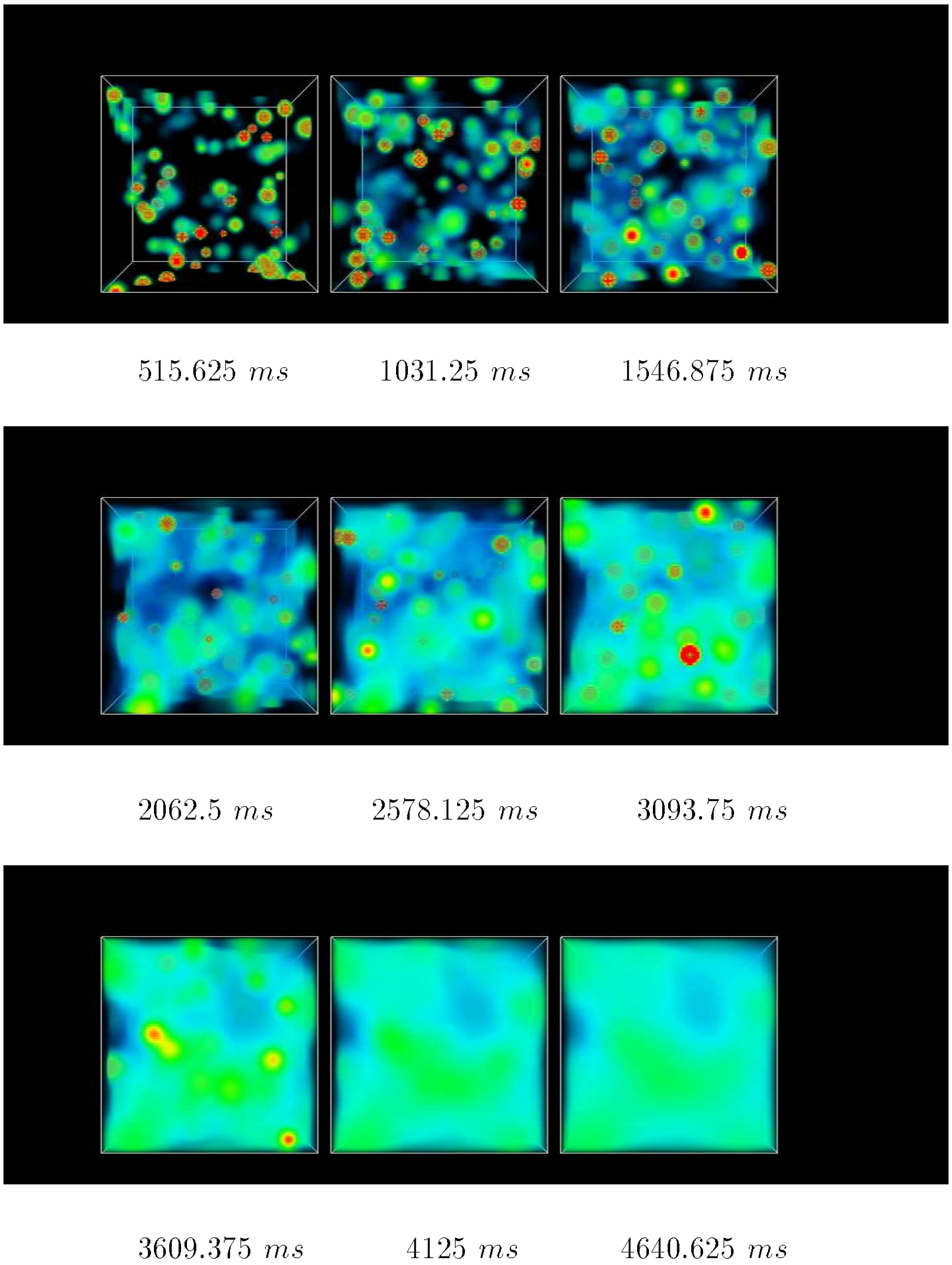


Figure 4.4: Problem with active sources. Average activity per iteration is 30,000% of the initial load average. Activity ceased at step 1000. Error in condition (2.1.b) decreased by three orders of magnitude in 500 additional iterations.

Chapter 5

Scalable load balancing methods

The load balancing problem has been the subject of considerable discussion in the past decade [2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 16, 19, 20, 22, 23, 25, 28, 29, 30, 31, 32, 33, 37, 38, 39, 42, 43]. This discussion has suffered from a lack of consensus regarding the architectures to be considered and the constraints under which the problem should be solved. An unfortunate consequence of this fact is that the discussion has little continuity and new methods rarely build on previous work. Most methods rely on simulation to demonstrate their effectiveness and rarely have formal proofs of correctness. As a result it can be difficult to extrapolate their behavior to architectures other than those on which they were simulated. Very few methods consider the issues of scalability and adjacency. Some noteable approaches among this group of methods are gradient based models [29], bidding algorithms [31], a drafting algorithm [32] and a method based on queing theory [11]. Most recently a spectral bisection method has become popular for problems which involve complex geometries [38].

5.1 Diffusion

The first published work on diffusion methods is due to Cybenko [14]. This work proposes a method to compute a balanced distribution of work in an arbitrarily connected graph. The method assigns a weighting factor $\alpha_{i,j}$ to every edge i, j of a graph where each vertex i is associated with a workload w_i. The method computes an iteration at each vertex i concurrently

$$w_i^{(t+1)} \leftarrow w_i^{(t)} + \sum_j \alpha_{i,j} \left(w_j^{(t)} - w_i^{(t)} \right) + \eta_i^{(t+1)}$$

In this model the α are constrained so that $\sum_{j\neq i}\alpha_{i,j} = 1$. Let M denote the matrix where $M_{i,j} = \alpha_{i,j}$ for every edge i,j in the graph and $M_{i,j} = 0$ if there is no edge between vertices i and j. η represents a stochastic source of new work arising during the rebalancing process. Then when $\eta = 0$ the iteration can be expressed as a matrix equation

$$\vec{w}^{(t+1)} \leftarrow M\vec{w}^{(t)} \tag{5.1}$$

The article demonstrates convergence of (5.1) and hence correctness of the algorithm under stationary conditions.[1] The constraint on α requires that M has row and column sums equal to 1 and hence the spectral radius is bounded $\rho(M) \leq 1$ ([24] thm. 8.1.22). Since M is nonnegative and irreducible the Frobenius generalization implies that M has a single eigenvalue of modulus 1 and that the corresponding eigenvectors are uniform distributions in w. M can have an eigenvalue of -1 only if it can be partitioned into block form

$$\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}$$

which is a condition for the interconnection graph to be bipartite. Therefore the algorithm does not oscillate so long as the interconnection graph is not bipartite.

This work does not explicitly consider the rate of convergence but suggests an acceleration technique to minimize $\rho(M)$

$$M_\kappa = \frac{M + \kappa I}{1+\kappa}, \kappa = -(\lambda_n + \lambda_2)/2$$

Adding κI to a diagonal M shifts the eigenvalues λ_i of M to $\lambda_i + \kappa$. The division by $1+\kappa$ normalizes the row and column sums of M to 1 which is necessary in order for the proof of convergence to hold.

The work considers the problem of active sources $\eta > 0$ and demonstrates that if σ is the variance in an initial distribution and n the number of processors then the asymptotic expected value of the variance is

$$E = \sigma^2 \left(\frac{N-1}{1-\rho(M)^2} \right)$$

The article concludes by considering a *dimension exchange* algorithm for binary hypercubes in which all $\alpha_{i,j} = 1/d$ where d is the dimensionality of the hypercube.

[1] The proof is by Frobenius's generalization for nonnegative irreducible matrices of Perron's theorem ([24] thm. 8.2.11). It proves boundedness of the spectral radius and gives conditions for a -1 eigenvalue.

This is a recursive algorithm in which each dimension in turn balances the work among it's set of computers. The algorithm terminates after d serial steps and scales with elapsed time $\mathcal{O}(\log_d N)$.

This article has been a precursor of several subsequent scalable and correct load balancing methods for distributed memory computers [6, 13, 21, 25, 43]. For example, implementations of this algorithm can bear a strong resemblance to algorithm **DIFFUSION**. In the absence of source terms ($\eta = 0$) and under the assumption that the graph represents the interconnection structure of a three dimensional mesh connected multicomputer matrix M in (5.1) is equivalent to the expression $\Lambda\mathcal{L} + I$ where L is taken from (3.5). Cybenko's method also bears a strong resemblance to a proposal by Boillat [6]. This method is also formulated for an arbitrary interconnection graph. When applied to a three dimensional interconnection mesh for a multicomputer the algorithm is

$$\vec{x}^{(t+1)} = P_G \vec{x}^{(t)} \tag{5.2}$$

where P_G is a matrix with the same sparsity structure as L in equation (3.5) but where every nonzero term is 1/7. While this does not correspond directly to a finite difference expression it is obviously convergent by the same arguments presented in [14]. The article uses Markov techniques to demonstrate that the iteration (5.2) is acyclic and converges to equilibrium. It considers the rate of convergence and derives an upper bound for eigenvalues. On a three dimensional mesh this upper bound is

$$\lambda \leq 1 - \frac{4}{7}\sin^2\frac{\pi}{2n}$$

Using this bound it demonstrates that worst case convergence has an upper bound elapsed time complexity of $\mathcal{O}(n^2)$. It is unfortunate that this method has received little attention as it is one of the few which has been implemented in the context of real scientific calculations [7, 9]. This may be partly the result of an obscurity of notation and presentation.

5.2 Transfer function

A novel formulation of the problem of calculating $\nabla\vec{u}$ is introduced by Conley [13]. The formulation, which arose in part from discussions of the load balancing problem with this author, requires solution of an elliptic equation

$$\nabla^2\vec{T} = -\nabla\vec{u} \tag{5.3}$$

where the resulting transfer of loads is effected by

$$\Delta\vec{u} = \nabla \cdot \vec{T} \tag{5.4}$$

If u_b represents the average workload then correctness of (5.3),(5.4) follow from

$$\begin{aligned}
\nabla \cdot \vec{T} &= u_b - \vec{u} \\
\nabla\left(\nabla \cdot \vec{T}\right) &= \nabla u_b - \nabla\vec{u} \\
-\nabla \times \left(\nabla \times \vec{T}\right) + \nabla^2\vec{T} &= -\nabla\vec{u} \\
\nabla^2\vec{T} &= -\nabla\vec{u}
\end{aligned} \tag{5.5}$$

The final step of (5.5) assumes an irrotational constraint $\nabla \times \vec{T} = 0$. Irrotational solutions are a subset of the set of valid solutions of problem 2.1.

Problems like (5.3) are well suited to solution by scalable concurrent iterations similar to (3.4). Simulations by this author using iterative solutions of a finite difference formulation of (5.3) have shown that overall solution times to fine $\Delta\vec{u}$ are similar to times for algorithm **DIFFUSION**.

Transfers must occur as a second step of the algorithm. This means that a load balancing method based on this calculation would have a serial dependency. This dependency could be expensive for large disturbances. Diffusion methods do not suffer from this problem because they transfer work over many steps concurrently with the calculation of $\vec{u}'$.

5.3 A multilevel method

Multigrid methods [40] are a popular way to accelerate convergence of iterative solution methods for linear systems of equations. Horton [25] suggests that a similar approach can accelerate the convergence of diffusion methods.

The article presents a "multilevel" algorithm for load balancing which has logarithmic elapsed time complexity. This algorithm requires that the aggregate workload among a subset of computers is known at each step. Although this is certainly feasible it suggests the use of an algorithm similar to **SIMPLE** and **SIMPLE2** of the second chapter. This suggests that this algorithm may not be particularly efficient in comparison to diffusion methods.

The inner loop of a "basic diffusion model" is presented as

`for all neighbors` j `of` ψ_i `do`

`transfer` $(u_i - u_j)/2$ `units of work from` ψ_i `to` ψ_j

In one dimension this transaction is equivalent to

$$\begin{aligned} u'^{(m+1)}_i &= u'^{(m)}_i - \frac{1}{2}\left(u'^{(m)}_i - u'^{(m)}_{i+1}\right) - \frac{1}{2}\left(u'^{(m)}_i - u'^{(m)}_{i-1}\right) \\ &= \frac{1}{2}(u'^{(m)}_{i+1} + u'^{(m)}_{i-1}) \end{aligned}$$

This is algorithm **AVERAGE** which was shown in the second chapter to be incorrect in the absence of periodic boundary conditions. On a hypercube architecture this algorithm is identical to Cybenko's dimension exchange.

5.4 A distributed task pool

The task pool algorithm of Hofstee et al [22] is scalable and correct. This algorithm is concerned with distributing a pool of tasks to a set of computers in a way that ensures load balance. Processes are assumed to be independent and no assumptions are made regarding the order of execution or adjacency relationship among the processes. An example of an application for which this algorithm could be beneficial might be an online transaction processing system in which a large number of tasks (transactions) queue in a task pool until they can be serviced by a multicomputer. The algorithm is not intended for domain decomposed calculations in which tasks must execute concurrently and in which the adjacency constraint (2.1.c) must be observed. Because of this the algorithm appears to be less useful for grid computations.

Bibliography

[1] Amdahl, G. Validity of the single processor approach to achieving large scale computing capabilities. *Proc. AFIPS Comput. Conf.* **30** (1967) 483–485.

[2] Athas, W. C. & Seitz, C. L. Multicomputers: message passing concurrent computers. *IEEE Comp.* **21** (1988) 9–24.

[3] Baden, S. B. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. *SIAM J. Sci. Stat. Comp.* **12**:1 (1991) 145–157.

[4] Barak, A. & Shiloh, A. A distributed load-balancing policy for a multicomputer. *Software—Pract. Exp.* **15**:9 (1985) 901–913.

[5] Bertsekas, D. P. & Tsitsiklis, J. N. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[6] Boillat, J. E. Load balancing and Poisson equation in a graph. *Concurrency: Pract. Exp.* **2** (1990) 289–313.

[7] Boillat, J. E., Brugé, F. & Kropf, P. G. A dynamic load balancing algorithm for molecular dynamics simulation on multiprocessor systems. *J. Comp. Phys.* **96**:1 (1991) 1–14.

[8] Bokhari, S. H. On the Mapping Problem. *IEEE Trans. Comput.* **C-30**:3, pp. 550–557 (1981).

[9] Brugé, F. & Fornili, S. L. A distributed dynamic load balancer and it's implementation on multi-transputer systems for molecular dynamics simulation. *Comp. Phys. Comm.* **60** (1990) 39–45.

[10] Bokhari, S. H. On the mapping problem. *IEEE Trans. Comp.* **C-30**:3 (1981) 207–214.

[11] Chou, T. C. K. & Abraham, J. A. Load balancing in distributed systems. *IEEE Trans. Soft. Eng.* **SE-8**:4 (1982) 401–412.

[12] Chow, Y. & Kohler, W. H. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Trans. Comp.* **C-28**:5 (1979) 354–361.

[13] Conley, A. J. Using a transfer function to describe the load balancing problem. Argonne National Laboratory report ANL-93/40 (1993).

[14] Cybenko, G. Dynamic load balancing for distributed memory multiprocessors. *J. Par. Distrib. Comp.* **7** (1989) 279–301.

[15] Eager, D. L., Lazowska, E. D. & Zahorjan, J. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Soft. Eng.* **SE-12**:5 (1986) 662–675.

[16] Efe, K. Heuristic models of task assignment scheduling in distributed systems. *IEEE Comp.* **18** (1982) 50–56.

[17] Gupta, A. & Kumar, V. Performance properties of large scale parallel systems. *J. Par. Distrib. Comp.* **19** (1993) 234–244.

[18] Gustafson, J. L. Reevaluating Amdahl's law. *Comm. ACM* **31** (1988) 532–533.

[19] Hać, A. Load balancing in distributed systems: a summary. *Perf. Eval. Rev.* **16** (1989) 17–19.

[20] Hanxleden, R. V. & Scott, L. R. Load balancing on message passing architectures. *J. Par. Dist. Comp.* **13** (1991) 312–324.

[21] Heirich, A. & Taylor, S. How to balance a million nodes. *Caltech Computer Science Department Technical Report* (1993).

[22] Hofstee, H. P., Lukkien, J. J. & van de Snepscheut, J. L. A. A distributed implementation of a task pool. In *Research Directions in High Level Parallel Programming Languages,* Banatre, J. P. & Le Metayer, D. (eds.) (1992) 338-348, Springer-Verlag.

[23] Hong, J., Tan, X. & Chen, M. From local to global: an analysis of nearest neighbor balancing on hypercube. *Proc. ACM Sigmetric Conf. on Measurement and Modeling of Comp. Sys.* (1988) 73–82.

[24] Horn, R. A. & Johnson, C. R. *Matrix Analysis.* (1991) Cambridge University Press, New York.

[25] Horton, G. A multi-level diffusion method for dynamic load balancing. *Par. Comp.* **19** (1993) 209–218.

[26] Karp, R. M. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, Miller, R. E. & Thatcher, J. W. (eds). (1972) New York: Plenum Press 85–104.

[27] Keckler, S. W. & Dally, W. J. Processor coupling: integrating compile time and run time scheduling for parallelism. *Proc. ACM 19th Int. Symp. on Comp. Arch.*, Queensland, Australia (1992) 202-213.

[28] Kumar, V., Ananth, G. Y. & Rao, V. N. Scalable load balancing techniques for parallel computers. Preprint 92-021. (1992) Army High Performance Computing Research Center, Minneapolis, MN.

[29] Lin, F. C. H. & Keller, R. M. The gradient model load balancing method. *IEEE Trans. Soft. Eng.* **SE-13** (1987) 32–38.

[30] Marinescu, D. C. & Rice, J. R. Synchronization and load imbalance effects in distributed memory multi-processor systems. *Concurrency: Pract. Exp.* **3** (1991) 593–625.

[31] Mirchandaney, R., Towsley, D. & Stankovic, J. A. Adaptive load sharing in heterogeneous distributed systems. *J. Par. Dist. Comp.* **9** (1990) 331–346.

[32] Ni, L. M., Xu, C. & Gendreau, T. B. A distributed drafting algorithm for load balancing. *IEEE Trans. Soft. Eng.* **SE-11** (1985).

[33] Nicol, D. M. & Saltz, J. H. Dynamic remapping of parallel computations with varying resource demands. *IEEE Trans. Comp.* **37** (1988).

[34] Noakes, M. & Dally, W. J. System design of the J-machine. In *Proc. 6th MIT Conf. on Advanced Research in VLSI.* Dally, W. J. (ed.). (1990) MIT Press, Cambridge, MA. 179–194.

[35] Rowell, J. Lessons Learned on the Delta. *High Perf. Comput. Rev.* **1** (1993) 21–24.

[36] Seitz, C. L. Mosaic C: an experimental fine-grain multicomputer. *Proc. International Conference Celebrating the 25th Anniversary of INRIA, Paris, France, December 1992*, Springer-Verlag.

[37] Shen, S. Cooperative Distributed Dynamic Load Balancing. *Acta Informatica* **25** (1988) 663–676.

[38] Pothen, A., Simon, H. D. & Liou, K. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM J. Matrix Anal.* **11** (1990) 430–452.

[39] Van Tilborg, A. M. & Wittie, L. D. Wave scheduling: decentralized scheduling of task forces in multicomputers. *IEEE Trans. Comp.* **C-33**:9 (1984) 835–844.

[40] Vandewalle, S. & Horton, G. Multicomputer-multigrid solution of parabolic partial differential equations. Report TW 196, Department of Computing Science, Katholieke Universiteit Leuven, Leuven, Belgium (1993).

[41] Wang, J. C. T. & Taylor, S. A concurrent navier-stokes solver for implicit multibody calculations. To appear in *Proc. Parallel-CFD '93*, Paris, France, June 1993.

[42] Williams, R. D. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Pract. Exp.* **3**:5, pp. 457–481, 1991.

[43] Xu, C. Z. & Lau, F. C. M. Analysis of the generalized dimension exchange method for dynamic load balancing. *J. Par. Dist. Comp.* **16** (1992) 385–393.