Analysis of Scalable Algorithms for Dynamic Load Balancing and Mapping with Application to Photo-realistic Rendering

Thesis by

Alan Heirich

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

California Institute of Technology
Pasadena, California

1998
(Submitted May 1997)

Acknowledgements

The material in this thesis has previously appeared in four journals and four conference proceedings: the journals *Parallel Computing, The Journal of Supercomputing, The International Journal of Foundations of Computer Science,* and *The International Journal of Advances in Engineering Software*; and proceedings of *the 1995 International Conference on Parallel Processing, Eurographics'97 (workshop on programming paradigms in graphics), the First Eurographics workshop on parallel rendering and visualization,* and *the 4th National Symposium on Large Scale Analysis and Design on High Performance Computers and Workstations.*

For support, in addition to the N.S.F. Center for Research in Parallel Computation, I have to thank Paul Messina of Caltech's Center for Advanced Computing Research, and Don Greenberg of the Program of Computer Graphics at Cornell University.

For everything else I have to thank my chairman, Jim Arvo, and thesis committee Al Barr, Mani Chandy, Carl Kesselman, Peter Schröder, and Steve Wiggins, each of whom has made my experience of Caltech remarkable in a different way; previous committees and/or advisors Steve Taylor, Christof Koch, Dan Meiron, Yaser Abu-Mustafa, and Alain Martin; others who have made Caltech unforgettable: Herb Keller, Andrew Conley, Michael Holst, Roy Williams, Cheryl Carey, Eric van de Velde, Peter Hofstee, Jan van de Snepscheut, Joel Franklin, and Thomas Caughey; my parents, for showing me the value of advanced education; Michele Titus, for putting up with me; and our children Laura, Nicole, and John, for teaching the things that we forget as we get older.

Abstract

This thesis presents and analyzes scalable algorithms for dynamic load balancing and mapping in distributed computer systems. The algorithms are distributed and concurrent, have no central thread of control, and require no centralized communication. They are derived using spectral properties of graphs: graphs of physical network links among computers in the load balancing problem, and graphs of logical communication channels among processes in the mapping problem. A distinguishing characteristic of these algorithms is that they are scalable: the expected cost of execution does not increase with problem scale. This is proven in a *scalability theorem* which shows that, for several simple disturbance models, the rate of convergence to a solution is independent of scale. This property is extended through simulated examples and informal argument to general and random disturbances. A worst case disturbance is presented and shown to occur with vanishing probability as the problem scale increases. To verify these conclusions the load balancing algorithm is deployed in support of a photo-realistic rendering application on a parallel computer system based on Monte Carlo path tracing. The performance and scaling of this application, and of the dynamic load balancing algorithm, are measured on different numbers of computers. The results are consistent with the predictions of scalability, and the cost of load balancing is seen to be non-increasing for increasing numbers of computers. The quality of load balancing is evaluated and compared with the quality of solutions produced by competing approaches for up to 1,024 computers. This comparison shows that the algorithm presented here is as good as or better than the most popular competing approaches for this application. The thesis then presents the dynamic mapping algorithm, with simulations of a model problem, and suggests that the pair of algorithms presented here may be an ideal complement to more expensive algorithms such as the well-known recursive spectral bisection.

Contents

Acknowledgements **iii**

Abstract **iv**

1 Introduction **1**

1.1 Summary of original contributions . . . 3

2 Dynamic Load Balancing **6**

2.1 Field equations . . . 6

2.2 Computer graphics . . . 7

2.3 The load balancing problem . . . 10

2.4 Some approaches to load balancing . . . 12

2.4.1 A naive approach . . . 12

2.4.2 Recursive bisection . . . 13

2.4.3 Diffusion . . . 14

2.5 A load balancing algorithm . . . 16

3 Convergence and Scaling **18**

3.1 Derivation of algorithm 1 . . . 18

3.2 Correctness of algorithm 1 . . . 21

3.3 Worst case analysis . . . 21

3.4 A scalability theorem . . . 24

3.4.1 Multiple and random disturbances . . . 25

3.5 Time dependent behavior . . . 26

4 An Experimental Validation **32**

4.1 A pinhole camera model . . . 32

4.2 A computational procedure . . . 34

4.2.1 A path tracing iteration . . . 35

4.2.2 Sampling strategies 36
4.3 A concurrent implementation 38
4.4 Analysis of expected initial conditions 42
4.5 Measurements of observed scaling 45
4.5.1 Measurement of algorithm 1 50

5 Dynamic Mapping **54**
5.1 The mapping problem 54
5.2 A dynamic mapping algorithm 55
5.3 Simulations of a model problem 57
5.4 Discussion 58

6 A Load Balancing Trace **61**

Bibliography **70**

List of Figures

2.1 *Characteristics of the* 16×16 *Laplacian matrix of a regular mesh. Left, the complete eigenvalue spectrum. Right, two representative eigenvectors. The eigenvector with the larger variation is the Fiedler vector.* 15

3.1 *The "Achilles heel" of equations 3.2 and 3.5. This sinusoidal disturbance is the eigenvector* $\vec{X}_{i,j}$ *that corresponds to the most slowly converging eigenvalue* $\lambda_{i,j}$ *of equation (3.2). The standard iterations cannot converge this disturbance effectively.* 22

3.2 *The initial conditions and first five iterates of a point disturbance under equation 3.2. The center point shows oscillatory convergence.* 27

3.3 *Point disturbance under equation 3.5 shows monotone convergence.* 28

3.4 *Left, height of the point disturbance over 10 iterates of equations 3.2 and 3.5. Equation 3.2 shows oscillation while algorithm 3.5 converges monotonically. Right, eigenvalue* $\lambda_{i,8}$ *for* $n = 16$ *of iteration matrices* A *and* AA *corresponding to the two equations.* A *has mixed eigenvalues while* AA *is positive definite.* 28

3.5 *Convergence of a problem on a* 64×64 *grid of* 4096 *unknowns. The height of the initial disturbance is the same in this series as in figure 3.2 affirming the conclusion of the Scalability Theorem.* 29

3.6 *Convergence of a Gaussian disturbance under equation 3.5.* 30

3.7 *Convergence of a pair of Gaussian disturbances under equation 3.5.* 31

4.1 *Left: computing Monte Carlo samples of direct illumination. The image plane corresponds to the rear wall of a pinhole camera. A path extends from the image plane out through the camera aperture to a visible surface at x. The light intensity reflected into the camera along this path is equal to the sum of the light intensities at x contributed from all visible light sources, multiplied by a reflection coefficient. Right: the reflectance phenomena described by equation (4.1). One out of the many paths by which light is transported to contribute to the value of a pixel. Light from surface point x' is transported to point x in direction ω'. Point x' may be a light emitter or may simply reflect light from other points, or both. A fraction of the total light reaching x at any instant is reflected in the direction ω which in this figure leads into the camera aperture. In the general case ω could lead to another reflector.* 33

4.2 *Clockwise from upper left: glass, conference, office, soda, and bath images. These were computed at NTSC resolution* (640×480) *and in full color.* . . 39

4.3 *The result of applying a random assignment strategy to a uniformly distributed sample population with mean $\mu = 10^6$ and variance $\sigma^2 = 10^{12}$. Left figure shows the probability Pr^n (see equation 4.6) of obtaining acceptable ϵ for (left to right) ϵ of 0.05, 0.10, 0.15, 0.20, and 0.25 for increasing numbers of computers (horizontal axis). The probability of obtaining $\epsilon \leq 0.10$ drops below 0.5 somewhere between 128 and 192 computers. Right figure shows the predicted ϵ resulting from a random assignment of this population to increasing numbers of computers (horizontal axis).* 44

4.4 *Distribution histograms and predicted imbalances for (top to bottom) office, soda, and bath images. All three images have highly nonuniform distributions as illustrated by their histograms (left column). These histograms show frequency of occurrence (vertical axis) versus floating point operations (horizontal bins). For each image the predicted imbalance ϵ (right column) is shown for three static load balancing strategies (naive, scatter, and random) on increasing numbers of computers (horizontal axis). In all cases the naive strategy produces the largest predicted ϵ (vertical axis) and the random strategy produces the smallest. The scatter decomposition is generally comparable to the random strategy but suffers from isolated peaks at multiples of 320 pixels, exactly one half the width of the image. In all cases the predicted ϵ is greater than 0.10 when the number of computers is 128 or higher.* 46

4.5 *Clockwise from lower left: measured ϵ versus n for the bath, for the cases shown in table 4.2, including four load balancing scenarios; the same for the office, and soda images; and predictions of ϵ resulting from a random initial mapping based on counts of floating point operations from figure 4.4. For every image the best results (lowest line) occur when algorithm 1 follows an initial scatter method. The predicted ϵ is offered for comparison to this best case, and appears to correlate moderately well, particularly for the soda image. This correlation suggests that the increase in measured ϵ under algorithm 1 may be due to the initial imbalance rather than to any failure of scaling. Unfortunately there is not enough data to prove or disprove this.* 47

4.6 *Comparison of effective imbalance ϵ with and without dynamic load balancing by algorithm 1, for the three images in table 4.2. Clockwise from upper left: naive method only; naive method followed by algorithm 1; scatter method followed by algorithm 1; scatter method alone. In several cases the initial imbalance seems to correlate with the the effective imbalance under algorithm 1.* 48

4.7 *Average number of milliseconds per computer spent in algorithm 1 versus number of computers. See table 4.5. The office, soda, and bath images were computed using identical parameters on 16, 32 and 64 computers. This data was collected from the same runs as the data in table 4.2 but shows only the cases that include algorithm 1. For an explanation of how the measurements were collected please see the final chapter. The left figure shows the office and soda measurements, and the right figure shows the bath measurements. All times are in milliseconds. In every case the naive method led to more time being spent in algorithm 1 than the scatter method did. The number of computers showed no influence on these measurements. All of these results are consistent with a hypothesis that algorithm 1 is insensitive to problem scale for realistic problem instances.* 52

5.1 *Perfect equilibria are not possible for all cases, but an approximation to equilibrium is still a good solution to the mapping problem. Left, the setting for a model problem on a grid of 16 computers. A rectangular region has been partitioned so that each computer occupies a subregion and the interfaces between these subregions represent network links. Middle, an ideal solution for an easy problem with a set of 1024 processes that communicate in a regular grid. In this solution each process is equidistant from all processes with which it communicates. Right, a solution for a realistic problem in which set of processes perform a structured multigrid calculation. This process graph is non-planar, and the solution shown here is the best that can be achieved even though it is not in perfect equilibrium.* 56

5.2 *An example of a regular graph sorting itself out from a difficult initial condition. This required constraining the corner positions. Note the persistent sinusoid.* 59

5.3 *The same problem, started from a nice initial condition. The final figure still has a persistent sinusoid, with points clustered toward the periphery.* 59

5.4 *Two incorrect solutions that can result from the initial condition of figure 5.2.* 59

List of Tables

4.1 *Probability (Pr^n) of obtaining $\epsilon \leq 0.10$ by random assignment. See equation (4.6). μ and σ^2 represent the mean and variance of the number of floating point operations required to compute an NTSC resolution image using an adaptive sampling strategy of from 1 to 25 samples per pixel. In no case is random assignment likely to be effective for more than 128 computers. This conclusion is independent of the type of ray tracing algorithm used and the distribution of the sample population w_i.* 45

4.2 *Empirical results measuring effective imbalance ϵ in rendering the office, soda, and bath images with different load balancing strategies on varying numbers of computers. See figure 4.5. In every case the best results came from dynamic load balancing by algorithm 1 preceded by the scatter method ("balanced") In most cases algorithm 1 by itself produced better results than scattering even from poorly balanced initial conditions ("unbalanced"). In general the problem becomes more difficult with increasing numbers of computers. These results have been reported in publications [21, 22, 25].* 49

4.3 *Results of an initial study on the SP2. In all of these runs the naive strategy was used for the initial mapping followed by algorithm 1. This study revealed the existence of errors in the original distributed algorithm for termination detection. These errors showed up the runs on 128 and 256 computers, where they caused a slowdown rather than a speedup. These results have been reported in publications [24, 26].* 51

4.4 *Rendering times on Intel cluster. These runs were computed with parameters identical to table 4.3 and are directly comparable. In every respect the performance was slightly better than on the SP2. These results have been reported in publications [24, 26].* 51

4.5 *Time measured within algorithm 1, in milliseconds, and total elapsed time, in seconds, for six cases. See figure 4.7. Each of three images was started from two initial mappings. One mapping had a reasonably well balanced workload while another was poorly balanced. These measurements show that the amount of work expended by algorithm 1 is affected by the amount of initial imbalance but not by problem scale.* 51

6.1 *Parameters that determine the sampling rates used in algorithm 2 and values used for typical runs. Some of the runs used values of PIXEL_SAMPLES as high as* 100*. For the office, soda, and conference models the value of PATH_GENERATIONS was 2 so that only direct lighting was computed.* . 61

Chapter 1 Introduction

This thesis presents scalable dynamic algorithms to solve the problems of load balancing and mapping in distributed computer systems. The *load balancing* problem partitions a large set of concurrent processes into equally balanced subsets for execution on a small set of computers. The *mapping problem* partitions the same set of processes into connected subgraphs for the same purpose.[1] Both problems have the goal of optimizing the performance of the processes on the computers: in the case of load balancing, to equalize (and therefore minimize) the time required by each computer; in the case of mapping, to minimize the amount of communication among computers.

These algorithms are constructed using an approach based on spectral properties of graphs: graphs of physical network links among computers in the load balancing problem, and graphs of logical communication channels among processes in the mapping problem. In both problems the approach starts by characterizing problem solutions as equilibrium distributions of appropriately chosen scalar quantities. For the load balancing problem this characterization is easily arrived at, since every solution is an equilibrium distribution of workload among a set of computers. For the mapping problem the characterization is somewhat less obvious, and the equilibrium sought is an equal distribution of *communication distance* as measured along paths through a physical interconnection network.

After the problem has been formulated in this way an algorithm is constructed from an iterative procedure to solve the Laplace equation on the vertices of the graph. The procedure to solve the Laplace equation is analogous to a process in which an initial disequilibrium condition diffuses (or relaxes) to an equilibrium. The iterative procedure yields a matrix of coefficients known as the *Laplacian matrix* of the graph. Spectral properties of this matrix can be used to analyze the convergence and scaling properties of the resulting algorithms. The thesis uses these properties to prove a *scalability theorem* which says that the resulting algorithms will reduce a discrete disturbance at a constant rate regardless of the scale of the system in which the disturbance occurs. Constant convergence can be a

[1] The mapping and load balancing problems are similar to a number of other important problems, such as the problems of partitioning circuits for VLSI placement and simulation. The approach that is demonstrated here may be applicable to some of these other problems.

useful property for algorithms that execute on scalable computer systems, and a necessary property for the largest systems.

The scalability theorem assumes a simplified model in which individual imbalances arise, one after another, with each imbalance confined to a single computer. In real applications this simplified model may not be realistic. Imbalances may arise on several computers simultaneously, as a result of several unrelated events, or as a result of a single event that involves several computers. When the number of such events is fixed the rate of convergence of the algorithms is not significantly affected by problem scale, just as in the case of a single disturbance. Scale-independent convergence is also the expected behavior for any number of events, when they occur at random times, on randomly selected computers, and with random magnitude.

These claims of scalability are tested in an implementation of a distributed algorithm for image synthesis. A dynamic load balancing algorithm is implemented according to the approach presented here, and used to support a photo-realistic Monte Carlo path tracing algorithm. The scaling of these two algorithms are measured, and the observed load balancing solutions are compared to solutions produced by competing load balancing algorithms. The measurements show that the performance of the load balancing and Monte Carlo algorithms are consistent with the scalability theorem. The comparison shows that the load balancing algorithm is superior, in both cost and quality of solution, to a popular load balancing algorithm for ray-tracing on parallel computer systems.

A subsequent chapter presents the dynamic mapping problem and derives an algorithm to solve it. The mapping algorithm is applied to a simple model problem that illustrates some of the issues that arise in a distributed application. A final chapter discusses issues in developing the software for this study and shows a partial program trace for one of the many data points that were collected. The purpose of presenting this information is to aid in reproducibility, by showing the many ancillary issues that must be addressed in order to implement the complete application. It also shows explicitly how the data was gathered.

Scalability is the principal feature that distinguishes these algorithms from competing approaches. The algorithms are distributed and concurrent, have no central thread of control, and require no centralized communication. These properties are necessary for achieving scalable performance on the largest distributed computer systems. Examples of such systems include "massively parallel" computer systems which have hundreds and in some cases thousands of processors connected by high performance communication networks. Another example is the Internet which has millions of computers connected by a relatively poor network in which communication is slow and synchronization is impractical.

When scalability is not important the algorithms presented in this thesis have little advantage over competing approaches. The load balancing and mapping problems can always be solved by serial algorithms that compute a solution on a single computer and then broadcast the result to the other computers. A serial approach may often be the simplest to implement, and may be satisfactory for small parallel computer systems. However this approach will not give scalable performance on large computer systems, and may not even be feasible on a system such as the Internet. In these cases algorithms such as those presented here may be the only ones that are practical.

1.1 Summary of original contributions

This thesis presents novel problem formulations, algorithms, literature surveys, algorithm analysis, and empirical data. The key contributions are listed here.

- Problems 1 and 2 define the *dynamic load balancing* problem in a form that is amenable to analysis. These definitions make explicit the relationship between the problems of load balancing and mapping in a form that has not appeared previously. This form makes it possible to present a unified discussion of these two problems, and to analyze them both within a common formal framework.

- Section 2.4.3 ("diffusion") presents the most complete survey to date of a class of load balancing algorithms that are based on the informal concept of *diffusion*. This survey demonstrates that over the past several years a consensus has emerged about the preferred local load balancing algorithms for distributed applications. The emergence of this consensus has not been widely recognized.

- Algorithm 1 offers a distributed dynamic solution to the problems of load balancing, or of load balancing and mapping. The derivation and correctness (sections 3.1, 3.2) make explicit the relationship between this algorithm (and others discussed in section 2.4.3) and the Laplace equation. The explicitness of this relationship makes it possible to analyze these algorithms in terms of the spectral properties of the Laplacian matrix of a graph. The Laplacian matrix has been used extensively to analyze properties of algorithms for the mapping problem, and in particular, the *recursive spectral bisection* algorithm. It has not previously been available for analysis of load balancing algorithms.

- A *scalability theorem* proves that, for a purely local disturbance, the rate of convergence of algorithm 1 is constant on computer systems of arbitrarily large scale. This property also holds for disturbances that are not purely local as is shown through discussion and simulation. A model of the worst-case disturbance is presented for which algorithm 1 is ineffective. This worst-case is shown to have a vanishing probability of occurrence with increasing problem scale. In addition, it is shown that if the worst-case characteristics are not present in a problem at small scale, then they will not be present at large scales either. These two characteristics are in sharp contrast to problems that commonly arise in solving differential equations by iterative algorithms and that give rise to research in *multigrid methods* [43]. In such problems the worst-case behavior has an increasing probability of occurrence with increasing problem scale, and problems that converge rapidly at small scales may converge slowly or not at all when the scale is increased.

- Algorithm 2 describes a path tracing iteration to sample the photo-realistic values of image pixels in a Monte Carlo procedure. It is unconventional to compute these values using an iteration, and this computation is typically formulated as a tail recursion. The advantage of formulating the computation as an iteration rather than as a recursion is that it becomes easy to redistribute work among computers as a result of dynamic load balancing.

- Algorithm 3 shows one way to use algorithm 1 to provide dynamic load balancing for a Monte Carlo image synthesis application based on algorithm 2. The discussion in section 4.3 ("a concurrent implementation") brings to light practical issues of implementation that may be relevant to load balancing in support of other applications.

- The analysis of randomization strategies in section 4.4 presents a model to calculate the expected workload imbalance on any number of computers as a function of statistical properties of an image. This analysis shows that the difficulty of achieving a balanced workload increases with problem scale, and that tiling strategies commonly used for load balancing in image synthesis have a detrimental effect. It illustrates the difficulty of achieving a balanced workload by static methods and shows the necessity for dynamic load balancing in distributed image synthesis computations. This section presents predicted and simulated data for load imbalance of representative images on up to 1,024 computers. This data may be useful to future studies in scalable image synthesis.

- An empirical comparison of the performance of four different load balancing strategies on systems of up to 64 computers appears in table 4.2. These results reinforce the need for dynamic load balancing for problems in image synthesis. They show that algorithm 1 was consistently as good as or superior to one of the most popular load balancing algorithms for ray-tracing on parallel computers.

- Algorithm 4 offers a scalable solution to the dynamic mapping problem based on the computational kernel of equation (3.5). This is one of the first algorithms ever proposed for the dynamic mapping problem on general interconnection topologies. The local nature of the algorithm ensures that it will produce better solutions than random placement algorithms [9]. The similarity of algorithms 1 and 4 serves to both unify and highlight the differences between the problems of load balancing and mapping.

Chapter 2 Dynamic Load Balancing

The load balancing problem is intrinsic to any distributed computation. In order to achieve scalable speedup it is necessary to distribute the computational workload evenly so that no individual computer becomes a bottleneck for the computation.

This chapter describes two classes of applications that make use of parallel computer systems, and considers the nature of imbalances that occur in these applications. It defines the load balancing problem in terms of processes and computers, with an objective to minimize the time required to run the processes on the computers. It describes the principal approaches that have been taken toward the load balancing and mapping problems, and proposes a dynamic load balancing algorithm that is appropriate for these applications.

2.1 Field equations

A number of problems in mechanics and other fields can be addressed by solving computer models of systems of equations that represent discretized differential or integral equations. For example, Computational Fluid Dynamics (CFD) applications solve Navier-Stokes or Euler equations of fluid mechanics, discretized according to finite-difference or finite-element methods [15]. Solutions to these equations are most often described by fields of velocity vectors, pressure, and sometimes temperature, organized into grids of two or three spatial dimensions. Automobile designers use CFD to simulate the cabin acoustics and ventilation inside a passenger compartment under various operating conditions. Food industry manufacturers use CFD calculations to minimize the friction and heat dissipation of factory equipment. Aerospace designers use CFD to calculate lift and drag of airfoils and rockets under operating conditions that are unattainable in wind tunnels.

In a similar way, problems in structural mechanics can be solved by using finite element methods to discretize equations of elasticity and heat transfer, and a variety of other equations for problems in electromagnetic scattering, radiative transfer, and other subjects [32]. Most of these problems use spatial grids to represent the discretized equations, and typical solution algorithms iteratively refine values on the grid until the values are consistent with the governing equations.[1] In solving these types of discretized equations, which are sometimes known as *field equations*, it is usually necessary to tailor the structure of the spatial grid in response to properties of the solution. For example, a CFD calculation will usually need to refine a grid by adding points around a shock wave or discontinuity in order to fulfill the requirements for consistency between the solution and the governing equations. A calculation that models the propagation of a fracture in a solid material will refine a grid to capture the structure of the fracture.

In solving field equations on parallel computers the grid refinement operations give rise to load imbalance in the solution algorithms. Load imbalance arises because the solution algorithms typically perform the same amount of work for each point in the grid and the workload of each computer is proportional to the number of grid points managed by that computer. When a grid is refined in a local region, the number of grid points managed by a small set of computers is increased, and the workload for those computers is increased relative to the average workload. These refinements are usually performed locally in response to a local estimation of solution error. Larger refinements can occur as a succession of local refinements, but are not usually implemented in a single step. One reason for this is that solution algorithms can become unstable if the grid changes dramatically, and therefore it is important to refine a grid gradually.

2.2 Computer graphics

Computer graphics is a rapidly developing subject that is readily amenable to parallel implementation [14]. This thesis will be concerned with *image synthesis*, the problem of computing a two dimensional image from a three dimensional geometric model. Image synthesis is fundamentally a transport problem in which light is transported from emitters to reflectors and ultimately reflected into a virtual camera to create pixels on an image plane. The intensity of light is termed *radiance* and has the property that it is unaffected by distance as it travels along a straight line through space. Radiance is measured in units of W/m^2sr (watts per square-meter steradian). Steradian is a measure of solid angle, with a sphere subtending a total of 4π steradians. Radiance is affected by reflection and trans-

[1] This describes typical solution algorithms that are used on parallel computers. A variety of other algorithms that employ so-called direct solution methods are typically used on uniprocessors and shared memory multiprocessors.

mission, and in particular by the molecular properties and surface geometry of reflectors and transmitters. The radiance of light at any frequency is assumed to be independent of the radiance at other frequencies, and images are typically computed at several frequencies, usually the "red", "green" and "blue" channels of a computer monitor. The value of a picture element or *pixel* in an image is a set of radiance values at specific frequencies, for a given model viewed under given lighting conditions from a given viewing location. A closely related quantity is *irradiance* which is the density of power arriving at a particular point from all directions. Since this is a non-directional quantity the unit of measure of irradiance is simply W/m^2.

The term *photo-realism* is used to refer to images that are sufficiently realistic that they might be mistaken for photographs. This term has various definitions and in the ultimate case requires accounting for all physical phenomena of light emission, reflection, and transmission, including subsurface scattering, flourescence, polarization, and properties of lenses and films. This level of realism is not presently attainable, because of the computational cost and even a lack of practical algorithms for such tasks as computing realistic surface reflections. Every recognized algorithm for photo-realistic image synthesis computes solutions to the *global illumination* problem in which the radiance transported into a camera from a surface in the model depends on the radiance transported to that surface from all other surfaces.

Algorithms based on *local illumination* compute the transport of radiance based on properties of only a single surface, ignoring multiple reflections and transmission. The simplicity of these approaches makes it possible to implement them in VLSI. This is the basis of real-time implementations of the *graphics pipeline* that are becoming pervasive in personal computers and workstations [1]. This approach only allows light to reflect once between the emitter and the camera, and it ignores the fact that objects can cast shadows on each other. The images produced by this approach lack realism but they are fast. Current algorithms perform a simple geometric transformation and lighting calculation for each object vertex and then rasterize the objects into fragments. Some number of depth comparisons are performed for each object fragment followed by a simple shading and texturing calculation for object fragments that are visible to the image plane.

Global illumination approaches take into account the characteristics of light source emission, and surface reflection and transmission. These approaches accurately model the ability of objects to cast shadows on each other. Image synthesis algorithms for global illumination are usually implemented in software and do not achieve real-time performance; however they produce images that are dramatically more realistic than the images product by local illumination approaches.

Finite element and Monte Carlo methods, long staples of numerical analysis, have become powerful tools for solving global illumination [20, 24, 56]. Other image synthesis algorithms that have been developed specifically for global illumination include ray tracing, and radiosity methods. Ray tracing algorithms simulate individual photon paths through space and thereby compute reflections very accurately [63]. They are often used to render models with mirror-like surface properties in which realistic reflection is important. Radiosity algorithms ignore the effects of individual paths in favor of computing the total transfer of energy between objects with purely diffuse, omni-directional surface reflection [55]. Radiosity computations are equivalent to finite element discretizations of the integral equations of light transport [20]. These equations are usually known as the *rendering* or *radiance* equations [33] but they can be formulated in a number of ways including formulations in terms of radiance, spectral radiance and transport intensity. Images produced by radiosity algorithms have a natural quality that is appealing to the eye, but because these algorithms are based on pairwise interactions they scale as $\mathcal{O}\left(n^2\right)$ and are therefore expensive. Hierarchical radiosity methods have been developed to reduce this $\mathcal{O}\left(n^2\right)$ complexity to $\mathcal{O}\left(n\right)$ by representing the surface interactions at multiple levels of detail [19, 58]. These algorithms are effective in reducing the complexity of the computation but it is very difficult to assess the error that is introduced into the solution as a result of the geometric simplifications.

Monte Carlo methods are an alternative to finite elements for explicitly estimating solutions to the integral equations of light transport [33, 56, 61, 62]. Like ray tracing methods Monte Carlo methods calculate the contributions of light transported along individual paths through space, and they are usually (but not necessarily) implemented using ray tracing inner kernels. Like radiosity methods Monte Carlo methods can model diffuse surface reflection. They have a major advantage in modeling sophisticated surface reflection and light source properties. Unfortunately their computational cost can be very high, since Monte Carlo analysis has a quadratic cost: in order to reduce the error in a solution by a factor of n it is generally necessary to compute n^2 new samples.

Load balancing is generally harder for image synthesis algorithms than for field equations because the computational workload is fundamentally data dependent and therefore can only be evaluated by executing the algorithm. When a Monte Carlo method is implemented using a ray tracing kernel the cost of computing the visibility for each sample depends on the relative positions of all of the complete set of surfaces. The lengths and numbers of paths that must be followed depend on which surfaces are visible from a given viewpoint. These can change abruptly when the objects in the model are also in motion. In a sequence of images with a moving viewpoint the workload distribution can change substantially from one image to the next.

2.3 The load balancing problem

Consider a set of n computers running a larger set of concurrent processes between two barrier synchronization events. Call t_i the time required to run all of the processes that execute on computer i. Given a mapping function M that assigns processes to computers the time p_j required to run process j then

$$t_i = \sum_{j \in M^{-1}(i)} p_j. \tag{2.1}$$

Equation (2.1) says that the time required by computer i is the sum of the times required for all of the processes that are mapped onto that computer. When there is only one computer ($n = 1$) the mapping is trivial,

$$T_1 = \sum_j p_j. \tag{2.2}$$

We can measure the effective utilization of a parallel computer system by comparing the serial execution time T_1 to the time required when $n > 1$,

$$s = T_1 / \max_i t_i. \tag{2.3}$$

Under ideal circumstances the speedup of a set of concurrent processes on n computers is $\mathcal{O}(n)$, that is, $s = c_0 n$ for some constant c_0. Then

$$\max_i t_i = T_1 / c_0 n. \tag{2.4}$$

Maximal speedup occurs at minima of equation (2.4) where, for all i,

$$\begin{aligned} t_i &= T_1/c_0 n \\ &= \frac{1}{c_0 n}\sum_j p_j \\ &\equiv T_{min}. \end{aligned} \tag{2.5}$$

At this maximum $s = T_1/T_{min}$. The objective of the load balancing problem is therefore to minimize $\max_i t_i$ by finding a mapping M that allows $t_i = T_{min}$ for all computers i. In a balanced solution all computers will have the same workload T_{min} and in addition the total amount of work will not change as a result of the load balancing algorithm.

Problem 1 *(Dynamic Load Balancing)*
Given $\vec{t}$ indexed by i, a set of times p_j, a mapping function M, and the relation $t_i = \sum_{j\in M^{-1}(i)} p_j$. Find a new mapping function $\hat{M}$ to yield a new vector $\hat{\vec{t}}$ so that $\|\hat{\vec{t}}\|_1 = \|\vec{t}\|_1$ and $\|\hat{\vec{t}}\|_\infty$ is minimal.

Problem 1 describes the *dynamic load balancing* problem.[2] The condition on $\|\hat{\vec{t}}\|_\infty$ requires that every computer have the same workload T_{min}. The requirement that $\|\hat{\vec{t}}\|_1 = \|\vec{t}\|_1$ requires that the total amount of work is the same before and after the algorithm is run. The *static* version of this problem is to find a mapping function $\hat{M}$ at the start of a computation. The static problem is just a special case of the dynamic problem in which M is trivial. It may be beneficial to use different algorithms for the static and dynamic problems, since the initial placement is only computed once and can afford a more expensive algorithm than subsequent remappings can.

Problem 1 at first appears to be straighforward. Given a set of processes with times p_j, and a set of computers, find a way to map the processes onto computers so that the total time t_i for each computer i is equal. Despite this straightforwardness problem 1 is NP-complete as can be shown by transformation from the *partition* problem.[3]

It is often necessary to solve the dynamic load balancing problem in the presence of a constraint that limits the mobility of the items that are remapped. For example, in

[2] The usual vector norms are defined $\|\vec{u}\|_1 \equiv \sum_{i,j} |u_{i,j}|$ and $\|\vec{u}\|_\infty \equiv \max_{i,j} |u_{i,j}|$.

[3] In the *partition* problem a set of integers is to be divided into two subsets of equal sum. This problem is NP-complete [34].

solving field equations it is necessary to maintain spatial locality among grid points. A load balancing algorithm that remapped grid points to randomly chosen computers would cause the field equation solvers to slow down severely because of the resulting increase in communication, for example. The same considerations would apply to finite-element image synthesis algorithms, and any other image synthesis algorithms that use distributed data structures. Image synthesis algorithms such as ray-tracers that exploit scene coherence would benefit from maintaining spatial locality because this would lead to better cache reuse, and minimize the need for communication in computing intersection tests.

Problem 2 *(Locality-Constrained Dynamic Load Balancing)*
Define vectors $\vec{m}$ *and* $\hat{\vec{m}}$ *so that* $m_i = M(i)$ *and* $\hat{m}_i = \hat{M}(i)$. *Solve problem 1 with the constraint that* $\|\hat{\vec{m}} - \vec{m}\|_2$ *is minimal.*

Problem 2 adds a constraint that remapped processes should move as little as possible. In a field equation solver this constraint would imply that every grid point should remain as close as possible to the grid points on which it depends. In a distributed radiosity calculation this would imply that the radiosity mesh would be distributed among computers in a way that preserves the relative ordering of the mesh elements.

This seemingly simple constraint makes solution of the load balancing problem considerably more difficult. The next section reviews common approaches to constructing load balancing algorithms, some of which address this constraint and some of which do not. Chapter 5 will address this constraint explicitly when it considers the mapping problem.

2.4 Some approaches to load balancing

2.4.1 A naive approach

It might appear that problem 1 can be solved by very simple algorithms. For example, in a master-slave scenario a single master process could collect the workload status of every slave, compute the average workload, and broadcast this average to all slaves. The slaves would then negotiate among themselves until they all achieve the specified average workload, or come as close to this as is possible given the actual distribution of the p_j.

The master-slave algorithm contains an inherently serial component, because the master process must handle all of the messages from the slaves, and must compute this average workload. The time required to broadcast the result increases with the number of slaves, on any conceivable network technology. As a result this proposal is not scalable because the cost of these operations increases linearly with the number of slave computers. The linear scaling could be partially alleviated by distributing the computation in the form of a tree. The resulting distributed algorithm would scale with a logarithmic cost, rather than linearly.

The master-slave algorithm might be adequate for solving problem 1 when scaling is not a requirement. However it cannot address the locality constraint of problem 2. It is no longer a matter of deciding merely how many processes should be assigned to each computer, but in addition, where to place each process in order to minimize the cost of communication. The problem of placing the processes is combinatorial, and is an instance of the NP-complete *quadratic assignment* problem [9]. This is the essence of the mapping problem, which will be discussed in a later chapter of this thesis. Many existing proposals for solving the load balancing problem can be distinguished by how explicitly they treat this consideration.

The master-slave approach ignores the NP-completeness of problem 1 that results from its relationship to the partition problem. It is possible that there may be no way to partition a given set of processes into equally weighted subsets. Every practical approach to load balancing also ignores this issue, either by simplifying the problem so that processes have equal weight, or by accepting approximate solutions that are close to optimal.

2.4.2 Recursive bisection

One way to solve problem 2 is to recursively bisect graphs of processes and of computers. At each step the results of the bisections are combined, so that each sub-graph of processes is assigned to one of the two sub-graphs of computers. The resulting solution has a balanced workload since the process graph is bisected into equally weighted sub-graphs at each step. Recursive bisection is guaranteed to satisfy the locality constraint if it chooses a minimal bisector of the process graph at each step.[4] It has been used successfully in various applications and has become popular for solving field equations. In these problems the individual grid points are mapped onto the computers, and each grid point is assumed to have weight $p_j = 1$.

[4]A minimal bisector is one for which the sum of the cut edge weights is minimal.

The most efficient of these approaches is *recursive coordinate bisection* in which the spatial coordinates associated with grid points are used to find the bisector [64]. In this algorithm the bisector computation is inexpensive, but unfortunately the algorithm is not applicable to the general problem in which spatial coordinates are not present. The general problem can be treated by any number of graph partitioning algorithms. Advocates of heuristic partitioners claim that they produce good solutions at low cost [35, 36, 54]. A competing approach, *recursive spectral bisection,* uses spectral properties of the graph to compute minimal bisectors [5, 50, 57]. Having a minimal bisector at each step does not guarantee that the recursive process will yield a minimal partition of the graph, and it is a matter of debate whether spectral bisection yields better results than the heuristic approaches [35]. One thing that is not in dispute is that spectral bisection is more expensive than other methods [5, 35].

Spectral bisection finds a minimal bisector by computing the *Fiedler vector* of the graph, and then uses the elements of the Fiedler vector to assign graph vertices to one subgraph or the other. In order to find the Fiedler vector the Laplace equation $\nabla^2 u = 0$ is discretized on the graph using the method of finite differences, with one unknown u_i for each vertex of the graph. The matrix of coefficients that results from this discretization is called the *Laplacian matrix* of the graph. The Fiedler vector is the eigenvector of this Laplacian matrix corresponding to the smallest non-zero eigenvalue.

The role of the Fiedler vector in computing minimal graph bisectors is well established [18, 44]. Spectral bisection typically computes the Fielder vector by using Lanczos iteration, but any other algorithm for computing eigensystems of symmetric matrices could be used, such as a Householder-QR sequence [17]. Figure 2.1 illustrates the complete spectrum of eigenvalues of a 16×16 Laplacian matrix corresponding to regular 4×4 grid of processes, and two representative eigenvectors of that matrix, including the Fiedler vector.

2.4.3 Diffusion

Recursive bisection approaches all have characteristics that make them undesirable for dynamic load balancing. Recursive spectral bisection, in particular, is too expensive to use routinely to remap an application [5]. All of these methods might be described as “all or nothing”: they require that all of the computers participating in a computation stop and synchronously compute a new mapping. This requirement is clearly impractical for envi-

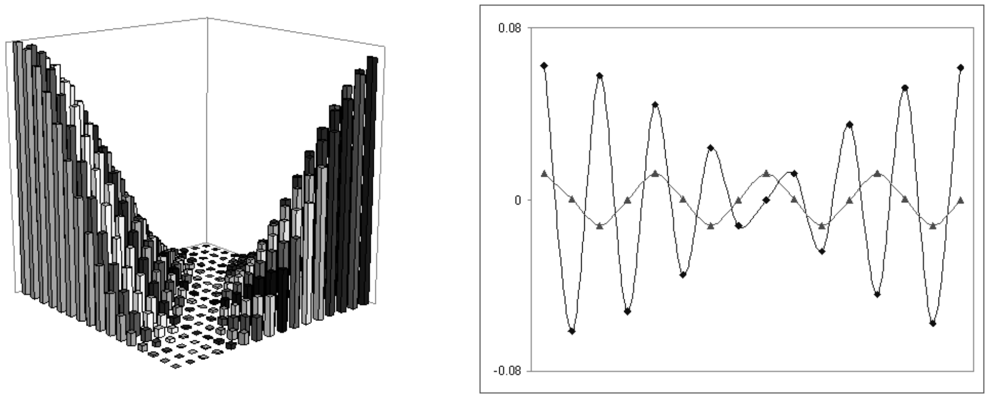

Figure 2.1: *Characteristics of the* 16×16 *Laplacian matrix of a regular mesh. Left, the complete eigenvalue spectrum. Right, two representative eigenvectors. The eigenvector with the larger variation is the Fiedler vector.*

ronments such as the Internet, and can be undesirable on other platforms. The requirement also violates the goal of scalability since the cost of these algorithms is in proportion to the number of computers.

Another class of algorithms can be loosely described as *diffusion* methods, and are based on local transfers of work among nearby computers. These algorithms don't have the disadvantages of recursive bisection methods. They are generally inexpensive and may be locally active without requiring the participation of all of the computers in a system. Unfortunately these methods have a weakness that recursive bisection methods don't have: because these are local methods, it is more difficult to address the mapping problem.

The first formal proposal in this class was due to Cybenko [13]. Cybenko's proposal considers the load balancing operation as a matrix-vector iteration in which computers are represented by a vector of workloads. Necessary conditions are formulated for convergence to equilibrium of the matrix-vector iteration. Unlike the algorithms presented in this thesis, these conditions don't assume nearest-neighbor communication among computers, and don't address the locality constraint, the mapping problem, or the issue of scalability.

Several other proposals were formulated within a short time of Cybenko's [7, 8, 10, 12, 23, 27, 30, 31, 41, 45, 47, 65, 66, 67]. Although these proposals were conceived independently the authors with few exceptions use the term "diffusion" in the descriptions of their algorithms. Few of these proposals are explicitly concerned with nearest neighbor communication [28]. A few of them rely on the theory of the iterative solution of linear systems of equations in order to prove convergence properties of their algorithms [65, 66, 67]. Still others derive their algorithms from models of differential equations [7, 12, 23, 27].

This author proposed a load balancing algorithm based on the heat equation $u_t = K\nabla^2 u$ [27]. The algorithm used an implicit numerical integration scheme to ensure numerical stability for very large time steps. Analysis of this algorithm showed that it required only a small fixed number of steps to accomplish a matrix inversion that guaranteed numerical stability. Analysis also showed that the rate of convergence to equilibrium for a simple model problem, based on a point disturbance, was independent of the size of the computer system, and therefore that the algorithm was scalable. Related algorithms were derived from the Laplace equation and applied to both the load balancing and mapping problems [23]. This is one of the first proposed solutions to the dynamic mapping problem. It is these algorithms that are presented in this thesis.

2.5 A load balancing algorithm

This section presents an algorithm to solve problems 1 and 2. For the sake of analysis the algorithm will be presented for a set of computers that communicate in a grid pattern. A similar result could be obtained for any interconnection topology. In order to model this grid $\vec{t}$ will be re-indexed by i, j. The following algorithm executes on every computer (i, j).

Algorithm 1 *(Dynamic Load Balancing)*

$u^{(0)}_{i,j} = t_{i,j}$; $\hat{M} = M$; $\epsilon = 0.1$

$\nu_{max} = PREDICT(\epsilon)$

for $\nu = 1$ *to* ν_{max} *begin*

  $du = 0$

  for $k = 1$ *to* m *begin*

    $(i', j') = NEIGHBOR(i, j, k)$

    $du_k = \frac{1}{m}\left(u^{(\nu)}_{i,j} - u^{(\nu)}_{i',j'}\right)$

    $du = du + du_k$

    while $(du_k > 0)$ *begin*

      choose l *so that* $p_l \leq du_k$ *(and* $\|\hat{\vec{m}} - \vec{m}\|_2$ *is minimal)*

      $\hat{M}(l) = (i', j')$

      $du_k = du_k - p_l$

    end

  end

end

$\hat{t}_{i,j} = u^{(\nu_{max})}_{i,j}$

In this algorithm the scalar quantity m is the number of immediate neighors of computer (i, j) and is 4 in the case of a regular grid in two dimensions. (This m should not be confused with the vectors $\vec{m}$ and $\hat{\vec{m}}$.) The value $\epsilon = 0.1$ was chosen arbitrarily, and could have been expressed as $\epsilon = h^{(\infty)}/h^{(0)}$ where $h^{(0)}$ is the height of an initial disturbance and $h^{(\infty)}$ is a desired height at termination. The function $PREDICT(\epsilon)$ returns a value for ν_{max} according to equation (3.12). $PREDICT$ could also return the value 1 to allow for an on-demand startup mechanism, which we will see demonstrated in chapter 4. The function $NEIGHBOR(i, j, k)$ just provides indices of neighboring points, which in this example simply differ by one in either i or j.

Chapter 3 Convergence and Scaling

This chapter discusses the derivation, correctness and scaling of algorithm 1. It illustrates the process by which the algorithm was derived as an iteration to solve the Laplace equation $\nabla^2 u = 0$ on the vertices of a graph. It shows that this derivation satisfies the requirements of problems 1 and 2. It shows a model for the worst case disturbance and the probability of occurrence of this worst case. It discusses the nature of the distribution of load imbalances that can occur in applications and presents a formal model of an individual disturbance. It presents a *scalability theorem* for individual disturbances which says that algorithm 1 reduces these disturbances at a rate that is independent of the size of the computer system. It argues that these properties also apply to multiple and random disturbances. Finally it shows simulations of the time-dependent evolution of single and multiple disturbances which are in agreement with the scalability theory.

3.1 Derivation of algorithm 1

The objectives of problem 1 are twofold: to compute the value of an equilibrium workload at every one of a set of computers, and to transfer processes among computers until the actual workloads equal this equilibrium. If the processes have unequal weights then it is usually necessary to accept approximate solutions to this second goal, since it may be impossible to achieve the exact equilibrium workload. Problem 2 adds a third objective in the form of the locality constraint, so that processes are transferred in a way that is consistent with the mapping problem.

The first objective can accomplished in a scalable way. Solutions of the Laplace equation $\nabla^2 u = 0$ are equilibria when boundary conditions are constant. The *standard iterations* (Jacobi, Gauss-Seidel, SOR) [17] are well-studied algorithms for solving certain classes of systems of linear equations. Linear systems of the form $A\vec{u} = \vec{b}$ can be solved by the standard iterations if the matrix A is *diagonally dominant*. A matrix A is diagonally dominant if in each i^{th} row the value of the diagonal element $A_{i,i}$ is greater than or equal to the sum of the other elements in that row. These conditions are sufficient to ensure that

A is positive definite, which is sufficient to ensure the success of the standard iterations. When the Laplace equation is discretized by the method of finite differences[1] the resulting linear system $Au = 0$ has a diagonally dominant matrix and can be solved by the standard iterations.[2]

The finite difference approximation to $\nabla^2 u = 0$ on a domain that is a two dimensional regular grid, containing n points indexed by row i and column j, is a set of equations

$$u_{i,j} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}). \tag{3.1}$$

Solutions of equation (3.1) are equilibria and are attracting fixed points of the iteration

$$u_{i,j}^{(\nu+1)} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1})^{(\nu)}. \tag{3.2}$$

Equation (3.2) is a Jacobi iteration for the Laplace equation with constant boundary conditions on a two-dimensional regular grid. The other standard iterations are procedurally similar. Gauss-Siedel iteration always uses the newest values of the right-hand side terms when they are available, and therefore converges faster than Jacobi. Successive over-relaxation (SOR) computes a weighted average of the right-hand side with the current value of the left-hand side. If the weighting is chosen correctly SOR converges faster than either Jacobi or Gauss-Seidel. The choice of an optimal SOR weighting for load balancing on various interconnection topologies is the subject of [65, 66, 67].

Equation (3.2) is one of the most studied algorithms in numerical analysis. When it is discretized in this way on a regular grid in two dimensions, the iteration has a well-known basis with eigenvalues $\lambda_{i,j}$ and eigenvectors $\vec{X}_{i,j}$ [69]. For example, in the Jacobi iteration these take the form

$$\begin{aligned} \lambda_{i,j} &= \frac{1}{2}\left(\cos \pi \frac{i}{\sqrt{n}} + \cos \pi \frac{j}{\sqrt{n}}\right) && (3.3) \\ (X_{i,j})_{x,y} &= k_{i,j} \cos \pi \frac{ix}{\sqrt{n}} \cos \pi \frac{jy}{\sqrt{n}}. && (3.4) \end{aligned}$$

In the case of Gauss-Seidel the eigenvalues are $\lambda_{i,j} = \frac{1}{4}\left(\cos \pi \frac{i}{\sqrt{n}} + \cos \pi \frac{j}{\sqrt{n}}\right)^2$ [69]. The

[1] A finite difference approximation to $\partial^2 u / \partial x^2$ on a domain of equally spaced points $\ldots, x_{i-1}, x_i, x_{i+1}, \ldots$ is defined as $2u_i - u_{i-1} - u_{i+1}$.

[2] These conditions are sufficient but not necessary. For a detailed discussion of these issues see [49].

Gauss-Seidel iteration converges faster than Jacobi, but at a cost: the unknowns must be updated in a fixed order. This order requirement isn't practical for algorithm 1 since it executes independently on each computer. With independent execution the updates will occur in random order, and the convergence rate will lie between the rates of the Jacobi and Gauss-Seidel iterations.

From equation (3.3) it is apparent that the iteration can have negative eigenvalues. Negative eigenvalues can produce oscillatory convergence to a solution which may have undesirable consequences (we will see an example of why this is undesirable in chapter 5). This oscillation can be eliminated by iterating equation (3.2) twice,

$$\begin{aligned} u_{i,j}^{(\nu+1/2)} &= \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1})^{(\nu)} \\ u_{i,j}^{(\nu+1)} &= \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1})^{(\nu+1/2)}. \end{aligned} \tag{3.5}$$

The resulting compound iteration has the same eigenvalues as Gauss-Seidel and converges monotonically.

The eigenvalues of equation (3.3) are for a Jacobi iteration derived from the Laplacian matrix of a two-dimensional regular grid. When a graph is not a two-dimensional regular grid the corresponding eigenvalues are still similar to equation (3.3) in that they cover the same interval, but with different spacing. This result follows from theorem 3.2 of Mohar [44]:

> Call $\vec{\lambda}$ the eigenvalues of the Laplacian matrix $Q(G)$ of a graph G. Construct a new graph G' by adding an edge between any two vertices of G. Then if $\vec{\lambda'}$ are the eigenvalues of $Q(G')$,
>
> $$0 = \lambda_0 = \lambda'_0 \leq \lambda_1 \leq \lambda'_1 \leq \cdots \leq \lambda_{max} \leq \lambda'_{max}$$

Since all of the eigenvalues occur in the interval from -1 to 1 this implies that the convergence properties for a regular grid are similar to the properties for any graph.

3.2 Correctness of algorithm 1

Algorithm 1 provides a scalable correct solution to problems 1 and 2. A perfectly scalable algorithm is one that does not increase in cost as the number of computers on which it executes increases. Algorithm 1 executes concurrently and independently on all computers, so there is no serialization due to dependencies among computers. Individual instances of the algorithm contain no components that would increase in cost on larger computer systems. The inner loop depends on m, the number of immediately connected neighbors, and not on n, the number of computers. Communication is with immediate neighbors and has cost that is also dependent on m rather than n. The outer loop depends on ν_{max} which is a computed quantity. The relationship between ν_{max} and n is complex and will be the subject of most of the remainder of this chapter. This discussion will demonstrate that ν_{max} is independent of n. As a result algorithm 1 is scalable in that the cost does not increase with n.

The correctness of algorithm 1 follows from the method by which it was derived. In the case $m = 4$ the result of the inner loop is to compute equation (3.2). In the case of general m the result is to compute the corresponding Jacobi iteration for the Laplace equation with m neighbors. Solutions of the Laplace equation are equilibria which satisfy problem 1.

Algorithm 1 stipulates that for each transferred process l it is necessary that $p_l \leq du_k$. This stipulation ensures that the correct amount of work is transferred among computers. Algorithm 1 also stipulates the locality constraint, that l should be chosen in order to minimize $\|\hat{m} - m\|_2$. A greedy strategy that minimizes this expression on each individual transfer will minimize it over a set of transfers.

3.3 Worst case analysis

Any workload imbalance $\vec{v}$ is a composition of eigenvectors

$$\vec{v} = \sum_{i,j} b_{i,j} \vec{X}_{i,j}. \tag{3.6}$$

Since the $\vec{X}_{i,j}$ are orthonormal it follows that $\vec{X}_{i,j} \cdot \vec{X}_{k,l} = \delta_{i,k}\delta_{j,l}$. Then

$$\vec{v} \cdot \vec{X}_{i,j} = b_{i,j}. \tag{3.7}$$

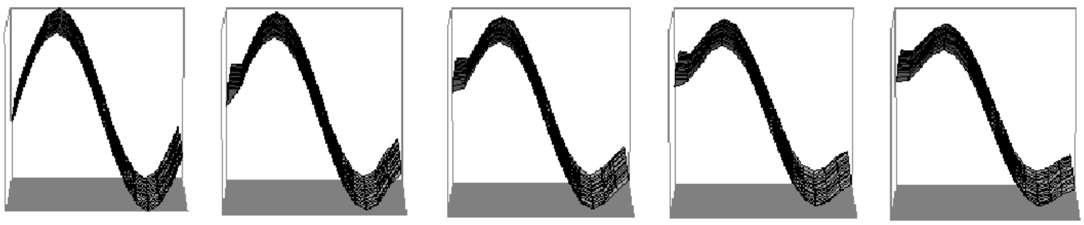

Figure 3.1: *The "Achilles heel" of equations 3.2 and 3.5. This sinusoidal disturbance is the eigenvector $\vec{X}_{i,j}$ that corresponds to the most slowly converging eigenvalue $\lambda_{i,j}$ of equation (3.2). The standard iterations cannot converge this disturbance effectively.*

An imbalance that consists of $\vec{X}_{i,j}$ is diminished by a factor $\lambda_{i,j}$ after each iterate of equation (3.2). An arbitrary imbalance, described by equation (3.6), has the value

$$\sum_{i,j} \lambda_{i,j} b_{i,j} \vec{X}_{i,j} \tag{3.8}$$

after one iterate of equation (3.2). If $\vec{u}^{(0)}$ represents an initial $\vec{v}$ and $\vec{u}^{(\nu)}$ the value of $\vec{u}^{(0)}$ after ν iterates of equation (3.2) then

$$\vec{u}^{(\nu)} = \sum_{i,j} (\lambda_{i,j})^{\nu} b_{i,j} \vec{X}_{i,j}. \tag{3.9}$$

The worst case disturbance is the $\vec{X}_{i,j}$ for which $\lambda_{i,j}$ is largest, which are the two cases $i + j = 1$.

Figure 3.1 illustrates an imbalance composed of the eigenvector that corresponds to the worst-case eigenvalue. Disturbances of this sort are very challenging to any algorithm derived from the standard iterations. These disturbances have been discussed extensively in the literature on numerical analysis, particularly with respect to solving discretized partial differential equations, where they have given rise to *multigrid methods* [43]. Multigrid methods address the slow convergence of these components by computing approximate solutions at several levels of resolution and then combining these approximate solutions to obtain an accurate solution. These methods can be used successfully but they require grids with regular structure, something that is not always available. Regular structure cannot be assumed in problems 1 and 2. In addition multigrid methods are not perfectly *scalable* according to the definitions introduced above. The cost of each iterate increases logarithmically with problem size, and the algorithm requires the synchronous participation of all computers, rather than the independent execution required by algorithm 1.

When the standard iterations are used to solve linear systems of equations the probability of occurrence of these slowly converging components increases as the problem scale increases. For any fixed λ between 0 and 1 the number of eigenvalues $\lambda_{i,j}$ for which $|\lambda_{i,j}| \geq \lambda$ increases as n increases. The cost of solving these problems is determined by the amount of error that is present in an initial solution, and primarily by the error in these slowly converging components. The expected amount of initial error is the same in every component. Since the number of slowly converging components increases with n the expected cost of the standard iterations also increases with n.

This phenomenon of cost that increases with problem scale is commonly seen in applications that solve discretized partial differential equations by iterative algorithms. A problem may converge at an acceptable rate on a grid of a certain resolution. When the grid resolution is increased the convergence may diminish until it is unacceptable. This is because the number of slowly converging components has increased along with the number of grid points, and so has the total error in these components. In addition, the worst-case eigenvalues also change slowly as n increases, approaching the value 1.0 asymptotically. This change reduces the decay rate of the slowly decaying components even further.

Algorithm 1 does not have the problem described above. The expected initial error is not the same in every component, in contrast to the general case of solving linear systems of equations. The initial error is the workload imbalance that is present in the problem. The expected workload imbalance is not the same in every component of an initial condition of problems 1 and 2. If a given problem develops a particular pattern of workload imbalance on a small number of computers, it will develop a related and similar pattern of imbalance on a larger of computers. Therefore the probability of occurrence of a slowly converging component does not increase with n. If this problem of slow convergence is not present when algorithm 1 executes on a small number of computers, then it will not be present on a large number of computers.

3.4 A scalability theorem

When workload imbalances are localized the scaling of algorithm 1 is excellent. The following theorem shows that in the best case, where the imbalance is confined to a single computer, the convergence rate is independent of n. Define this best case by

$$\vec{u}^{(0)} = \sum_{i,j} a_{i,j} \vec{X}_{i,j} \tag{3.10}$$

where all elements of $\vec{u}^{(0)}$ are 0 except for $u^{(0)}_{0,0}$ which is equal to 1.

Lemma 1 $(X_{i,j})_{0,0} = a_{i,j}$ *for all* i, j.

Proof: the definition of $\vec{u}^{(0)}$ *implies that* $\vec{u}^{(0)} \cdot \vec{X}_{i,j} = (X_{i,j})_{0,0}$. *Equation (3.7) implies that* $\vec{u}^{(0)} \cdot \vec{X}_{i,j} = a_{i,j}$. *The conclusion follows immediately.*

Lemma 2 $(X_{i,j})_{0,0} = k_{i,j}$ *for all* i, j.

Proof: follows immediately from equation (3.4) by subsituting $x = y = 0$ *into equation (3.4).*

Lemma 3 $a_{i,j} = k_{i,j}$.

Proof: from the conjunction of lemmas 1 and 2.

Lemma 4 $a_{i,j} = \frac{1}{n^2}$ *for all* i, j.

Proof:

$$\begin{aligned} 1 &= \vec{X}_{i,j} \cdot \vec{X}_{i,j} \\ &= \sum_{x,y} \left(k_{i,j} \cos \pi \frac{ix}{n} \cos \pi \frac{jy}{n} \right)^2 \\ &= k_{i,j}^2 n^4. \end{aligned} \tag{3.11}$$

From equation (3.11) it follows that $k_{i,j} = \frac{1}{n^2}$. *Therefore* $a_{i,j} = \frac{1}{n^2}$ *by lemma 3 .*

Theorem 1 *(Scalability Theorem)*

For any fixed $\nu > 0$ the value of $u_{0,0}^{(\nu)}$ converges to $u_{0,0}^{(\infty)} = \hat{\vec{t}}$ as n increases.

Proof: equation (3.9) implies that the height of the point disturbance after ν iterates is

$$u_{0,0}^{(\nu)} = \sum_{i,j} \frac{1}{n^2} \left(\lambda_{i,j}\right)^{\nu} . \tag{3.12}$$

The term $\frac{1}{n^2}$ dominates equation (3.12). This factor converges to zero as n increases.

In informal terms, the scalability theorem says that as the problem scale increases, the number of steps ν required to reduce the initial disturbance by an arbitrary factor ϵ converges to a constant value. This value is the smallest ν such $\sum_{i,j} \frac{1}{n^2} \left(\lambda_{i,j}\right)^{\nu} \vec{X}_{i,j} \leq \epsilon$.

3.4.1 Multiple and random disturbances

The scalability theorem describes an idealized problem in which workload imbalance is confined to a single computer. A purely local disturbance may be a realistic model for some applications, but in general disturbances may arise at several computers simultaneously. A result analogous to the scalability theorem could be proven for any specific spatial pattern of multiple disturbances. The rate of convergence of algorithm 1 on this set of disturbances does not increase significantly as the number of computers increases, again as a result of equation (3.9). A stronger statement can be made, that for any fixed number of disturbances, independent of spatial pattern, the probability of occurrence of a worst-case imbalance vanishes as n increases. This statement follows from the observation that the number of points required to create a worst-case imbalance increases as n increases. If this number of points is fixed while n increases then the magnitude of the slowly converging components must steadily decrease.

Similar arguments can be made for disturbances with any number of points, if these disturbances occur at random times and locations and with random intensity. When these disturbances occur in sufficiently large numbers the expected imbalance at every computer is the same, as implied by the central limit theorem [42]. In this case there is no imbalance since all computers have equal workloads.

A more interesting case occurs when disturbances are random but not in sufficient number to have equal expected imbalance. A simulation was performed to explore this case in the context of the heat equation algorithm [27]. The simulation was of system of 1,000,000 computers that was initially balanced. The simulation was run for 1,000 steps, and a random disturbance was created on each of the first 700 steps. The position and magnitude of each disturbance was chosen randomly, but the magnitude was very large, on average 30,000 times the average workload. After 700 steps the worst case of imbalance at any computer was only 15,737 times the average workload, less than the magnitude of the average disturbance. After 800 steps the worst case was only 50 times the average. This demonstrated that the algorithm was removing imbalance faster than it was occurring. This demonstration took place on a very large computer system with very large average imbalances. This result would become even better if the number of computers were to to increase further, or if the magnitudes were to decrease.

3.5 Time dependent behavior

Figure 3.2 shows the first few iterates of equations 3.2 on the point disturbance. Although the convergence is rapid, the height of the initial point disturbance oscillates between zero and a diminishing series of positive values as shown in figure 3.4.

Iteration of equation 3.2 creates oscillation in Jacobi form because the spectrum of the Laplacian matrix contains negative eigenvalues. Equation 3.5 achieves monotone convergence by performing two steps of equation 3.2. If A is the iteration matrix for equation 3.2 with eigenvalues $\lambda_{i,j}$ then AA is the corresponding matrix for equation 3.5 with corresponding eigenvalues $\lambda_{i,j}^2$. These eigenvalues are always positive, and so convergence is monotonic. This is illustrated in figures 3.4 and 3.3.

The scalability theorem says that the rate of convergence for a point disturbance is constant at large scales. Figure 3.5 shows that the same convergence occurs in a problem that is sixteen times bigger than in figure 3.3.

Figure 3.6 and 3.7 shows more general disturbances characterized by Gaussian distributions. The Gaussian disturbances appear to take slightly longer to disperse than the point disturbances, but this difference is very small. A pair of Gaussian disturbances spaced closely together disperse in roughly the same amount of time as a single disturbance does, showing that the interaction between the two is small. The taller Gaussian takes longer to disperse, as would be expected from the preceding discussion.

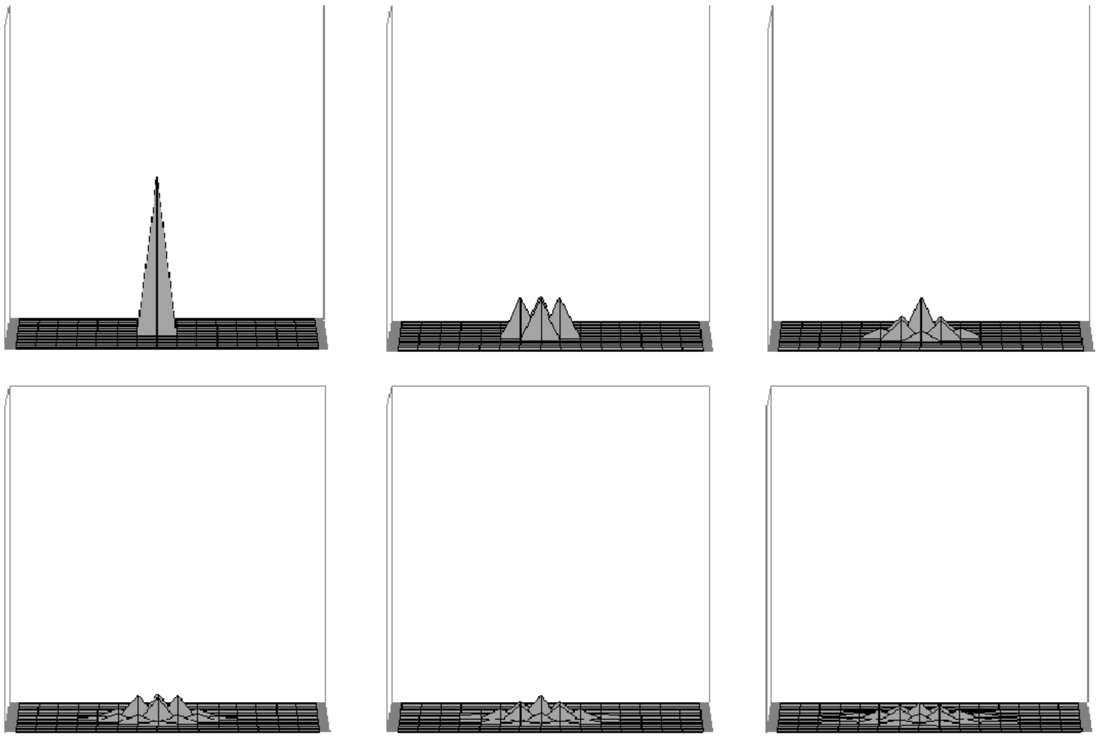

Figure 3.2: *The initial conditions and first five iterates of a point disturbance under equation 3.2. The center point shows oscillatory convergence.*

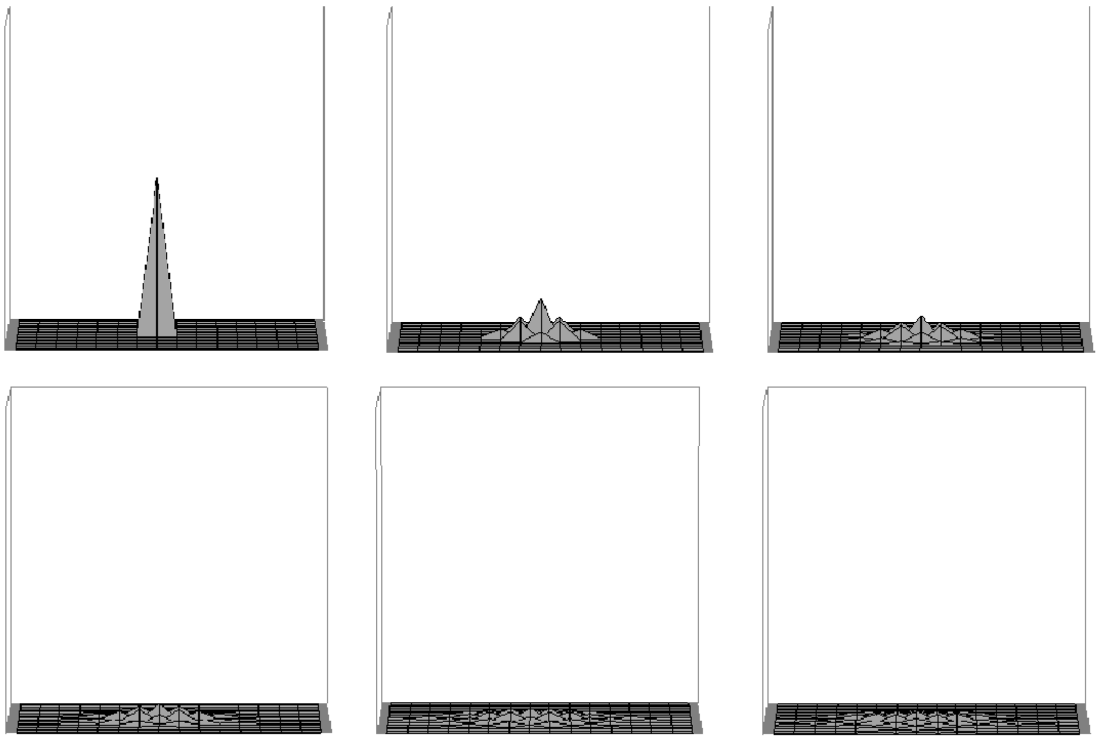

Figure 3.3: *Point disturbance under equation 3.5 shows monotone convergence.*

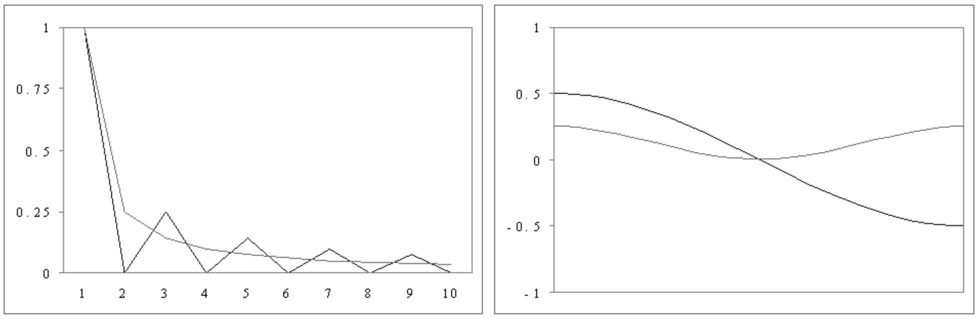

Figure 3.4: *Left, height of the point disturbance over 10 iterates of equations 3.2 and 3.5. Equation 3.2 shows oscillation while algorithm 3.5 converges monotonically. Right, eigenvalue* $\lambda_{i,8}$ *for* $n = 16$ *of iteration matrices* A *and* AA *corresponding to the two equations.* A *has mixed eigenvalues while* AA *is positive definite.*

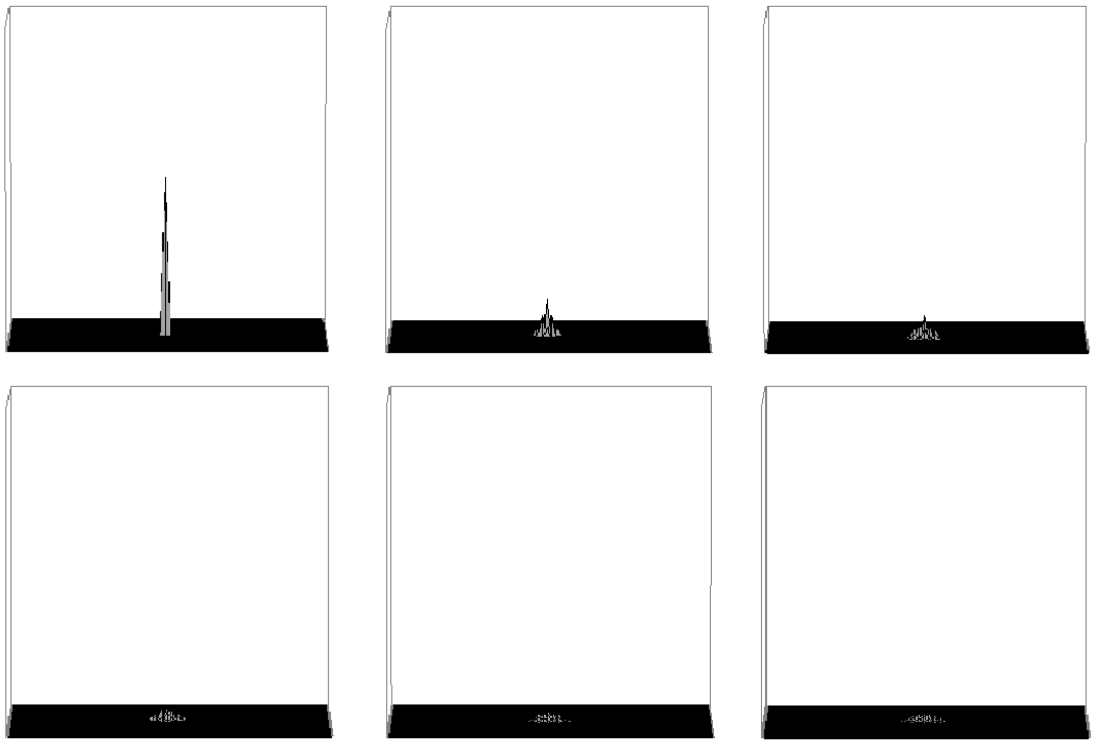

Figure 3.5: *Convergence of a problem on a* 64×64 *grid of* 4096 *unknowns. The height of the initial disturbance is the same in this series as in figure 3.2 affirming the conclusion of the Scalability Theorem.*

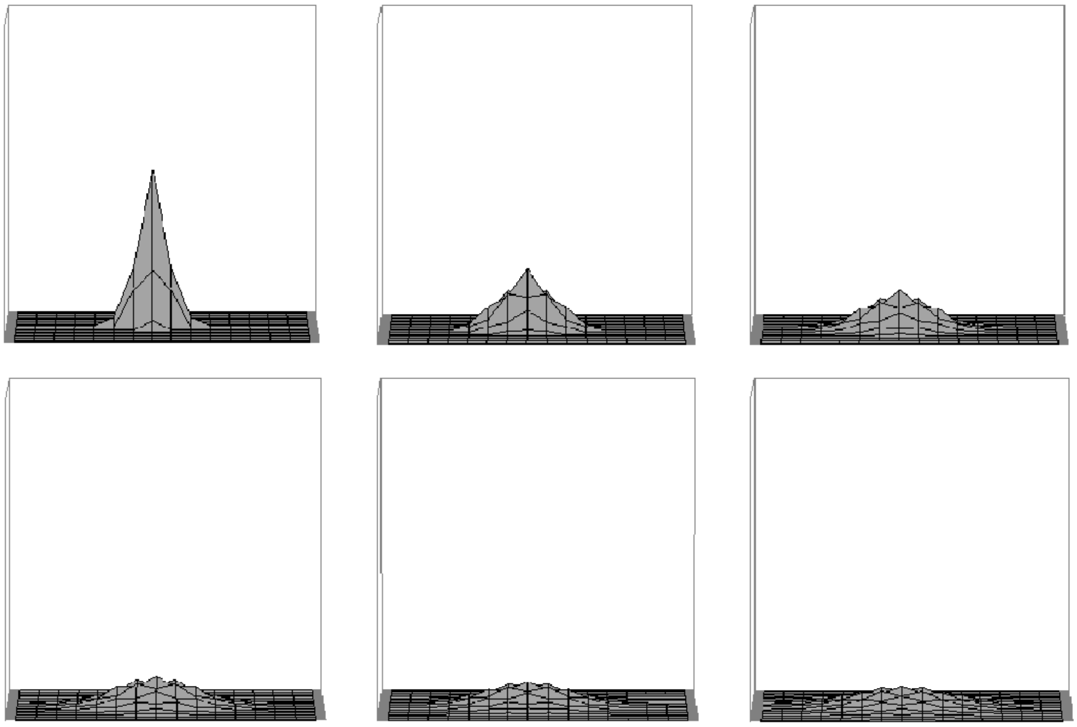

Figure 3.6: *Convergence of a Gaussian disturbance under equation 3.5.*

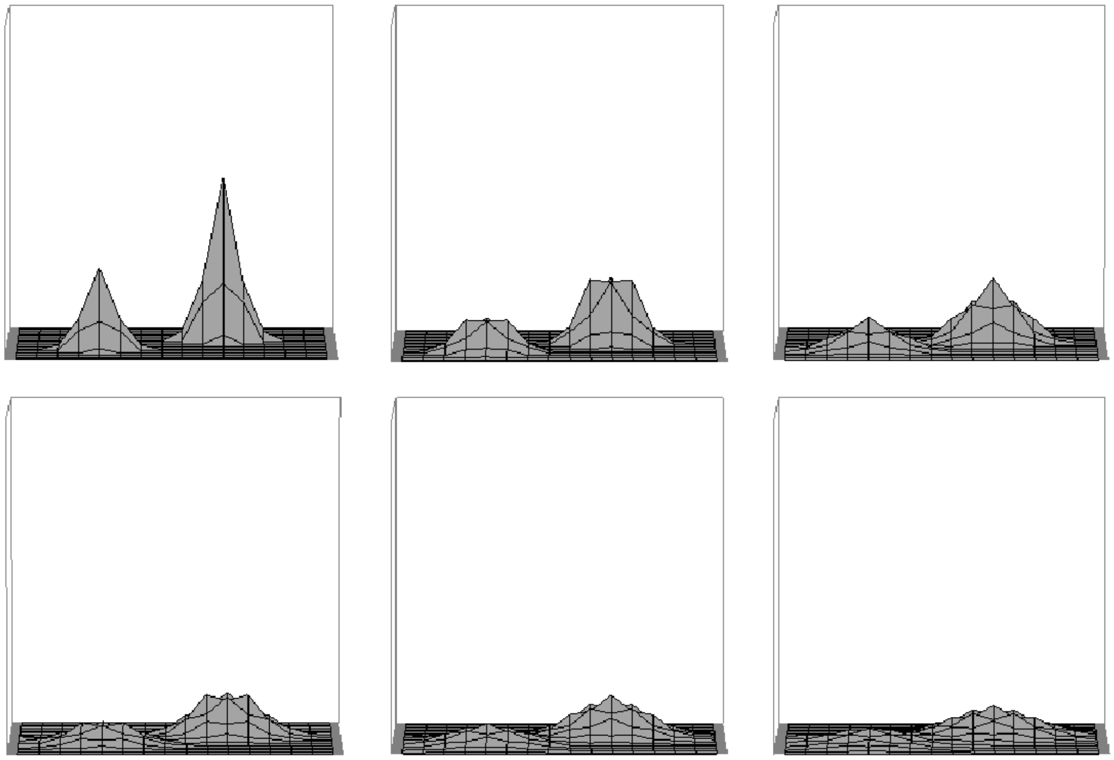

Figure 3.7: *Convergence of a pair of Gaussian disturbances under equation 3.5.*

Chapter 4 An Experimental Validation

The value of algorithm 1 depends on the validity of two assumptions. The first assumption is that a discrete workload imbalance, represented by a point disturbance, is erased quickly by the algorithm. The second assumption is that a point disturbance is a valid model for real phenomena. The first assumption can be shown mathematically but the second assumption must be tested empirically. We can construct an experiment to test the hypothesis of the second assumption. The experiment will use algorithm 1 in a scalable application. Since the application is assumed to be scalable, then if the combined program fails to demonstrate scalability we will attribute this failure to algorithm 1.

Although both field equations and computer graphics offer scalable applications it is image synthesis that offers the greatest challenges for load balancing. Monte Carlo rendering methods offer an ideal combination of features for this experiment. They are "embarrassingly" parallel and scale easily to large numbers of computers. It is difficult or impossible to predict load imbalance in advance. Furthermore balanced solutions are very unstable, so that small changes in the data set (such as moving viewpoints or geometry) can and routinely do lead to large and abrupt changes in workload distribution.

4.1 A pinhole camera model

Figure 4.1 shows a model of a pinhole camera taking a photograph. The pinhole camera has a sheet of film at the back called the *image plane.* A pixel is a rectangular region of the image plane. The value of a pixel is a distribution of radiance values at a set of frequencies. We will compute the radiance at three frequencies that correspond to the r, g, b channels of a computer monitor. We will sample these radiance values at a set of discrete points within a pixel. We will average these samples at each frequency to represent the radiance for the pixel at that frequency.

Figure 4.1 also shows one path of light transport in the pinhole camera model. This path shows that light can reflect multiple times on the way to a destination, for example reflecting once at x' in direction ω', then again at x in direction ω. At each surface point

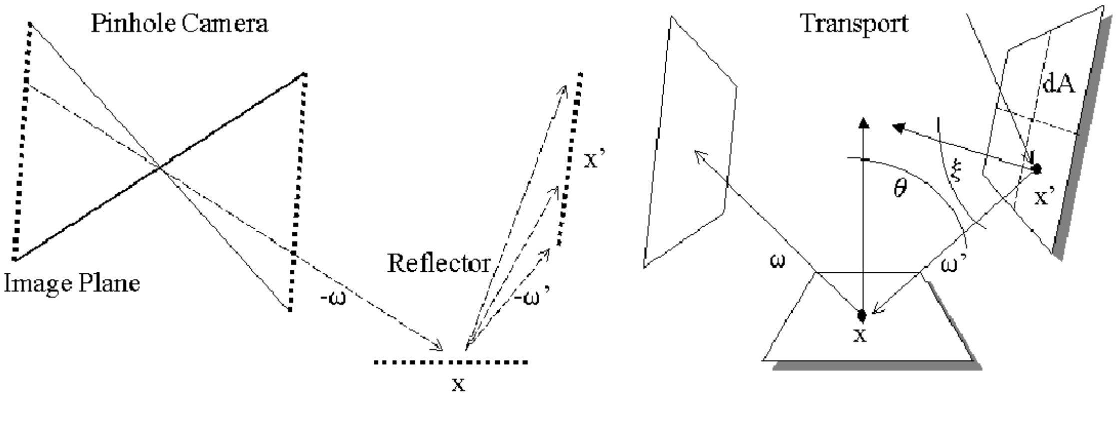

Figure 4.1: *Left: computing Monte Carlo samples of direct illumination. The image plane corresponds to the rear wall of a pinhole camera. A path extends from the image plane out through the camera aperture to a visible surface at x. The light intensity reflected into the camera along this path is equal to the sum of the light intensities at x contributed from all visible light sources, multiplied by a reflection coefficient. Right: the reflectance phenomena described by equation (4.1). One out of the many paths by which light is transported to contribute to the value of a pixel. Light from surface point x' is transported to point x in direction ω'. Point x' may be a light emitter or may simply reflect light from other points, or both. A fraction of the total light reaching x at any instant is reflected in the direction ω which in this figure leads into the camera aperture. In the general case ω could lead to another reflector.*

there is a coefficient of reflection that determines how much of the light is reflected in the new direction. This coefficient is sensitive to direction in the most general case and can be modeled $k(x,\omega,\omega')\cos\theta$ as shown in the *reflectance equation* (4.1).

$$\mathcal{L}(x,\omega) = \int_{\Omega} \mathcal{L}(x',\omega')\, k(x,\omega,\omega') \cos\theta d\omega'. \tag{4.1}$$

The quantity $\mathcal{L}(x,\omega)$ represents the radiance leaving x in the direction ω. The radiance after multiple surface reflections, as illustrated in figure 4.1, is described by expanding equation (4.1) recursively as shown in equation (4.2). At the base of the recursion is a light source with an emissive radiance e. We will simulate the pinhole camera by sampling the value of equation (4.2) for a single frequency at a single point within a pixel. We will compute the value of a pixel at that frequency by averaging many such samples from different points within the area of the pixel.

$$\mathcal{L}(x,\omega) = k(x,\omega,\omega') \cos\theta \left(\int_{\Omega} k(x',\omega',\omega'') \cos\theta' (\cdots e \cdots)\, d\omega' \right). \tag{4.2}$$

4.2 A computational procedure

Equation (4.2) can be computed by a branching procedure that follows a tree of individual paths from the image plane to a multitude of surface points $x, x', x'', \ldots$ and accumulates their total contributions to $\mathcal{L}(x,\omega)$. At each surface point x the procedure generates a new set of outgoing paths to explore. Each path is defined by a direction $-\omega'$ originating from x. An efficient Monte Carlo sampling strategy will concentrate these new directions in areas where the reflectance function $k(x,\omega,\omega')$ is large. The branching procedure follows each of these new paths independently, until a path either encounters a light source, in which case the light source makes a contribution to the sample $\mathcal{L}(x,\omega)$, or until the product of reflection coefficients along the path becomes so small that the further contributions would be unnoticeable.

This branching procedure can be implemented iteratively if the reflection coefficient $k(x,\omega,\omega')\cos\theta$ can be computed at each step. From the pinhole camera of figure 4.1 it is apparent that at each reflection point x the values of x and ω are known, but ω' (and therefore θ) remains to be determined.

We want to be able to move a computation from one computer to another as a result of dynamic load balancing. This is difficult to do in a recursive implementation, because the original invocation context is no longer present upon return from a recursive function. For this reason we would like to rearrange terms of equation (4.2) so that the computation can be performed by a forward sweep along a set of paths of points $(x, x', x'', \ldots)$. As a result the computation can be organized as an iteration rather than a recursion. An iteration never needs to return, and therefore there is no need to recreate an invocation context.

4.2.1 A path tracing iteration

We assume that path descriptors (x, ω', I, x_0) are processed out of a queue of pending work. The queue initially contains a descriptor $(x_0, \omega, 1, x_0)$ for each desired sample where x_0 is a point on the image plane, and the direction $-\omega$ extends through the camera pinhole and out the aperture as shown in figure 4.1. The initial value $I = 1.0$ will be multiplied by each reflection coefficient along a path in order to determine the amount of light reflected along the path and into the camera.

The algorithm will find the next visible point x' in direction $-\omega'$. If this point is on the surface of a light source then the light emission is added to the sample after weighting by I. The sample is accumulated in $p(x_0)$ which initially has the value 0. If x' is a reflector then the algorithm will generate a set of new entries (x', ω'', I', x_0) in the queue. This process continues for some number of steps or until the queue is empty.

The algorithm computes independent trees of paths for specular, diffuse, and glossy surface properties, and combines the results of these independent computations. These surface properties could be extended with additional properties, or could be combined into general models of reflectance functions [40].

Algorithm 2 *(Path Tracing Iteration)*

for (some number of steps) do begin

choose a descriptor (x, ω', I, x_0) *from the queue*

find point x' *visible from* x *in direction* $-\omega'$

if (x' *is on a light source with radiance emission* e*) then*

$p(x_0) = p(x_0) + Ie/\pi$

else if (I *is not too small) then begin*

for (each separate surface property) do begin

generate a new set of v *direction unit vectors*

for each new direction unit vector ω'' *do begin*

compute r *according to the surface property*

enqueue descriptor $(x', \omega'', rI/v, x_0)$

end

end

end

end

The value of the reflection coefficient r is different for different surface properties. This is explained in the next section.

4.2.2 Sampling strategies

We can improve the efficiency of the rendering algorithm by using informed sampling strategies known as variance reduction techniques. For the purposes of photo-realism it is important to ensure that these strategies do not introduce bias into the result, as is the case with many common methods of antialiasing and path termination strategies [3, 37]. One reliable strategy is to explicitly sample the surfaces of the light sources, and we have done so in this implementation. Another reliable strategy is to model different surface properties independently, in ways that are unbiased, and then to combine samples from the different properties [61]. In this implementation we have separate models of the purely specular and purely diffuse surface properties, and of a "glossy" property that is similar to a directional Phong shading model. Each of these separate properties leads to a separate calculation for the reflection coefficient r.

Sampling the light sources is accomplished by an explicit calculation of the radiance integrated over the solid angle subtended by the light source. If A is the area of a light source i this calculation is

$$\Phi_i(x) = \int_A \mathcal{L}(x', \omega') \frac{\cos\theta \cos\xi}{\|x' - x\|_2^2} dA. \tag{4.3}$$

In equation (4.3) the quantity $\Phi_i(x)$ represents the irradiance (density of incident power) arriving at surface point x from light source i. The angle between the surface normal of the light source and the angle of emission corresponding to ω' is denoted by ξ. The quantity dA is the differential surface area of the light source. To reduce variance we stratify the light surface into rectangles of equal area dA and sample equation (4.3) over the set of rectangles.[1] The distance term $\|x' - x\|_2^2$ reflects the reduction in solid angle that occurs with increasing distance. This computation is performed for each light source i and the resulting $\Phi_i(x)$ are added into the calculation of the irradiance $\Phi(x)$.

The purely specular property is modeled by a scalar coefficient of reflection. For this model we sample a single path in the direction of mirror reflection. In this case k is equal to the scalar coefficient.

The glossy property is modeled by a distribution of rays clustered in a lobe around the mirror reflection direction. Samples are generated on the surface of this lobe, and the value of k is taken from another scalar coefficient.

The diffuse property is modeled by yet another scalar coefficient. Surfaces that are purely diffuse reflect equal radiance in all directions. The value of $\mathcal{L}(x, \omega)$ in each direction is $1/\pi$ times the value of the irradiance $\Phi(x)$. Since we sample the diffuse model over the entire incoming hemisphere there is an opportunity to use an efficient sampling strategy to concentrate samples in the areas where they will provide the most information. We will use a cosine distribution which concentrates samples around the surface normal. To be precise, we will estimate the irradiance

[1] The total number of samples per light source in our implementation is the product of two parameters, the number of samples per pixel (PIXEL_SAMPLES) and the number of samples per light source per pixel sample (DIRECT_SAMPLES). Realistic soft shadows have been obtained when this product is 16 or higher.

$$\Phi(x) = \int_{\cap} \mathcal{L}\left(x', \omega'\right) k\left(x, \omega, \omega'\right) \cos\theta d\omega' \tag{4.4}$$

by a set of samples of the integrand over the unit hemisphere. To generate samples with the correct distribution, we compute samples on the surface of a unit disk and project them upwards to the hemisphere. This method of generating samples clusters the samples around the unit normal vector and multiplies each incoming radiance by a factor of $1/\cos\theta$. This removes the factor of $\cos\theta$ from equation (4.4) and so r is just the scalar parameter associated with the surface.

These properties can be seen in the images in figure 4.2. In the office image the interior is reflected in the specular window surface, and the other surfaces contain both diffuse and specular components. The stained glass image shows a reflection of the butterfly on a glossy surface.

4.3 A concurrent implementation

Algorithm 2 is scalable, in that it can be implemented concurrently on a set of computers and achieve, at least in theory, perfect speedup. This speedup occurs when individual iterates have equal access to the model data so that they can be computed concurrently. Since most of the geometric models used in the experiments require about a megabyte of storage it is reasonable to reproduce them on every computer. One model with over 300,000 polygons was rendered in order to test the implementation. Even this model fit easily in the memory available on each computer.

In order to test the hypothesis that algorithm 1 is scalable it will be necessary to use it in a distributed implementation of algorithm 2. If both algorithms are scalable then the combined program should also be scalable. Algorithm 2 is scalable, since it executes concurrently as a set of independent processes, and contains no procedural dependencies on the number of computers on which it runs. If the combined program does not scale then we can infer that algorithm 1, or some artifact of the implementation, is not scalable.

A program that combines algorithms 1 and 2 was implemented using a prioritized message-driven scheme [25]. The program consisted of a set of logical processes connected through a set of communication channels. A large number of rendering processes were connected in a mesh arrangement that matched the grid of equation 3.2. A user interface

Figure 4.2: *Clockwise from upper left: glass, conference, office, soda, and bath images. These were computed at NTSC resolution* (640 × 480) *and in full color.*

process and a model-input process were connected to a master process that controlled the grid of rendering processes. Each rendering process had a direct connection to the user interface process that was used to send pixel updates. This was more efficient than trying to relay the pixel updates through the master process.

A message queue was created for each logical process. Processes interacted by sending messages to each other. The messages would reside in the queue until the process was ready to handle them. Each process consisted of a set of message handlers that were associated with specific message types, and expected to receive only messages of recognized types. Sending a message to a process that lacked a handler for that message type would result in a run-time error. Message types were prioritized so that higher priority messages would be handled before lower priority messages. Within a priority level, and within a message type, messages were processed in non-deterministic order.

Algorithms 1 and 2 were combined into algorithm 3. This algorithm describes the message handler that was responsible for the rendering computation. It is one out of one hundred and eighteen message handlers that were constructed for the implementation. Most of these other handlers performed mundane tasks, such as updating the value of a pixel sample, defining portions of the geometric model, or constructing a binary space partition (BSP) tree for use in visibility calculations [46].

Algorithm 3 *(Path Tracing Iteration with Dynamic Load Balancing)*

```
while (workload is nonzero) do begin
    do algorithm 2
    if (workload is below a threshold) then begin
        while (local load imbalance exists) do begin
            send a "rebalance" message to immediate neighbors
            do algorithm 1
        end
    end
end
send a "request release" message to the master process
```

Before executing algorithm 3 we assume the geometric model has been loaded and all necessary data structures have been constructed. In order to start the algorithm the master process sends a "render" message to every rendering process. The "render" message handler loads an initial set of path descriptors into the message queue and then exits. These path descriptors are then processed as messages by the general message handling mechanism, resulting in algorithm 2. At regular intervals this mechanism checks for differences in workload with all immediate neighbors, where workload is counted as the number of unprocessed path descriptors in the message queue. If local imbalance exists algorithm 1 is initiated, which continues until the local imbalance is removed. Neighbors are drawn into a rebalancing operation when they receive a "rebalance" message. This type of message has high priority, higher than path descriptor messages, and so a "rebalance" message will interrupt a process that is within algorithm 2 and cause it to execute algorithm 1.[2] If the workload drops to zero the rendering process sends a message to the master process requesting permission to terminate. The master process maintains a count of rendering processes that have zero workload. When this number rises to equal the total number of rendering processes the master process broadcasts a "render finished" message.

Algorithm 3 will only initiate algorithm 1 when the workload falls below a threshold. In this application there is a finite amount of work to be processed, and the goal of load balancing is to minimize the amount of time computers waste in an idle state. In this application it proved useful to avoid rebalancing until there was a significant danger that a computer could run out of work. Avoiding unnecessary rebalancing reduced the amount of work that was expended in algorithm 1 and sped up the overall computation. In other applications there could be other reasons to invoke load balancing, for example limitations on queue size, in which cases this strategy of reluctant rebalancing could be harmful.

As a result of these policies algorithm 1 is initiated by a type of spreading activation, so that a set of computers will participate in rebalancing if they detect imbalance with their immediate neighbors, but will continue to do useful work if there is no such imbalance. Any computer that detects a local imbalance can pull its neighbors into a rebalancing operation

[2]The pseudocode of algorithm 3 hides some ugly details of implementation. A few different strategies for initiating and propagating the rebalancing operation were implemented in order to understand what impact, if any, they would have on performance. For example, if a processor is already involved in a rebalancing operation it was sometimes convenient to ignore new rebalance requests. We also experimented with explicitly propagating rebalance requests to neighbors where imbalance exists, rather than allowing the neighbors to discover this imbalance on their own. Neither of these issues showed a significant impact on application scaling. In contrast it turned out to be important to use a coarse grained interleaving of algorithms 1 and 2 because it was very easy to waste resources in checking needlessly for load imbalances. The main reason for this was the overhead associated with monitoring the workload of neighbors.

but these neighbors will not propagate this activity unless they also detect imbalances with other neighbors of theirs. Once a computer enters a rebalancing operation it continues as long as it detects a local imbalance. This proved to be usually only one or two steps.

Some other practical considerations became apparent only after experimenting with an initial implementation. For example, it turned out to be preferable to transfer path descriptors that were near the root of the tree of paths, rather than near the leaves. A path descriptor that is near the root of a tree will have many descendents. Transferring such a descriptor to a computer assigns a large amount of work to that computer, in contrast to transferring a descriptor that is closer to the leaves of a tree. Since computers only initiate load balancing if their workload falls below a threshold this transfer strategy increased the average amount of time between rebalancing operations.

Since path descriptors are messages, and are processed out of the general message queues, workloads could be redistributed by one rendering process sending a series of "path descriptor" messages to another rendering process, while removing those messages from it's own message queue. We originally implemented work transfers in this way but this proved to be too costly since it required a large number of messages, and the overhead associated with sending each message was non-trivial. Since the application was running in some cases with a TCP/IP transport over ethernet it wasn't practical to reduce the message overhead to the point where this strategy was feasible. Instead a "bundle of descriptors" message type was created so that an arbitrarily large number of path descriptors could be transferred in a single message.

4.4 Analysis of expected initial conditions

In order to test algorithm 1 it's necessary to start from a balanced condition. This is not as simple as it may first appear. This section examines three strategies for achieving initial balance and finds problems with all of them. The best of these strategies, "scattering", will be used to initialize the experiment.

In ray tracing it is useful to begin a computation with an initially balanced workload. A number of strategies have been explored for achieving this initial balance [4, 6, 11, 16, 38, 48, 51, 53, 68]. Among the more successful strategies has been the use of a random initial mapping [53]. The expected behavior of this approach can be predicted from the central limit theorem. This theorem states that when a sufficiently large number of samples w_i are drawn from a sample population the sums of any sets of p samples will take on a normal distribution. If the sample population is uniform then it is possible to quantify the probability that the sum of any individual set of p samples is below a given bound y [42]. In the general case we have

$$P\left[\sum_{i=1}^{p} w_i \leq y\right] = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{(y-p\mu)/\sigma\sqrt{p}} \left(e^{-u^2/2} du\right) = \frac{1}{2} + \frac{1}{2}\mathrm{Erf}\left(\frac{y-p\mu}{\sigma\sqrt{2p}}\right). \tag{4.5}$$

In this expression μ and σ^2 are the mean and variance of the sample population. When this is applied to the problem of ray tracing for NTSC resolution images we take p equal to m/n where m is 307,200, the total number of pixels in a 640×480 image, and n is the number of computers.

We can use this expression to determine the probability that a random mapping will achieve a load balanced within a factor ϵ. We will call ϵ the workload imbalance that results from a mapping, defined as $1 - t_i/T_{min}$ (see equation 2.5). For example, if ϵ is 0.1 then t_i is 10% greater than T_{min} and therefore the workload is imbalanced by 10%. We define y to be $(1+\epsilon)T_{min}$ to obtain a formula for a new quantity Pr

$$Pr \equiv P\left[\sum_{i=1}^{p} w_i \leq (1+\epsilon)\, T_{min}\right] = \frac{1}{2} + \frac{1}{2}\mathrm{Erf}\left(\frac{\epsilon\mu m}{n\sigma\sqrt{2p}}\right). \tag{4.6}$$

The probability that all n computers will be balanced to within a factor of ϵ is the product of n instances of Pr, that is, Pr^n. This quantity can be computed for various numbers of computers. The result of such a computation is shown in figure 4.3. This figure shows that the probability of achieving $\epsilon \leq 0.10$ by a random strategy declines rapidly with increasing numbers of computers. This decline should not be surprising since the number of pixels is fixed at NTSC resolution. As more computers are applied to the rendering problem fewer pixels are assigned to each computer and the law of large numbers breaks down. This problem of achieving equal workload distribution is made more difficult by tiling strategies, which are a common way to partition work among computers. These strategies partition an image into rectangles and compute one rectangle on each computer. The result of these strategies is to reduce the value of p by the size of the tiles, which reduces Pr.

The accuracy of equation (4.6) can be affected by skew in the sample population.[3]

[3]The nature of the skew is important. So-called "right skew", in which the distribution has a long tail

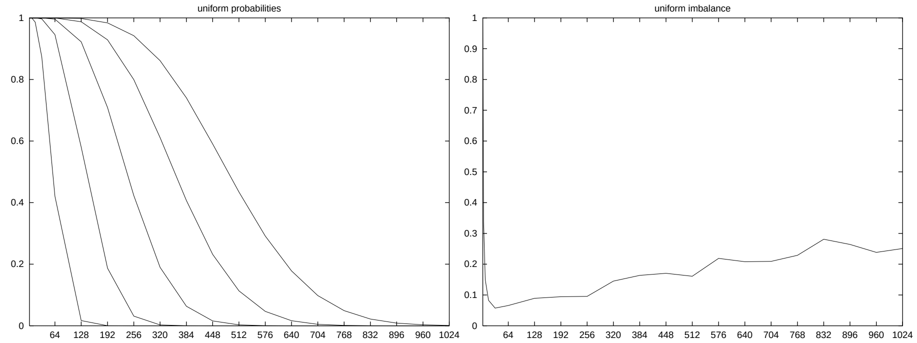

Figure 4.3: *The result of applying a random assignment strategy to a uniformly distributed sample population with mean* $\mu = 10^6$ *and variance* $\sigma^2 = 10^{12}$. *Left figure shows the probability* Pr^n *(see equation 4.6) of obtaining acceptable* ϵ *for (left to right)* ϵ *of 0.05, 0.10, 0.15, 0.20, and 0.25 for increasing numbers of computers (horizontal axis). The probability of obtaining* $\epsilon \leq 0.10$ *drops below 0.5 somewhere between 128 and 192 computers. Right figure shows the predicted* ϵ *resulting from a random assignment of this population to increasing numbers of computers (horizontal axis).*

Complex images generally do not have uniformly distributed w_i as is evident from figure 4.4. As a result the predictions of (4.6) may err on the side of optimism. For example, compare the simulated results of figure 4.4 with the predictions of table 4.1. As ray tracing algorithms become increasingly sophisticated they generally increase the skew in the sample population, decreasing the chances of success for random load balancing strategies. The absolute magnitudes of μ and σ^2 are irrelevant since ϵ is defined as a percentage of μ rather than an absolute value. For all of the above reasons we conclude that these predictions apply to a wide range of images and ray tracing algorithms.

Many static load balancing strategies partition the image plane among a set of computers and then render the different portions of the image concurrently [14, 16, 21, 51, 53, 68]. Naive strategies that partition contiguous segments of the image fare poorly [21]. These strategies suffer from the effects of locality in the image which can lead to wide variations in workload for different computers. A more effective strategy assigns pixels pseudo-randomly to computers in an attempt to obtain a balanced workload. The most straightforward version of this strategy assigns pixels in an alternating sequence so that $M(i) = (i \bmod n)$. This strategy is sometimes called the "scatter" method [53].

The effectiveness of various static load balancing strategies can be predicted using data

to the right of the mean, leads to poor performance because it is hard to balance large statistical outliers. In contrast "left skew" does not present these problems. The cases we have examined all exhibit right skew and this seems to be typical of ray traced images and probably most other types of images as well.

			Number of computers n				
Model	μ	σ^2	16	32	64	128	256
Office	9.72×10^5	1.38×10^{12}	0.99	0.98	0.87	0.34	0.00
Soda	8.70×10^5	5.10×10^{12}	0.96	0.71	0.18	0.00	0.00
Bath	1.37×10^6	8.95×10^{12}	0.76	0.30	0.01	0.00	0.00

Table 4.1: *Probability (Pr^n) of obtaining $\epsilon \leq 0.10$ by random assignment. See equation (4.6). μ and σ^2 represent the mean and variance of the number of floating point operations required to compute an NTSC resolution image using an adaptive sampling strategy of from 1 to 25 samples per pixel. In no case is random assignment likely to be effective for more than 128 computers. This conclusion is independent of the type of ray tracing algorithm used and the distribution of the sample population w_i.*

obtained from program traces generated in the course of rendering complex images. A trace captured empirical values of w_i (required floating point operations) for each pixel. The three images were then rendered using an adaptive sampling strategy that generated from one to twenty five primary rays through each pixel. After rendering, the w_i were collected from the program traces and were used to predict workload distributions for a naive scanline-order partitioning and the scatter method on varying numbers of computers. These predictions are shown in figure 4.4.

These results suggest that the naive strategy fares poorly in general, and that even a random strategy will fail to obtain $\epsilon \leq 0.10$ when the number of computers is 128 or more. They also reveal that the scatter method, while often assumed to be equivalent to a random assignment, is in fact subject to pathologies related to spatial regularity in the image, as illustrated by the peaks that occur at multiples of 320 pixels.

4.5 Measurements of observed scaling

The accuracy of these predictions was tested empirically by measuring the elapsed times to render the three images on various numbers and types of computers using the naive and scatter strategies. The resulting measurements are shown in table 4.2 and plotted in figure 4.5. The measurements are generally in close agreement with the predictions but there are a few surprises. For example, the scatter decomposition worked well for the bath image on 16 computers but performed poorly on 15 computers. The converse was true for the image

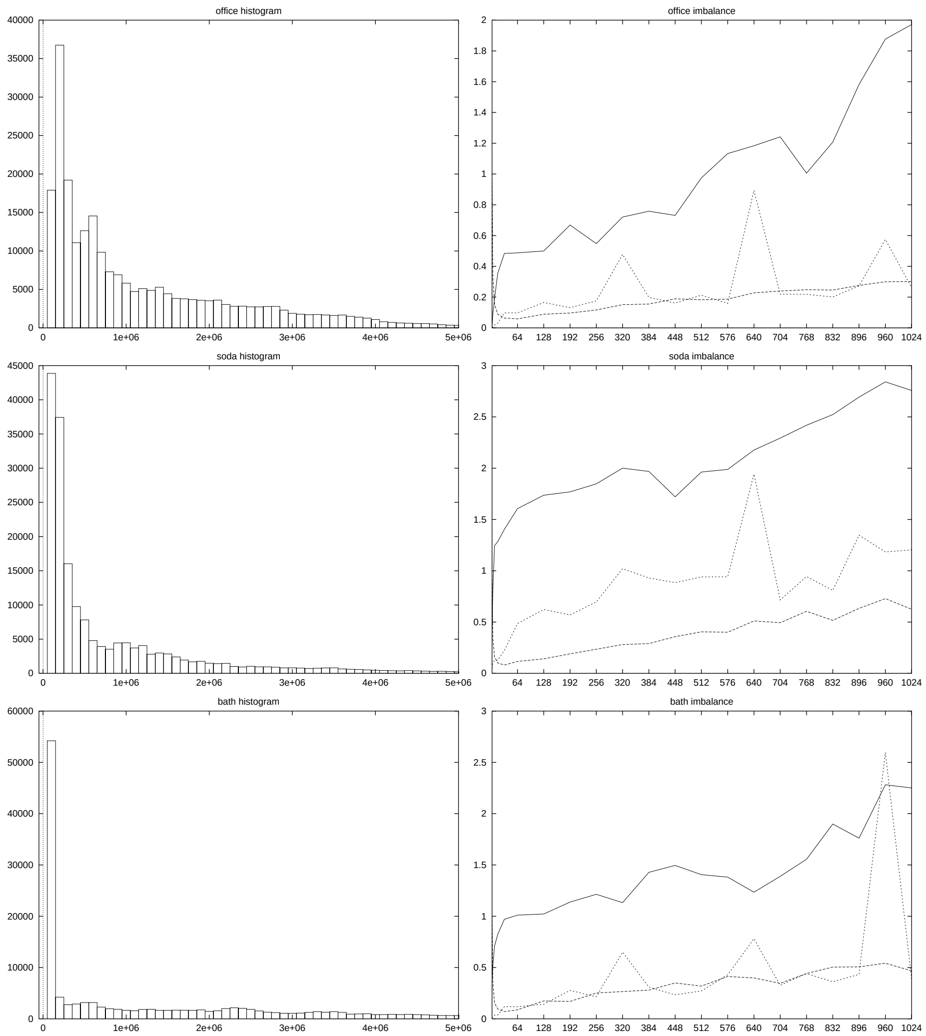

Figure 4.4: *Distribution histograms and predicted imbalances for (top to bottom) office, soda, and bath images. All three images have highly nonuniform distributions as illustrated by their histograms (left column). These histograms show frequency of occurrence (vertical axis) versus floating point operations (horizontal bins). For each image the predicted imbalance ϵ (right column) is shown for three static load balancing strategies (naive, scatter, and random) on increasing numbers of computers (horizontal axis). In all cases the naive strategy produces the largest predicted ϵ (vertical axis) and the random strategy produces the smallest. The scatter decomposition is generally comparable to the random strategy but suffers from isolated peaks at multiples of 320 pixels, exactly one half the width of the image. In all cases the predicted ϵ is greater than 0.10 when the number of computers is 128 or higher.*

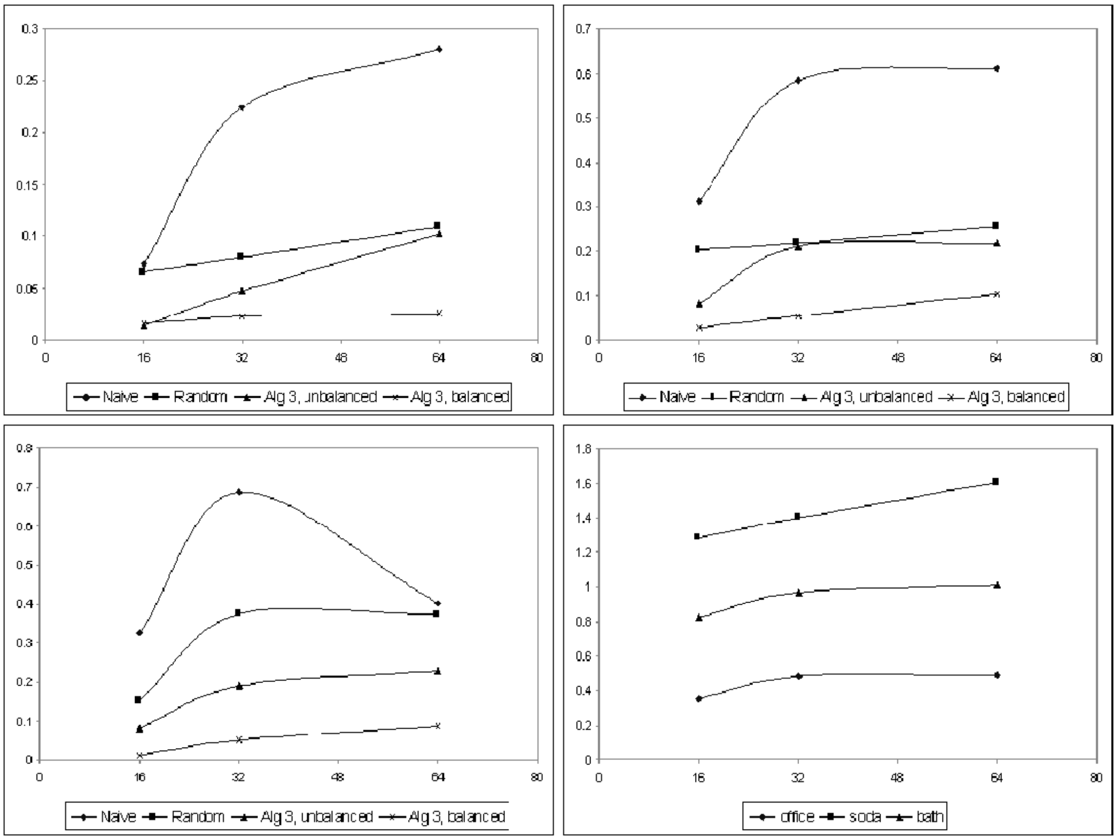

Figure 4.5: *Clockwise from lower left: measured ϵ versus n for the bath, for the cases shown in table 4.2, including four load balancing scenarios; the same for the office, and soda images; and predictions of ϵ resulting from a random initial mapping based on counts of floating point operations from figure 4.4. For every image the best results (lowest line) occur when algorithm 1 follows an initial scatter method. The predicted ϵ is offered for comparison to this best case, and appears to correlate moderately well, particularly for the soda image. This correlation suggests that the increase in measured ϵ under algorithm 1 may be due to the initial imbalance rather than to any failure of scaling. Unfortunately there is not enough data to prove or disprove this.*

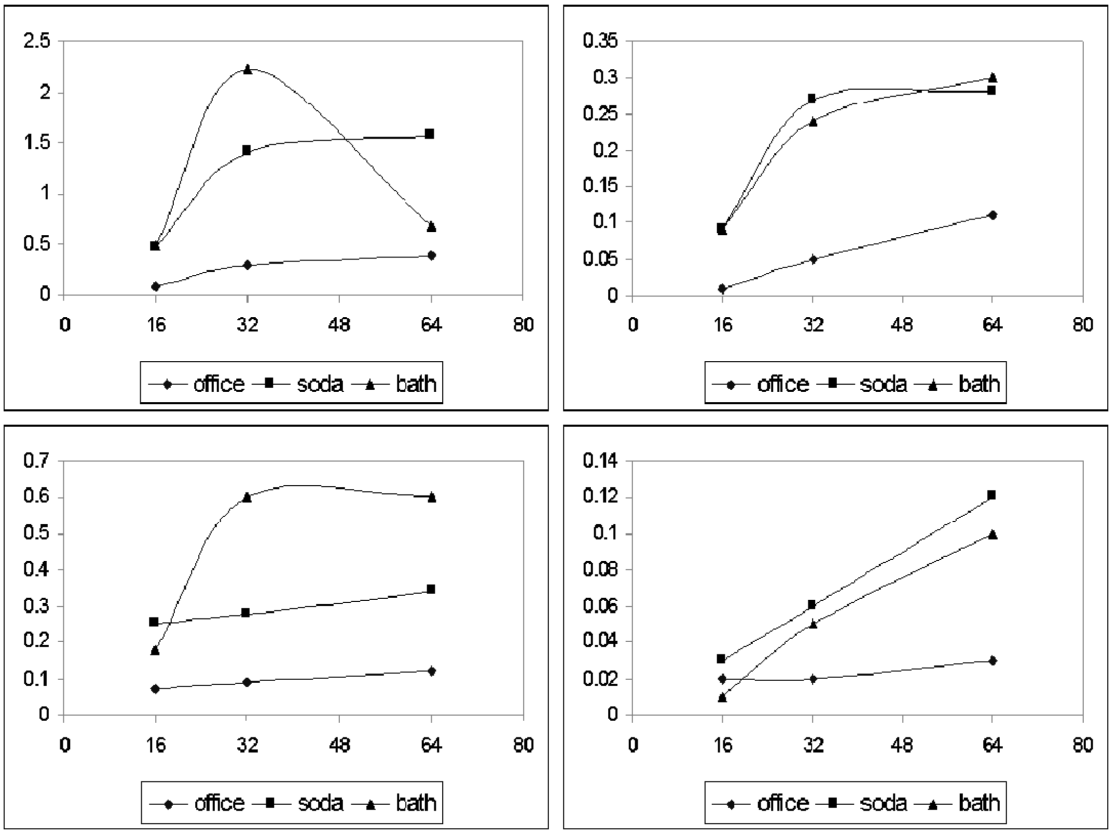

Figure 4.6: *Comparison of effective imbalance ϵ with and without dynamic load balancing by algorithm 1, for the three images in table 4.2. Clockwise from upper left: naive method only; naive method followed by algorithm 1; scatter method followed by algorithm 1; scatter method alone. In several cases the initial imbalance seems to correlate with the the effective imbalance under algorithm 1.*

Model	n	naive T	naive ϵ	scatter T	scatter ϵ	unbalanced T	unbalanced ϵ	balanced T	balanced ϵ	T_{min}
Office	15	1:08:10	0.22	1:00:46	0.05	58:41	0.02	57:51	0.00	57:38
	16	1:00:53	0.08	1:00:24	0.07	57:15	0.01	57:22	0.02	56:25
	32	36:29	0.29	30:46	0.09	29:42	0.05	28:58	0.02	28:18
	64	19:44	0.39	15:56	0.12	15:49	0.11	14:35	0.03	14:12
Soda	15	1:34:08	0.30	1:16:31	0.06	1:17:22	0.07	1:13:03	0.01	1:12:13
	16	1:50:57	0.46	1:35:27	0.25	1:23:06	0.09	1:18:13	0.03	1:16:07
	32	1:32:08	1.41	48:51	0.28	48:28	0.27	40:25	0.06	38:12
	64	49:05	1.57	25:37	0.34	24:25	0.28	21:21	0.12	19:05
Bath	15	2:14:46	0.40	2:06:09	0.31	1:42:46	0.07	1:38:58	0.03	1:36:28
	16	2:24:17	0.48	1:55:15	0.18	1:46:06	0.09	1:38:35	0.01	1:37:24
	32	2:36:44	2.22	1:17:59	0.60	1:00:04	0.24	51:14	0.05	48:38
	64	40:29	0.67	38:43	0.60	31:24	0.30	26:34	0.10	24:11

Table 4.2: *Empirical results measuring effective imbalance ϵ in rendering the office, soda, and bath images with different load balancing strategies on varying numbers of computers. See figure 4.5. In every case the best results came from dynamic load balancing by algorithm 1 preceded by the scatter method ("balanced") In most cases algorithm 1 by itself produced better results than scattering even from poorly balanced initial conditions ("unbalanced"). In general the problem becomes more difficult with increasing numbers of computers. These results have been reported in publications [21, 22, 25].*

of the soda shop interior. In all cases the naive decomposition produced unacceptably high values of ϵ, as high as 2.22 for the bath image on 32 computers. Figure 4.4 suggests that all of these methods will break down as n increases, and that these static load balancing strategies are inadequate except at small scales.

Other results that support these conclusions appear in tables 4.3 and 4.4. These results are from an initial study that tested algorithm 1 starting from the naive initial mapping. These results are comparable to the "unbalanced" results of table 4.2. This initial study used a fully distributed algorithm to detect termination. These measurements showed that the termination algorithm had a flaw that caused it to waste a considerable amount of time at large n. This flaw resulted in a large Amdahl fraction[4] and prevented further speedup beyond 100 computers. In fact the office and conference images slowed down between 128 and 256 computers, which is clearly a pathology. The source of this error was never fully discovered. In order to circumvent this error a master-slave termination mechanism was implemented as described in algorithm 3.

[4]The Amdahl fraction is the portion of a computation that cannot be sped up through parallelism. It is the ultimate limiter of scaling for any concurrent algorithm [2].

This study was repeated on a cluster of inexpensive Intel workstations connected by a fast ethernet switch, as reported in table 4.4. All of the rendering times were slightly better than on the SP2. Although both systems offered the same peak floating point performance of 200 MFlops this result is somewhat surprising because current Intel processors give notoriously poor floating point performance for codes such as this that mix floating point and integer operations.

The purpose of these measurements was twofold. One purpose was to compare the effectiveness of algorithm 1 against competing approaches. In this respect the algorithm has performed remarkably well, in that by itself it produces results at least as good as the scatter method, one of the most popular load balancing methods for ray tracing.

It's instructive to examine figure 4.6 which provides direct comparison of runs with and without algorithm 1. There appears to be some correlation between the measured ϵ resulting from a purely static load balancing strategy, and the measured ϵ when the static strategy is followed by algorithm 1. Such a correlation would suggest that, if the initial conditions for each run had been perfectly balanced, then algorithm 1 would have produced a constant value of ϵ in every case. Unfortunately this study does not have enough data to show such a correlation definitively and this can only be speculated. There are some cases that would appear to argue against this speculation, such as the scattered cases for the office and soda images which both show increasing ϵ but with very different slopes.

A second purpose for these measurements is to test the hypothesis of scalability. These data for ϵ have done this indirectly, from the implication that a failure of scalability in algorithm 1 must necessarily manifest itself in a failure of scalability of the application. Since the application showed good scaling we might conclude that algorithm 1 also showed good scaling. Unfortunately it's not safe to draw this conclusion, because it is possible that algorithm 1 could scale poorly, but represent such a small part of the overall computation that the effect of this does not appear in the measurement of ϵ. It's necessary to examine an explicit measurement of the time spent in algorithm 1.

4.5.1 Measurement of algorithm 1

Figure 4.7 and table 4.5 show data that measure the amount of time spent explicitly within algorithm 1 during the overall execution of the application. Due to the way that data

	Office		Glass		Conference	
n	T	ϵ	T	ϵ	T	ϵ
1	29:11	0	39:53	0	180:37	0
8	3:50	0.048	5:04	0.016	22:44	0.007
16	1:58	0.073	2:33	0.022	11:25	0.011
32	1:02	0.117	1:20	0.065	5:53	0.041
64	0:41	0.333	0:40	0.065	3:02	0.070
128	0:33	0.585			1:38	0.136
256	0:35	0.805			2:23	0.704

Table 4.3: *Results of an initial study on the SP2. In all of these runs the naive strategy was used for the initial mapping followed by algorithm 1. This study revealed the existence of errors in the original distributed algorithm for termination detection. These errors showed up the runs on 128 and 256 computers, where they caused a slowdown rather than a speedup. These results have been reported in publications [24, 26].*

	Office		Glass		Conference	
n	T	ϵ	T	ϵ	T	ϵ
1	26:15	0	34:19	0	162:21	0
8	3:26	0.044	4:21	0.014	20:34	0.013
15	1:51	0.054	2:23	0.040	11:02	0.019

Table 4.4: *Rendering times on Intel cluster. These runs were computed with parameters identical to table 4.3 and are directly comparable. In every respect the performance was slightly better than on the SP2. These results have been reported in publications [24, 26].*

	Office				Soda				Bath			
	balanced		unbalanced		balanced		unbalanced		balanced		unbalanced	
n	ms-lb	sec	ms-lb	sec	ms-lb	sec	ms-lb	sec	ms-lb	sec	ms-lb	sec
16	68	3442	38	3435	48	4693	231	4986	1305	5915	60534	6366
32	60	1738	68	1782	38	2425	254	2908	2013	3074	67668	3604
64	51	875	48	949	50	1283	243	1465	1003	1595	37197	1884

Table 4.5: *Time measured within algorithm 1, in milliseconds, and total elapsed time, in seconds, for six cases. See figure 4.7. Each of three images was started from two initial mappings. One mapping had a reasonably well balanced workload while another was poorly balanced. These measurements show that the amount of work expended by algorithm 1 is affected by the amount of initial imbalance but not by problem scale.*

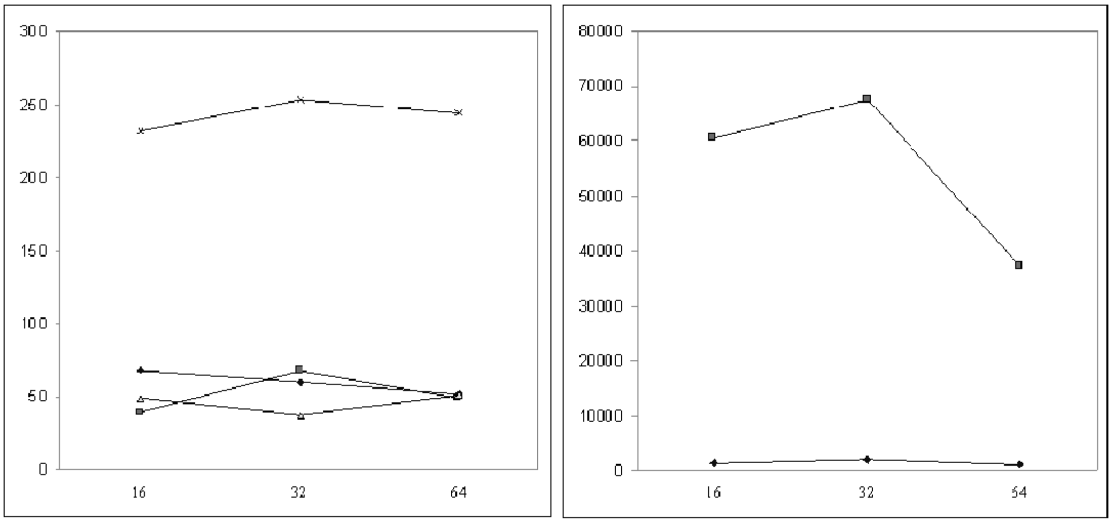

Figure 4.7: *Average number of milliseconds per computer spent in algorithm 1 versus number of computers. See table 4.5. The office, soda, and bath images were computed using identical parameters on 16, 32 and 64 computers. This data was collected from the same runs as the data in table 4.2 but shows only the cases that include algorithm 1. For an explanation of how the measurements were collected please see the final chapter. The left figure shows the office and soda measurements, and the right figure shows the bath measurements. All times are in milliseconds. In every case the naive method led to more time being spent in algorithm 1 than the scatter method did. The number of computers showed no influence on these measurements. All of these results are consistent with a hypothesis that algorithm 1 is insensitive to problem scale for realistic problem instances.*

was collected it was only possible to accurately measure the time that algorithm 1 spent transferring work to neighbors, and not the time spent in receiving work from neighbors. There are compelling reasons to believe that these times are very close to equal, and therefore these measurements are approximately half the total time spent in the algorithm. This data is adequate for the purpose of this analysis, since the scalability of either of these portions of the algorithm should be the same. These results clearly show that the time spent in algorithm 1 is not correlated with n. In most cases the times are roughly constant. In no case are they consistently increasing. This allows to draw our final conclusion, that in this experiment the cost of algorithm 1 at each computer did not increase as the number of computers increased.

The bath image showed two to three orders of magnitude more time spent in algorithm 1 than the soda or office images. Although this image required more load balancing, the overall computation time was not dramatically higher than for the other images, as table 4.2 shows. The bath image contains several inter-reflecting surfaces. Computers that were responsible for pixels in the mirror or window would be sources of frequent imbalance since they would develop very deep trees of paths. This would account for the increased time spent in algorithm 1.

These results are consistent with the hypothesis of scalability. In combination with the other results presented here, the evidence is consistent with the following hypotheses: that the cost of algorithm 1 for any individual disturbance is independent of problem scale; that the cost of repeated invocations of algorithm 1 within this complex application was also independent of scale; and that algorithm 1 is as good or better than the most popular known algorithm for a specific complex application, and in particular, is better than random assignment. These are all positive results, so this experiment can be considered a success.

Chapter 5 Dynamic Mapping

This chapter presents a solution to the dynamic mapping problem. This completes the goal of solving the combined problems of load balancing and mapping. In contrast to *recursive spectral bisection* which is too expensive to use routinely for dynamic problems the algorithms presented in this thesis are ideally suited for dynamic problems. In fact the mapping construction will be seen to have issues that pose challenges in the static case, but that are easily resolved in the dynamic case. This is convenient because, while there are many known algorithms for static mapping, very little has been written about the dynamic problem. One of the only such proposals uses random placement [9]. Such an approach clearly violates the locality constraint of problem 2 and therefore can't provide an optimal solution.

5.1 The mapping problem

The setting for the mapping problem is the same as for load balancing. A set of n computers runs a set of concurrent processes between two barriers. The processes communicate through logical channels and the computers are connected by a communication network. The goal of the *mapping problem* is to map the processes onto computers in a way that minimizes the utilization of the communication network. Most approaches to this problem assume the communication network is built so that the cost of communication between two points is proportional to the distance between them. This notion of "distance" is related to the structure of the communication network, and might be measured by the number of network links and switches that must be traversed from one point to the other. We will assume that this quantity can be measured for any pair of computers. We'll leave the details of this measurement for implementation.

Under this assumption a common approach is to partition a graph of processes into subgraphs in a way that minimizes the size of the interfaces between subgraphs. If each subgraph is mapped onto a different computer then only the interface connections have to use bandwidth from the communication network. The cost of communication is then the sum of the costs of these interfaces. If subgraphs that were adjacent in the original graph are mapped onto computers that are adjacent in the network then the partition and the mapping will be optimal. This is the approach taken by recursive bisection algorithms.

To define the mapping problem assume that we have a set of P processes, a set of n computers, and a mapping function M from processes to computers. Let's construct a $P \times n$ *assignment matrix* A with the property that $A_{i,j} = 1$ if $M(i) = j$ and is zero otherwise. We'll use a nonnegative $P \times P$ *communication matrix* C where $C_{i,j}$ is the bandwidth required between processes i and j. And also a symmetric positive $n \times n$ *distance matrix* D where $D_{i,j}$ represents the distance from computer i to computer j.

Problem 3 *(Dynamic Mapping)*
Given a set of computers and a set of processes, both indexed by integers, a mapping M from processes to computers, and matrices $A(M)$, C and D as described above.
Find a new mapping $\hat{M}$ to minimize $Obj \equiv \|CA(\hat{M})DA(\hat{M})^{-1}C^{-1}\|_2$.

This definition is similar to the original formulation of this problem by Kung & Stevenson [39]. In informal terms the dynamic mapping problem seeks a remapping that minimizes the product of communication bandwidth and routing distance over all pairs of processes.

5.2 A dynamic mapping algorithm

After a distributed application has been mapped onto a computer system, remapping will be necessary if changes occur in the pattern of communication among processes (represented by C) or in the topology or capacity of the interconnection network (represented by D). In this section we'll apply the construction method presented in the second chapter to define an algorithm for dynamic remapping. The intuition behind this construction is an observation that solutions to the mapping problem resemble equilibrium distributions of communication cost. This cost is the product of two quantities: the bandwidth $C_{j,j'}$ required between processes j and j'; and the routing distance $D_{M(j),M(j')}$ between the computers that processes j and j' are mapped to.

Solutions that are perfect equilibria are not guaranteed to exist, and if they exist, they may not be unique. For example, suppose that the computer network is a mesh, which can be embedded in the plane, but that the graph of processes is non-planar. To make this

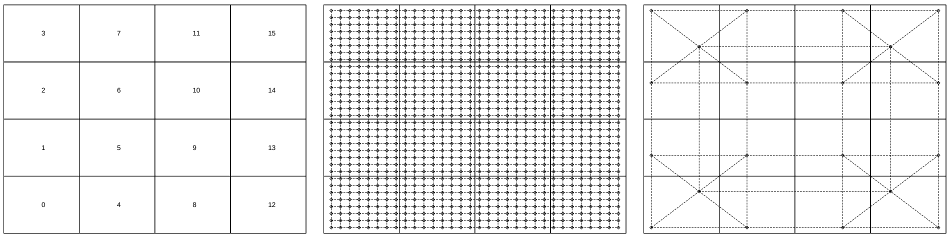

Figure 5.1: *Perfect equilibria are not possible for all cases, but an approximation to equilibrium is still a good solution to the mapping problem. Left, the setting for a model problem on a grid of 16 computers. A rectangular region has been partitioned so that each computer occupies a subregion and the interfaces between these subregions represent network links. Middle, an ideal solution for an easy problem with a set of 1024 processes that communicate in a regular grid. In this solution each process is equidistant from all processes with which it communicates. Right, a solution for a realistic problem in which set of processes perform a structured multigrid calculation. This process graph is non-planar, and the solution shown here is the best that can be achieved even though it is not in perfect equilibrium.*

example simple, suppose $C_{i,j}$ is always either zero or one. Then there is no way to embed the process graph into the network with equal communication cost between every pair of processes. An example of this appears in figure 5.1.

The possible non-existence of solutions is not a problem. The important point is that there is an ideal characterization of a solution, namely a perfect equilibrium, which may or may not be a feasible solution for a given problem instance. Any algorithm that makes monotonic progress toward this ideal solution is capable of generating a series of increasingly good approximate solutions, and it is this series that we want to compute. An algorithm that can guarantee monotone convergence toward an equilibrium, such as algorithm 3.5, might therefore be a good candidate solution to the mapping problem.

We'll discuss some problems with this approach, and explain why it is more suitable for the dynamic mapping problem than for the static problem. But first let's define an algorithm. This algorithm is executed for every process j and is computed by computer $M(j)$. The mapping function M defines locations in a real numbered space as shown in figure 5.1.

Algorithm 4 *(Dynamic Mapping)*

$\hat{M} = M$

for (some number of steps) do begin

$sum_x = sum_y = sum_w = 0$

for $j' = 1$ *to* P *do begin*

$(x, y) = M(j')$

$sum_x = sum_x + x$

$sum_y = sum_y + y$

$sum_w = sum_w + C_{j,j'}$

end

$\hat{M}(j) = \left(\frac{sum_x}{sum_w}, \frac{sum_y}{sum_w}\right)$

if (step is odd) then remap process j *to* $\hat{M}(j)$

end

It should be apparent that this is a variation of algorithm 3.5 that shows monotone convergence to an equilbrium. On every step each process j computes a new location, and on every other step, j remaps itself to that new location. This new location is equidistant from the most recent positions of all processes with j communicates. Clearly when all processes are equidistant the algorithm is at a fixed point. Under all circumstances the algorithm converges the solution toward an equilibrium until it becomes trapped in a local optimum and can make no further progress. Because the algorithm only remaps on alternate steps it avoids the potential for oscillation that is illustrated in figure 3.4. As a result it shows monotone convergence to equilibrium, and therefore monotone improvement in the approximate solution.

5.3 Simulations of a model problem

Figures 5.2 and 5.3 illustrate algorithm 4 operating on the model problem with a regular grid of processes. In both cases the algorithm converges to an approximation of the ideal solution. Both of these approximations have a flaw, in the form of a persistent sinusoid such as the one illustrated in figure 3.1.

We'll talk about this, but first let's look at a more serious problem illustrated in figure

5.4. This figure shows two solutions that can result from the initial conditions of figure 5.2. The only difference between these solutions, which are clearly incorrect, and the solutions of figures 5.2 and 5.3 which are approximately correct, are in the way the corners are constrained. In figures 5.2 and 5.3 the corners of the process graph have been constrained in an order preserving way, and so resulting solutions must approximate the perfect solution shown in figure 5.1. In contrast the two solutions shown in figure 5.4 have their corners constrained so that correct solutions are impossible.

Figure 5.4 illustrates what can happen if the algorithm becomes trapped in a pathological local optimum. The solution to this is to avoid these pathological regions. In the case of the dynamic mapping problem this is easy to do, if we assume that the application is initially well-mapped. In order to avoid converging to a pathological solution it is only necessary to ensure that some of the processes are fixed in place so that the processes that move do so in a way that avoids pathologies. This will always be possible in a remapping operation, because in the worst case a periphery can be identified for any sub-graph, and this periphery can be fixed while all of the other processes reposition themselves. In practice it is probably best to initiate the algorithm with a type of spreading activation, and to remap small local regions when they become disrupted.

The problem of removing persistent sinusoidal error from the converged solution is more difficult. It is unreasonable to expect algorithm 4 to remove them by itself, as is illustrated in figure 3.1. But when algorithm 4 is combined with algorithm 1, or any of several other local load balancing algorithms, the two interact in a way that can alleviate this problem. The cases in figures 5.2 and 5.3 would be resolved by algorithm 1 implemented in a way that observes the locality constraint of problem 2.

5.4 Discussion

Algorithm 4 is similar to a well-known algorithm for embedding a graph in the plane [60]. This variant extends the plane to arbitrary dimensions so that a computer network of any topology could be an embedding target. It also relaxes the definition in a way that allows a notion of continuous convergence to an approximate solution, so that instances can be handled for which an optimal solution does not exist, such as embedding a non-planar graph in the plane.

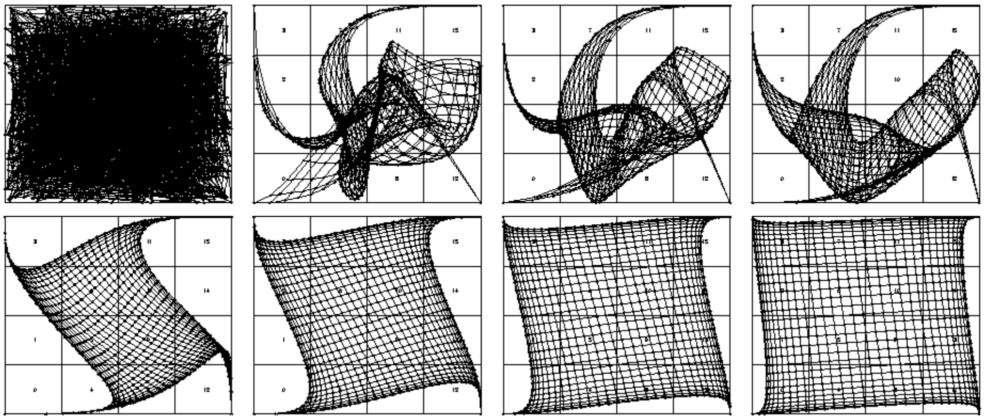

Figure 5.2: *An example of a regular graph sorting itself out from a difficult initial condition. This required constraining the corner positions. Note the persistent sinusoid.*

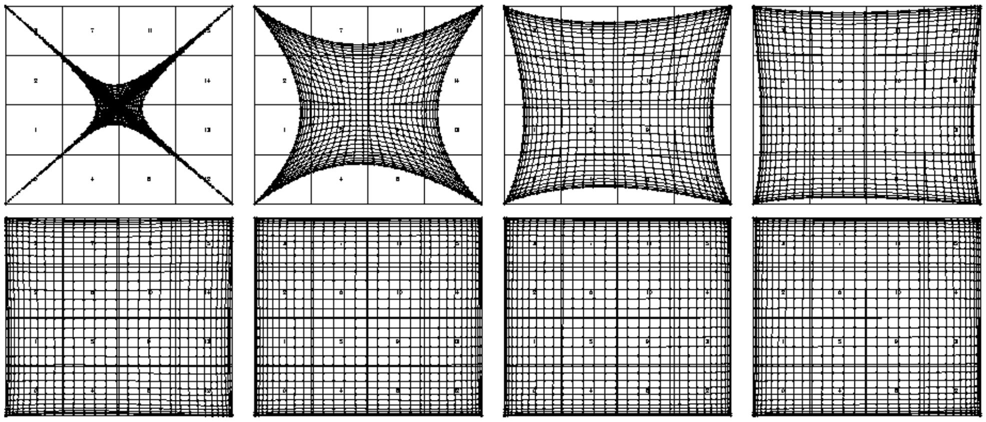

Figure 5.3: *The same problem, started from a nice initial condition. The final figure still has a persistent sinusoid, with points clustered toward the periphery.*

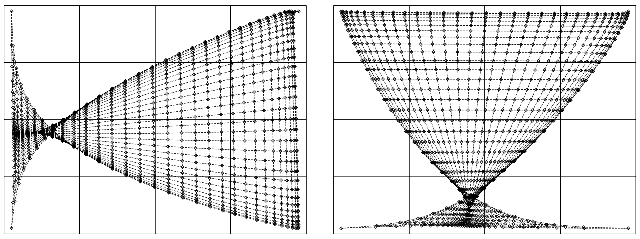

Figure 5.4: *Two incorrect solutions that can result from the initial condition of figure 5.2.*

The purpose of this discussion has been to show that the approach to deriving algorithms that is presented in this thesis is not restricted to a single problem (load balancing) but instead may be applicable to other problems that resemble load balancing and mapping. There are number of problems which are known to be similar to these: circuit partitioning and layout, network capacity planning, and sparse matrix reordering, to name just a few [9, 52].

This thesis began by mentioning recursive spectral bisection (RSB) and discussing properties of the Laplacian matrix of a graph. This construction method has been used for the same two problems that are solved by RSB, namely mapping and load balancing. Some comments about RSB are in order to place these algorithms in perspective. RSB is very expensive since it requires solving an eigenvalue problem many times, once for each bisector. This is normally done by Lanczos iteration [17] which can be implemented in a distributed way. But Lanczos iteration scales poorly, with characteristics similar to the standard iterations.[1] RSB is strictly more powerful than the algorithms presented here, because it can solve the mapping problem without requiring any constraints, in contrast to algorithm 4 which requires constraints in order to ensure correctness.

These considerations suggest that RSB may be well complemented by algorithms 1 and 4. RSB is frequently advocated to compute an initial mapping M for problems in technical computing, where M solves the load balancing and mapping problems as they are defined in this thesis. But RSB is recognized as being too expensive for repeated use in dynamic instances of these same problems. The algorithms introduced here are good solutions for the dynamic problems, because they have low cost and are scalable. Algorithm 4 seems to be useful for the dynamic mapping problem when a valid initial mapping M has already been computed, because it can take its constraints from this initial M.

[1] Lanczos iteration is closely related to the *conjugate gradient* algorithm for solving linear systems of equations. This algorithm shares many of the convergence properties of the standard iterations.

Chapter 6 A Load Balancing Trace

About two dozen images were computed under a variety of conditions and operating parameters. Most of these were of models from the *Materials and Geometry Format Data Base* which is published on the world wide web by the Lawrence Berkeley Laboratories. The largest model (the battleship) had over 300,000 polygons and several hundred light sources. The smallest model (the office) had 3,256 objects including three light sources. Most of these images were computed in the course of developing the implementation and experimenting with policy details of algorithm 3. Detailed statistics were collected for six images most of which were computed at very high sampling rates.

The cost of algorithm 2 was controlled by a set of parameters that are described in table 6.1. These parameters describe the number of initial descriptors for each pixel (PIXEL_SAMPLES), the maximum number of points $(x, x', x'', \ldots)$ that will be followed along a single path (PATH_GENERATIONS) and the degree of branching at each reflection point. The diffuse characteristic usually decayed below the threshold of perceptibility after a path had encountered two or three reflections, so a typical tree with a diffuse branching factor of 100 would comprise $10,000$ or $1,000,000$ path descriptors for the diffuse characteristic.

Problems were run on a variety of platforms but most studies were done on the IBM SP series of machines and on a cluster of Intel workstations connected by a dedicated fast ethernet switch. In general the performance was slightly better on the Intel platform, and

PIXEL_SAMPLES	Image plane samples per pixel	16
DIRECT_SAMPLES	Shadow rays toward each light	9
LAMBERTIAN_SAMPLES	Reflected rays at a diffuse surface	100
PHONG_SAMPLES	Reflected rays at a glossy surface	64
PATH_GENERATIONS	Maximum depth of a path tree	99

Table 6.1: *Parameters that determine the sampling rates used in algorithm 2 and values used for typical runs. Some of the runs used values of PIXEL_SAMPLES as high as* 100. *For the office, soda, and conference models the value of PATH_GENERATIONS was 2 so that only direct lighting was computed.*

the compile time under *gcc* with all optimizations enabled was roughly five times faster than with the Aix compiler on the SP2. The SP systems were located at Caltech, the Argonne National Laboratory, the Cornell Theory Center, and the Maui High Performance Computing Center. The Intel cluster was located at Caltech and constructed by staff at the Center for Advanced Computing Research. The performance on the SP2 was evaluated by profiling and by using the AIX *monitor* program on individual nodes while the application was running. *Monitor* showed that system call overhead was consistently below 5% demonstrating that the application was dominated by computation and not by communication. The floating point performance was evaluated on the Intel cluster and was found to be consistently below 10% of peak. This result is unsurprising because the code mixed integer and floating point operations indiscriminately and did not optimize for floating point utilization. Floating performance on current Intel processors is notoriously bad when it is mixed with integer operations due to inter-dependencies between the floating point and integer cpu pipelines. It is certain that careful implementation could improve the efficiency of this code, but this is unlikely to have any impact on the conclusions of this study.

The following pages show a representative trace from a run on 32 rendering computers which has been edited for brevity. This trace includes a detailed description of the load balancing behavior. This run took place on the IBM SP2 system at the Maui High Performance Computing Center. The job starts by launching a “daemon” process on every computer. A console process computes an initial mapping of application processes onto computers and communicates that mapping to the daemons. The daemons start all of the application processes and establish communication channels among them. There are 32 rendering processes named ‘obj’ and three additional processes named ‘ui’, ‘mas’ and ‘inp’. Each rendering processes is on a dedicated computer while the other three processes are on an additional computer that was not involved in rendering. All of the runs were mapped in this way. The user interface process ‘ui’ parses a script that loads a geometric model, constructs a BSP tree, defines a camera, establishes parameters, and then causes the ‘obj’ processes to compute an image by executing algorithm 3.

At the end of the trace each ‘obj’ reports three cumulative statistics. The first statistic is the elapsed time the process spent in algorithm 3. It is reported in this way:

```
#object 0 elapsed 40 min 25 sec
```

The second statistic is the total time the process spent in load balancing that involved exporting work to neighboring processes. This statistic is proportional to the total time spent in algorithm 1 but is lower because it reports only one side of each two sided transaction. It seems reasonable to expect that the total time spent in algorithm 1 is approximately twice this figure since the operations of exporting and importing work are very similar. This statistic is reported in this way:

```
#object 0 spent 70 ms transferring work to others
```

The third statistic is the sum of the elapsed times spent in algorithm 1 with concurrent execution of algorithm 2. This statistic is neither the time spent in algorithm 1 nor the total time spent in algorithm 2 nor the total time spent in algorithm 3. Instead it represents the sum of two quantities: the total time the process spent in load balancing that involved importing work from neighboring processes; and the total time spent in algorithm 2 while waiting for neighboring processes to respond to requests to participate in load balancing operations. This first quantity is approximately equal to the second statistic, the time spent exporting work. This second quantity is considerably larger than the first quantity and in the cases measured here is roughly proportional to the first statistic, the total elapsed time in algorithm 3. It is reported in this way:

```
#object 0 spent 116811 ms in load balancing (self)
```

```
::
:: Start of AdaptSeriesSoda.33proc.script
::
Tue Feb  4 19:07:37 HST 1997
fr5n14.mhpcc.edu
/u/heirich/private/RayTracing/Run
::
:: Making node list.
:: Making ghost map.
::
Tue Feb  4 19:07:37 HST 1997
make ghost map for mhpcc jid SODA33
::
:: Making config.
::
Tue Feb  4 19:07:37 HST 1997
here is the config file:
ghost.map.SODA33
Config/Hosts/host.33.graph
Config/Guests/guest.grid.33.graph
Config/child.type.mhpcc
::
:: Starting daemons.
::
Tue Feb  4 19:07:38 HST 1997
::
:: All daemons are started.
::
Tue Feb  4 19:08:11 HST 1997
::
:: Creating console input.
::
::
:: Here is the console input:
::
run file 'Cmd/AdaptSeriesSoda.cmd';
quit;
::
:: Start console program
::
Tue Feb  4 19:08:11 HST 1997
Connecting to initial daemons defined in
'ghost.map.SODA33'.
Connecting host graph defined in
'Config/Hosts/host.33.graph'.
Host graph connected ok.
Starting 35 application processes.
Issued 35 invocations.
All application processes have started.
Connecting guest graph 'Config/Guests/guest.grid.33.graph'
Issued 150 connection requests.
All connection requests honored.
#synchronized child 'obj' pid 22244 host fr5n04.mhpcc.edu
#synchronized child 'obj' pid 21786 host fr6n02.mhpcc.edu
#synchronized child 'obj' pid 21652 host fr10n12.mhpcc.edu
#synchronized child 'obj' pid 18588 host fr18n06.mhpcc.edu
#synchronized child 'obj' pid 17368 host fr18n14.mhpcc.edu
#synchronized child 'obj' pid 17948 host fr6n05.mhpcc.edu
#synchronized child 'obj' pid 23582 host fr18n09.mhpcc.edu
#synchronized child 'obj' pid 18316 host fr18n15.mhpcc.edu
#synchronized child 'inp' pid 22168 host fr5n14.mhpcc.edu
#synchronized child 'obj' pid 22526 host fr10n10.mhpcc.edu
#synchronized child 'obj' pid 17380 host fr18n11.mhpcc.edu
#synchronized child 'obj' pid 24280 host fr10n04.mhpcc.edu
#synchronized child 'obj' pid 21846 host fr10n03.mhpcc.edu
#synchronized child 'obj' pid 37428 host fr10n09.mhpcc.edu
#synchronized child 'obj' pid 18934 host fr18n07.mhpcc.edu
#synchronized child 'obj' pid 21288 host fr18n03.mhpcc.edu
#synchronized child 'obj' pid 2628 host fr5n08.mhpcc.edu
#synchronized child 'obj' pid 16214 host fr5n06.mhpcc.edu
#synchronized child 'obj' pid 17214 host fr5n07.mhpcc.edu
#synchronized child 'obj' pid 19198 host fr5n02.mhpcc.edu
#synchronized child 'obj' pid 17556 host fr5n09.mhpcc.edu
#synchronized child 'obj' pid 21302 host fr5n16.mhpcc.edu
#synchronized child 'obj' pid 18632 host fr5n13.mhpcc.edu
#synchronized child 'obj' pid 37306 host fr18n10.mhpcc.edu
#synchronized child 'obj' pid 20214 host fr10n08.mhpcc.edu
#synchronized child 'obj' pid 16224 host fr18n02.mhpcc.edu
#synchronized child 'obj' pid 18740 host fr10n15.mhpcc.edu
#synchronized child 'obj' pid 37896 host fr17n16.mhpcc.edu
#synchronized child 'obj' pid 17760 host fr6n14.mhpcc.edu
#synchronized child 'obj' pid 23390 host fr17n12.mhpcc.edu
#synchronized child 'obj' pid 14598 host fr10n02.mhpcc.edu
#synchronized child 'obj' pid 17338 host fr5n15.mhpcc.edu
#synchronized child 'obj' pid 11900 host fr5n05.mhpcc.edu
#synchronized child 'ui' pid 20378 host fr5n14.mhpcc.edu
#synchronized child 'mas' pid 23190 host fr5n14.mhpcc.edu
#Completed initial synchronization.
#Pausing for user input ...
#CHILDasynchronous
RUN FILE 'Cmd/AdaptSeriesSoda.cmd';
real SCALE;
real XRES, YRES;
SCALE=1;
XRES=(640*SCALE);
YRES=(480*SCALE);
synchronize;
#requesting global synchronization
#global synchronization achieved
time;
#time=855119347
compile mgf 'Models/soda.mgf';
synchronize;
#requesting global synchronization
#Read MGF 'Models/soda.mgf'
#loading MGF file 'Models/soda.mgf'
#loaded 22725 objects from 'Models/soda.mgf'
#274 spheres
#22451 polygons
#5 lights
#global synchronization achieved
time;
#time=855119892
BSP_MAX_LIST_LENGTH=64;
make bsp;
synchronize;
#requesting global synchronization
#built Bsp tree of 28579 nodes
#global synchronization achieved
time;
#time=855119922
define camera 0, (2.54, 1.52, -1.9), ((2.54-0.440225),
(1.52-0.17609),
(-1.9+0.880451)), (0, 1, 0), 1, 1, 0.75, XRES, YRES, 24;
#lens has resolution 640 by 480
load schedule 0 'soda.schedule';
synchronize;
#requesting global synchronization
#Distributing schedule for image 0 to 32 objects
#global synchronization achieved
RENDERING_MODE=3;
PATH_GENERATIONS=99;
CHECKPOINT_INTERVAL=10000;
SCATTER=1;
BALANCE_WORKLOAD=1;
printlist 'diffusion and scattering';
diffusion and scattering
synchronize;
#requesting global synchronization
#global synchronization achieved
time;
#time=855119976
render image 0;
# in finishRender, renderHold=0 renderActive=1
synchronize;
#requesting global synchronization
#there are 32 active objects
#object 21 getting work after 30 min 15 sec
#object 21 balances with 4 peers to new workload 11936
#object 9 getting work after 31 min 22 sec
#object 9 balances with 4 peers to new workload 10598
#object 25 getting work after 31 min 44 sec
#object 25 balances with 4 peers to new workload 12888
#object 13 getting work after 32 min 0 sec
#object 13 balances with 4 peers to new workload 11960
#object 17 getting work after 32 min 11 sec
#object 17 balances with 4 peers to new workload 14104
#object 5 getting work after 32 min 28 sec
#object 5 balances with 4 peers to new workload 13125
#object 20 getting work after 32 min 28 sec
#object 20 balances with 3 peers to new workload 8781
#object 29 getting work after 33 min 31 sec
#object 29 balances with 3 peers to new workload 11666
#object 8 getting work after 33 min 37 sec
#object 8 balances with 3 peers to new workload 4917
#object 24 getting work after 34 min 5 sec
#object 24 balances with 3 peers to new workload 4701
#object 12 getting work after 34 min 18 sec
#object 12 balances with 2 peers to new workload 1932
#object 16 getting work after 34 min 25 sec
#object 16 balances with 3 peers to new workload 3554
#object 12 getting work after 34 min 42 sec
#object 12 balances with 3 peers to new workload 1897
#object 28 getting work after 34 min 50 sec
#object 28 balances with 2 peers to new workload 4253
#object 4 getting work after 35 min 2 sec
#object 4 balances with 3 peers to new workload 6852
#object 1 getting work after 35 min 7 sec
#object 1 balances with 3 peers to new workload 10169
#object 8 getting work after 35 min 17 sec
#object 8 balances with 3 peers to new workload 3767
#object 20 getting work after 35 min 37 sec
#object 20 balances with 3 peers to new workload 1052
#object 28 getting work after 35 min 40 sec
#object 28 balances with 2 peers to new workload 1553
#object 24 getting work after 35 min 47 sec
#object 24 balances with 3 peers to new workload 2311
#object 25 getting work after 35 min 51 sec
#object 25 balances with 4 peers to new workload 2060
#object 13 getting work after 35 min 52 sec
#object 13 balances with 4 peers to new workload 3062
#object 22 getting work after 35 min 53 sec
#object 22 balances with 3 peers to new workload 2871
#object 18 getting work after 36 min 1 sec
#object 18 balances with 3 peers to new workload 5685
#object 10 getting work after 36 min 3 sec
#object 10 balances with 3 peers to new workload 6088
#object 20 getting work after 36 min 17 sec
#object 20 balances with 3 peers to new workload 1127
#object 24 getting work after 36 min 19 sec
#object 24 balances with 3 peers to new workload 1424
#object 0 getting work after 36 min 24 sec
#object 0 balances with 1 peers to new workload 6434
#object 22 getting work after 36 min 30 sec
#object 22 balances with 2 peers to new workload 3289
#object 24 getting work after 36 min 31 sec
#object 24 balances with 3 peers to new workload 979
#object 4 getting work after 36 min 34 sec
#object 4 balances with 3 peers to new workload 2929
#object 24 getting work after 36 min 36 sec
#object 24 balances with 3 peers to new workload 906
```

```
#object 20 getting work after 36 min 38 sec
#object 20 balances with 2 peers to new workload 353
#object 20 getting work after 36 min 39 sec
#object 20 balances with 2 peers to new workload 1049
#object 26 getting work after 36 min 41 sec
#object 26 balances with 2 peers to new workload 7665
#object 20 getting work after 36 min 45 sec
#object 20 balances with 2 peers to new workload 1479
#object 25 getting work after 36 min 46 sec
#object 25 balances with 4 peers to new workload 1348
#object 24 getting work after 36 min 49 sec
#object 24 balances with 3 peers to new workload 1544
#object 23 getting work after 36 min 53 sec
#object 23 balances with 2 peers to new workload 3267
#object 22 getting work after 36 min 59 sec
#object 22 balances with 4 peers to new workload 934
#object 9 getting work after 37 min 4 sec
#object 9 balances with 4 peers to new workload 761
#object 22 getting work after 37 min 5 sec
#object 24 getting work after 37 min 5 sec
#object 22 balances with 4 peers to new workload 491
#object 24 balances with 3 peers to new workload 1042
#object 22 getting work after 37 min 7 sec
#object 22 balances with 4 peers to new workload 537
#object 22 getting work after 37 min 12 sec
#object 22 balances with 3 peers to new workload 252
#object 20 getting work after 37 min 12 sec
#object 20 balances with 1 peers to new workload 677
#object 22 getting work after 37 min 14 sec
#object 22 balances with 2 peers to new workload 790
#object 20 getting work after 37 min 14 sec
#object 20 balances with 1 peers to new workload 103
#object 24 getting work after 37 min 14 sec
#object 24 balances with 2 peers to new workload 412
#object 20 getting work after 37 min 15 sec
#master hears 'out of work' from pid 21302, activeObjs now
31
#object 28 getting work after 37 min 15 sec
#object 29 getting work after 37 min 16 sec
#object 29 balances with 2 peers to new workload 2222
#object 13 getting work after 37 min 17 sec
#object 13 balances with 3 peers to new workload 2059
#object 9 getting work after 37 min 18 sec
#object 9 balances with 2 peers to new workload 840
#master hears 'out of work' from pid 37896, activeObjs now
30
#object 28 balances with 1 peers to new workload 1224
#object 20 balances with 3 peers to new workload 320
#master hears 'got work' from pid 37896, activeObjs now 31
#object 22 getting work after 37 min 19 sec
#object 22 balances with 1 peers to new workload 140
#master hears 'got work' from pid 21302, activeObjs now 32
#object 22 getting work after 37 min 19 sec
#object 22 balances with 1 peers to new workload 92
#object 8 getting work after 37 min 20 sec
#object 8 balances with 1 peers to new workload 294
#object 22 getting work after 37 min 20 sec
#object 22 balances with 1 peers to new workload 537
#object 24 getting work after 37 min 20 sec
#object 24 balances with 1 peers to new workload 762
#object 9 getting work after 37 min 20 sec
#object 25 getting work after 37 min 20 sec
#object 25 balances with 2 peers to new workload 468
#object 9 balances with 1 peers to new workload 83
#object 22 getting work after 37 min 20 sec
#object 20 getting work after 37 min 20 sec
#object 9 getting work after 37 min 20 sec
#object 20 balances with 1 peers to new workload 442
#master hears 'out of work' from pid 20214, activeObjs now
31
#object 9 balances with 2 peers to new workload 317
#object 22 balances with 1 peers to new workload 205
#object 22 getting work after 37 min 21 sec
#object 22 balances with 1 peers to new workload 36
#object 22 getting work after 37 min 21 sec
#object 9 getting work after 37 min 22 sec
#object 9 balances with 2 peers to new workload 232
#object 22 balances with 1 peers to new workload 18
#object 22 getting work after 37 min 22 sec
#object 10 getting work after 37 min 22 sec
#object 10 balances with 3 peers to new workload 653
#object 9 getting work after 37 min 22 sec
#object 22 balances with 1 peers to new workload 201
#object 9 balances with 1 peers to new workload 215
#object 22 getting work after 37 min 22 sec
#object 22 balances with 1 peers to new workload 300
#object 8 getting work after 37 min 22 sec
#object 8 balances with 1 peers to new workload 495
#object 20 getting work after 37 min 23 sec
#object 20 balances with 2 peers to new workload 238
#object 9 getting work after 37 min 23 sec
#object 9 balances with 1 peers to new workload 220
#object 22 getting work after 37 min 23 sec
#object 9 getting work after 37 min 23 sec
#object 9 balances with 1 peers to new workload 105
#object 20 getting work after 37 min 23 sec
#object 20 balances with 1 peers to new workload 5
#object 20 getting work after 37 min 23 sec
#object 22 balances with 1 peers to new workload 24
#master hears 'out of work' from pid 21302, activeObjs now
30
#object 22 getting work after 37 min 23 sec
#object 9 getting work after 37 min 24 sec
#object 9 balances with 1 peers to new workload 331
#object 20 balances with 1 peers to new workload 208
#object 22 balances with 1 peers to new workload 7
#object 22 getting work after 37 min 24 sec
#object 8 getting work after 37 min 24 sec
#object 22 balances with 1 peers to new workload 144
#master hears 'out of work' from pid 17380, activeObjs now
29
#object 12 getting work after 37 min 24 sec
#master hears 'out of work' from pid 18934, activeObjs now
28
#object 28 getting work after 37 min 24 sec
#object 28 balances with 1 peers to new workload 242
#object 9 getting work after 37 min 24 sec
#object 9 balances with 1 peers to new workload 157
#object 28 getting work after 37 min 24 sec
#object 12 balances with 1 peers to new workload 131
#object 28 balances with 1 peers to new workload 690
#master hears 'got work' from pid 18934, activeObjs now 29
#object 8 balances with 1 peers to new workload 56
#object 12 getting work after 37 min 24 sec
#object 22 getting work after 37 min 24 sec
#object 12 balances with 1 peers to new workload 43
#object 22 balances with 1 peers to new workload 12
#object 22 getting work after 37 min 24 sec
#object 22 balances with 1 peers to new workload 2
#object 22 getting work after 37 min 25 sec
#object 20 getting work after 37 min 25 sec
#object 22 balances with 1 peers to new workload 229
#object 10 getting work after 37 min 25 sec
#object 8 getting work after 37 min 25 sec
#object 10 balances with 3 peers to new workload 696
#object 8 balances with 1 peers to new workload 180
#object 9 getting work after 37 min 25 sec
#object 9 balances with 1 peers to new workload 63
#object 12 getting work after 37 min 25 sec
#object 22 getting work after 37 min 25 sec
#object 8 getting work after 37 min 25 sec
#object 20 balances with 1 peers to new workload 260
#master hears 'out of work' from pid 18934, activeObjs now
28
#object 9 getting work after 37 min 25 sec
#object 20 getting work after 37 min 25 sec
#master hears 'out of work' from pid 24280, activeObjs now
27
#object 28 getting work after 37 min 25 sec
#object 28 balances with 1 peers to new workload 17
#object 20 balances with 1 peers to new workload 162
#object 20 getting work after 37 min 25 sec
#object 12 balances with 1 peers to new workload 78
#object 28 getting work after 37 min 26 sec
#object 28 balances with 1 peers to new workload 110
#object 9 balances with 1 peers to new workload 15
#object 9 getting work after 37 min 26 sec
#master hears 'got work' from pid 18934, activeObjs now 28
#object 8 balances with 1 peers to new workload 33
#object 12 getting work after 37 min 26 sec
#object 20 balances with 1 peers to new workload 73
#master hears 'out of work' from pid 18934, activeObjs now
27
#object 28 getting work after 37 min 26 sec
#object 28 balances with 1 peers to new workload 336
#object 20 getting work after 37 min 26 sec
#object 22 balances with 1 peers to new workload 38
#object 20 balances with 1 peers to new workload 168
#object 10 getting work after 37 min 26 sec
#object 10 balances with 3 peers to new workload 700
#object 28 getting work after 37 min 27 sec
#object 28 balances with 1 peers to new workload 376
#object 28 getting work after 37 min 27 sec
#object 28 balances with 1 peers to new workload 541
#object 10 getting work after 37 min 27 sec
#object 10 balances with 3 peers to new workload 490
#object 28 getting work after 37 min 28 sec
#master hears 'out of work' from pid 37896, activeObjs now
26
#object 28 balances with 1 peers to new workload 174
#master hears 'got work' from pid 37896, activeObjs now 27
#object 28 getting work after 37 min 28 sec
#object 28 balances with 2 peers to new workload 250
#object 28 getting work after 37 min 29 sec
#object 28 balances with 1 peers to new workload 249
#object 25 getting work after 37 min 29 sec
#object 25 balances with 2 peers to new workload 401
#object 20 getting work after 37 min 29 sec
#object 20 balances with 1 peers to new workload 180
#object 28 getting work after 37 min 29 sec
#object 28 balances with 1 peers to new workload 142
#object 28 getting work after 37 min 29 sec
#master hears 'out of work' from pid 37896, activeObjs now
26
#object 10 getting work after 37 min 29 sec
#object 10 balances with 2 peers to new workload 445
#object 20 getting work after 37 min 29 sec
#object 28 balances with 1 peers to new workload 267
#master hears 'got work' from pid 37896, activeObjs now 27
#object 10 getting work after 37 min 30 sec
#object 10 balances with 2 peers to new workload 1733
#object 28 getting work after 37 min 30 sec
#object 28 balances with 1 peers to new workload 332
#object 8 getting work after 37 min 30 sec
#object 24 getting work after 37 min 30 sec
#object 24 balances with 1 peers to new workload 86
#object 25 getting work after 37 min 30 sec
#master hears 'out of work' from pid 11900, activeObjs now
26
#object 28 getting work after 37 min 30 sec
#master hears 'out of work' from pid 37896, activeObjs now
25
```

```
#object 18 getting work after 37 min 31 sec
#object 9 balances with 1 peers to new workload 1
#object 9 getting work after 37 min 31 sec
#object 18 balances with 3 peers to new workload 774
#object 9 balances with 1 peers to new workload 20
#object 22 getting work after 37 min 31 sec
#object 22 balances with 2 peers to new workload 32
#object 9 getting work after 37 min 32 sec
#object 9 balances with 1 peers to new workload 63
#object 9 getting work after 37 min 32 sec
#object 9 balances with 1 peers to new workload 1091
#object 12 balances with 1 peers to new workload 74
#master hears 'got work' from pid 18934, activeObjs now 26
#object 8 balances with 1 peers to new workload 28
#object 12 getting work after 37 min 32 sec
#master hears 'out of work' from pid 18934, activeObjs now
25
#object 13 getting work after 37 min 33 sec
#object 22 getting work after 37 min 33 sec
#object 5 getting work after 37 min 33 sec
#object 13 balances with 1 peers to new workload 663
#object 5 balances with 2 peers to new workload 516
#object 8 getting work after 37 min 33 sec
#object 10 getting work after 37 min 34 sec
#object 10 balances with 1 peers to new workload 202
#object 24 getting work after 37 min 34 sec
#master hears 'out of work' from pid 18632, activeObjs now
24
#object 13 getting work after 37 min 34 sec
#object 10 getting work after 37 min 34 sec
#object 28 balances with 1 peers to new workload 238
#master hears 'out of work' from pid 21846, activeObjs now
23
#master hears 'got work' from pid 37896, activeObjs now 24
#object 24 balances with 1 peers to new workload 87
#object 13 balances with 1 peers to new workload 978
#object 28 getting work after 37 min 34 sec
#object 12 balances with 1 peers to new workload 72
#object 10 balances with 2 peers to new workload 106
#master hears 'got work' from pid 18934, activeObjs now 25
#object 8 balances with 1 peers to new workload 33
#object 12 getting work after 37 min 34 sec
#object 10 getting work after 37 min 34 sec
#object 10 balances with 1 peers to new workload 270
#object 26 getting work after 37 min 35 sec
#object 10 getting work after 37 min 35 sec
#object 10 balances with 1 peers to new workload 101
#object 26 balances with 2 peers to new workload 665
#object 22 balances with 1 peers to new workload 88
#object 10 getting work after 37 min 35 sec
#object 10 balances with 1 peers to new workload 145
#object 22 getting work after 37 min 35 sec
#object 22 balances with 1 peers to new workload 35
#object 10 getting work after 37 min 36 sec
#object 10 balances with 2 peers to new workload 22
#object 22 getting work after 37 min 36 sec
#object 10 getting work after 37 min 36 sec
#object 9 getting work after 37 min 36 sec
#object 10 balances with 1 peers to new workload 7
#object 10 getting work after 37 min 36 sec
#object 10 balances with 1 peers to new workload 7
#object 10 getting work after 37 min 36 sec
#object 22 balances with 1 peers to new workload 300
#object 22 getting work after 37 min 37 sec
#object 22 balances with 1 peers to new workload 216
#object 13 getting work after 37 min 37 sec
#object 10 balances with 1 peers to new workload 142
#object 13 balances with 1 peers to new workload 599
#object 10 getting work after 37 min 37 sec
#object 16 getting work after 37 min 37 sec
#object 16 balances with 1 peers to new workload 856
#object 22 getting work after 37 min 37 sec
#object 22 balances with 1 peers to new workload 80
#object 28 balances with 1 peers to new workload 27
#object 22 getting work after 37 min 37 sec
#object 10 balances with 1 peers to new workload 32
#object 28 getting work after 37 min 38 sec
#master hears 'out of work' from pid 37896, activeObjs now
24
#object 24 getting work after 37 min 38 sec
#master hears 'out of work' from pid 18934, activeObjs now
23
#object 10 getting work after 37 min 38 sec
#object 10 balances with 1 peers to new workload 51
#object 18 getting work after 37 min 38 sec
#object 18 balances with 1 peers to new workload 320
#object 10 getting work after 37 min 38 sec
#object 21 getting work after 37 min 38 sec
#object 28 balances with 1 peers to new workload 24
#object 21 balances with 1 peers to new workload 622
#object 25 balances with 1 peers to new workload 66
#object 24 balances with 1 peers to new workload 77
#object 13 getting work after 37 min 38 sec
#object 8 getting work after 37 min 38 sec
#object 13 balances with 1 peers to new workload 231
#object 28 getting work after 37 min 38 sec
#object 24 getting work after 37 min 38 sec
#object 9 balances with 1 peers to new workload 17
#object 16 getting work after 37 min 39 sec
#object 16 balances with 1 peers to new workload 401
#object 25 getting work after 37 min 39 sec
#object 10 balances with 1 peers to new workload 37
#object 13 getting work after 37 min 39 sec
#object 13 balances with 1 peers to new workload 561
#object 21 getting work after 37 min 39 sec
#object 21 balances with 1 peers to new workload 393
#object 10 getting work after 37 min 39 sec
#object 13 getting work after 37 min 39 sec
#object 13 balances with 1 peers to new workload 380
#object 21 getting work after 37 min 39 sec
#object 21 balances with 1 peers to new workload 146
#object 10 balances with 1 peers to new workload 2
#object 10 getting work after 37 min 40 sec
#object 16 getting work after 37 min 40 sec
#object 16 balances with 1 peers to new workload 18
#object 29 getting work after 37 min 40 sec
#object 29 balances with 1 peers to new workload 52
#object 13 getting work after 37 min 40 sec
#object 13 balances with 1 peers to new workload 145
#object 21 getting work after 37 min 40 sec
#object 21 balances with 1 peers to new workload 52
#object 29 getting work after 37 min 40 sec
#object 5 getting work after 37 min 40 sec
#object 29 balances with 1 peers to new workload 139
#object 5 balances with 1 peers to new workload 828
#object 9 getting work after 37 min 40 sec
#object 29 getting work after 37 min 40 sec
#object 13 getting work after 37 min 40 sec
#object 13 balances with 1 peers to new workload 202
#master hears 'out of work' from pid 17760, activeObjs now
22
#object 16 getting work after 37 min 40 sec
#object 16 balances with 1 peers to new workload 49
#object 29 balances with 1 peers to new workload 213
#object 21 getting work after 37 min 41 sec
#object 21 balances with 1 peers to new workload 324
#object 16 getting work after 37 min 41 sec
#object 16 balances with 1 peers to new workload 142
#object 13 getting work after 37 min 41 sec
#object 16 getting work after 37 min 41 sec
#master hears 'out of work' from pid 17948, activeObjs now
21
#master hears 'out of work' from pid 16214, activeObjs now
20
#object 29 getting work after 37 min 41 sec
#object 29 balances with 1 peers to new workload 167
#object 13 balances with 1 peers to new workload 205
#object 10 balances with 1 peers to new workload 152
#object 16 balances with 1 peers to new workload 213
#object 16 getting work after 37 min 41 sec
#object 16 balances with 1 peers to new workload 90
#object 13 getting work after 37 min 42 sec
#object 29 getting work after 37 min 42 sec
#object 29 balances with 1 peers to new workload 348
#object 13 balances with 1 peers to new workload 153
#object 10 getting work after 37 min 42 sec
#object 29 getting work after 37 min 42 sec
#object 21 getting work after 37 min 42 sec
#object 29 balances with 1 peers to new workload 148
#master hears 'out of work' from pid 17338, activeObjs now
19
#object 18 getting work after 37 min 42 sec
#master hears 'out of work' from pid 19198, activeObjs now
18
#object 21 balances with 1 peers to new workload 205
#object 18 balances with 1 peers to new workload 91
#object 29 getting work after 37 min 42 sec
#object 13 getting work after 37 min 42 sec
#object 29 balances with 1 peers to new workload 311
#object 16 getting work after 37 min 42 sec
#object 21 getting work after 37 min 43 sec
#object 21 balances with 1 peers to new workload 72
#object 13 balances with 1 peers to new workload 19
#object 18 getting work after 37 min 43 sec
#object 13 getting work after 37 min 43 sec
#object 9 balances with 1 peers to new workload 28
#object 29 getting work after 37 min 43 sec
#object 13 balances with 1 peers to new workload 43
#object 9 getting work after 37 min 43 sec
#object 18 balances with 1 peers to new workload 11
#object 9 balances with 1 peers to new workload 203
#object 18 getting work after 37 min 43 sec
#object 29 balances with 1 peers to new workload 142
#object 13 getting work after 37 min 43 sec
#object 29 getting work after 37 min 43 sec
#object 21 getting work after 37 min 44 sec
#object 29 balances with 1 peers to new workload 84
#object 9 getting work after 37 min 44 sec
#object 21 balances with 1 peers to new workload 252
#object 16 balances with 1 peers to new workload 154
#object 13 balances with 1 peers to new workload 35
#object 23 getting work after 37 min 44 sec
#master hears 'out of work' from pid 37306, activeObjs now
17
#object 29 getting work after 37 min 44 sec
#object 21 getting work after 37 min 44 sec
#object 29 balances with 1 peers to new workload 298
#object 16 getting work after 37 min 44 sec
#object 13 getting work after 37 min 45 sec
#object 23 balances with 1 peers to new workload 80
#object 23 getting work after 37 min 45 sec
#object 29 getting work after 37 min 45 sec
#object 10 balances with 1 peers to new workload 11
#object 10 getting work after 37 min 46 sec
#object 29 balances with 1 peers to new workload 43
#object 29 getting work after 37 min 46 sec
#object 29 balances with 1 peers to new workload 131
#object 13 balances with 1 peers to new workload 115
#object 21 balances with 1 peers to new workload 85
#object 29 getting work after 37 min 46 sec
#object 16 balances with 1 peers to new workload 29
#object 16 getting work after 37 min 47 sec
```

```
#object 13 getting work after 37 min 47 sec
#object 9 balances with 1 peers to new workload 227
#object 21 getting work after 37 min 47 sec
#object 18 balances with 1 peers to new workload 4
#object 18 getting work after 37 min 47 sec
#object 29 balances with 1 peers to new workload 106
#object 9 getting work after 37 min 48 sec
#object 29 getting work after 37 min 48 sec
#object 8 balances with 1 peers to new workload 146
#object 5 getting work after 37 min 49 sec
#object 5 balances with 2 peers to new workload 1362
#object 29 balances with 1 peers to new workload 71
#object 8 getting work after 37 min 50 sec
#object 8 balances with 1 peers to new workload 82
#object 8 getting work after 37 min 50 sec
#object 29 getting work after 37 min 50 sec
#object 5 getting work after 37 min 50 sec
#object 5 balances with 1 peers to new workload 486
#object 23 balances with 1 peers to new workload 106
#object 23 getting work after 37 min 51 sec
#object 4 getting work after 37 min 52 sec
#object 5 getting work after 37 min 52 sec
#object 5 balances with 1 peers to new workload 397
#object 4 balances with 1 peers to new workload 1006
#object 26 getting work after 37 min 52 sec
#master hears 'out of work' from pid 16224, activeObjs now
16
#object 17 getting work after 37 min 52 sec
#master hears 'out of work' from pid 17214, activeObjs now
15
#object 19 getting work after 37 min 53 sec
#object 5 getting work after 37 min 54 sec
#object 5 balances with 1 peers to new workload 476
#object 19 balances with 1 peers to new workload 2555
#object 4 getting work after 37 min 54 sec
#object 4 balances with 1 peers to new workload 540
#object 23 balances with 1 peers to new workload 43
#object 8 balances with 1 peers to new workload 44
#object 23 getting work after 37 min 55 sec
#object 18 balances with 1 peers to new workload 38
#object 8 getting work after 37 min 55 sec
#object 5 getting work after 37 min 56 sec
#object 5 balances with 1 peers to new workload 392
#object 18 getting work after 37 min 56 sec
#object 23 balances with 1 peers to new workload 22
#object 23 getting work after 37 min 57 sec
#object 5 getting work after 37 min 58 sec
#object 5 balances with 1 peers to new workload 288
#object 5 getting work after 37 min 58 sec
#object 5 balances with 1 peers to new workload 383
#object 5 getting work after 38 min 0 sec
#object 5 balances with 1 peers to new workload 0
#object 5 getting work after 38 min 0 sec
#master hears 'out of work' from pid 22526, activeObjs now
14
#object 5 balances with 1 peers to new workload 173
#object 5 getting work after 38 min 1 sec
#object 5 balances with 1 peers to new workload 388
#object 5 getting work after 38 min 2 sec
#object 5 balances with 1 peers to new workload 177
#object 5 getting work after 38 min 3 sec
#object 5 balances with 1 peers to new workload 148
#object 5 getting work after 38 min 4 sec
#object 5 balances with 1 peers to new workload 246
#object 5 getting work after 38 min 6 sec
#object 5 balances with 2 peers to new workload 785
#object 8 balances with 1 peers to new workload 752
#object 8 getting work after 38 min 7 sec
#object 9 balances with 1 peers to new workload 94
#object 27 getting work after 38 min 8 sec
#object 27 balances with 1 peers to new workload 3123
#object 11 getting work after 38 min 9 sec
#object 11 balances with 2 peers to new workload 3385
#object 10 balances with 1 peers to new workload 32
#object 10 getting work after 38 min 9 sec
#object 10 balances with 1 peers to new workload 307
#object 6 getting work after 38 min 9 sec
#object 6 balances with 2 peers to new workload 3857
#object 4 getting work after 38 min 10 sec
#object 4 balances with 1 peers to new workload 1199
#object 1 getting work after 38 min 11 sec
#object 1 balances with 3 peers to new workload 1651
#object 4 getting work after 38 min 11 sec
#object 4 balances with 2 peers to new workload 623
#object 9 getting work after 38 min 11 sec
#object 9 balances with 1 peers to new workload 258
#object 9 getting work after 38 min 12 sec
#object 9 balances with 1 peers to new workload 58
#object 9 getting work after 38 min 12 sec
#object 10 getting work after 38 min 12 sec
#object 10 balances with 1 peers to new workload 77
#object 4 getting work after 38 min 12 sec
#object 4 balances with 1 peers to new workload 1216
#object 10 getting work after 38 min 13 sec
#object 10 balances with 1 peers to new workload 286
#object 10 getting work after 38 min 14 sec
#object 5 getting work after 38 min 14 sec
#object 10 balances with 1 peers to new workload 161
#object 5 balances with 1 peers to new workload 1395
#object 10 getting work after 38 min 14 sec
#object 10 balances with 1 peers to new workload 179
#object 10 getting work after 38 min 15 sec
#object 10 balances with 1 peers to new workload 82
#object 10 getting work after 38 min 15 sec
#object 10 balances with 2 peers to new workload 293
#object 4 getting work after 38 min 15 sec
#object 4 balances with 1 peers to new workload 445
#object 10 getting work after 38 min 15 sec
#object 10 balances with 1 peers to new workload 40
#object 10 getting work after 38 min 15 sec
#object 10 balances with 1 peers to new workload 64
#object 10 getting work after 38 min 16 sec
#object 10 balances with 1 peers to new workload 145
#object 10 getting work after 38 min 16 sec
#object 10 balances with 1 peers to new workload 144
#object 10 getting work after 38 min 17 sec
#object 10 balances with 1 peers to new workload 38
#object 9 balances with 1 peers to new workload 2
#object 9 getting work after 38 min 17 sec
#object 4 getting work after 38 min 17 sec
#object 4 balances with 1 peers to new workload 1046
#object 10 getting work after 38 min 18 sec
#object 19 getting work after 38 min 18 sec
#object 19 balances with 1 peers to new workload 675
#object 10 balances with 1 peers to new workload 24
#object 10 getting work after 38 min 19 sec
#object 10 balances with 1 peers to new workload 139
#object 10 getting work after 38 min 19 sec
#object 10 balances with 1 peers to new workload 124
#object 9 balances with 1 peers to new workload 180
#object 4 getting work after 38 min 19 sec
#object 4 balances with 2 peers to new workload 1300
#object 10 getting work after 38 min 19 sec
#object 10 balances with 1 peers to new workload 27
#object 10 getting work after 38 min 21 sec
#object 5 getting work after 38 min 21 sec
#object 5 balances with 1 peers to new workload 648
#object 10 balances with 1 peers to new workload 85
#object 10 getting work after 38 min 22 sec
#object 0 getting work after 38 min 22 sec
#object 10 balances with 1 peers to new workload 56
#object 0 balances with 1 peers to new workload 291
#object 0 getting work after 38 min 23 sec
#object 0 balances with 1 peers to new workload 461
#object 8 balances with 1 peers to new workload 202
#object 9 getting work after 38 min 23 sec
#object 19 getting work after 38 min 24 sec
#object 9 balances with 1 peers to new workload 635
#object 8 getting work after 38 min 24 sec
#object 8 balances with 1 peers to new workload 1270
#object 19 balances with 1 peers to new workload 303
#master hears 'got work' from pid 17380, activeObjs now 15
#object 9 getting work after 38 min 24 sec
#object 9 balances with 2 peers to new workload 41
#object 19 getting work after 38 min 24 sec
#object 19 balances with 1 peers to new workload 449
#object 10 getting work after 38 min 24 sec
#object 9 getting work after 38 min 25 sec
#object 0 getting work after 38 min 25 sec
#object 19 getting work after 38 min 25 sec
#object 12 balances with 1 peers to new workload 46
#master hears 'got work' from pid 18934, activeObjs now 16
#object 9 balances with 1 peers to new workload 349
#object 13 balances with 1 peers to new workload 130
#object 19 balances with 1 peers to new workload 95
#object 16 balances with 1 peers to new workload 50
#object 13 getting work after 38 min 25 sec
#object 12 getting work after 38 min 25 sec
#object 19 getting work after 38 min 25 sec
#object 12 balances with 1 peers to new workload 24
#object 16 getting work after 38 min 25 sec
#object 27 getting work after 38 min 25 sec
#object 27 balances with 1 peers to new workload 1071
#object 12 getting work after 38 min 25 sec
#master hears 'out of work' from pid 18934, activeObjs now
15
#object 13 balances with 1 peers to new workload 12
#object 19 balances with 1 peers to new workload 88
#object 13 getting work after 38 min 25 sec
#object 19 getting work after 38 min 25 sec
#object 19 balances with 1 peers to new workload 73
#object 19 getting work after 38 min 26 sec
#object 19 balances with 1 peers to new workload 20
#object 19 getting work after 38 min 26 sec
#master hears 'out of work' from pid 17556, activeObjs now
14
#object 19 balances with 1 peers to new workload 146
#object 15 getting work after 38 min 27 sec
#master hears 'out of work' from pid 2628, activeObjs now
13
#object 19 getting work after 38 min 27 sec
#object 5 getting work after 38 min 28 sec
#object 5 balances with 2 peers to new workload 294
#object 5 getting work after 38 min 29 sec
#object 5 balances with 1 peers to new workload 157
#object 0 balances with 2 peers to new workload 546
#object 5 getting work after 38 min 29 sec
#object 5 balances with 1 peers to new workload 238
#object 0 getting work after 38 min 30 sec
#object 0 balances with 1 peers to new workload 97
#object 5 getting work after 38 min 30 sec
#object 5 balances with 1 peers to new workload 317
#object 0 getting work after 38 min 30 sec
#object 0 balances with 1 peers to new workload 98
#object 4 getting work after 38 min 31 sec
#master hears 'out of work' from pid 22244, activeObjs now
12
#object 0 getting work after 38 min 31 sec
#object 0 balances with 1 peers to new workload 108
#object 5 getting work after 38 min 31 sec
#object 0 getting work after 38 min 31 sec
#master hears 'out of work' from pid 21786, activeObjs now
```

```
11
#object 5 balances with 1 peers to new workload 214
#object 8 getting work after 38 min 32 sec
#master hears 'out of work' from pid 17380, activeObjs now
10
#object 0 balances with 1 peers to new workload 203
#object 0 getting work after 38 min 32 sec
#object 0 balances with 1 peers to new workload 131
#object 0 getting work after 38 min 32 sec
#object 5 getting work after 38 min 32 sec
#object 0 balances with 1 peers to new workload 79
#object 5 balances with 1 peers to new workload 11
#object 5 getting work after 38 min 32 sec
#object 5 balances with 1 peers to new workload 67
#object 0 getting work after 38 min 33 sec
#object 0 balances with 1 peers to new workload 17
#object 9 getting work after 38 min 33 sec
#object 5 getting work after 38 min 33 sec
#object 5 balances with 1 peers to new workload 4
#object 0 getting work after 38 min 33 sec
#object 5 getting work after 38 min 33 sec
#object 30 getting work after 38 min 34 sec
#object 30 balances with 1 peers to new workload 138
#object 1 getting work after 38 min 34 sec
#object 1 balances with 1 peers to new workload 239
#object 1 getting work after 38 min 34 sec
#object 1 balances with 1 peers to new workload 590
#object 1 getting work after 38 min 35 sec
#object 1 balances with 1 peers to new workload 34
#object 30 getting work after 38 min 36 sec
#object 30 balances with 1 peers to new workload 43
#object 14 getting work after 38 min 36 sec
#object 1 getting work after 38 min 36 sec
#master hears 'out of work' from pid 21288, activeObjs now
9
#object 1 balances with 1 peers to new workload 257
#object 30 getting work after 38 min 36 sec
#object 30 balances with 1 peers to new workload 530
#object 1 getting work after 38 min 36 sec
#object 1 balances with 1 peers to new workload 143
#object 1 getting work after 38 min 37 sec
#object 1 balances with 1 peers to new workload 295
#object 27 getting work after 38 min 37 sec
#object 27 balances with 1 peers to new workload 676
#object 30 getting work after 38 min 37 sec
#object 30 balances with 1 peers to new workload 208
#object 1 getting work after 38 min 37 sec
#object 1 balances with 1 peers to new workload 132
#object 30 getting work after 38 min 37 sec
#object 30 balances with 1 peers to new workload 6
#object 30 getting work after 38 min 37 sec
#master hears 'out of work' from pid 23390, activeObjs now
8
#object 27 getting work after 38 min 37 sec
#master hears 'out of work' from pid 18740, activeObjs now
7
#object 27 balances with 1 peers to new workload 214
#object 1 getting work after 38 min 38 sec
#object 1 balances with 1 peers to new workload 314
#object 27 getting work after 38 min 38 sec
#object 1 getting work after 38 min 38 sec
#master hears 'out of work' from pid 21652, activeObjs now
6
#object 27 balances with 1 peers to new workload 288
#object 1 balances with 1 peers to new workload 184
#object 30 balances with 1 peers to new workload 331
#object 27 getting work after 38 min 38 sec
#object 27 balances with 1 peers to new workload 25
#object 27 getting work after 38 min 38 sec
#object 1 getting work after 38 min 38 sec
#object 1 balances with 1 peers to new workload 146
#object 30 getting work after 38 min 39 sec
#object 30 balances with 1 peers to new workload 90
#object 27 balances with 1 peers to new workload 109
#object 1 getting work after 38 min 39 sec
#object 27 getting work after 38 min 39 sec
#object 30 getting work after 38 min 39 sec
#object 30 balances with 1 peers to new workload 62
#object 27 balances with 1 peers to new workload 24
#object 30 getting work after 38 min 39 sec
#object 1 balances with 1 peers to new workload 305
#object 27 getting work after 38 min 39 sec
#object 1 getting work after 38 min 39 sec
#object 1 balances with 1 peers to new workload 135
#object 30 balances with 1 peers to new workload 50
#object 27 balances with 1 peers to new workload 38
#object 27 getting work after 38 min 40 sec
#object 1 getting work after 38 min 40 sec
#object 1 balances with 1 peers to new workload 71
#object 27 balances with 1 peers to new workload 49
#object 1 getting work after 38 min 40 sec
#object 30 getting work after 38 min 40 sec
#object 27 getting work after 38 min 40 sec
#object 30 balances with 1 peers to new workload 21
#object 1 balances with 1 peers to new workload 77
#object 30 getting work after 38 min 40 sec
#object 27 balances with 1 peers to new workload 52
#object 1 getting work after 38 min 41 sec
#object 27 getting work after 38 min 41 sec
#object 1 balances with 1 peers to new workload 140
#object 1 getting work after 38 min 41 sec
#object 27 balances with 1 peers to new workload 57
#object 1 balances with 1 peers to new workload 140
#object 30 balances with 1 peers to new workload 58
#object 27 getting work after 38 min 42 sec
#object 1 getting work after 38 min 42 sec
#object 1 balances with 1 peers to new workload 218
#object 30 getting work after 38 min 42 sec
#object 1 getting work after 38 min 42 sec
#object 1 balances with 1 peers to new workload 10
#object 1 getting work after 38 min 42 sec
#object 1 balances with 1 peers to new workload 28
#object 1 getting work after 38 min 43 sec
#object 1 balances with 1 peers to new workload 86
#object 1 getting work after 38 min 44 sec
#object 1 balances with 1 peers to new workload 55
#object 1 getting work after 38 min 45 sec
#object 27 balances with 1 peers to new workload 81
#object 27 getting work after 38 min 45 sec
#object 1 balances with 1 peers to new workload 103
#object 1 getting work after 38 min 46 sec
#object 31 getting work after 38 min 48 sec
#master hears 'out of work' from pid 14598, activeObjs now
5
#object 10 balances with 1 peers to new workload 169
#object 5 balances with 1 peers to new workload 150
#object 1 balances with 1 peers to new workload 135
#object 10 getting work after 38 min 49 sec
#object 5 getting work after 38 min 49 sec
#object 1 getting work after 38 min 49 sec
#object 11 getting work after 38 min 51 sec
#object 11 balances with 1 peers to new workload 406
#object 11 getting work after 38 min 52 sec
#object 11 balances with 1 peers to new workload 299
#object 11 getting work after 38 min 52 sec
#object 11 balances with 1 peers to new workload 520
#object 11 getting work after 38 min 52 sec
#object 11 balances with 1 peers to new workload 304
#object 11 getting work after 38 min 53 sec
#object 11 balances with 1 peers to new workload 95
#object 11 getting work after 38 min 54 sec
#object 11 balances with 1 peers to new workload 460
#object 11 getting work after 38 min 54 sec
#master hears 'out of work' from pid 37428, activeObjs now
4
#object 11 balances with 1 peers to new workload 403
#object 11 getting work after 38 min 55 sec
#object 11 balances with 1 peers to new workload 285
#object 11 getting work after 38 min 55 sec
#object 11 balances with 1 peers to new workload 355
#object 11 getting work after 38 min 56 sec
#object 11 balances with 1 peers to new workload 5
#object 11 getting work after 38 min 56 sec
#object 11 balances with 1 peers to new workload 112
#object 11 getting work after 38 min 57 sec
#object 11 balances with 1 peers to new workload 542
#object 11 getting work after 38 min 57 sec
#object 11 balances with 1 peers to new workload 151
#object 11 getting work after 38 min 58 sec
#object 11 balances with 1 peers to new workload 135
#object 11 getting work after 38 min 58 sec
#object 11 balances with 1 peers to new workload 218
#object 11 getting work after 39 min 0 sec
#object 11 balances with 1 peers to new workload 84
#object 11 getting work after 39 min 2 sec
#object 11 balances with 1 peers to new workload 17
#object 11 getting work after 39 min 2 sec
#object 11 balances with 1 peers to new workload 2
#object 11 getting work after 39 min 4 sec
#object 11 balances with 1 peers to new workload 46
#object 11 getting work after 39 min 5 sec
#object 6 getting work after 39 min 5 sec
#master hears 'out of work' from pid 23582, activeObjs now
3
#object 7 getting work after 39 min 12 sec
#object 7 balances with 1 peers to new workload 5967
#object 11 balances with 1 peers to new workload 935
#object 6 balances with 1 peers to new workload 447
#object 6 getting work after 39 min 19 sec
#object 6 balances with 1 peers to new workload 181
#object 6 getting work after 39 min 19 sec
#object 6 balances with 1 peers to new workload 34
#object 6 getting work after 39 min 20 sec
#object 6 balances with 1 peers to new workload 88
#object 6 getting work after 39 min 20 sec
#object 6 balances with 1 peers to new workload 282
#object 6 getting work after 39 min 20 sec
#object 6 balances with 1 peers to new workload 234
#object 6 getting work after 39 min 21 sec
#object 6 balances with 1 peers to new workload 122
#object 6 getting work after 39 min 21 sec
#object 6 balances with 1 peers to new workload 165
#object 6 getting work after 39 min 22 sec
#object 6 balances with 1 peers to new workload 147
#object 6 getting work after 39 min 22 sec
#object 6 balances with 1 peers to new workload 234
#object 6 getting work after 39 min 22 sec
#object 6 balances with 1 peers to new workload 116
#object 6 getting work after 39 min 23 sec
#object 6 balances with 1 peers to new workload 118
#object 6 getting work after 39 min 24 sec
#object 6 balances with 1 peers to new workload 9
#object 6 getting work after 39 min 24 sec
#object 6 balances with 1 peers to new workload 28
#object 6 getting work after 39 min 24 sec
#object 6 balances with 1 peers to new workload 32
#object 6 getting work after 39 min 25 sec
#object 6 balances with 1 peers to new workload 77
#object 6 getting work after 39 min 27 sec
#object 6 balances with 1 peers to new workload 27
#object 6 getting work after 39 min 28 sec
#object 11 getting work after 39 min 28 sec
```

```
#object 11 balances with 1 peers to new workload 46
#object 11 getting work after 39 min 35 sec
#object 2 getting work after 39 min 38 sec
#object 2 balances with 1 peers to new workload 2217
#object 7 getting work after 39 min 40 sec
#object 7 balances with 1 peers to new workload 1150
#object 11 balances with 1 peers to new workload 53
#object 11 getting work after 39 min 41 sec
#object 6 balances with 1 peers to new workload 73
#object 6 getting work after 39 min 41 sec
#object 6 balances with 1 peers to new workload 56
#object 11 balances with 1 peers to new workload 49
#object 11 getting work after 39 min 41 sec
#object 6 getting work after 39 min 41 sec
#object 11 balances with 1 peers to new workload 52
#object 11 getting work after 39 min 42 sec
#object 6 balances with 1 peers to new workload 36
#object 6 getting work after 39 min 43 sec
#object 11 balances with 1 peers to new workload 46
#object 6 balances with 1 peers to new workload 36
#object 6 getting work after 39 min 45 sec
#object 11 getting work after 39 min 45 sec
#object 6 balances with 1 peers to new workload 1
#object 6 getting work after 39 min 49 sec
#object 11 balances with 1 peers to new workload 76
#object 6 balances with 1 peers to new workload 28
#object 11 getting work after 39 min 49 sec
#object 6 getting work after 39 min 49 sec
#object 7 getting work after 39 min 54 sec
#object 7 balances with 1 peers to new workload 115
#object 7 getting work after 39 min 56 sec
#object 7 balances with 1 peers to new workload 95
#object 7 getting work after 39 min 56 sec
#object 7 balances with 1 peers to new workload 920
#object 7 getting work after 39 min 57 sec
#object 7 balances with 1 peers to new workload 10
#object 7 getting work after 39 min 57 sec
#master hears 'out of work' from pid 18316, activeObjs now
2
#object 7 balances with 1 peers to new workload 442
#object 7 getting work after 39 min 57 sec
#object 7 balances with 1 peers to new workload 576
#object 7 getting work after 39 min 59 sec
#object 7 balances with 1 peers to new workload 270
#object 7 getting work after 39 min 59 sec
#object 7 balances with 1 peers to new workload 442
#object 7 getting work after 40 min 0 sec
#object 7 balances with 1 peers to new workload 184
#object 7 getting work after 40 min 1 sec
#object 7 balances with 1 peers to new workload 495
#object 7 getting work after 40 min 2 sec
#object 7 balances with 1 peers to new workload 133
#object 7 getting work after 40 min 2 sec
#object 7 balances with 1 peers to new workload 33
#object 7 getting work after 40 min 3 sec
#object 7 balances with 1 peers to new workload 88
#object 7 getting work after 40 min 3 sec
#object 7 balances with 1 peers to new workload 26
#object 7 getting work after 40 min 4 sec
#object 7 balances with 1 peers to new workload 447
#object 7 getting work after 40 min 6 sec
#object 7 balances with 1 peers to new workload 25
#object 7 getting work after 40 min 9 sec
#object 7 balances with 1 peers to new workload 45
#object 7 getting work after 40 min 10 sec
#object 2 getting work after 40 min 16 sec
#master hears 'out of work' from pid 18588, activeObjs now
1
#object 3 getting work after 40 min 25 sec
#master hears 'out of work' from pid 17368, activeObjs now
0
#object 0 elapsed 40 min 25 sec
#object 1 elapsed 40 min 25 sec
#object 2 elapsed 40 min 25 sec
#object 3 elapsed 40 min 25 sec
#object 4 elapsed 40 min 25 sec
#object 5 elapsed 40 min 25 sec
#object 7 elapsed 40 min 25 sec
#master notified UI that render is complete, renderActive=0
#object 8 elapsed 40 min 25 sec
#object 9 elapsed 40 min 25 sec
#object 11 elapsed 40 min 25 sec
#object 12 elapsed 40 min 25 sec
#object 14 elapsed 40 min 25 sec
#object 15 elapsed 40 min 25 sec
#object 16 elapsed 40 min 25 sec
#object 19 elapsed 40 min 25 sec
#object 0 spent 116811  ms in load balancing (self)
#object 4 spent 114747  ms in load balancing (self)
#object 13 elapsed 40 min 25 sec
#object 15 spent 116976  ms in load balancing (self)
#object 16 spent 158944  ms in load balancing (self)
#object 18 elapsed 40 min 25 sec
#object 19 spent 122386  ms in load balancing (self)
#object 20 elapsed 40 min 25 sec
#object 21 elapsed 40 min 25 sec
#object 27 elapsed 40 min 25 sec
#object 30 elapsed 40 min 25 sec
#object 31 elapsed 40 min 25 sec
#object 0 spent 70  ms transferring work to others
#object 1 spent 103369  ms in load balancing (self)
#object 2 spent 9033  ms in load balancing (self)
#object 3 spent 20  ms in load balancing (self)
#object 4 spent 70  ms transferring work to others
#object 5 spent 114135  ms in load balancing (self)
#object 7 spent 21800  ms in load balancing (self)
#object 8 spent 158360  ms in load balancing (self)
#object 9 spent 151116  ms in load balancing (self)
#object 15 spent 14  ms transferring work to others
#object 16 spent 12  ms transferring work to others
#object 18 spent 160672  ms in load balancing (self)
#object 19 spent 1  ms transferring work to others
#object 22 elapsed 40 min 25 sec
#object 1 spent 129  ms transferring work to others
#object 2 spent 58  ms transferring work to others
#object 3 spent 50  ms transferring work to others
#object 5 spent 40  ms transferring work to others
#object 6 elapsed 40 min 25 sec
#object 7 spent 101  ms transferring work to others
#object 8 spent 11  ms transferring work to others
#object 9 spent 7  ms transferring work to others
#object 10 elapsed 40 min 25 sec
#object 11 spent 70239  ms in load balancing (self)
#object 12 spent 178282  ms in load balancing (self)
#object 13 spent 162183  ms in load balancing (self)
#object 14 spent 108429  ms in load balancing (self)
#object 17 elapsed 40 min 25 sec
#object 18 spent 17  ms transferring work to others
#object 20 spent 174132  ms in load balancing (self)
#object 21 spent 157040  ms in load balancing (self)
#object 22 spent 175956  ms in load balancing (self)
#object 23 elapsed 40 min 25 sec
#object 24 elapsed 40 min 25 sec
#object 25 elapsed 40 min 25 sec
#object 26 elapsed 40 min 25 sec
#object 27 spent 103756  ms in load balancing (self)
#object 28 elapsed 40 min 25 sec
#object 29 elapsed 40 min 25 sec
#object 30 spent 104892  ms in load balancing (self)
#object 31 spent 95582  ms in load balancing (self)
#object 6 spent 67116  ms in load balancing (self)
#object 10 spent 155372  ms in load balancing (self)
#object 11 spent 33  ms transferring work to others
#object 12 spent 15  ms transferring work to others
#object 13 spent 22  ms transferring work to others
#object 14 spent 42  ms transferring work to others
#object 17 spent 148884  ms in load balancing (self)
#object 20 spent 26  ms transferring work to others
#object 21 spent 68  ms transferring work to others
#object 22 spent 5  ms transferring work to others
#object 23 spent 160390  ms in load balancing (self)
#object 24 spent 164074  ms in load balancing (self)
#object 25 spent 172237  ms in load balancing (self)
#object 26 spent 151309  ms in load balancing (self)
#object 27 spent 7  ms transferring work to others
#object 28 spent 177371  ms in load balancing (self)
#object 29 spent 159395  ms in load balancing (self)
#object 30 spent 33  ms transferring work to others
#object 31 spent 42  ms transferring work to others
#object 6 spent 44  ms transferring work to others
#object 10 spent 29  ms transferring work to others
#object 17 spent 80  ms transferring work to others
#object 23 spent 9  ms transferring work to others
#object 24 spent 56  ms transferring work to others
#object 25 spent 27  ms transferring work to others
#object 26 spent 15  ms transferring work to others
#object 28 spent 27  ms transferring work to others
#object 29 spent 61  ms transferring work to others
#global synchronization achieved
time;
#time=855122414
save image 0 ppm 'soda.ppm';
quit;
#ui process pid 20378 requests termination
Requesting shutdown of all processes.
$console.c$
Tue Feb  4 23:09:58 HST 1997
::
:: Job finished!
::
```

Bibliography

[1] Akeley, K. (Reality engine) graphics. *Proceedings of SIGGRAPH* (1993) pp. 109-116.

[2] Amdahl, G. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings AFIPS Conference,* vol. 30 (1967) pp. 483-485.

[3] Arvo, J. & Kirk, D. Particle transport and image synthesis. *Computer Graphics,* vol. 24 (1990) pp. 63-66.

[4] Badouel, D. & Priol, T. An efficient parallel ray-tracing scheme for highly parallel architectures. *Proceedings of the Fifth Eurographics Workshop on Graphics Hardware* (1989).

[5] Barnard, S.T. & Simon, H. A parallel implementation of multi-level recursive spectral bisection for application to adaptive unstructured meshes. In *Proceedings of the* 7^{th} *SIAM Conference on Parallel Processing for Scientific Computing* (1995) pp. 627-632.

[6] Barrett, M.L. A load balancing experiment for parallel ray-tracing. *Proceedings of Ausgraph* (1990) pp. 145-155.

[7] Boillat, J.E. Load balancing and Poisson equation in a graph. *Concurrency: Practice and Experience,* vol. 2 (1990) 289-313.

[8] Boillat, J.E., Brugé, F. & Kropf, P.G. A dynamic load balancing algorithm for molecular dynamics simulations on multiprocessor systems. *Journal of Computational Physics,* vol. 96 (1991) pp. 1-14.

[9] Bokhari, S. *Assignment Problems in Parallel and Distributed Computing* (Kluwer, Boston, 1987).

[10] Brugé, F. & Fornili, S.L. A distributed dynamic load balancer and it's implementation on multi-transputer systems for molecular dynamics simulation. *Computer Physics Communications,* vol. 60 (1990) pp. 39-45.

[11] Caspary, E. & Scherson, I.D. A self-balanced parallel ray-tracing algorithm. *Proceedings of the International Conference on Parallel Processing for Computer Vision and Display* (1988).

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

[13] Cybenko, G. Dynamic load balancing for distributed memory multiprocessors. *The Journal of Parallel and Distributed Computing,* vol. 7 (1989) pp. 279-301.

[14] Delany, H.C. Ray tracing on the Connection Machine. *Proceedings of SIGGRAPH* (1988) pp. 659-664.

[15] Fletcher, C.A.J. *Computational Techniques for Fluid Dynamics* (Springer, New York, 1991).

[16] Green, S.A. & Paddon, D.J. Exploiting coherence for multiprocessor ray-tracing. *IEEE Computer Graphics and Applications,* vol. 9 (1989) pp. 12-26.

[17] Golub, G. H. & Van Loan, C. F. *Matrix Computations* (Johns Hopkins University Press, Baltimore, 1989).

[18] Hall, K. An r-dimensional quadratic placement algorithm. *Management Science,* vol. 17, no. 3 (1970) pp. 219-229.

[19] Hanrahan, P., Salzman, D. & Aupperle, L. A rapid hierarchical radiosity algorithm. *Proceedings of SIGGRAPH* (1991) pp. 197-206.

[20] Heckbert, P. Radiosity in flatland. *Proceedings of Eurographics'92,* vol. 11 (1992) pp. 181-192.

[21] Heirich, A. & Arvo, J. A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing,* vol. 12, no. 1 & 2 (1998) pp. 57-68.

[22] Heirich, A. & Arvo, J. Parallel Radiometric Image Synthesis. *The International Journal of Advances in Engineering Software* (to appear, 1998).

Previously appeared in *Proceedings of the Fourth National Symposium on Large Scale Analysis and Design on High-Performance Computers and Workstations,* Williamsburg, VA (October 1997).

[23] Heirich, A. A scalable diffusion algorithm for dynamic mapping and load balancing on networks of arbitrary topology. *The International Journal of Foundations of Computer Science,* vol. 8, no. 3 (1997) pp. 329-346.

[24] Heirich, A. & Arvo, J. Scalable Monte Carlo image synthesis. *Parallel Computing,* vol. 23, no. 7 (1997) pp. 845-859.

[25] Heirich, A. & Arvo, J. Parallel rendering with an Actor model. *Proceedings of Eurographics '97, Workshop on Programming Paradigms for Graphics,* Budapest, Hungary (September 1997).

[26] Heirich, A. & Arvo, J. Scalable photo-realistic rendering of complex scenes. *Proceedings of the First Eurographics Workshop on Parallel Graphics and Visualization,* Bristol, England (September 1996).

[27] Heirich, A. & Taylor, S. A parabolic load balancing method. *Proceedings of the 24th International Conference on Parallel Processing,* vol. III (1995) pp. 192-202.

[28] Hong, J., Tan, X. & Chen, M. From local to global: an analysis of nearest-neighbor balancing on hypercube. *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (1988) pp. 73-82.

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

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

[31] Hosseini, S. et al. Analysis of graph coloring based distributed load balancing algorithm. *Journal of Parallel and Distributed Computing,* vol. 10 (1990) pp. 160-166.

[32] Hughes, T.J.R. *The Finite Element Method* (Prentice-Hall, Englewood Cliffs, 1987).

[33] Kajiya, J.T. The rendering equation. *Computer Graphics,* vol. 20 (1986) pp. 143-150.

[34] Karp, R. Reducibility in combinatorial problems. In *Complexity of Computer Computations,* Miller & Thatcher (eds.) (Plenum, New York, 1972) pp. 85-103.

[35] Karypis, G. & Kumar, V. Multilevel graph partitioning schemes. *Proceedings of the 24th International Conference on Parallel Processing,* vol. III (1995) pp. 113-122.

[36] Karypis, G. & Kumar, V. A parallel algorithm for multilevel graph partitioning and sparse matrix reordering. *Journal of Parallel and Distributed Computing,* vol. 48 (1998) pp. 71-95.

[37] Kirk, D. & Arvo, J. Unbiased sampling techniques for image synthesis. *Proceedings of SIGGRAPH* (1991) pp. 153-156.

[38] Kobayashi, H., Nakamura, T. & Shigei, Y. Parallel processing of an object space for image synthesis using ray-tracing. *The Visual Computer,* vol. 3 (1987) pp. 13-22.

[39] Kung, H.T. & Stevenson, D. A software technique for reducing the routing time on a parallel computer with a fixed interconnection network. In *High Speed Computer and Algorithm Organization,* Kuck, Lawrie & Sameh (eds.) (Academic Press, New York, 1977) pp. 423-433.

[40] Lafortune, E.P.F., Foo, S.-C., Torrance, K.E. & Greenberg, D. Non-linear approximation of reflectance functions. *Proceedings of SIGGRAPH* (1997) pp. 117-126.

[41] Lin, F.C.H. & Keller, R.M. The gradient model load balancing method. *IEEE Transactions on Software Engineering,* vol. SE-13 (1987) pp. 32-38.

[42] Lindgren, B. W. *Statistical Theory* (MacMillan, New York, 1962).

[43] McCormick, S. *Multigrid Methods* (SIAM, Philadelphia, 1987).

[44] Mohar, B. The Laplacian Spectrum of Graphs. In *Graph Theory: Combinatorics and Applications,* Alavi et al (eds.) (Wiley, New York, 1988) pp. 871-898.

[45] Muniz, F.J. & Zaluska, E.J. Parallel load balancing: an extension to the gradient model. *Parallel Computing,* vol. 21 (1995) pp. 287-301.

[46] Naylor, B. & Thibault, W. Application of (BSP) trees to ray-tracing and (CGS) evaluation. *Technical report GIT-ICS 86/03,* Georgia Institute of Technology, School of Information and Computer Science (1986).

[47] Ni, L.M., Xu, C. & Gendreau, T.B. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering,* vol. SE-11 (1985).

[48] Notkin, I. & Gotsman, C. Parallel progressive ray-tracing. *Computer Graphics Forum,* vol. 17 (1997) pp. 43-55.

[49] Ortega, J.M. *Introduction to Parallel and Vector Solution of Linear Systems* (Plenum, New York, 1988).

[50] Pothen, A., Simon, H. & Liou, K. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM Journal of Matrix Analysis,* vol. 11 (1990) pp. 430-452.

[51] Priol, T. & Bouatouch, K. Static load balancing for (a) parallel ray-tracing on a (MIMD) hypercube. *The Visual Computer,* vol. 5 (1989) pp. 109-119.

[52] Rosenberg, A. Issues in the study of graph embedding. In *Graph Theoretic Concepts in Computer Science,* Noltemeier (ed.) (Springer, New York, 1981) pp. 150-176.

[53] Salmon, J. & Goldsmith, J. A hypercube ray-tracer. *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications* (1988) pp. 1194-1206.

[54] Schloegel, K., Karypis, G., Kumar, V., Biswas, R. & Oliker, L. A performance study of diffusive versus remapped load balancing schemes. *Technical report 98-018,* Army High Performance Computing Center, University of Minnesota (1998).

[55] Sillion, F.X. & Puech, C. *Radiosity and Global Illumination* (Morgan Kaufmann, San Francisco, 1994).

[56] Shirley, P., Wang, C.Y. & Zimmerman, K. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics,* vol. 15 (1996) pp. 1-36.

[57] Simon, H. Partitioning unstructured problems for parallel processing. *Computer Systems in Engineering,* vol. 2, no. 2/3 (1991) pp. 135-148.

[58] Smits, B., Arvo, J. & Greenberg, D. A clustering algorithm for radiosity in complex environments. *Proceedings of SIGGRAPH* (1994) pp. 435-442.

[59] Sterling, T., Becker, D., et al. Beowulf: a parallel workstations for scientific computation. *Proceedings of the 24th International Conference on Parallel Processing,* vol. I (1995) pp. 11-14.

[60] Tutte, W. T. How to draw a graph. *Proceedings of the London Mathematical Society,* vol. 13 (1963) pp. 743-768.

[61] Veach, E. & Guibas, L. Optimally combining sampling techniques for Monte Carlo rendering. *Proceedings of SIGGRAPH* (1995) pp. 419-428.

[62] Ward, G.J., Rubenstein, F.M. & Clear, R. A ray tracing solution for diffuse interreflection. *Proceedings of SIGGRAPH* (1988) pp. 85-92.

[63] Whitted, T. An improved illumination model for shaded display. *Communications of the ACM,* vol. 23 (1980) pp. 343-349.

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

[65] Xu, C. & Lau, F.C.M. Analysis of the generalized dimension exchange method for dynamic load balancing. *Journal of Parallel and Distributed Computing,* vol. 16 (1992) pp. 385-393.

[66] Xu, C. & Lau, F.C.M. The generalized dimension exchange method for load balancing in k-ary n-cubes and variants. *Journal of Parallel and Distributed Computing,* vol. 24 (1995) pp. 72-85.

[67] Xu, C. & Lau, F.C.M. *Load Balancing in Parallel Computers: Theory and Practice* (Kluwer, Boston, 1997).

[68] Yoon, H.J., Fun, S. & Cho, J.W. An image parallel ray-tracing using static load balancing and data prefetching. *Proceedings of the First Eurographics Workshop on Parallel Graphics and Visualization,* Bristol, England (September 1996) pp. 53-66.

[69] Young, D. M. *Iterative Solution of Large Linear Systems* (Academic Press, New York, 1971).