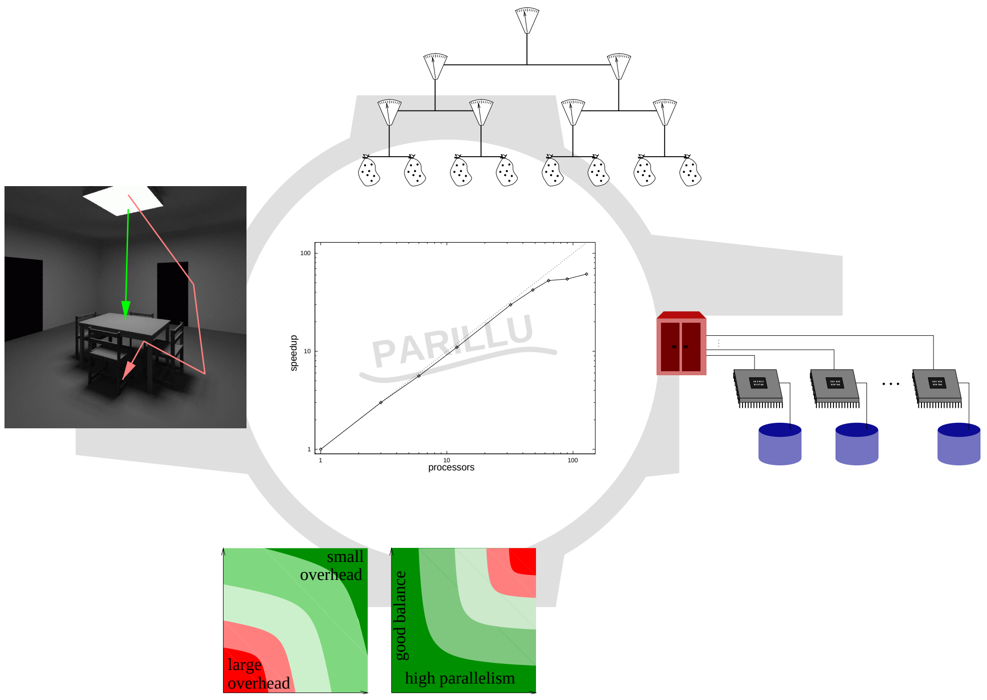
New Contributions To Spatial Partitioning And Parallel Global Illumination Algorithms



Dissertation zur Erlangung des Grades des Doktors der Naturwissenschaften der Universität Dortmund am Fachbereich Informatik von

Robert Garmann

Dortmund 2000

Robert Garmann

Lehrstuhl VII - Graphische Systeme
Fachbereich Informatik
Universität Dortmund
D-44221 Dortmund

Tag der mündlichen Prüfung: 15. September 2000

Preface

This thesis would not have been possible without the help and support of many people. First of all I thank Heinrich Müller, who actively supported me during the last years by giving valuable hints and finding the time for productive discussions. All my colleagues and student assistants are responsible for having provided a pleasant working atmosphere and have supported me with many good ideas. I give thanks to Christian Bohn, Ulrich Lang, and Matthias Müller for providing access to and assisting me with the parallel computers. I want to thank Christoph Lindemann, Peter Marwedel, and Peter Kemper as members of the Doctoral Commitee.

My gratitude I give to my parents, who layed the foundations for making me curious about everything mathematical and technical, and for making my studies and research generally possible. Especially I want to thank Nicole for motivating and supporting me and for enduring my absence especially during the last months.

Dortmund, September 2000 Robert Garmann

Contents

Chapter 1

Introduction

This thesis resides around the interface of two disciplines in computer science: computer graphics and parallel computing (Fig. 1.1). On the one hand an efficient algorithm (A) for global illumination simulation is examined

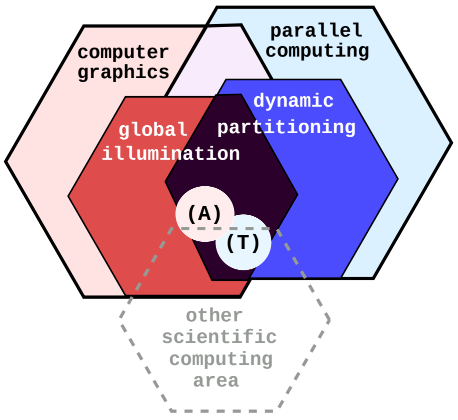


Figure 1.1: Localizing this thesis.

with respect to its capability of being parallelized. On the other hand we develop a general tool (T) for the dynamic partitioning of spatially mapped tasks, that is furthermore analyzed theoretically and experimentally.

The above simulation algorithm (A) is a special instance of algorithms that can be formulated as a collection of spatially mapped tasks. As a proof of practicability of our tool (T) we apply the tool to the simulation algorithm and get useful speedup values. Algorithms of other scientific computing areas apparently can benefit the same way from our tool, even though studying this is beyond the scope of this thesis.

1.1 Global Illumination

The creation of raster images out of a given description of geometrical and optical characteristics of a 3D scene is the subject of Rendering. A broad range of applications from engineering to architecture benefit from realistically looking images. They differ in the extent of the required realism.

In the entertainment industry fast rendering algorithms for visually satisfying outputs are prevailing. Mostly they include only local illumination – i. e. illumination directly from the light sources – into their calculations (see Fig. 1.2).

An increasing number of applications needs higher degrees of realism. Actually, the light emitted by light sources is reflected in the scene across several surfaces. Hence, the light at a point indirectly depends on all light globally transmitted through the scene. Formally this fact is described by a linear operator equation of the form $L = L^e + \mathscr{T} L$ with the unknown quantity L. Solving this equation is the subject of global illumination algorithms.

There are two basic approaches for solving the global illumination problem, the Finite Element approach and the Monte-Carlo approach, also known as the Radiosity approach and the Raytracing or Particle approach. Both approaches are very time consuming raising the natural question for a parallel computer implementation. There is no common agreement, which of these two approaches is generally preferable. Both approaches have their

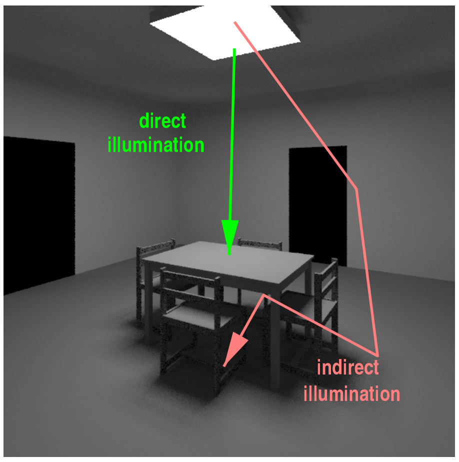


Figure 1.2: Direct and indirect illumination.

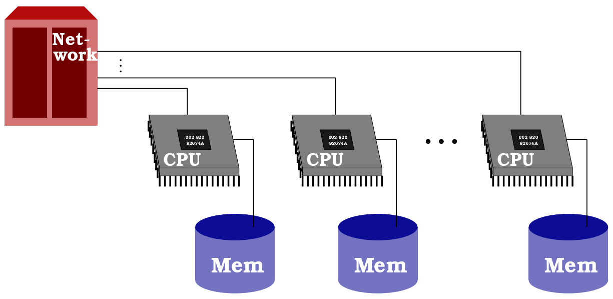


Figure 1.3: Distributed memory.

advantages and disadvantages; both techniques have been improved greatly over the last years. Sophisticated variance reduction techniques have pushed the Monte-Carlo methods. Finite Element techniques have been made useable by multiresolution techniques and by extending them to glossy reflection.

1.2 Dynamic Partitioning

When implementing an algorithm on parallel processors one has to partition the algorithm into equally sized blocks such that all processors are kept busy while interprocessor-communication is kept as low as possible.

The accurate solution of many problems in science and engineering requires the resolution of unpredictable physical phenomena. Such problems may involve complicated partial differential or integral equations and originate from materials design, computational fluid dynamics, astrophysics, molecular dynamics, and – last but not least – global illumination. The most important feature of all these numerical problems is that some regions of the computational domain require deeper resolution than others, and that these regions are not known from the beginning. Hence, the partitioning of the algorithm should be adapted dynamically while the computation proceeds.

We are mostly interested in distributed memory (DM) environments (see Fig. 1.3), since this architecture class is very well scalable. Another reason is that onwardly improving local area network technology puts more and more people in a position where they can solve their problems cost- and time-efficiently on a network of workstations (NOW).

Because there are so many different kinds of parallel and distributed computing environments, there is no agreement to universal abstract models of a parallel environment and of its communication overhead. Such models would be advantageous since they allow portable implementation and universal complexity statements. Agreement has been reached about a specific programming model, the message-passing model, which is now standardized as the *message passing interface (MPI)* [77, 76].

Regardless how far network technology will reduce communication overhead in the future, we will probably always need to deal economically with remote data communication in parallel programs. The larger the discrepancy between processor speed and network bandwidth — typically in NOWs this tends to be very large — the more a partitioning should support coarse grained parallel approaches.

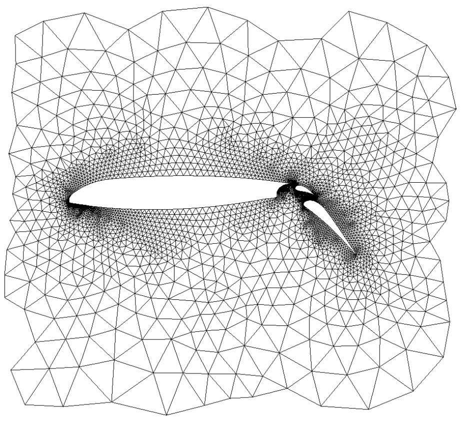


Figure 1.4: A 2D finite element mesh example (“airfoil1”, taken from [85]).

1.3 Motivation And Contributions

Many researchers have devised parallel algorithms for global illumination. Most of these are specifically tailored to special assumptions about the physical properties of the scene’s objects or to a single solution approach. For instance much work has been done on parallelizing the “flat” classical radiosity [22], but not on the much more irregular hierarchical counterpart.

The primary conception for this thesis was to develop a parallel algorithm for the Hierarchical Radiosity algorithm (HRA) [53]. There have been quite successful attempts to implement HRA on a cache-coherent shared-address-space multicomputer [95], since HRA exhibits high fine grained parallelism. But developing a coarse grained parallel algorithm meeting the requirements of a distributed memory (DM) environment is difficult. One contribution[1] of this thesis is a detailed examination of the bottlenecks of a DM implementation of HRA. We show in Chapter 6 that HRA is a very dynamic and communication intensive algorithm, and that the communication paths are quite interweaved and unstructured. We propose metrics and graph partitioning methods for analyzing the “partitionability” in terms of load balancing, communication and congestion. This is a new experimental technique, which could also be useful and important for the analysis of other scientific computing methods.

An important task to be solved when parallelizing hierarchical finite element algorithms is the dynamic partitioning problem. Above in Sect. 1.2 we enumerated a few scientific application domains. There is an important difference between other applications and the hierarchical finite element algorithms. Those other applications exhibit irregular, but locally structured meshes to represent the changing numerical computation (see for example Fig. 1.4). The locality property is important on parallel processors, since it allows to reduce the communication cost by exploiting data locality. At first sight global illumination algorithms miss this locality as the name “global” eloquently suggests. The visibility term in the transport equation (2.1, page 6) takes the responsibility. A contribution of this thesis is a clever mapping of hierarchical finite element algorithms to high-dimensional spaces (Chapter 5). This mapping manifests a locality property, which can greatly reduce communication

Recently there are trends to integrate both Finite Element and Monte-Carlo approaches into a single global illumination framework [96]. We searched for a parallel solution of hierarchical finite element algorithms that is well adaptable also to particle-based global illumination algorithms. Finite element methods, particle methods, and many other scientific computing algorithms can be seen as operating on a set of geometric objects. We decided to deepen the aspect of dynamic partitioning for geometric objects in this thesis. One contribution is the definition of a dynamic *spatial* partitioning strategy (Chapter 7), which preserves locality during dynamic load balancing of geometric objects on parallel processors. An important feature of this strategy, which distinguishes it from previously published partitioners, is that one can show that the worst case amortized time complexity per dynamic object-update is small. We studied the performance and scalability of the load balancer experimentally on a simple artificial application (Chapter 8).

As a realistic “live test” the final contribution of this thesis is an efficient parallel implementation of hierarchical radiosity based on dynamic spatial partitioning (Chapter 9). The speedup curve for the HRA obtained on a Cray T3E is almost linear up to 64 processors. This is better than previously published HRA implementations on massively parallel distributed memory computers.

[1] For a detailed list of all original contributions of this thesis see Sect. 10.2.

1.4 Structure Of This Thesis

This thesis starts with the definition of the global illumination problem by formulating the radiance equation. Chapter 2 gives a short overview of the two predominating methods — the finite element method and the Monte-Carlo method. The description focuses on the algorithmic properties that are important in a parallel implementation. A generic hierarchical finite element algorithm is described in greater detail, since this algorithm is subject of a deeper analysis later in this thesis.

Chapter 3 engages in the problem that we are lacking commonly agreed abstract models of parallel environments. We introduce several basic models (machine, programming, cost) and state reasons about our particular choices that are fundamental to the rest of this thesis.

In Chapter 4 we review the general graph embedding problem, that needs to be solved, when a sequential algorithm is parallelized. We discuss the main incentives that should guide the search for a good solution strategy. Only briefly we skim the wide field of existing heuristic solution approaches.

The following Chapter 5 develops partitioning strategies for the two global illumination methods, finite elements and Monte-Carlo. We aimed at a formulation that makes a joint utilization of both methods in a single program feasable. We compare our approach with related implementations. Again, the hierarchical finite element algorithm is discussed in greater detail. We define a *task access graph* on top of this algorithm which will be examined experimentally in the following chapter.

Chapter 6 presents a quantitative study of the parallelization of the hierarchical radiosity algorithm expressed as a graph partitioning problem. We present metrics about the quality of partitions generated by different graph partitioning techniques. The results highlight the bottlenecks of the parallelization of this algorithm and confirm that the HRA is poorly partitionable. We see that the total communication volume of the HRA cannot be reduced arbitrarily, even if it were possible to use a very complex graph partitioning software. One main result of this chapter is that a simple spatial partitioner, which partitions tasks with respect to their geometric position, seems to be the best compromise when considering the influence of all sources of overhead in a parallel program, namely the load imbalance, the communication overhead and the congestion.

These findings have led us to search for a good spatial partitioner. Existing methods have been reviewed already in Chapter 4. The orthogonal recursive bisection methods stick out because of their ability to compactly represent a directory of distributed objects and tasks — a feature that is very useful in parallel radiosity implementations. In Chapter 7 we describe a dynamic adaptation of the orthogonal recursive bisection method to the load balancing problem. Main merits are a distributed imbalance detection condition and a proof that in the worst case every single dynamic update involves only a small amortized time overhead. The imbalance detection can be performed locally on each processor without global communication. Rebalancing operations take place infrequently which leads to the desirable effect of bundled communication.

In order to show the practicability of our dynamic load balancer, we extensively tested its performance and robustness on a simple synthetic application that treats a set of objects in multi-dimensional space. Chapter 8 shows that for near-practice-circumstances our load balancer achieves good speedups. The small overhead of the theoretical worst-case of the previous chapter is discovered to get even smaller in practice.

We made a truly real experimental test by implementing and studying the HRA in parallel. Besides the spatial partitioning technique we have identified three key concepts for an efficient implementation. First, an asynchronous formulation of the HRA is needed to avoid time consuming barrier synchronizations. Second, elements and links should be grouped to reduce administration overhead. Third, the order of link refinement is crucial for cache efficiency. These concepts, the implementation details, and time measuring results are presented in Chapter 9.

We end with some closing remarks (Chapter 10) and an appendix with a directory of many symbols and abbreviations used in this thesis.

Chapter 2

Rendering Basics

Contents

Rendering means simulating real optics in order to finally produce images from a virtual scene model. Clearly, it is too expensive to simulate real optics exactly. Instead, research has focused on a formulation of light transport making several simplifying assumptions about the behaviour of light. In this chapter we introduce the *radiance equation* and discuss methods that solve this equation approximately.

Our descriptions of the algorithms are focused on exposing those characteristics that are important in a parallel environment. More detail is given about the *representation* of the radiance function and the transport operator at the expense of the physical and mathematical background.

2.1 Modeling Physics

2.1.1 The Radiance Equation

The *radiance equation* is the central equation of image synthesis. It completely captures the distribution of light in a scene. Common simplifications concern eliminating polarization, phosphorescence and flourescence and assuming

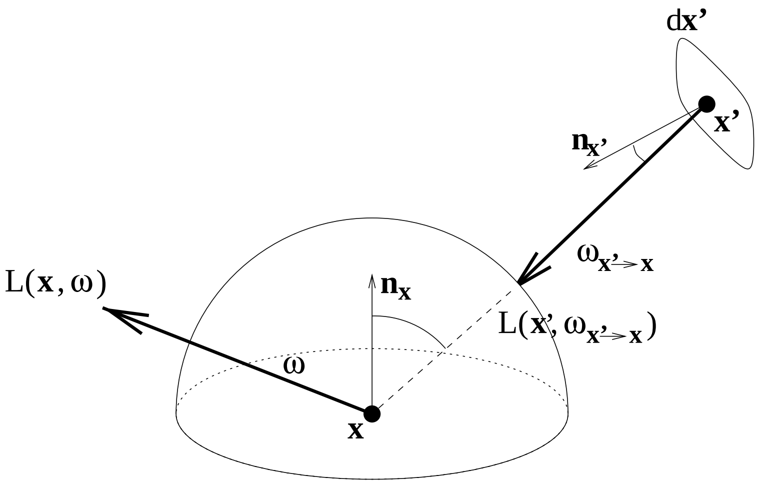


Figure 2.1: The geometry for the radiance equation.

a vacuum surrounding the surfaces of the scene [46]. A further simplification is to only consider reflection by assuming that all objects are opaque. The resulting equation is:

$$L(\vec{x},\vec{\omega}) = L^{\mathrm{e}}(\vec{x},\vec{\omega}) + \int_{M^2} G(\vec{x},\vec{x}')\, f_r(\vec{x},\vec{\omega}_{\vec{x}\to\vec{x}'} \to \vec{\omega}) L(\vec{x}',\vec{\omega}_{\vec{x}'\to\vec{x}}) d\vec{x}', \tag{2.1}$$

where the *geometric term* is defined as

$$G(\vec{x},\vec{x}') = v(\vec{x},\vec{x}')\frac{\cos(\vec{\omega}_{\vec{x}'\to\vec{x}},\vec{n}_{\vec{x}'})\cos(\vec{\omega}_{\vec{x}\to\vec{x}'},\vec{n}_{\vec{x}})}{\|\vec{x}'-\vec{x}\|^2}.$$

Fig. 2.1 shows the corresponding geometry. The term $L(\vec{x},\vec{\omega})$ measures *radiance* at a point $\vec{x}$ in a direction $\vec{\omega}$ and is the unknown quantity in the equation. The radiance's unit is watt per steradian-square meter ($\frac{W}{sr\ m^2}$), a quantity that is directly related to the retina response of the human eye. By $L^{\mathrm{e}}(\vec{x},\vec{\omega})$ we denote the *emitted radiance* at the point $\vec{x}$ in direction $\vec{\omega}$. Those regions, where $L^{\mathrm{e}} > 0$, are called *light sources*. The rest of the equation is basically an integral over the radiance of all incoming directions from the hemisphere at $\vec{x}$ multiplied by the *bidirectional reflection-distribution function (brdf)* and some geometric quantities. The brdf f_r (unit sr^{-1}) characterizes the reflectivity of the surface at some surface point for a given incoming ($\vec{\omega}_{\vec{x}\to\vec{x}'}$) and outgoing ($\vec{\omega}$) direction. The parameterization of incoming directions in (2.1) is by points $\vec{x}'$ that are located on all surfaces M^2 in the scene. The term $v(\vec{x},\vec{x}') \in \{0,1\}$ is dimensionless and is zero if and only if the two points are invisible to each other. $\cos(\vec{\omega},\vec{n}_{\vec{x}})$ denotes the cosine of the angle between the vector $\vec{\omega}$ and the surface normal at $\vec{x}$. $d\vec{x}'$ is an infinitesimal area at the point $\vec{x}'$. The term $\frac{d\vec{x}'\cos(\vec{\omega}_{\vec{x}'\to\vec{x}},\vec{n}_{\vec{x}'})}{\|\vec{x}'-\vec{x}\|^2}$ describes an infinitesimal incident *solid angle* (unit *sr*). The brdf f_r satisfies the following physical constraints:

- reciprocity:

$$f_r(\vec{x},\vec{\omega} \to \vec{\omega}') = f_r(\vec{x},\vec{\omega}' \to \vec{\omega}).$$

- conservation of energy:

$$\rho(\vec{x},\vec{\omega}^{in}) = \int_{H^2} f_r(\vec{x},\vec{\omega}^{in} \to \vec{\omega}^{out})\cos(\vec{\omega}^{out},\vec{n}_{\vec{x}})d\vec{\omega}^{out} \leq 1,$$

  where ρ is called the *total hemispherical reflectivity* and H^2 denotes the set of directions to the upper hemisphere.

A thorough derivation of the radiance equation can be found in [46]. This equation is classified as a Fredholm equation of the second kind [90]. It cannot be solved analytically, except for some trivial cases.

2.1.2 The Radiosity Equation

An often at least approximately valid assumption is that all surfaces in a scene are diffuse surfaces, which reflect light equally and independent of the incoming and outgoing directions. In this case we can switch to the *radiosity* *equation*:

$$B(\vec{x}) = B^e(\vec{x}) + \frac{\rho(\vec{x})}{\pi} \int_{M^2} G(\vec{x},\vec{x}')B(\vec{x}')d\vec{x}'. \tag{2.2}$$

The difference to the radiance equation is the notation in terms of *radiosity* $B(\vec{x}) = \pi L(\vec{x})$, which is independent from the outgoing direction. Its unit is watt per square meter ($\frac{W}{m^2}$). Also reflection does not depend on directions, hence we use the *reflectance factor* $\rho(\vec{x}) = \pi f_r(\vec{x})$ to express reflectivity at $\vec{x}$. The radiosity equation is much simpler, since we were able to drop the dependency from directions. Nevertheless it is still difficult to solve.

2.1.3 Formal Solution

Equations (2.1) and (2.2) both can be expressed in a simple form using linear operator notation. In the case of radiance we write

$$L = L^{\mathrm{e}} + \mathscr{T}L, \tag{2.3}$$

where L, L^{e} are functions defined on the product space of surface points M^2 and hemisphere directions H^2. $\mathscr{T}$ is called the *transport operator*. For the radiosity equation we write

$$B = B^e + \mathscr{T}B, \tag{2.4}$$

where the domain of B, B^e is M^2.

We can solve (2.3) (and analogically (2.4)) formally by inverting the transport operator:

$$L = (1 - \mathscr{T})^{-1}L^{\mathrm{e}}.$$

Given the physical interpretation of the transport operator, we know that the inverse exists. Expansion into a von Neumann series leads to

$$L = \sum_{i=0}^{\infty} \mathscr{T}^i L^{\mathrm{e}}. \tag{2.5}$$

An intuitive physical interpretation of equation (2.5) is that the solution of (2.3) is given by the summation of the directly emitted radiance plus the radiance that is reflected 1, 2, 3, ... times in the scene.

2.1.4 Notations

Radiance is the most important quantity in physically based rendering, since it directly corresponds to the retina response of the human eye. But there are many more quantities that are useful in solving the radiance equation.

Radiant intensity is defined as the power emanating from a point $\vec{x}$ in a direction $\vec{\omega}$ per unit solid angle in that direction:

$$dI(d\vec{x},\vec{\omega}) = L(\vec{x},\vec{\omega})\cos(\vec{\omega},\vec{n}_{\vec{x}})d\vec{x}.$$

Radiant intensity is measured in $\frac{W}{sr}$ and is independent of the orientation of a surface. Point light sources may be modelled by a point emitting varying radiant intensities in different directions. Also clusters of surfaces may be assumed to emit light as a point source.

Incident radiance is measured like radiance in $\frac{W}{m^2 sr}$. Since radiance is invariant along a ray, the radiance incident at a point $\vec{x}'$ from direction $\vec{\omega}$ equals the outgoing radiance at $\vec{x}$ in direction $\vec{\omega}$, if $v(\vec{x},\vec{x}') = 1$:

$$L^{\mathrm{in}}(\vec{x}',\vec{\omega}_{\vec{x}'\to\vec{x}}) = v(\vec{x},\vec{x}')L(\vec{x},\vec{\omega}_{\vec{x}\to\vec{x}'}).$$

We parameterize incident quantities with the direction pointing to the upper hemisphere $(\vec{x}', \vec{\omega}_{\vec{x}'} \to \cdot)$.

Irradiance is defined as the energy per unit area at a point $\vec{x}$ received from direction $\vec{\omega}$:

$$dE(\vec{x},d\vec{\omega}) = L^{\mathrm{in}}(\vec{x},\vec{\omega})\cos(\vec{\omega},\vec{n}_{\vec{x}})d\vec{\omega}$$

and is measured in $\frac{W}{m^2}$. We can use irradiance to calculate radiance from it (see (2.1)):

$$L(\vec{x},\vec{\omega}) = L^{\mathrm{e}}(\vec{x},\vec{\omega}) + \int_{\vec{\omega}^{\mathrm{in}}\in H^2} f_r(\vec{x},\vec{\omega}^{\mathrm{in}} \to \vec{\omega})dE(\vec{x},d\vec{\omega}^{\mathrm{in}}). \tag{2.6}$$

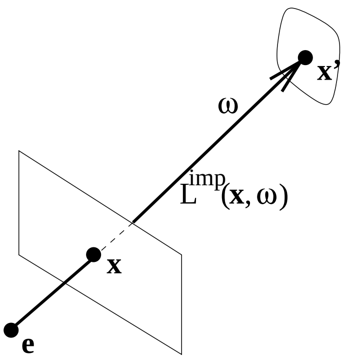


Figure 2.2: The geometry for the importance equation.

Sometimes we need an incident quantity that is not tied to a surface but nevertheless is measured with respect to a differential solid angle. We define *perpendicular irradiance* as irradiance in the case that the direction $\vec{\omega}$ is perpendicular to some imaginary surface at $\vec{x}$:

$$dE^{\perp}(\vec{x}, d\vec{\omega}) = L^{\text{in}}(\vec{x}, \vec{\omega})d\vec{\omega}.$$

The *radiant flux* (or *power*) flowing in the beam from a differential surface element $d\vec{x}$ to another differential surface element $d\vec{x}'$ is defined as the product of radiance, the solid angle subtended by $d\vec{x}'$ and the projected area of $d\vec{x}$:

$$\begin{aligned} d^2\Phi(d\vec{x}, d\vec{x}') &= L(\vec{x}, \vec{\omega}_{\vec{x}\to\vec{x}'})v(\vec{x}, \vec{x}')\frac{\cos(\vec{\omega}_{\vec{x}'\to\vec{x}}, \vec{n}_{\vec{x}'})d\vec{x}'}{\|\vec{x}-\vec{x}'\|^2}\cos(\vec{\omega}_{\vec{x}\to\vec{x}'}, \vec{n}_{\vec{x}})d\vec{x} \\ d^2\Phi(d\vec{x}, d\vec{\omega}) &= dE(\vec{x}, d\vec{\omega})d\vec{x}. \end{aligned}$$

Flux is measured in W (watts) and is used in particle tracing, where each particle carries a small amount of flux.

For notational convenience we may pack the two parameters of L into a single variable $\mu := (\vec{x}, \vec{\omega})$. Radiance functions L are square-integrable functions (often denoted $\mathscr{L}^2$-functions) [21]. We define the (standard) inner product on the space of $\mathscr{L}^2$-functions by $\langle f, g\rangle = \int_{M^2\times H^2} f(\mu)g(\mu)d\mu$. Two functions are called *orthogonal*, if their inner product is zero.

Sometimes we split the operator $\mathscr{T}$ into two operators as $\mathscr{T} = \mathscr{R}\mathscr{G}$, where the operator $\mathscr{R}$ defines outgoing radiance from an irradiance function as follows:

$$(\mathscr{R}E)(\vec{x}, \vec{\omega}) = \int_{\vec{\omega}'\in H^2} f_r(\vec{x}, \vec{\omega}' \to \vec{\omega})dE(\vec{x}, d\vec{\omega}')$$

and $\mathscr{G}$ denotes the propagation including the visibility calculations

$$(\mathscr{G}L)(\vec{x}, \vec{\omega}_{\vec{x}\to\vec{x}'}) = v(\vec{x}, \vec{x}')L(\vec{x}', \vec{\omega}_{\vec{x}'\to\vec{x}})\cos(\vec{\omega}_{\vec{x}\to\vec{x}'}, \vec{n}_{\vec{x}}).$$

2.1.5 Importance

Often we are interested in generating an image from just a single or a small set of viewpoints. Then a view-dependent solution of the radiance equation would suffice saving much effort. But, therefore we need to know, in what regions a detailed solution is important and where not.

We introduce *(directional) importance* L^{imp} as in [20] as a dimensionless quantity that satisfies an equilibrium equation like the radiance equation (2.3):

$$L^{\text{imp}} = L^{\text{imp,e}} + \mathscr{T}L^{\text{imp}}. \tag{2.7}$$

The emitted importance $L^{\text{imp,e}}$ is defined as:

$$L^{\text{imp,e}}(\vec{x}, \vec{\omega}) = \begin{cases} 1, & \text{if } \vec{x} \text{ is a point on the image and } \vec{\omega} \text{ points from the eye to } \vec{x} \\ 0, & \text{otherwise.} \end{cases}$$

Fig. 2.2 shows the corresponding geometry.

Intuitively $L^{\text{imp}}(\vec{x}, \vec{\omega}_{\vec{x}\to\vec{x}'})$ can be seen as the fraction of $G(\vec{x}, \vec{x}')L(\vec{x}', \vec{\omega}_{\vec{x}'\to\vec{x}})$ that reaches the eye (Fig. 2.3).

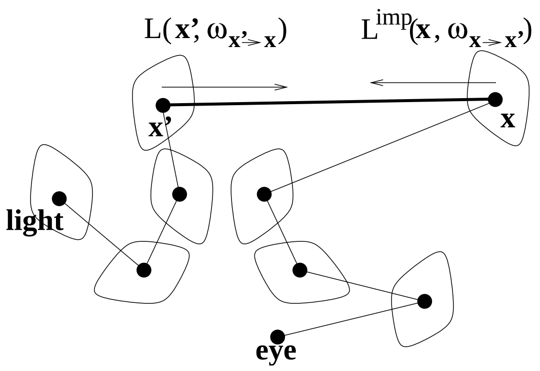


Figure 2.3: Connecting a radiance and an importance path.

The directional importance can be used to calculate the power that radiance L contributes to the image:

$$\int\int G(\vec{x}',\vec{x})L^{\mathrm{imp}}(\vec{x},\vec{\omega}_{\vec{x}\to\vec{x}'})L(\vec{x}',\vec{\omega}_{\vec{x}'\to\vec{x}})d\vec{x}'d\vec{x}. \tag{2.8}$$

The contribution of radiance to the image can be found by solving a transport problem with the image emitting importance. Since importance is transported like radiance we can use the same data structures as for radiance and treat importance as if it were another set of color channels.

Of course we can define analogues to radiant intensity, incident radiance, irradiance and perpendicular irradiance: *importance intensity* I^{imp}, *incident importance* $L^{\mathrm{imp,in}}$, *projected incident importance* E^{imp} and *perpendicular incident importance* $E^{\perp,\mathrm{imp}}$:

$$\begin{aligned}
dI^{\mathrm{imp}}(d\vec{x},\vec{\omega}_{\vec{x}\to\vec{x}'}) &= L^{\mathrm{imp}}(\vec{x},\vec{\omega}_{\vec{x}\to\vec{x}'})\cos(\vec{\omega}_{\vec{x}\to\vec{x}'},\vec{n}_{\vec{x}})d\vec{x},\\
L^{\mathrm{imp,in}}(\vec{x},\vec{\omega}_{\vec{x}\to\vec{x}'}) &= v(\vec{x},\vec{x}')L^{\mathrm{imp}}(\vec{x}',\vec{\omega}_{\vec{x}'\to\vec{x}}),\\
dE^{\mathrm{imp}}(\vec{x},d\vec{\omega}_{\vec{x}\to\vec{x}'}) &= \cos(\vec{\omega}_{\vec{x}\to\vec{x}'},\vec{n}_{\vec{x}})L^{\mathrm{imp,in}}(\vec{x},\vec{\omega}_{\vec{x}\to\vec{x}'})d\vec{\omega}_{\vec{x}\to\vec{x}'},\\
dE^{\perp,\mathrm{imp}}(\vec{x},d\vec{\omega}_{\vec{x}\to\vec{x}'}) &= L^{\mathrm{imp,in}}(\vec{x},\vec{\omega}_{\vec{x}\to\vec{x}'})d\vec{\omega}_{\vec{x}\to\vec{x}'}.
\end{aligned}$$

The generation of an image out of a given solution for one of the transport equations (radiance or importance) is beyond the scope of this thesis. The interested reader can find an excellent derivation in [30].

2.1.6 Representing Radiance And Importance

The radiance and importance function must be stored in a suitable form in a computer. Since radiance and importance are defined as functions on the same domain, they can be stored in the same way. Hence we will focus here on representing the radiance function.

The problem with storing the radiance function is, that we do not know in advance, where the function is smooth and where it is discontinuous. Of course we could estimate the shape of L by considering the reflection behaviour of the surfaces. Thus, L mostly is smooth on diffusely reflecting surfaces. But if incident light is partly obstructed, then L may have large gradients even on a diffuse reflector. Also caustics may introduce large gradients into the radiance function.

There are two basic principles used in global illumination to store the radiance function: finite elements and particles. Finite element techniques are especially useful, if the function to be represented is smooth. They can also be employed for storing an average value of the function over some large support. Representing fine small details is very costly in finite element techniques but nevertheless possible.

Particles represent function values at sample points. Since usually there is no coherence assumption between sample points, particles are especially useful for representing large local gradients. Nevertheless, particles also have been used to represent smooth functions [104].

2.2 Finite Element Approach

The description of the finite element approach presented in this section is somewhat over-detailed for readers interested in parallel computing. We discuss clustering, the representation of functions over surfaces and that over clusters. We do this in the present detail because we aim at a new generic formulation of the shooting approach

(Sect. 2.2.5). This formulation abstracts from the specific function (radiosity, radiance, irradiance, ...) stored at the elements. The parallel algorithm described later in this thesis is based on this generic formulation, hence it does not necessarily need to know about the represented function. Thus, it is easily extendable to all finite element algorithm variants.

2.2.1 Galerkin Approach

We start with the infinite-dimensional space $\mathscr{L}^2(M^2 \times H^2)$ of square-integrable functions. Let $\{N_j\}_{j=1,2,3,...}$ denote a basis of this space. A function f can be expressed as a linear combination of the basis functions:

$$f(\vec{x},\vec{\omega}) = \sum_j c_j N_j(\vec{x},\vec{\omega}). \tag{2.9}$$

In the case of a finite dimensional subspace this representation is called a *discretization* of f. The coefficients are calculated by projection [8].

Here we follow the Galerkin approach, where the coefficients are simply inner products of f with the dual basis functions $\tilde{N}_j$:

$$c_j = \langle \tilde{N}_j, f \rangle.$$

The dual basis is characterized by the relation $\langle N_i, \tilde{N}_j \rangle = \delta_{ij}$, where δ_{ij} is Kronecker's delta. Orthonormal bases are *selfdual*. In the following we will assume an orthonormal basis.

There is some freedom in choosing a basis. Classically for radiosity constant basis functions were used [47]. Later higher order functions were employed in order to better approximate the real radiosity function [111, 98]. Hierarchical bases [48, 20] are useful for an adaptive solution.

2.2.2 The Multiresolution Approach

In a real computation we must restrict ourselves to a finite subspace of $\mathscr{L}^2(M^2 \times H^2)$. The simplest way to do this is to choose a priori a large, fixed set of basis functions. A simple but common choice is a basis of piecewise constant functions [19].

Let us first consider univariate functions $f \in \mathscr{L}^2([0,1])$. We may approxmiate such functions by piecewise constant functions at a given resolution (or *scale*) s with discontinuities at $\{0, \frac{1}{2^s}, \frac{2}{2^s}, \ldots, 1\}$. This subspace of approximating functions is spanned by the so-called *Haar scaling functions* $\phi_t^s(u)$, $t \in \{0,\ldots,2^s-1\}$:

$$\phi_t^s(u) = \begin{cases} \sqrt{2}^s, & \text{for } u \in [\frac{t}{2^s}, \frac{t+1}{2^s}), \\ 0, & \text{otherwise.} \end{cases}$$

s is called the *scale* and t is called the *translation* of ϕ_t^s. The basis is selfdual, since $\langle \phi_t^s, \phi_{t'}^s \rangle = \int_0^1 \phi_t^s(u)\phi_{t'}^s(u)du = \delta_{tt'}$.

For multivariate functions like the radiance function we construct a basis using tensor products of the above univariate basis functions. We will follow the approach of [19] and assume a mapping of the domain of radiance functions such that $M^2 = [0,1]^2$ and $H^2 = [0,1]^2$. The following functions form a basis of piecewise constant approximations of $\mathscr{L}^2([0,1]^4)$-functions:

$$\phi\phi\phi\phi^s_{t_u,t_v,t_\theta,t_\rho}(u,v,\theta,\rho) := \phi^s_{t_u}(u)\phi^s_{t_v}(v)\phi^s_{t_\theta}(\theta)\phi^s_{t_\rho}(\rho),$$

where $t_u, t_v, t_\theta, t_\rho \in \{0,\ldots,2^s-1\}$.

The representation of f by coefficients c_i at a fixed resolution scale s has the drawback, that computation time and storage is wasted. Many of the c_i coefficients are approximately equal, hence they could be represented sufficiently exact by a single coefficient. A better approach would start with a coarse representation of f and then add more detail where necessary. Hierarchical methods [53, 9, 48, 19, 89] take such an approach.

Again, let us consider the univariate case first. Let $V := \mathscr{L}^2([0,1])$ and let V^s denote the subspace of V that is spanned by all ϕ_t^s functions ($t \in \{0,\ldots,2^s-1\}$). Subspaces of increasing scale form a hierarchy of subspaces of V, i. e. $V^0 \subset V^1 \subset V^2 \subset \cdots \subset V$. By W^s we denote the orthogonal complement space of V^s in V^{s+1} defined by

$$W^s = \{ f \in V^{s+1} \quad : \quad \langle f, g \rangle = 0 \text{ for all } g \in V^s \}.$$

The W^s are sometimes called *wavelet spaces*. A basis of W^s is given by the *wavelets* $\psi_t^s(u)$, $t \in \{0,\ldots,2^s-1\}$:

$$\psi_t^s(u) = \begin{cases} \sqrt{2}^s, & \text{for } u \in [\frac{2t}{2^{s+1}}, \frac{2t+1}{2^{s+1}}), \\ -\sqrt{2}^s, & \text{for } u \in [\frac{2t+1}{2^{s+1}}, \frac{2t+2}{2^{s+1}}), \\ 0, & \text{otherwise.} \end{cases}$$

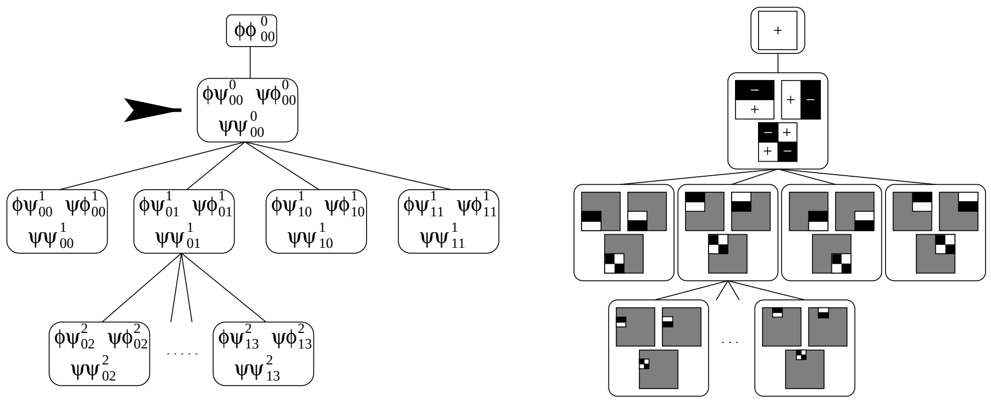


Figure 2.4: Storing a non-standard constructed basis for $\mathscr{L}^2([0,1]^2)$ in a tree.

The space of approximate univariate functions at a given scale s now can be written as the direct sum $V^s = V^0 \oplus W^0 \oplus W^1 \oplus \cdots \oplus W^{s-1}$, and a basis of V^s can be constructed similarly as $\{\phi_0^0, \psi_0^0, \psi_1^0, \psi_1^1, \ldots, \psi_{2^{s-1}-1}^s\}$. The scaling function spanning V^0 represents coarse shape, while the wavelets provide detail at increasing resolutions.

A basis of the multivariate functions $\mathscr{L}^2([0,1]^4)$ can be constructed in different ways. The *standard construction* results in the following four-dimensional basis:

$$\left\{\phi_0^0, \psi_0^0, \psi_1^0, \psi_1^1, \ldots, \psi_{2^{s-1}-1}^s\right\}^4.$$

In the *nonstandard construction* each tensor product consists of univariate basis functions in the same space V^j, including scaling functions at all levels:

$$\{\phi_0^0\phi_0^0\phi_0^0\phi_0^0\} \cup \left\{ \begin{array}{r} \phi_{t_u}^j\phi_{t_v}^j\phi_{t_\theta}^j\psi_{t_\rho}^j, \phi_{t_u}^j\phi_{t_v}^j\psi_{t_\theta}^j\phi_{t_\rho}^j, \ldots, \psi_{t_u}^j\psi_{t_v}^j\psi_{t_\theta}^j\psi_{t_\rho}^j : \\ 0 \le j < s, 0 \le t_u, t_v, t_\theta, t_\rho < 2^j \end{array} \right\}.$$

Let us consider the radiosity function $B \in \mathscr{L}^2([0,1]^2)$ for a moment. A multiresolution basis using the non-standard construction can be stored in a tree as it is shown in figure 2.4. The right tree visualizes the sign of each basis function on $[0,1]^2$. In grey regions the respective function equals zero, in white regions it is positive, in black regions negative.

2.2.3 Clustering

Up to now we assumed that the geometry of a scene was given by a set M^2 of surface points, that may easily be mapped onto $[0,1]^2$. In a real application scenes consist of thousands of different more or less complicated separate surface geometries. It is difficult to represent all these objects by one set M^2 that is easily mapped onto $[0,1]^2$. But often it is straightforward to map each single object's surface onto $[0,1]^2$. Hence, from now on M^2 will denote the set of points on a single object's surface.

The first multiresolution global illumination algorithm [53] was designed as follows. The illumination is represented separately on each object. At the beginning of the algorithm the illumination on every object is assumed to be constant independent of position and direction. A single transport coefficient accounts for the transport of energy from a sending object to a receiving object. The linear equation system consisting of all these transport coefficients is constructed and solved iteratively. Then some of the objects are *refined*, i. e. a multiresolution representation of the illumination on the surface is constructed.

Let k denote the number of separate objects. Then initially we have to compute $\Theta(k^2)$ transport coefficients (see below). This is too expensive, especially if we aim at a progressive algorithm, that is able to compute a fast coarse solution for preview purposes. Time can be saved by recognizing that the transport coefficients between surfaces in far distant object clusters do not vary significantly. By clustering objects together and establishing transport links between pairs of object clusters instead of pairs of individual objects we represent all individual transports by a single average transport coefficient.

Let $B^3 \subset \mathbb{R}^3$ denote the axis aligned bounding box of the scene geometry. Clusters are created by dividing the box into two subboxes. Each scene object is put into one subbox of B^3 and then we recurse for each subbox. Objects that do not fit completely in a subbox are grouped directly at the parent. Recursion stops, when a leaf contains only few objects. Fig. 2.5 shows a cluster with its subboxes and some larger objects grouped at the parent.

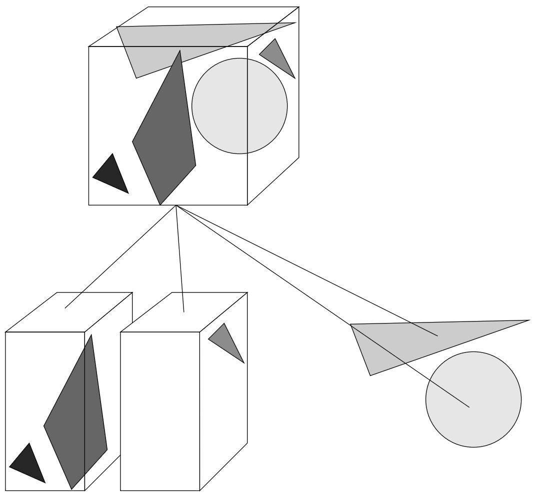


Figure 2.5: Clustering objects.

A hierarchy of clusters can be used to coarsely approximate the radiance function in 3D space. Radiance is not only defined on surfaces but also between them. As a function in 3D space radiance is not associated any longer with a single hemisphere of a surface. Let S^2 denote the surface of a sphere. In order to store the function $L \in \mathscr{L}^2(B^3 \times S^2)$ we assume that the domain $B^3 \times S^2$ is mapped to $[0,1]^5$.

2.2.4 Hierarchical Representation

We describe a strategy to store a finite element representation of radiance (whether defined as four- or five-variate) in a tree. Of course, importance can be stored in the same way. To simplify the presentation, we only discuss the representation of radiance.

The whole support $[0,1]^{4|5}$ of the function is associated with the root node of the tree and is divided into sub-supports[1]. Each node of the tree has its own support which is contained in the support of the parent node. The support of a node is described exactly by the scale $s \in I\!N \cup \{0\}$ and the vector $t \in \prod_{i=1}^{4|5}\{0,\ldots,2^s-1\}$. These are interpreted as the support:

$$\prod_{i=1}^{4|5}\left[\frac{t_i}{2^s},\frac{t_i+1}{2^s}\right).$$

Each node has associated a set of basis functions that describe the detail of the function over the support of the node. The corresponding coefficients are stored in the node. The nodes of the tree are called *elements.*

2.2.4.1 Surface Elements

We start with a description of the element representation of radiance for surfaces of scene objects. Since these are associated with surfaces we call them S-elements. First we have to decide, which quantity should be stored. If we store an outgoing quantity like radiance, we can quickly evaluate radiance – the quantity we are interested in when generating an image. Incoming quantities like irradiance instead must be reflected before, but has the advantage that it carries the information from which direction light is impinging on the surface. This is especially useful in Monte-Carlo importance sampling, where one wants to generate secondary rays pointing to the main light contributors. In this thesis we assume that irradiance is stored at the surfaces. Outgoing radiance is calculated on the fly, if needed.

The coarsest possible finite element representation of a function is by a single basis function $\phi_0^0\phi_0^0\phi_0^0\phi_0^0$ (u,v,θ,ρ) whose support is the whole domain $[0,1]^4$.

An element of this type is called *constant on surface (CS)* and contains only a single coefficient. Such an element may have a single child, if more detail is required.

Fig. 2.6 shows two surfaces (a sphere and a Bezier patch) and their corresponding representation. The sphere is assumed to be lit smoothly and therefore a single element suffices to represent illumination properly. The Bezier patch contains a hierarchy of elements as they are described below.

[1] We use the notation $X^{a|b}$ as shorthand for "X^a or X^b".

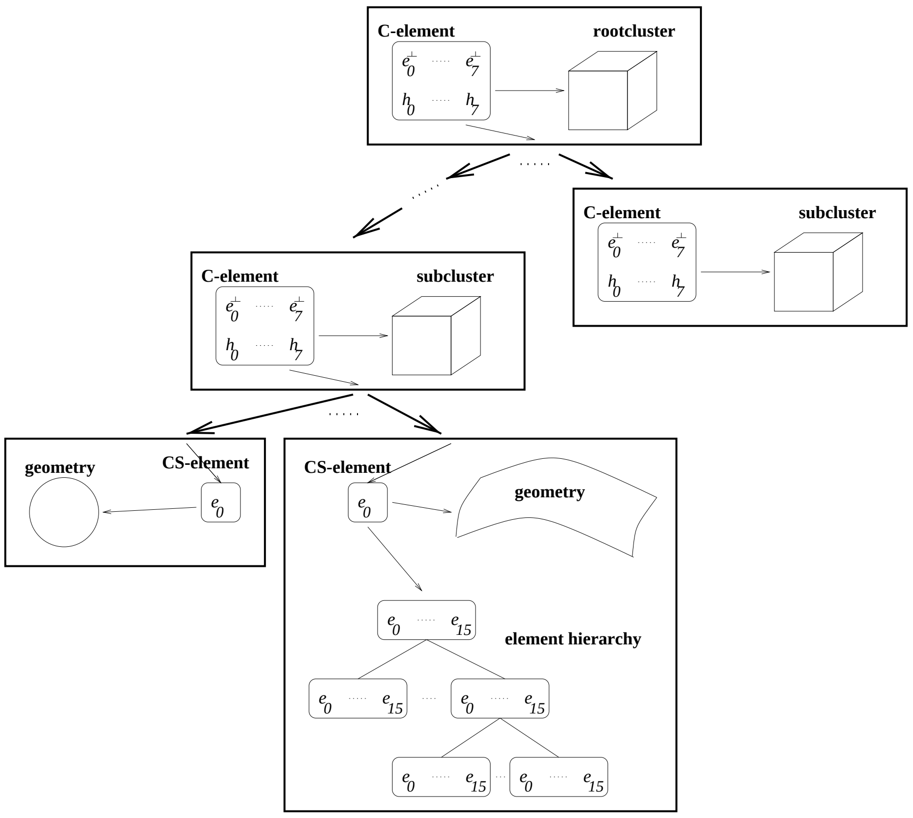


Figure 2.6: Element representation of radiance and importance.

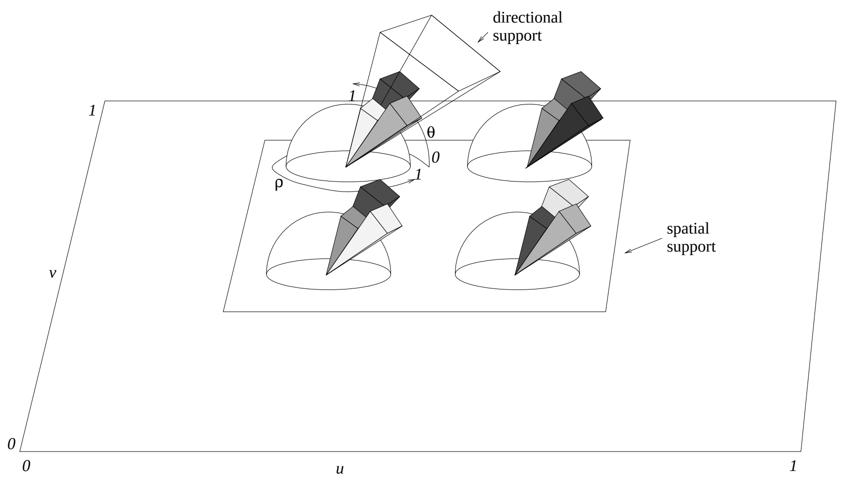


Figure 2.7: A function represented by a VS-element.

We represent detail of the illumination function by considering the following 15 basis functions:

$$\begin{array}{cccc}
 & \phi\psi\phi\phi^s_{t_u,t_v,t_\theta,t_\rho} & \psi\phi\phi\phi^s_{t_u,t_v,t_\theta,t_\rho} & \psi\psi\phi\phi^s_{t_u,t_v,t_\theta,t_\rho} \\
\phi\phi\phi\psi^s_{t_u,t_v,t_\theta,t_\rho} & \phi\psi\phi\psi^s_{t_u,t_v,t_\theta,t_\rho} & \psi\phi\phi\psi^s_{t_u,t_v,t_\theta,t_\rho} & \psi\psi\phi\psi^s_{t_u,t_v,t_\theta,t_\rho} \\
\phi\phi\psi\phi^s_{t_u,t_v,t_\theta,t_\rho} & \phi\psi\psi\phi^s_{t_u,t_v,t_\theta,t_\rho} & \psi\phi\psi\phi^s_{t_u,t_v,t_\theta,t_\rho} & \psi\psi\psi\phi^s_{t_u,t_v,t_\theta,t_\rho} \\
\phi\phi\psi\psi^s_{t_u,t_v,t_\theta,t_\rho} & \phi\psi\psi\psi^s_{t_u,t_v,t_\theta,t_\rho} & \psi\phi\psi\psi^s_{t_u,t_v,t_\theta,t_\rho} & \psi\psi\psi\psi^s_{t_u,t_v,t_\theta,t_\rho}
\end{array}$$

These basis functions are associated with so-called *variable on surface (VS)* elements and add detail in both spatial and directional variables. Fig. 2.7 shows a visualization of the function that could be represented by a single VS-element.

A VS-element may be put immediately below a CS-element as the only child with $s = 0$. A VS-element may have 16 children, each being a VS-element with an s-value one larger than the parent's scale. VS-elements contain 15 coefficients.

When we want to evaluate the illumination at some sample (u, v, θ, ρ) we would have to evaluate an average value at the CS-element and then add the detail, which is evaluated at the VS-elements. If the height of the tree grows, we would have to follow a complete path from the CS-element downto a leaf and add contributions to the resulting function value at all levels of the tree. We could accelerate this by storing an additional coefficient at a VS-element that corresponds to the scaling basis function

$$\phi\phi\phi\phi^s_{t_u,t_v,t_\theta,t_\rho}.$$

Then we know the average value locally at the VS-element and do not need information from the ancestor elements any longer. [2]

If a scene model is given that consists of diffuse surfaces only, then the representation of detail in the directional variables θ, ρ is superfluous. In this case VS-elements are associated only with the following four basis functions

$$\phi\phi\phi\phi^s_{t_u,t_v,t_\theta,t_\rho} \quad \phi\psi\phi\phi^s_{t_u,t_v,t_\theta,t_\rho} \quad \psi\phi\phi\phi^s_{t_u,t_v,t_\theta,t_\rho} \quad \psi\psi\phi\phi^s_{t_u,t_v,t_\theta,t_\rho}.$$

Note, that this definition differs from the definition of elements in the original hierarchical radiosity algorithm [53], where each surface element was associated with only a single basis function $\phi\phi\phi\phi^s_{t_u,t_v,t_\theta,t_\rho}$. This has the effect, that a link carries only a single transport coefficient. This is a fairly tiny task compared to the overhead to manage a link's datastructure and to assign the link to an idle processor. We believe that in a distributed memory environment it is reasonable to burden each link by a more complex task in order to reduce the total number of links, which should result in a smaller management overhead. The resulting approximation of the radiance/importance function will be not coarser than in the original hierarchical radiosity algorithm.

2.2.4.2 Cluster Elements

Radiance of clusters is a function of three spatial and two directional variables. The problem with clusters is that they may contain scene objects. Radiance of scene objects is represented as described in Sect. 2.2.4.1 by a four-variate function where both the spatial and the directional support are $[0,1]^2$. As a principle the support of a child element must be contained in the support of its parent element. But what does that mean, when the child element is a CS-element with spatial support $I_{child} = [u_0,u_1] \times [v_0,v_1] \subseteq [0,1]^2$ and the parent is a cluster with spatial support $I_{parent} = [x_0,x_1] \times [y_0,y_1] \times [z_0,z_1] \subseteq [0,1]^3$? We define the support I_{child} as being contained in I_{parent}, if the 3D surface points that correspond to the piece I_{child} of the surface are contained in I_{parent}. With this definition the clustering algorithm of Sect. 2.2.3 guarantees that the spatial support of surfaces is contained in the spatial support of clusters.

Containment of directional support is problematic, too, because the directional support $[0,1]^2$ of CS-element describes a hemisphere centered at a local coordinate system and the directional support of clusters is mapped to a whole sphere in a canonical coordinate system. Therefore a similar definition is given as above. The directional support of a surface element is defined as being contained in the directional support of the parent cluster, if it is contained after transformation to the cluster's coordinate system. Because of arbitrarily oriented surfaces inside a cluster, the cluster must have full directional support in order to contain the directional support of the contained surface elements. This forces clusters to never split their directional support.

If we want a detailed representation of radiance in the directional arguments, then the basis functions of a cluster must provide detail in the directional variables. Since we also want a detailed representation in the spatial

[2] By storing scaling coefficients at each level, of course we overrepresent the function. Especially all detail coefficients at some inner node at level l could be easily calculated from the scaling coefficients at the next deeper level $l+1$. If we would omit the detail coefficients at level l, then transporting this detail information to other elements had to be done by sixteen links at the deeper level $l+1$ instead of one link at level l. Hence, by using detail coefficients at inner nodes, we are essentially saving memory for links.

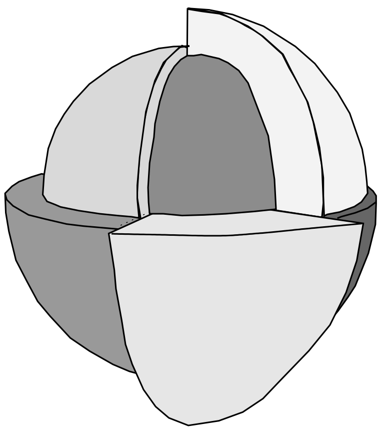


Figure 2.8: A function of direction that is constant on each octant.

variables, a cluster would have to store at least 31 coefficients. This is prohibitively large in the solution process (see below) where a link established between two clusters would contain $31^2 = 961$ propagation coefficients. Hence, we decided to represent radiance of clusters only detailed in the directional variables. The spatial detail is accounted for by subdividing clusters spatially.

Radiance of a cluster is represented by a so-called *cluster element (C-element)*. The following functions are used as a basis leading to a representation of octantwise constant functions (see Fig. 2.8). :

$$\begin{array}{ll}
\phi^s_{t_x}\phi^s_{t_y}\phi^s_{t_z}\phi^0_0\phi^0_0(x,y,z,\theta,\rho), & \phi^s_{t_x}\phi^s_{t_y}\phi^s_{t_z}\psi^0_0\phi^0_0(x,y,z,\theta,\rho), \\
\phi^s_{t_x}\phi^s_{t_y}\phi^s_{t_z}\phi^0_0\psi^0_0(x,y,z,\theta,\rho), & \phi^s_{t_x}\phi^s_{t_y}\phi^s_{t_z}\psi^0_0\psi^0_0(x,y,z,\theta,\rho), \\
\phi^s_{t_x}\phi^s_{t_y}\phi^s_{t_z}\phi^0_0\psi^1_0(x,y,z,\theta,\rho), & \phi^s_{t_x}\phi^s_{t_y}\phi^s_{t_z}\psi^0_0\psi^1_0(x,y,z,\theta,\rho), \\
\phi^s_{t_x}\phi^s_{t_y}\phi^s_{t_z}\phi^0_0\psi^1_1(x,y,z,\theta,\rho), & \phi^s_{t_x}\phi^s_{t_y}\phi^s_{t_z}\psi^0_0\psi^1_1(x,y,z,\theta,\rho).
\end{array}$$

Note that every cluster has got a scaling function $\phi\phi\phi\phi\phi$ meaning a slight overrepresentation. This ensures that we represent detail in space thereby avoiding a large number of coefficients in each cluster.

If a C-element should have children, then its spatial support is split into two spatial subboxes. The directional support of the children is that of the parent.

We did not mention the quantity that should be stored at clusters yet. Irradiance is problematic, since irradiance is tied to a single surface. Hence we decided to represent *perpendicular irradiance* $E^{\perp}$ at clusters.

Above we argued, why storing an incoming quantity is advantageous for Monte-Carlo importance sampling. In the case of surfaces it is fairly easy to calculate the essential outgoing radiance from irradiance, since this means only a single reflection. On the cluster level we do not know, how to reflect the perpendicular irradiance. We would have to descend the tree everytime we wish to evaluate the outgoing radiance for a given outgoing direction. In order to speed up this calculation we store an outgoing quantity at the clusters, too. Since clusters are represented as point sources (no detail in space), we use radiant intensity I as a description of light emitted by clusters.

A C-element now contains 16 coefficients:

$e^{\perp}_0,\ldots,e^{\perp}_7$	representing perpendicular irradiance,
$h_0,\ldots h_7$	representing intensity,

Fig. 2.6 shows the root cluster and subclusters in the tree representation. Clusters contain subclusters and/or surfaces (i. e. CS-elements).

2.2.5 Shooting

Radiance and importance are defined by operator equations (2.3, page 7) and (2.7, page 8), which can be solved by shooting radiance and importance iteratively through the environment. In finite element methods an explicit representation of the transport operator $\mathscr{T}$, a formfactor matrix or a set of links, is used to account for every single transport between different subsupports of the radiance function.

Consider the formal solution of the radiance equation as in (2.5, page 7):

$$L = \sum_{i=0}^{\infty} \mathscr{T}^i L^{\mathrm{e}}.$$

From this equation we could derive the following algorithm that solves for the unknown function L. Since importance L^{imp} is handled analogously, here we discuss only the radiance case. First we initialize a data structure to represent radiance:

```
divide the support of $L$ into a finite set $\mathscr{M}$ of disjoint subsupports.
for each subsupport $m \in \mathscr{M}$
    set up an approximate representation for radiance: $\Lambda_m := 0$.
    set up an “unshot” portion of radiance $\Lambda_m^{\mathrm{unshot}} := L^{\mathrm{e}}|_m$
    set up a “next iteration unshot” portion $\Lambda_m^{\mathrm{next}} := 0$
endfor
```

Here $X|_m$ means the restriction of function X to support m.

Note, that the representations Λ_m^* need not store radiance explicitly, but may store something from which radiance is easily computed (e. g. irradiance). As a placeholder for the subsupports we could use the elements of the element hierarchy. Then $(\Lambda_m, \Lambda_m^{\mathrm{unshot}}, \Lambda_m^{\mathrm{next}})$ simply describes a single S- or C-element.

Now, during several iterations unshot portions of radiance are transported via $\mathscr{T}$ to next iteration unshot portions. A single iteration is realized as follows:

```
for each subsupport $s \in \mathscr{M}$
    for each subsupport $r \in \mathscr{M}$
        $\Lambda_r^{\mathrm{next}} += \mathscr{T}\Lambda_s^{\mathrm{unshot}}$
    endfor
endfor
```

The notation of transports of unshot radiance in the inner iteration is a bit sloppy, since $\Lambda_s^{\mathrm{unshot}}$ and $\Lambda_r^{\mathrm{next}}$ do not necessarily store radiance. But, above we already required the representation in $\Lambda_s^{\mathrm{unshot}}$ to easily allow calculation of approximate outgoing radiance on s. This outgoing radiance is propagated to r maybe attenuated by blocking objects. At r the incident radiance is processed and summed up such that it properly fits into the representation $\Lambda_r^{\mathrm{next}}$. In appendix A we show how such a transport could be calculated.

In a computer program we create a *link* between the two elements s and r, where s is called the sender and r the receiver.

At the end of an iteration the next iteration is prepared:

```
$\Lambda += \Lambda^{\mathrm{next}}$
$\Lambda^{\mathrm{unshot}} := \Lambda^{\mathrm{next}}$
$\Lambda^{\mathrm{next}} := 0$
```

In fact, we do not have a single element tree, but we need — at least theoretically[3] — three such trees for the unshot light $\Lambda^{\mathrm{unshot}}$, the next iteration's unshot light Λ^{next} and the approximate solution Λ. The line $\Lambda += \Lambda^{\mathrm{next}}$ above means that coefficients of one element representation are added to coefficients of another representation.

We stated above, that we are going to assign an incident quantity (perpendicular irradiance) and an exitant quantity (intensity) to each C-element. The exitant quantity is used only as a data source during the iteration. The incident quantity is filled up successively with contributions from other elements. Hence for clusters Λ^{next} denotes the set of coefficients $\{e_0^{\perp}, \dots, e_7^{\perp}\}$ of a C-element and $\Lambda^{\mathrm{unshot}}$ denotes the coefficients $\{h_0, \dots h_7\}$.

For S-elements we could do the same by representing unshot light by radiance and next iterations unshot light by irradiance. Unfortunately the unshot light representation in such setting tends to “smooth out” fine details of the surface's brdf. Hence we use irradiance also for the unshot light representation.

2.2.6 Hierarchical Shooting

Just as there are two constructions (standard and nonstandard) for multidimensional bases, so are there two ways to decompose a linear operator (*standard* and *nonstandard*). We will use a *nonstandard decomposition* of the transport operator as in [97] where a shooting hierarchical radiosity algorithm is described that avoids storing the complete set of links.

At the beginning a single self-link is created from the root C-element to the root C-element. Links are refined if they are likely to approximate the transport operator too coarsely. Refining a link means replacing it by links on deeper levels. Refined links are positioned between child elements of the originally linked elements.

[3] In a real implementation we would merge the three representations into a single tree.

By doing so, we ignore the transports between the elements at higher levels. We must account for these highlevel transports at lower levels. Fortunately every element contains a pure scaling function on its support, which now can be used as “proxy” for scaling functions at higher levels. In order to propagate the low-level transports through the element hierarchy, *pushing* and *pulling* is needed.

We start by initializing the representations of light and unshot light by calling the following procedure for each node:

```
**initialize($m$):**
if ($m$ is leaf) then
    $\Lambda_m := 0$
    $\Lambda_m^{\text{unshot}} := L^{\text{e}}|_m$
    $\Lambda_m^{\text{next}} := 0$
endif
```

Pulling means representing light coarsely at inner nodes that is available more detailed at lower levels. We have to pull once before the first iteration, since we need a representation of unshot light also at the inner nodes. For the root element of the element hierarchy we call the following recursive procedure:

```
**pull($s$):**
if ($s$ is inner node) then
    $\Lambda_s^{\text{unshot}} := 0$
    $\Lambda_s := 0$
    for each child $c$ of $s$
        pull($c$)
        $\Lambda_s^{\text{unshot}} += \mathscr{Q}\Lambda_c^{\text{unshot}}$
        $\Lambda_s += \mathscr{Q}\Lambda_c$
    endfor
    $\Lambda_s^{\text{next}} := 0$
endif
```

The meaning of $\Lambda_s^{\text{unshot}} += \mathscr{Q}\Lambda_c^{\text{unshot}}$ is: incorporate the unshot light represented at the child c into the unshot light represented at s. Doing this usually some kind of transformation between the two representations is necessary, which is denoted by the operator $\mathscr{Q}$. E. g. if s is a cluster element and c a surface element, then we have to transform $\Lambda_c^{\text{unshot}}$, which is represented as irradiance, to the outgoing intensity at $\Lambda_s^{\text{unshot}}$ by calculating the reflection at surface c. But also in pulling from surface to surface or from cluster to cluster we have to transform representations from one basis into another.

The following procedure is now called for the root-link (root,root) and accounts for all transports between any pair of subsupports.

```
**transport($\{s,r\}$):**
if oracle($\{s,r\}$)
    let $i$ denote the element to be subdivided, $j$ the other element, $i,j \in \{s,r\}$
    subdivide($i$)
    for all children $i.c$ of $i$
        transport($\{i.c, j\}$);
    endfor
else
    link($s,r$)
    link($r,s$)
endif
```

The decision, when to refine a link and when to allow transports (we say *establish* a link), depends on the discretization error at the receiver and the error arising from formfactor integration. Also the amount of transported light and visibility issues could be integrated into the oracle. These details are beyond the scope of this thesis. The interested reader may consult [69, 20].

The subdivide method simply subdivides a given element into subelements. The link method calculates a light transport between elements:

```
**link($s,r$):**
$\Lambda_r^{\text{next}} += \mathscr{T}\Lambda_s^{\text{unshot}}$
```

This includes the very expensive calculation of visibility between the elements.

Pushing means distributing the light received in Λ^{next} downwards to the leaves by calling the following procedure with the root element as an argument:

```
**push($r$):**
if ($r$ is a leaf)
  then $\Lambda_r^{\text{unshot}} := \Lambda_r^{\text{next}}$
       $\Lambda_r += \Lambda_r^{\text{next}}$
       $\Lambda_r^{\text{next}} := 0$
  else for each child $c$ of $r$
           $\Lambda_c^{\text{next}} += \mathscr{P}\Lambda_r^{\text{next}}$
           push($c$)
       endfor
endif
```

In the then clause the received unshot light is transferred to the unshot light for the next iteration. Since at the leaves of the hierarchy we are concerned only with equal representations for Λ, Λ^{unshot} and Λ^{next} (all represented as irradiance on surfaces in the same basis), this is a pure assignment without transformations. If r is not a leaf, transformations between parent and child elements are necessary (operator $\mathscr{P}$).

For completeness we describe the whole algorithm:

```
for each element $m$:
    initialize($m$)
endfor

while not converged
    pull(root)
    transport({root,root})
    push(root)
endwhile
```

The algorithm terminates, when it reaches an iteration where the root-cluster self-link is not refined [97].

2.2.7 Computational Complexity

The first multiresolution algorithm — hierarchical radiosity [53] — has been shown to reduce the number of links dramatically compared to the classical full-matrix methods. Hanrahan has argued that

$$\#\text{links} = O(\#\text{elements}),$$

a statement that clearly heavily depends on the oracle and other parameters of the algorithm. The above equation means that when the scene complexity is doubled, then the complexity of the solution method at most is doubled. This is a great improvement compared to $\#\text{links} = \Theta((\#\text{elements})^2)$ for full-matrix methods.

The author of this thesis has shown that the above equation holds only for a fixed resolution quality:

$$\#\text{links}(\text{quality}_{\text{fix}}) = O(\#\text{elements}(\text{quality}_{\text{fix}})).$$

Surely, when the quality of the solution for a fixed scene is required to increase, then intensified refinement of links and elements leads to higher numbers of both links and elements. In [37] we have shown for a “flatland” example scene that the relation between links and elements develops as

$$\#\text{links}(\text{quality}) = \Omega((\#\text{elements}(\text{quality}))^2) \quad \text{for quality} \to \infty.$$

2.3 Monte-Carlo Approach

In this section we discuss the basic principles of Monte-Carlo methods. More material about Monte-Carlo methods in general can be found e. g. in [63]. An overview of solutions to the global illumination problem using Monte-Carlo methods can be found in Lafortune’s thesis [67].

In this thesis the discussion of finite element approaches on parallel architectures prevails. We describe Monte-Carlo approaches here relatively detailed, because we think that our general spatial partitioning strategy is useful for both finite element and Monte-Carlo approaches, and thus also for hybrid approaches (see Sect. 2.4). Readers that are interested in finite element approaches only, may skip this and the following section and continue reading at Chapter 3.

2.3.1 Monte-Carlo Integration

Consider an univariate function $f \in \mathscr{L}^2([0,1])$. We can represent f approximately by a number of samples

$$\{(\mu_i, f(\mu_i)) : 1 \leq i \leq n\}.$$

To calculate this set we have to generate the sample points μ_i and then evaluate f n times. We can use this representation, to estimate f at an intermediate point μ not in the above set of samples, by averaging the function values of the nearby samples. Of course this approximation is valid only if f is sufficiently smooth.

A particle representation is especially useful, if we want to integrate a function. Consider the irradiance function E in equation (2.6, page 7) which is integrated over the hemisphere. A common integration technique – the Monte-Carlo integration – is completely based on function sampling.

The basic idea in Monte-Carlo integration is to approximate an integral by taking a large number of samples:

$$I := \int_{[0,1]} f(\mu) d\mu \approx \frac{1}{n} \sum_{1 \leq i \leq n} \frac{f(\mu_i)}{p(\mu_i)} =: \hat{I}.$$

Here the n samples μ_i were drawn according to a *probability density function (pdf)* p.

The problem in Monte-Carlo image synthesis is to properly choose samples. Two common approaches are *stratification* and *importance sampling*. Stratification is used to divide the space $[0,1]$ into distinct regions that are sampled independent of each other. Importance sampling tries to draw samples at "important" regions of $[0,1]$ by using a pdf that is similar in shape to the integrand f. The ultimate goal in using these techniques is to reduce the variance of the estimator $var(\hat{I})$, since this value is directly coupled with the expected error of the result. It is known that the standard deviation of the estimator is porportional to $1/\sqrt{n}$, i. e. we have to quadruple the number of samples in order to reduce the standard deviation by a factor of 2.

2.3.2 Solving The Radiance Equation

Consider the radiance equation (2.1, page 6) in directional parameterization (cf. Fig. 2.1):

$$L(\vec{x},\vec{\omega}) = L^e(\vec{x},\vec{\omega}) + \int_{H^2} v(\vec{x},\vec{x}') f_r(\vec{x},\vec{\omega}_{\vec{x}\to\vec{x}'} \to \vec{\omega}) L(\vec{x}',\vec{\omega}_{\vec{x}'\to\vec{x}}) \cos(\vec{\omega}_{\vec{x}\to\vec{x}'},\vec{n}_{\vec{x}}) d\vec{\omega}_{\vec{x}\to\vec{x}'}. \tag{2.10}$$

If we are interested in the radiance value at some specific point and direction $(\vec{x}_0, \vec{\omega}_0)$, we could estimate the integral by choosing a single random sample. We choose a random direction $h_1 \in H^2$ according to some pdf p_1, trace a ray from $\vec{x}_0$ towards that direction and arrive at a unique surface point $\vec{x}_1$. We define $\vec{\omega}_1$ as the direction from $\vec{x}_1$ to $\vec{x}_0$. Since $v(\vec{x}_0,\vec{x}_1) = 1$ we can drop the visibility factor and get a first estimator:

$$L(\vec{x}_0,\vec{\omega}_0) \approx L^e(\vec{x}_0,\vec{\omega}_0) + \frac{f_r(\vec{x}_0,-\vec{\omega}_1 \to \vec{\omega}_0)\cos(-\vec{\omega}_1,\vec{n}_{\vec{x}_0})}{p_1(h_1)} L(\vec{x}_1,\vec{\omega}_1).$$

The radiance at $\vec{x}_1$ can be estimated the same way, and we arrive at the following estimator:

$$\begin{aligned} L(\vec{x}_0,\vec{\omega}_0) &\approx L^e(\vec{x}_0,\vec{\omega}_0) + \frac{f_r(\vec{x}_0,-\vec{\omega}_1 \to \vec{\omega}_0)\cos(-\vec{\omega}_1,\vec{n}_{\vec{x}_0})}{p_1(h_1)} \\ &\qquad \cdot \left(L^e(\vec{x}_1,\vec{\omega}_1) + \frac{f_r(\vec{x}_1,-\vec{\omega}_2 \to \vec{\omega}_1)\cos(-\vec{\omega}_2,\vec{n}_{\vec{x}_1})}{p_2(h_2)} L(\vec{x}_2,\vec{\omega}_2) \right) \\ &\vdots \\ &\approx \sum_{i=0}^{\infty} \left[\prod_{j=1}^{i} \frac{f_r(\vec{x}_{j-1},-\vec{\omega}_j \to \vec{\omega}_{j-1})\cos(-\vec{\omega}_j,\vec{n}_{\vec{x}_{j-1}})}{p_j(h_j)} \right] L^e(\vec{x}_i,\vec{\omega}_i). \end{aligned}$$

The radiance at $(\vec{x}_0, \vec{\omega}_0)$ is calculated by following an infinite *path* $\vec{x}_0, \vec{x}_1, \vec{x}_2, \ldots$ that bounces randomly through the scene. The inifinite series can be cut off in an unbiased probabilistic way using the technique of *Russian roulette* [7].

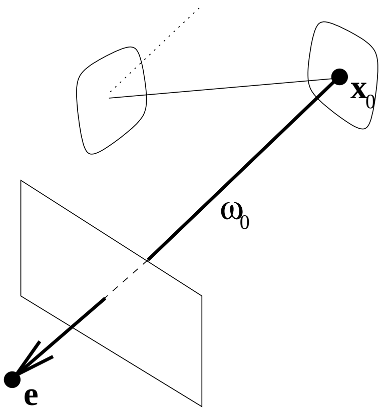


Figure 2.9: Start of the path at the eyepoint.

2.3.3 Tracing-Algorithms

Assume, the above path starts at the eyepoint. Let $\vec{x}_0$ denote a point in the scene that can be seen from the eye through the image window and $\vec{\omega}_0$ the direction from $\vec{x}_0$ to the eye (Fig. 2.9). Then we could describe the following algorithm to generate an image for a given eyepoint: shoot a single ray through each pixel; for each ray determine the first hitpoint, calculate a new ray starting at the hitpoint; follow the new ray and recurse; associate the resulting path with the pixel. For each pixel the associated path is evaluated as described in the previous section. We end up with a radiance value for each pixel — an image. This algorithm is called (backward-) raytracing.

Another possible algorithm would start at the light sources in the scene. From each light source a certain number of paths is generated. Some paths may luckily hit the eyepoint.[4] These paths are used to calculate the image. Such an approach is called (forward-) raytracing.

In this thesis we use the terms path tracing, particle tracing, and ray tracing as synonyms describing the same basic principle of tracing along straight lines through a scene either starting at a light source or at the eye.

2.3.4 A Particle Representation Of Radiance

Instead of aiming at the generation of a single image, the path calculation technique can be used to calculate a representation of the whole radiance function. Jensen [62], Pattanaik [83], Shirley [91] and others have described such methods.

Here we will describe briefly Jensen's approach, the *photon map*. The photon map represents a distribution of photons (particles) throughout the scene. It is created by emitting photons from all light sources into the scene. Each photon is traced through the scene. At the first intersection the photon's energy is stored with the hitpoint and the source direction. Ward's "real pixels" approach [103] can be used to pack a three wavelength representation of the energy into 4 bytes. By Russian roulette it is decided whether the photon is reflected resulting in a secondary photon that is traced through the scene again. The photons are stored in space in a 3-d-tree without any structural correspondence to the surface where the photon hits.

The 3-d-tree can be used in two ways. First, a backward raytracer can use the information about incident illumination to guide the spawning of secondary rays to those directions, where light is received. Second, the 3-d-tree can be seen as a representation that allows an easy approximation of the radiance function. Jensen proposes the following algorithm to calculate radiance from the photons stored in the tree. Locate the N photons with shortest distance to some point $\vec{x}$. Each photon represents a packet of energy (flux) $\Delta\Phi_j$ arriving at $\vec{x}$ from direction $\vec{\omega}_j$. Then the radiance is approximately calculated as

$$L(\vec{x},\vec{\omega}') = \int_{H^2} f_r(\vec{x},\vec{\omega} \to \vec{\omega}')dE(\vec{x},d\vec{\omega}) \approx \sum_{1\le j\le N} f_r(\vec{x},\vec{\omega}_j \to \vec{\omega}')\frac{\Delta\Phi_j}{\Delta A}. \tag{2.11}$$

As a rough but reportedly accurate enough approximation of the area ΔA the disc area of the disc with radius r is used, where r is the maximum distance between $\vec{x}$ and the N photons.

Exactly the same way, we store photons that were emitted from the light source in a photon-map, we could store so-called *importons* that were emitted from the eyepoint [84] inside an importance-map. An importance-map

[4]Of course this would be enormous luck, if we generate secondary rays at random, since then the probability to hit the infinitely small eyepoint is zero. Hence, in practice the paths are steered in a deterministic way towards the eyepoint.

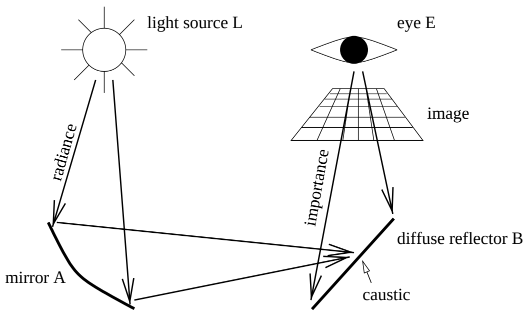


Figure 2.10: Scene with a caustic.

can be used to focus the generation of a photon-map to those regions that are most frequently evaluated during a subsequent (backward-) raytracer run.

2.4 Combined Approaches

Above we mentioned advantages and disadvantages of the two major approaches to global illumination. Finite element methods are useful, where the radiance function is smooth. Particles represent fine details more efficiently. Hence, specular reflection is best handled by particles, and diffuse reflection is the finite element's profession.

Combinations of these basic approaches have been developed, leading to hybrid or multi-pass techniques [93, 18, 96]. Hybrid methods exploit the advantages of both basic approaches. E. g. a *Lighting Network* [96] provides an infrastructure for flexibly combining specialized algorithms and adapting them to different environments. The global illumination computation is split into separate steps. In each step, one algorithm performs a part of the whole simulation process and makes the results available to other algorithms.

In this section we will discuss two examples, where a combined approach using particles and elements could be useful. The whole algorithm is sketched only blurry, since hybrid methods are not yet established and are subject of ongoing research. The purpose of this section is to convince the reader, that hybrid methods are eligible candidates for global illumination. Having seen this, it is clear that a parallelization should regard both particle and element methods equally.

Consider a scene with both highly specular and diffuse surfaces (Fig. 2.10). In finite element methods, the integral of a transport coefficient (cf. appendix A):

$$T_{ij}^{rs} = \int\int_{\text{support}(r)} \tilde{N}_i^r(\cdot)\cos(\cdot)v(\cdot)\int_{H^2} f_r(\cdot)N_j^s(\cdot)\,d\vec{\omega}'\,d\vec{x}\,d\vec{\omega}_{\vec{x}\to\vec{x}'}$$

between two far distant diffuse surfaces can be approximated confidently by only few samples. If instead f_r has large gradients, because it describes a specular surface (e. g. mirror A in Fig. 2.10), then we would refine the link between A and B recursively into thousands of sublinks. When we had a particle representation of radiance at the mirror A, then maybe we would simply shoot the particles towards surface B, saving lots of memory. In the proposed algorithm particles and elements should be kept simultaneously in memory, each describing distinct parts of the radiance function. They should be organized such that they can be converted into each other, depending on which representation is useful for a particular transport.

Another example, where a combined particle and element respresentation is useful, is the simultaneous generation of an image while the global illumination calculation proceeds. Importance is emitted as particles (a handful for each pixel) from the eye. Each particle carries the index of the corresponding pixel. The particles are converted to elements after their first reflection at diffuse surfaces. Inside the elements a list of corresponding pixel indices is stored. The importance information is exploited during the global illumination computation, which goes on simultaneously, to restrict the refinement of links to important regions. At the end, when the computation has converged, all scattered importance elements and particles are collected and their contribution is added directly to their corresponding image pixels.

One unclear point in this algorithm is, how imprecise importance information can be used in early iterations of a shooting algorithm to guide the illumination computations to important regions, but this is beyond the scope of this thesis.

Chapter 3

Models In Parallel Computing

Contents

We are going to describe algorithms for parallel processors. First, we will look at the various machines available for parallel computing. In order to be not restricted to a specific machine, we will abstract from the specific features by describing the *machine model*, which we are operating on.

For the description of a parallel algorithm we need a model with concepts for communication between processors. We will describe various *programming models* below.

In order to analyze the performance of our algorithms we need a *cost model.* This model specifies the architecture of a parallel system by some parameters.

3.1 Machine Model

This section gives an overview of machine models for parallel computers. The terminology is taken from [34].

Multicomputer. A *von Neumann computer* consists of a central processing unit (CPU) and a storage unit (memory).

A *multicomputer* comprises a number of von Neumann computers linked by an interconnection network (see Fig. 3.1). Each computer (or *node*) executes its own program and accesses local memory. Messages are used to communicate with other computers. In the idealized network, the cost of sending a message between two nodes is independent of both node location and other network traffic, but does depend on message length.

The difference between the above multicomputer and the *distributed-memory MIMD computer* (multiple instruction multiple data) is that in the latter, the cost of sending a message between two nodes may not be independent of node location and other network traffic. Examples of distributed-memory MIMD computers are IBM SP, Intel Paragon, Thinking Machines CM5, Cray T3E, Meiko CS-2, NEC Cenju 3, and nCUBE. Also a cluster of workstations connected by a local area network may be regarded as a DM-MIMD computer.

Multiprocessor A *multiprocessor* (or *shared-memory MIMD computer*) consists of a number of processors that access a common memory.

An idealized model for theoretical considerations is the *PRAM model* (parallel random access machine), where each processor can access any memory element equally efficient (see Fig. 3.2). Concurrent memory accesses may be allowed depending on the actual model used (for details cf. [61]).

In reality caches or hierarchies of caches introduce different costs to different memory accesses. Then locality of memory accesses is an important issue. Examples of a multiprocessor are the Silicon Graphics Origin and the Sun Enterprise.

A hybrid model, the distributed shared memory (DSM) computer, provides a common memory address space but distributed memory modules. The compiler and the runtime system are responsible for parallelising an application. Hence the system software is relatively complex.

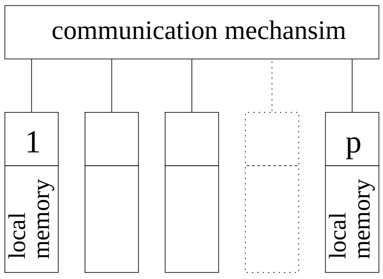


Figure 3.1: Multicomputer model.

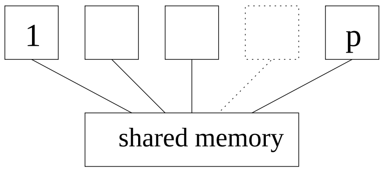


Figure 3.2: PRAM model.

SIMD computer In a *SIMD computer* (single instruction multiple data) all processors execute the same instruction stream on a different piece of data. Examples are the MasPar MP, and Thinking Machines CM2.

The multicomputer is a model of a well scalable architecture. Also low-price parallel computing on workstation clusters is covered by this model. Therefore we decide to use the multicomputer model.

3.2 Programming Model

Parallel programs are implemented in a programming language such as C or Fortran. Parallel constructs are either incorporated directly into the language or may be linked to the code as a library. The *programming model* describes the style of parallel programming. We distinguish the *data parallel*, *shared memory* and *message passing* models. There are other programming models (functional parallelism, etc.) possible [34].

Data parallel programming The *data parallel* model originates from programming synchronous SIMD parallel computers. The programmer implements one program that is executed separately on each processor. Each processor has its own local memory. The programmer defines, how data is to be partitioned into local parts. Applications that massively employ conditional execution are hard to implement efficiently on a SIMD computer.

Shared Memory Programming The *shared memory* model is a simple model, because the programmer can write a parallel program just like a sequential one. A program is executed independently on several processors. Synchronization is not needed. A processor may execute statements independently of the state of other processors.

All running programs access the same global memory. Care must be taken, if a processor wants to access data exclusively without interference of other processors. A program must mark such critical sections with special statements.

Message Passing Here we will use the *message passing* model in combination with the *SPMD implementation technique* (single program multiple data). A single program is executed independently on several processors. There is no need for synchronization (MIMD system). Every program instance accesses a local memory. Remote memory accesses are possible by sending messages. There are different kinds of communication:

point-to-point communication: a processor sends a data package to another processor, using a *send*-operation. The destination processor must call a *receive*-operation in order to get the data. We assume that a processor can communicate with only one other processor at the same time. The communication may be asynchronous, i. e. the receiving processor may call *receive* at any time after the send-operation.

collective communication: collective communications involve a group of processors.

cast: a single processor sends copies of the same data package to a group of processors.

combine: each processor of a group contributes a data item. A single *root*-processor receives the result – a single data item calculated out of the contributed data items (e. g. the sum or the maximum of real numbers).

Collective communications on m processors can be simulated by many point-to-point communications in $O(\log(m))$ parallel time in a tree-like manner, if we assume that the communication channels between each pair of processors are independent.

Language Constructs There are efforts in extending sequential languages for parallel computing. E. g. HPC++ [58] or HPF [59] try to implement various programming models into the languages C++ or Fortran. One possibility to integrate parallelism into a language is by providing a library with routines for parallel processes and communication. The more challenging and more problematic possibility is to extend the language by some new language elements and keywords.

The process of designing such parallel extensions is in a preliminary state. Employing such standards when established presumably is wise. But even then we are not freed from the need to decide on a programming model, since many different models will be incorporated in such standards. In the light of these facts it will not be a restriction to use message passing as a programming model. Future developments in the area of parallel language extensions probably will not cause us to revise this decision.

All programs presented in this thesis are written in the message-passing model. There has been much effort to define a standard in message-passing over the last years – the *Message-Passing-Interface* (MPI) [77, 76]. There are efficient MPI implementations for many current supercomputers. Also there is a freely available, high-performance, portable implementation of MPI called *MPICH* that runs on workstation clusters, too [79, 49]. Thus, our MPI-based implementation is easily portable to many parallel environments.

3.3 Cost Model

After describing a parallel algorithm in the message passing model we want to analyze the computational complexity of the algorithm. Unfortunately, the large number of different supercomputers makes it nearly impossible to predict the behaviour of the algorithm exactly. Therefore we have to specify a cost model as a basis of our analysis. Clearly the model should simulate current architectures as close as possible. There have been efforts to define universal models that allow exact prediction of complexity of a parallel algorithm, regardless of the programming model or hardware used. For an overview see [72].

Our machine model is the multicomputer. This model consists of p independent processors, each accessing its own local memory. The processors work asynchronously and communicate through a network that connects each pair of processors by a bidirectional channel of communication. Communication happens in a *point-to-point*-fashion, i. e. a processor sends a data package to another processor, using a *send*-operation. The destination processor calls a *receive*-operation in order to get the data. There is a vast number of possibilities to measure the costs of a point-to-point communication. In the following few of them are introduced.

Real supercomputers with many processors cannot offer a dedicated communication channel for each single pair of processors. Instead, processors share channels. Message congestions will occur, when two processor pairs try to communicate at the same time over the same channel. The approaches of cost measurement below do not consider this problem, since it depends on the specific topology of the network. But we will discuss some general rules for avoiding congestion in Sect. 4.1.3.

There have been considerations to incorporate another class of communication as atomic operations into distributed memory models, the so-called *collective communications* that involve groups of processors. Clearly such communications can be simulated by many point-to-point communications. We will only use these point-to-point communication, since we do not want to specialize in hardware which permits more efficient collective communications.

The postal model The *postal* model [10] is motivated by the communication between people where it is assumed that the only way for people to contact each other is by sending and receiving letters. The letters are picked up regularly by the postal service and are delivered to their destinations after some delay. This letter-model has two inherent features: first, it connects people completely and second, it creates communication channels of roughly the same delay.

The postal model incorporates a *latency*-parameter $\lambda \geq 1$ that measures the inverse of the ratio between the time it takes an originator of a message to send it and the time that passes until the recipient of the message receives it. More specifically, let us assume that an originator i of an atomic message M spends 1 unit of time preparing and sending M to its destination. After that i is free to perform other functions including sending other messages. Later M arrives at its destination j and the recipient spends 1 unit of time receiving and handling it. The communication latency λ is the total amount of time that passes from the time the originator i started preparing M until the time the recipient j finished handling M.

The postal model is a model of communication with no concepts for modeling computation. The following model introduces *time steps* to measure computation time against communication time.

The BSP model A *bulk-synchronous parallel (BSP) computer* [99] consists of the following:

- a set of identical sequential processors with local memory
- a communication system that delivers messages in a point-to-point manner
- a mechanism for globally synchronising the processors.

The model assumes that any parallel computation can be divided into sequences of steps, called *supersteps*. During each superstep, each processor has to carry out computation steps on values held locally and communication steps, i. e. send and receive operations. Each superstep is followed by a barrier synchronisation after that all communications are guaranteed to be completed. There are no specialised combining or broadcasting facilities.

We define a *time step* to be the time required for a single local operation. A BSP cost model takes the following system parameters:

p: number of processors

l: synchronisation periodicity, i. e. minimal number of time steps between successive synchronisation operations

g: the ratio of (total number of local operations performed in one second) / (total number of words delivered by the communications network in one second)

The parameter g corresponds to the frequency with which non-local memory accesses can be made. A machine with higher value of g should make remote-accesses less frequently. The parameter l is also called *latency*.

The complexity of a superstep in a BSP algorithm is determined as follows. Let the work w be the maximum number of local computation steps executed by any processor during the superstep. Let h be the maximum number of messages sent or received by any processor during the superstep. In the original BSP model, the cost of the superstep is $\max\{w+g\cdot h, l\}$ time steps. An analytically more tractable upper bound of this is $w+g\cdot h+l$, which often is used in predicting the costs of a BSP algorithm.

One disadvantage of the BSP model is that processors synchronize only between supersteps. A message sent at the beginning of a superstep can only be used in the next superstep, even if the length of the superstep is longer than the latency. Another drawback is that the model does not charge overhead for a message to be injected into the network. These deficiencies are recovered by the following model.

The LogP model In the *LogP* model [25] the processors work asynchronously, and a processor can use a message as soon as it arrives. Processors communicate by point-to-point messages. The model specifies the performance characteristics of the interconnection network, but does not describe the structure of the network. The main parameters of the model are:

p: number of processors. Unit time is assumed for a local operation (a *cycle*).

L: an upper bound on the *latency*, or delay, incurred in communicating a message containing a single word from its source to its target.

o: the *overhead* of transmission or reception of a message; during this time the processor cannot perform other operations

g: the *gap*, defined as the minimum time interval between consecutive message transmissions or receptions at a processor. The reciprocal of g corresponds to the available per-processor communication bandwidth.

The parameters L, g and o are measured as multiples of the processor cycle. L is an upper bound on the latency, i. e. particular messages may travel faster. The basic model assumes that all messages are of small size.

Sending a small message between two processors takes $o+L+o$ cycles: o cycles on the sending processor, L cycles for the communication latency, and finally another o cycles on the receiving processor. Sending a k word message is realized by sending k single word messages, which would take $o+(k-1)\cdot\max\{g,o\}+L+o$ cycles.

The LogGP model An extension of the LogP model – the *LogGP* model – concerns realistic modelling of sending long messages [4]. A new parameter characterizes the *gap* for long messages:

G: the *Gap per word* for long messages, defined as the time per word for a long message. The reciprocal of G characterizes the available per processor communication bandwidth for long messages.

g: the *gap* for messages, defined as the minimum time interval between consecutive message transmissions or receptions at a processor. The reciprocal of g corresponds to the available per-processor communication bandwidth for short messages.

Under the new LogGP model sending a k word message takes $o+(k-1)\cdot G+L+o$ cycles. The sending and receiving processors are busy only during the o cycles of overhead. The rest of the time they can overlap computation with communication. If a processor wants to send two long messages in a row it has to wait g cycles after the first message goes out before it can push the first byte of the second message into the network.

Our model The LogP model is a widely accepted model. Merely the sending of long messages is not modeled appropriately. The LogGP model is a little bit complicated, because it introduces two gap parameters. We are aiming at estimating the overhead of a dynamic load balancing algorithm in a more qualitative than quantitative way. For instance we will state an overhead result in Chapter 7 partly using big-oh notation. For such results the distinction between short and long messages is by far too pedantic. To simplify things we will assume that the gap for long and small messages is identical ($g=G$). Now sending a k word message takes $o+(k-1)g+L+o$ cycles.

In contrast to the LogGP model we assume that the sending and receiving processors both are busy during $ov(k)=o+(k-1)g$ cycles. This is reasonable, since setting up a long message takes longer than setting up a small message. The LogGP model does not include the time for message setup, since it aims at measuring only the cost of communication. Later in this thesis we will analyze the total overhead of a parallel algorithm, which surely is larger when longer messages are communicated.

The computer programs written in the context of this thesis all are designed as asynchronously communicating SPMD-style programs. There are only a few global synchronisation points. Between these points all processors are kept busy by dynamic load balancing procedures. In such programs latency is almost totally hidden by computation. Hence we will ignore the latency in our theoretical complexity consideration in Section 7.2 and only consider the overhead $ov(k)=o+(k-1)g$.

Chapter 4

Partitioning And General Solutions

Contents

In this chapter we discuss the general graph embedding problem that needs to be solved, when a sequential algorithm is parallelized. A simpler but nevertheless complicated problem is the graph partitioning problem. A solution to this problem can be characterized by some formal measures like load balance and cutsize, which are defined in this chapter.

Since the graph partitioning problem does not exactly describe all sources of overhead in a parallel implementation, we need to discuss the influence of the remaining sources on a more informal level. This is done in Sect. 4.1.3 below, where we present the main incentives that must be followed when aiming at a well scalable parallelization.

In Sect. 4.2 we briefly review some strategies that have been proposed in the literature to solve the graph partitioning problem in a universal fashion. We distinguish the embarassingly parallel problem type, the general type, and the spatially mapped graph type. Especially the latter type is interesting in our context, since global illumination algorithms can be formulated conformably to spatially mapped graphs.

4.1 The Problem

4.1.1 The Graph Embedding Problem

If we want to execute an algorithm on parallel processors, we have to solve the *graph-embedding* problem [78]. Let H denote the host graph, which describes processing nodes and communication channels. A guest graph G expresses the communication requirements between subprocesses of an application. An embedding of the application (G) into the architecture (H) leads to a parallel algorithm. Every processing node executes a certain set of application subprocesses. This approach sometimes is called the “owner computes”-rule.

In order to perform best, we must embed G into H thereby minimizing costs such as load, dilation, and congestion.

Load. Obtaining an equal load on all processors is the most important goal. The total runtime of an application depends on the runtime of the processor with maximum load.

Dilation describes the period of time (or better the length of the path) needed for any message travelling over edges of H that is communicated between two directly connected nodes of G. New interconnection architectures result in weaker demands on a good dilation.

Congestion measures the maximum number of edges from G routed via an arbitrary edge in H. Congestion is important from a practical point of view. But only few work has been done on mapping techniques considering congestion.

4.1.2 The Graph Partitioning Problem

Graph partitioning is a possible way for an approximate solution of the graph-embedding problem. Partitioning is the task of calculating an evenly balanced clustering of an application graph into p clusters while minimizing the *cut-size*, i. e. the number of edges crossing the cluster boundaries. Mapping of clusters to processors is not considered. Hence, the cost of dilation is considered only rudimentary. Congestion is fully ignored.

Bokhari has reduced a version of the partitioning problem (which he calls the *mapping problem* [14]), where the number of nodes of G is not greater than that of H, to the graph isomorphism problem – a problem, for which exact, polynomial time algorithms are not known although numerous researchers have attacked this problem. Heuristics are used in practice to compute a fairly balanced partition with a cut size as low as possible.

Matters get worse, if we assume that G is dynamically manipulated. Then we have to dynamically adapt an initial partitioning.

Formally the graph partitioning problem is stated as follows. Let $G = (V,E)$ denote a graph with vertices $V = \{v_0,\dots,v_{n-1}\}$ and undirected edges $E = \{e_0,\dots,e_{m-1}\}$. Each vertex v gets $h \in \mathbb{N}$ vertex weights $(W_i(v))_{0\le i\le h-1} \in \mathbb{R}^{+h}$. Every edge e has a unique edge weight $W(e) \in \mathbb{R}^+$. The weight $W_i(M)$ of a set M of vertices/edges is defined as the sum of the weights of the contained vertices/edges.

A *partition* of a graph G among p parts $V_0,V_1,\dots,V_{p-1}$ is defined by

$$\pi : V \to \{0,1,\dots,p-1\} \ .$$

The major characteristics of a partition are its balance and its cut size.

The *load balance* of the partition π is an h-tuple:

$$lb(\pi) = \left(p \max_{0\le j\le p-1} \left\{ \frac{W_i(V_j)}{W_i(V)} \right\} \right)_{0\le i\le h-1} \ .$$

The partition is *load balanced*, if $lb(\pi) = (1,\dots,1)$.

The *cut size* of the partition is the weight of all edges running across partition boundaries, i. e.:

$$cs(\pi) = \sum_{\{v,w\}\in E,\pi(v)\ne\pi(w)} W(\{v,w\}) \ .$$

The task of the partitioning problem is, to find a balanced partition that minimizes the cut size. Sometimes a perfectly balanced partition does not exist, then we seek for a partition that is as balanced as possible.

4.1.3 Communication And Congestion

Once an application has been partitioned, the cost models of Sect. 3.3 may guide us to invent an efficient way to communicate between processors. The particular incentives are:

(a) prefer local memory references,

(b) overlap communication with computation (*latency hiding*),

(c) use few large messages instead of many small messages.

The above cost models and also the graph partitioning approach treat the problem of avoiding traffic congestion to a limited extent only. The bandwidth parameters have only local impact in a way that a processor should send messages rarely since local bandwidth is limited. If a processor had a global view over the network traffic it might decide not to send a message when global network traffic is high. Since the processors do not know about global traffic, one could employ a strategy where messages are sent asynchronously and independently since then with high probability no congestion will occur. However, a synchronous communication strategy in phases could make use of efficient collective communications such as broadcasts or combinings. Which strategy is best depends on the amount of traffic for a specific problem as well as on potentiality to use collective communications.

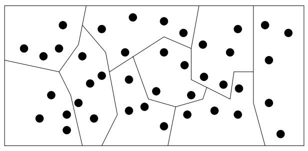


Figure 4.1: Spatial clustering of tasks.

Even if asynchronous message passing is employed there will occur congestion if the communication need is high. Congestion occurs because of the invalid assumption of totally connected processors. Clearly, in a communication network that connects each pair of processors directly, any two messages M_1 and M_2 may travel independently of each other without interference. However, in a sparse network there is a real chance of interference. Since we want to describe portable algorithms, we do not assume a specific network topology. Instead we aspire at an algorithm that uses only a small number of links of the assumed complete network simultaneously. Thus, we add the following incentive:

(d) use only few network links at the same time.

For a general application it is difficult to meet this requirement in a universal fashion. Fortunately many applications do not show totally unstructured communications. E. g. domain decomposed calculations such as computational fluid dynamics or finite elements define local dependencies between tasks that are located in space. Also Radiosity and Raytracing can be formulated in a way that only local communication is necessary (see sections 5.1.3 and 5.2.1 for details).

4.2 General Solution Approaches

Before inventing a new partitioning strategy we should briefly discuss some existing general solution approaches to the partitioning problem that can be found in the literature.

4.2.1 Embarassingly Parallel

The most simple application type is the *embarassingly parallel* type of applications, where little or no interdependency between subproblems is present [2, 71, 1]. In that case overloaded processors are instructed to move some of their tasks to randomly chosen or neighbouring underloaded processors. The choice of the destination processors is not affected by any considerations about the interdependency of the tasks.

Physically based rendering is by no means an embarassingly parallel application. There are many dependencies between subproblems.

4.2.2 Multi-purpose Graph Partitioning

There is a vast literature on solutions for the graph partitioning problem reaching from the pioneering work in [66, 33] up to software packages including fast multi-level methods [56, 55, 102, 65, 64, 86]. These methods are ideally suited to the static load balancing problem, where the workload is static, hence the tasks have to be distributed only once across the processors before the computation starts. If the workload varies over time usually first an initial partitioning is computed that then is rebalanced adaptively if the balance gets bad [107, 100, 27]. During the adaptive partitioning one has to take care that the amount of data to be moved is small.

In the past, we tried already to tackle the hierarchical radiosity algorithm using a general graph partitioning approach [36, 45, 13]. The results have been quite frustrating[1]. One possible reason for this is, that the formulation of the graph partitioning problem does not account for the cost of possible congestion. Spatial clustering methods are capable of reducing the chance of congestion.

4.2.3 Spatially Mapped Graphs

Many scientific applications define application graphs whose vertices can be mapped naturally into a multidimensional space. Applications are called to communicate *locally* if only a small subset of all objects, e. g. the marginal objects of an object-cluster, have to communicate with other clusters. In such a situation, building spatial clusters

[1] . . . which drove us to rethink the problem from scratch.

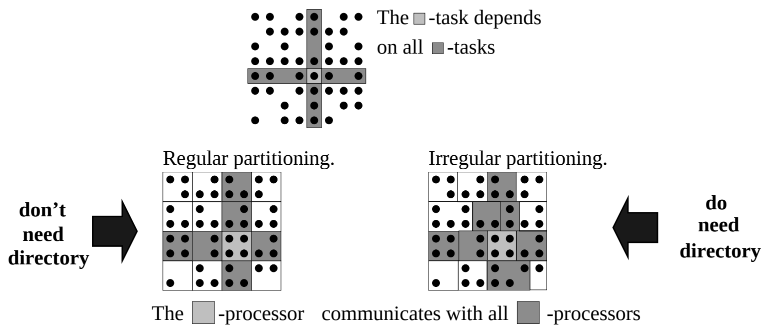


Figure 4.2: “Locally” communicating tasks in 2D.

is a good strategy (Fig. 4.1). In order to minimize communication one would use a cluster shape that minimizes the ratio of surface to volume. The ideal shape would be a hypersphere, but unfortunately it is not possible to overlay a bounded space completely by non-overlapping hyperspheres.

If the application needs to perform long distance remote data accesses, then a compact representation of a directory telling which processor hosts which tasks is needed, that can be replicated on each local memory. Long distance remote data accesses do not contradict the classification of an application as communicating locally. For example an application with tasks mapped to 2D space and each task communicating with the tasks sharing a common coordinate (see Fig. 4.2 top) is communicating locally, since any task depends only on a small subset of all tasks. The bottom of Fig. 4.2 shows a regular partitioning of the tasks, where the index of the processor hosting a specific task can be simply calculated. In an irregular partitioning, a directory, which tells the processor index of a given task, is useful. Unfortunately, generally we can reach a good load balance only by an irregular partitioning.

Diffusion methods [54, 108] remap tasks while keeping adjacent tasks on neighbouring processors. Neighbouring processors exchange load depending on local load differences. Because of this locality, the surface of clusters gets jaggy making a compact cluster representation nearly impossible.

Orthogonal recursive bisection methods [12, 17] have a compact tree representation. Rebalancing can be performed asynchronously and independently in subtrees (e. g. [70]). To minimize the ratio of surface to volume, a multidimensional bisection strategy is superior to a onedimensional clustering, which would produce thin long “slices”.

Rectangular grid clusterings as in Fig. 4.2 (bottom/left) are often used for matrix-vector-multiplication [57]. Here an exact load balance cannot be achieved by simply moving cutting hyperplanes. Instead the tasks must be permuted randomly in order to achieve balance with high probability [80].

Mapping multidimensional spaces onto a space-filling curve and then partitioning the resulting “pearl necklet” by a onedimensional procedure seems to be a straightforward method to overcome the problem of long, thin clusters. One problem with this are jaggy cluster boundaries, which make statements on the worst case communication complexity of the underlying application difficult.

Chapter 5

Partitioning Global Illumination Algorithms

Contents

In the previous chapter we reviewed very general partitioning strategies for any imaginable algorithm. In this chapter we are going to present partitioning strategies of the two major global illumination approaches, finite elements and Monte-Carlo. The description primarily focuses on the finite element approach. Besides reviewing existing strategies the main issue is the explanation of a spatial partitioning technique for each of these two algorithms. These techniques fit in the context of each other, making a joint utilization for upcoming combined particle-element-algorithms potentially possible.

5.1 Partitioning The Finite Element Approach

In order to examine the structure of the Finite Element approach, we split the algorithm into different tasks and describe the dependencies between these tasks. The result is a *task access graph*, amenable to further analysis by graph partitioners. Vertices of the graph are links and elements, while edges exist between links and elements and between elements.

This graph can be mapped to parallel processors by assigning spatial locations to its vertices. We show, why choosing a spatial partitioning approach seems to be a reasonable choice for radiosity algorithms.

There exist a few implementations of the hierarchical radiosity algorithm [53] — a special form of a finite element method — on distributed memory (DM) architectures. We will first dicuss these implementations in the following section.

5.1.1 Related Implementations

Excellent survey papers about parallel rendering can be found in [24, 87]. Here we will focus on hierarchical radiosity algorithms on DM architectures and on parallel solutions that employ a spatial partitioning technique.

5.1.1.1 Hierarchical Methods

Only a few implementations of the hierarchical radiosity algorithm (HRA) on distributed memory environments have been reported to date:

- The probably first DM-implementation of the HRA was undertaken by Carter [16] on an nCube 2.
- Singh et al. [95] report briefly on "exercises in frustration" when implementing a message passing HRA on an Intel iPSC/860, whose implementation is described in [94] in greater detail.
- Bohn and Garmann [13] described an implementation on a CM5.
- Zareski [110] had bad experiences with a PVM implementation on an IBM SP-1.
- Funkhouser [35] and
- Fellner et al. [31] described implementations for a network of workstations (NOW).
- Feng et al. [32] acieve good speedups for moderately complex scenes on a small NOW with a fine granular adaption of [35]
- Benavides et al.[11] implemented a simple mapping strategy on a Cray T3D
- Meneveaux and Bouatouch [74] describe a wavelet-radiosity implementation on a NOW for large scenes very similar to [35].

The implementations [110, 35, 31, 32] follow a master-slave approach, [16, 94, 13, 11, 74] are in SPMD style (*single program multi data*) with no dedicated master process, which could potentially limit scalability.

In a DM implementation of the HRA there are basically three problems to be solved. First, tasks need to be distributed such that each processor gets an equal amount of work (*functional parallelism*). Second, data needs to be partitioned such that local memories are utilized in a balanced way (*data parallelism*). And third, *communication* should be low, both by clever data/function parallelism and by reuse of communicated data using caching techniques and temporal locality. We now discuss solutions to these aspects found in the above implementations.

Functional Parallelism. In radiosity algorithms the overwhelming part (about 90%) of the calculation are formfactor calculations. In [110, 31] only these formfactor/visibility calculations are distributed among a couple of slaves while performing the solution part of the HRA serially on a master. The speedup of such approaches never will exceed 10.[1] The other implementations also distribute tasks of the solution part among processors. Funkhouser's (and Meneveaux's) approach is something in the middle, since his group iterative solver performs several solution steps on the slaves; only a few solution tasks are performed serially on the master by merging slave solutions together.

In order to achieve functional balance in the master-slave approaches a scheduling problem has to be solved, while in the SPMD style implementations load balancing must be performed. Funkhouser [35] achieves functional balance by combining the techniques of a first-fit-scheduler and of a scheduler aiming at small changes in the working set of a slave. Meneveaux et al. [74] use a graph partitioner that groups elements based on their distance to a center-of-gravity, combined with a task stealing strategy. A very similar strategy – a combination of a static ordering and dynamic task stealing – is described in [32]. Fellner et al. [31] use a simple FCFS-scheduler. Singh [94] used a demand driven load balancing approach (task stealing), but did not get satisfying results for a DM environment. The implementations [13, 16] assume that large data items are associated with complex tasks and that functional balance is good if data balance is. In [11] the area of surface elements is taken as a criterion of complexity for a demand-driven mapping in the first iteration.

Data Parallelism. The large data structures in the HRA are the element hierarchy, a tree containing geometry data for visibility calculations (which may or may not be distinct from the element hierarchy), and the set of links.

In all above implementations the visibility-tree is distinct from the element hierarchy. We may replicate all geometry on each local memory [16, 94, 13, 31, 11], since this data is much smaller than the remaining data structures. Zareski [110] tried to partition the visibility data structure, which led to so large communication overhead that no speedup resulted at all. In [35, 32, 74] so-called *working sets* are loaded to each slave, which contain all geometry that is potentially visible from a given target group of elements. In scenes with dense visibility graphs this strategy will reduce to the "replicate-all-geometry" approach. The use of working sets is "especially useful for building interiors with many rooms and corridors but not for a simple room with objects of highly detailed geometry" (cited from [74]).

[1] Speedup = $\frac{T_1}{T_p} \leq \frac{T_1}{0.1T_1+0.9\frac{T_1}{p}} < \frac{T_1}{0.1T_1+0} = 10$, where T_i = runtime on i processors.

The element and link data structures are mapped permanently to processors only in the SPMD style implementations [16, 94, 13, 11, 74]. The master-slave programs assign elements and links temporarily to processors during scheduling. In [16] data balance was achieved by randomly shuffling links between processors periodically. In [13] all data/tasks were represented by a undirected graph. On this graph the graph partitioning problem was solved by *simulated annealing*, thereby reducing communication and maintaining data balance. In [11] data balance is not explicitly accounted for. An initial data balance is provided in [74] by graph partitioning, which is dynamically adapted by task stealing.

Reducing Communication By Clever Partitioning. One important point is to reduce and to balance the communicated data per processor. But structuring communication to few communication channels (see Sects. 4.1.3, 6.6) should also be done in order to reduce the chance of congestion. In Carter's approach [16] element hierarchies were decomposed such that subtrees below some fixed level were mapped entirely to the same processor, which leads to low communication during the push and pull phases. Links were mapped irregularly and independently of elements. Link refinement showed scalable behaviour, but during shooting the communication time exceeded by far computation time. This was mainly due to the high volume random-all-to-all communication structure, which originates from the irregular partitioning of links.

Bohn et al. [13] used the general graph partitioning principle, which can reduce the communication overhead while maintaining functional and/or data balance. Both the overhead of the graph partitioner and the unstructured communication pattern reduced the efficiency of this approach significantly. Singh [94] used unidirectional links that are indivisibly coupled with their source element. The elements themselves are distributed randomly. Hence, as all the other implementations, also this approach ignores the aspects of structured communication and congestion.

Reducing Communication By Reuse. Singh [94] discusses two caching approaches. "Local quadtrees" creates huge caches for all elements needed during a single iteration and exchanges element data only between iterations. In this approach cached data is likely to be outdated, when used late inside an iteration. "Global quadtrees" manages smaller and more up-to-date caches. A disadvantage of this approach is that messages are communicated at a much finer granularity. Even worse, in [16, 13] every link task that needs remote element data sends individual request messages. Hence, remote element data is cached only during the calculations regarding a single link. The data is not reused. Funkhouser [35] (and also Meneveaux [74] and Feng [32]) reduces communication by carefully scheduling element groups to slaves, such that working sets change only slightly. As a result data downloaded once may be used for several target groups.

We may conclude from the above discussion that up to now there are no clearly perferable techniques that make up a good and largely scalable DM implementation of HRA. Caching techniques and high quality functional load balancing seem to be important. Also a well devised partitioning strategy that prevents many communications alone by its structure is required. Because of missing consensus about these questions, in Chapter 6 we analyze HRA on a more abstract level.

5.1.1.2 "Flat" Methods

The parallelization of the "flat" (or *classic* or *full-matrix*) radiosity algorithm was subject of many papers in the past. We focus on a few that used spatial partioning techniques.

Virtual walls have been used to subdivide the scene into several sub-scenes. The virtual walls act as translucent boundaries and are considered as additional surfaces of the scene database that exchange light between the sub-scenes. Arnaldi et. al. [6] and van Liere [101] made first experiments with this technique. A problem is that the virtual walls introduce error in the radiosity computation, which can be reduced by subdividing the walls into small patches.

Menzel [75] proposed virtual walls as a general tool. The user or programmer defines cells and exchange mechanisms between the cells. Dynamic load balancing was performed by heuristics such as rotation or cell splitting. Experimental but no theoretical results on the computational overhead have been reported.

5.1.2 The Algorithm's Graph

We define a graph, whose vertices and egdes describe tasks and task dependencies. This graph is intended for being analyzed with respect to the communicative behaviour of the Hierarchical Shooting algorithm of Sect. 2.2.6.

We identify the following distinct tasks performed during a run of the Hierarchical Shooting algorithm:

Oracle: Use data of two associated elements to estimate the error across an interaction.

Subdivide: Subdivide an element.

Link: Calculate a high quality approximation of a transport between two elements. Sum up a contribution from one element to another element. This task writes data to an element in contrast to "Oracle", that reads only element data.

PushPull: Propagate light through the element hierarchy from a parent to its children and back.

An additional task may be identified, that is hidden in the tasks Oracle and/or Link, but should be considered separately because of its large communication demands (see below):

Visibility: Estimate and/or calculate the visibility between two elements.

In the algorithm we basically manipulate two data structures: *elements* and *links*. Links need not be stored explicitly, but usually exist as temporary containers of visibility and other useful information regarding two connected elements.

We can naturally map the above tasks to these data structures in a manner that maximizes locality, i. e. we map a task to the data structure where most of the data needed by the task is located. The tasks Oracle, Link, and Visibility are mapped to the corresponding link. The tasks PushPull and Subdivide are mapped to the corresponding parent element.

Of course this mapping does not eliminate the need for communication between the tasks. Oracle for instance needs to access the data of the associated elements. PushPull communicates with the PushPull task of the children and with that of the parent. Even worse, Visibility may need to access a large part of the geometry, resulting in communication with many element tasks, if the geometry is not replicated separately on each processor.

We will subsume the tasks Oracle, Link, and Visibility in a super-task called *Link*-Task. The tasks PushPull and Subdivide are pooled in an *Element*-Task.

Each vertex of the *task graph* to be defined consists of a data structure and its associated task. The data structure represents either a single element or a single link. The vertices of the graph are classified into element and link vertices. The vertices each get two weights. Element vertices get the weight tuple (0,1), and links the weight (1,0). This allows load balancing of both elements and links simultaneously, as is discussed in greater detail below.

Edges between vertices exist in case of dependencies between tasks, i. e. either between links and elements or between elements. Everytime when one of the following actions occurs in the algorithm of Sect. 2.2.6, then a corresponding edge weight is increased by 1:

- pull(c) [called within pull(s)]: increase weight of edge between element s and child element c.
- oracle({i,j}), link({i,j}): increase two edge weights between the link vertex (i,j) and the element vertices i and j. Additionally increase the edge weights between the link vertex (i,j) and all elements, that are visited during visibility calculations. When using the cluster hierarchy for visibility calculations, usually not all primitives need to be checked, if they block the transport between i and j.
- subdivide(i): increase the edge weights between i and its child elements.
- push(c) [called within pull(r)]: increase edge weight between element r and child element c.

Now, the graph is defined completely. We will use this graph later in Chap. 6, where we analyze the partitionability of hierarchical radiosity using graph partititioning techniques.

5.1.3 Partitioning Strategy

In this section we describe a new spatial partitioning strategy of the hierarchical shooting algorithm. We will use this strategy later in Chapter 9 in an efficient implementation of hierarchical radiosity.

Why do we actually define a new partitioning strategy, instead of applying a general purpose graph partitioner to the graph just defined in Sect. 5.1.2? There are two reasons for this. First, a graph partitioner would need unavailable a priori knowledge about the graph that evolves during the run of the finite element algorithm. And second, general purpose graph partitioning does not necessarily produce good graph embeddings for any algorithm.[2]

As mentioned above, we are interested in a DM-implementation of the hierarchical shooting algorithm. Distributed memory machines – especially when programmed in the message passing paradigm – show large differences between latencies of local and remote memory accesses. Hence, only coarse grained parallel computations are expected to scale well when the number of processors is increased.

[2] Bruce Hendrickson — one of the authors of the famous *Chaco* graph partitioning software — mentioned in his talk at IRREGULAR 1998 that graph partitioning works, because it is applied mainly to simple partial differential equation solvers. If the application is more difficult and unstructured (as global illumination is), then the inherent simplification of the graph partitioning problem could make it impossible to generate a good embedding.

The hierarchical shooting algorithm exhibits high fine grained parallelism both on the link and on the element level. E. g. each Oracle task inside one iteration loop can be performed independent of each other, if the data of the associated elements is available. Also the order of task execution inside an interation loop is almost arbitrary, except that at the beginning of a loop a few high level links need to be processed sequentially until there are enough links on a deeper level of the hierarchy, that can be processed in parallel.

Unfortunately, due to its very dynamic nature, it is not easy to partition the computation into large, equally complex blocks, which would be necessary to achieve a balanced, coarse grained parallel algorithm.

In the hierarchical shooting algorithm we are concerned basically with three kinds of data items: elements, links, and boundary data. Boundary data consists of the scene geometry and illumination specific properties such as material and emission data. Material and emission data is needed everytime, when an associated element is involved in a link refinement and during local reflection operations. Hence these data is best stored together with the element. Also the geometry data is needed for these operations and should be stored there.

Unfortunately geometry is needed not only for local reflection operations but also for visibility calculations. In order to make visibility calculations fast, in our program each processor holds a separate copy of the whole scene geometry. Since geometry data is far less voluminous than the illumination representation (the elements), this should be a viable way even for very large scenes. Simulations in Chapter 6 show that the other approach of distributing the visibility calculations across processors would result in very poor performance.

For the mapping of links and elements we choose a spatial clustering method much like an orthogonal recursive bisection strategy.

- Elements are associated with either scene surfaces or with surface clusters. Hence, with each element we may associate a 3D range $[\vec{x}_{\min}, \vec{x}_{\max}]$ of the scene. The ranges of different elements may overlap, e. g. the range of a cluster element contains all ranges of the contained surface elements.

  Given a number of processors $p \geq 1$ we partition the 3-d-space into p regions using a 3-d-tree. Elements are mapped to a region based on the center of gravity $\vec{x} \in \mathbb{R}^3$ of the associated range. Here we solely use boxes as bounding ranges, but also other geometries could be reasonable.

  Every processor holds a copy of the 3-d-tree. The 3-d-tree has storage size $O(p)$ and is used as a directory for the elements. If we want to access a remote element, whose geometric location is known, then we determine the owning processor of the element using the directory and then directly send a query to that processor.

  With each surface primitive usually not a single element but a more or less complex hierarchy of elements is associated. We pool all surface elements that belong to the same surface primitive and map these elements together with the surface's boundary data to the same processor.

- Links are basically pairs of elements. Hence it is natural to associate a 6-d-range with links resulting from the cartesian product of the 3-d-ranges of the two elements. Mapping of links to processors again is done based on the center of gravity $(\vec{x}, \vec{y}) \in \mathbb{R}^6$ inside a 6-d-tree with p leaves. We store a copy of this tree for directory lookup purposes on each processor.

Computational load balance is achieved in our partitioning by moving the cutting hyperplanes of the k-d-trees geometrically and moving the elements and links between processors. In Chapter 7 it is described, how such a load balancer may be implemented with small worst case overhead.

So, why is this partitioning strategy a good strategy?

First, the hierarchical shooting algorithm is known to be very communication intensive. In Chapter 6 we discuss a few partitioners and show by experimental measurements that a spatial partitioning does reduce the amount of communication fairly well.

Second, the chance of congestion in an interconnection network gets high if there is high traffic in it. Incentive (d) in Sect. 4.1.3 says that we should aim at reducing the total number of communicating processor pairs. The spatial partitioning approach does exactly this, which is discussed in the following.

We consider $p = 16$ processors. Both the 3D space of elements and the 6D space of links is partitioned as shown in Fig. 5.1. We consider an element whose range is centered at a fixed $\vec{x} \in \mathbb{R}^3$. This element is involved in the computations of those links whose ranges are centered at some point from the following set:

$$\{(\vec{x}, \vec{y}) : \vec{y} \in \mathbb{R}^3\} \cup \{(\vec{y}, \vec{x}) : \vec{y} \in \mathbb{R}^3\}.$$

The set of processors owning all these links is shown in light gray in Fig. 5.1. The number of these processors is $2\sqrt{p}$ if we assume $p = 2^{6m}, m \in \mathbb{N}$, and it is $O(\sqrt{p})$ otherwise.

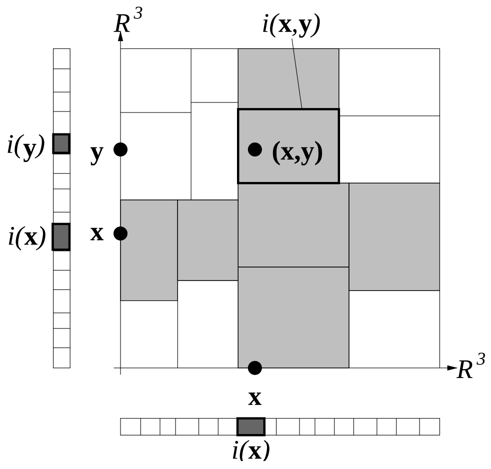


Figure 5.1: The 6-dimensional space of link locations $(\vec{x},\vec{y})$, is mapped to 16 processors. For clarity reasons only a schematic 2-dimensional mapping is shown. The 3-dimensional space of elements is shown schematically as a 1-dimensional line. An element, whose range is centered at $\vec{x}$ gets mapped to a processor index $i(\vec{x})$. The link centered at $(\vec{x},\vec{y})$ is mapped to processor $i(\vec{x},\vec{y})$.

Let us assume an even distribution of elements and links in 3-d/6-d space. Then all elements located on a given processor i are involved in the processing of links, that are located on $O(\sqrt{p})$ different processors. The constant factor in the big-oh depends on how even elements and links are distributed. The total number of communicating processor pairs under these assumptions is $O(p \cdot \sqrt{p}) = O(p^{1.5})$ which is much less than all possible pairs $(\Theta(p^2))$.

5.2 Partitioning The Monte-Carlo Approach

Spatial partitioning is an obvious approach to map particle based algorithms to parallel processors. Many researchers have followed this general principle already in the past. We can expect good speedups only if dynamic load balancing is performed. We first describe a simple, widely used, spatial partitioning strategy for the particle based *photon map* algorithm (cf. Sect. 2.3.4). Then we discuss briefly only a few interesting approaches found in literature that perform dynamic load balancing for a spatial partitioning of raytracing.

5.2.1 Partitioning Strategy

A particle tracer works on a scene model, which consists of many objects. From an algorithmic point of view decomposing the scene model is useless, since there are no tasks associated naturally with the objects. But in a parallel environment with local memories we will have to distribute the objects across many processors, because the whole scene may not fit into a single processor's memory.

The other data items in the algorithm are the particles that are spawned recursively. The task associated naturally with a particle is to firstly calculate a closest intersection, and then to spawn new particles at the intersection point. At an instant of time during the application's run every particle can be traced independently, since concurrent read access to the scene objects is possible. As time elapses the particles depend on their successors and predecessors. A particle cannot start its associated task before the parent particle has completed its task. Hence, the set of runnable tasks dynamically evolves.

Assume the 3D space of the scene is partitioned by a 3-d-tree T_{scene} (see Fig. 5.2 for a 2D-example). Each leaf cell contains a subset of the scene's objects and a subset of all particles. Objects that do not fit into a single cell are duplicated in each intersected cell.[3] For large real world scene models this usually consumes only little additional memory. The universe of all leaf cells defines the set of tasks — one task for every single leaf cell. A task consists of calculating intersections between all particle-paths and all objects contained in the cell. For those particles hitting an object, new reflected or transmitted rays are calculated. Particles that miss all objects of the cell are propagated to a neighbouring cell.

[3] The extent of objects contained in a leaf cell may be larger than the cell's extent. The object's extent ⬚ is stored inside the cell besides the cell's extent □. Particles are always contained in the □-box. When searching for particles near a given query point, which is needed in the calculation of equation (2.11, page 20), we can restrict ourselves to the non-overlapping □-boxes, which potentially reduces the number of processors to be visited during the search.

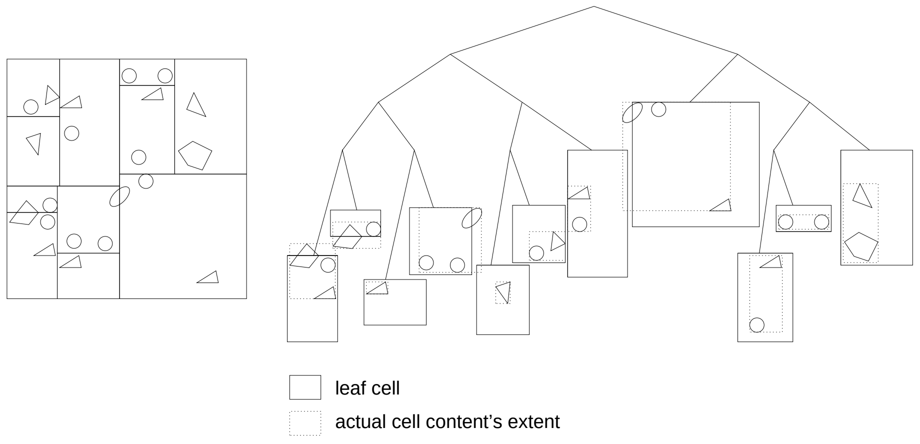


Figure 5.2: A 2D-example of a tree T_{scene} hosting the scene's objects.

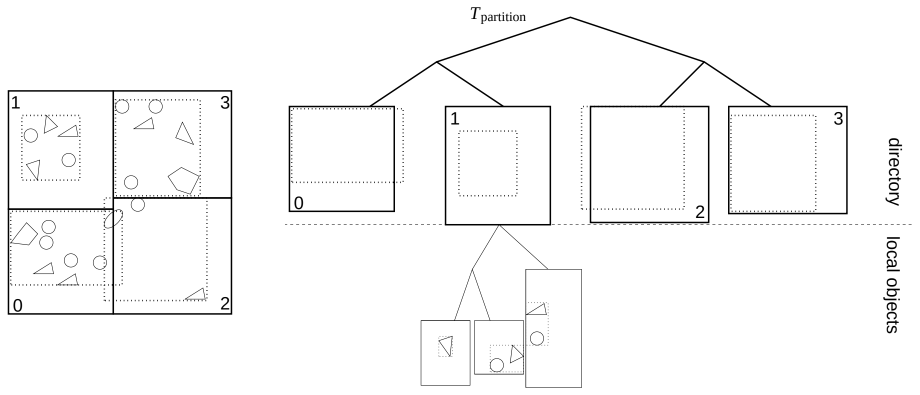


Figure 5.3: A 2D-example of a tree-partitioning $T_{\text{partition}}$ for 4 processors. On the right only the local objects on processor 1 are displayed.

As in the finite element partitioning we can use a 3-d-tree with p leaves to describe a partitioning for p processors (see tree labeled $T_{\text{partition}}$ in the 2D example of Fig. 5.3). Here we can simply reuse the upper levels of T_{scene} to define the partitioning. $T_{\text{partition}}$ can be queried directory-like about which processor holds which regions of the space. Computational load balance in this partitioning can be achieved by moving cutting hyperplanes of $T_{\text{partition}}$ (see Chapter 7). The complexity of a task may be estimated by taking the number of contained objects and particles and the surface complexity of the objects into account (see e. g. [88] on cost prediction in ray tracing). A cell of T_{scene} could be dynamically split, if its complexity is high. Neighbouring low complexity cells of T_{scene} may be merged together.

The communication in this partitioning is restricted to neighbouring cells of $T_{\text{partition}}$. If the distribution of the scene's objects is not too unbalanced, then every processor communicates only with a small constant number of neighbouring processors. Hence, following incentive (d) in Sect. 4.1.3, the total number of communicating processor pairs under these assumptions is $O(p)$ which is much less than all possible pairs ($\Theta(p^2)$).

5.2.2 Related Work

In an early paper [28] Dippe and Swensen proposed an adaptive subdivision of the 3d space into subregions. As in Sect. 5.2.1 scene objects are distributed among the subregions according to their position, and each subregion processes rays independently. Rays are tested for intersection only with those objects within the particular subregion. Rays leaving the subregion are communicated to neighbouring subregions. The shape of subregions is adapted dynamically based on local, bilaterally available load information. They analyzed this approach only theoretically and demonstrated promising speedups.

Caspary and Scherson [17] describe a partitioning approach based on a hierarchical data structure of bounding volumes. Similarly to Sect. 5.2.1 they divide the hierarchy into two parts at some level of the tree. The upper part is duplicated on each processor. The lower subtrees are distributed evenly to unique home processors. Each processor hosts instances of two processes: one relocatable process that computes intersections between rays and bounding volumes of the upper part of the hierarchy, and one non-relocatable process that is responsible for ray-object intersections. Load balance is achieved by performing the first process less frequently on those processors that have a heavy workload with the second process. Results have been published only as simulations on a uniprocessor machine.

In the parallel raytracing work of [60] a subdivision of 3d-space by orthogonal hyperplanes is maintained dynamically using a variant of the Fiduccia and Mattheyses graph partitioning heuristic [33].

Zara et al. [109] describe a *process-farming* approach on a cell-partitioning. With each cell a light-weighted process is associated that maintains all rays "living" currently inside the cell. Further heavy-weighted *RT-Core* processes are assigned dynamically to cells and essentially perform the intersection calculations. Dynamic load balance is achieved by assigning a varying number of RT-Cores to different cells, where overloaded cells get more RT-Cores than underloaded cells.

5.3 Issues For Combined Approaches

As there is currently no common agreement on the best hybrid global illumination algorithm, surely there isn't any on a good parallelization of it. In this section we discuss briefly the basic data access patterns occurring in hybrid algorithms and how these could be implemented in parallel based on the above spatial partitioning strategies.

In hybrid global illumination algorithms it is an essential operation to convert a finite element representation and a particle representation into each other. Since both, particles and elements, are mapped to the processors based on their position in 3D space, the conversion mostly can be performed locally on a single processor. For the calculation of a particle representation out of some surface element approximation we simply generate some particles and store them locally in the particle tree. For calculating an element approximation on some surface from a given particle representation, we search locally for nearest particles close to the surface and then evaluate equation (2.11, page 20).

Unfortunately sometimes communication could be necessary. First, when converting a large surface's element representation to particles, the generated particles could fall into several different T_{scene}-cells. If these cells are on different processors, communication is needed. Second, when searching particles close to a given surface, the closest particles may lie in different T_{scene}-cells, especially when the given surface is large and/or is located near a T_{scene}-cell boundary. Again, communication could be necessary.

Both effects can be mitigated by preprocessing large surfaces into smaller ones and by trading an increased size of T_{scene}-cells against the need for a manageable computational complexity of these cells. Finding a good tradeoff and implementing these conversion methods is an interesting area of future research.

Chapter 6

On The Partitionability Of Hierarchical Radiosity

Contents

The Hierarchical Radiosity Algorithm (HRA) [53] is a famous representative of the family of finite element methods for lighting simulations.

Parallelization, however, causes troubles because the HRA is a dynamic, hierarchical, irregularly structured algorithm, in contrast to full matrix radiosity methods [23] which are regularly structured and therefore better suited to parallel computations (see survey in [82]). There have been quite successful attempts to implement the HRA on a cache-coherent shared-address-space multicomputer [95], since the HRA exhibits high fine grained parallelism. But developing a coarse grained parallel algorithm meeting the requirements of a distributed memory (DM) environment is difficult.

In this chapter we present a detailed experimental analysis of the HRA and its partitionability as it was published first in [41, 42]. We give quantitative results underpinning “common knowlegde”, that HRA is difficult to parallelize. Measurements regarding three important issues of a parallel implementation are presented: load balance, communication, and congestion. The results are rated quantitatively and indicate that there exist several bottlenecks. Different graph partitioners are examined with respect to their ability to cope with these bottlenecks. One main finding is that a spatial partitioning method overcomes most of the problems well.

We do not perform the measurements on a real parallel implementation. Instead we abstract from several possible techniques of dynamic load balancing and communication reduction by analyzing each radiosity iteration in a static manner. However, as is explained below, this approach does not assume a static partitioning strategy. Instead, it allows conclusions regarding the performance characteristics of any possible real dynamic partitioning method.

Section 6.1 defines the graphs to be analyzed and argues, why these are a reasonable choice for getting an idea about the behaviour of the best imaginable parallel mapping strategy applied to the HRA.

Section 6.2 introduces three partitioners: a naive partitioner, a spatial partitioner and an “optimum” partitioner. The optimum partitioner cannot be used in a real implementation, since it needs a priori knowledge of the graph, which itself evolves during computations. We consider the optimum partitioner as a generator of a reference solution.

In Sect. 6.3 it turns out, that the graph defined so far is very poorly partitionable. Even the best partitioner achieves a cut size of only 85 percent of all edges running across processor boundaries, when partitioning into 16 parts. The reason of this poor behaviour is the overwhelming amount of communication due to visibility computations. Therefore, in Sect. 6.5, we remove the visibility access edges from the definition of the graph. The resulting task access graphs are better partitionable and allow greater insight in the structure of the HRA.

Sections 6.4-6.6 treat the following sources of overhead in a potential parallel implementation of the HRA: load imbalance, communication, and congestion.

In Sect. 6.4 we see, that the HRA is a very dynamic algorithm. The amount of work changes dramatically from iteration to iteration. We also see that load balancing is strictly necessary, since balance of load is poor, when not accounting for load balance at all. But, the changes in the balance of load are relatively small. Hence, one could guess that load balancing in early iterations pays off in later iterations. An important result of Sect. 6.4 is that load balancing amortizes, which is confirmed by *load continuity matrices*. These relate overloaded processor sets of different iterations to each other.

Section 6.5 presents measured data on the cut size, the connectivity balance, and the load balance for different partitioners. A result is that all partitioners do not achieve admirable partitions. Metis produces fairly well load balanced partitions with the best cut sizes among all partitioners, but with a poor communication balance. The spatial partitioner achieves excellent load balance, good communication balance and bad cut size. The spatial partitioner is much faster than Metis, and therefore might be the better choice.

This is confirmed in the following Sect. 6.6, where we examine the important aspect of congestion. The number of channels between processor pairs that is used in the HRA is much larger than the number of available channels in scalable supercomputers, which may lead to congestion. Moreover the channel usage is unevenly distributed across the available channels. We discuss this issue based on a new measure, the *channel usage*.

A short Sect. 6.7 takes a look at graphs from other domains and shows that these graphs are much better partitionable than the HRA graphs.

6.1 The Graphs

We define a single graph for each iteration of the algorithm as described in Sect. 5.1.2. The reader is kindly asked to read Sect. 5.1.2 now, if she skipped it previously. Below we will describe, how to statically partition each of these graphs separately using a graph partitioner.

A real parallel implementation of the algorithm of course lacks a priori knowledge of these graphs and therefore would have to employ other more dynamic partitioning strategies. Nevertheless considering the static partitions is useful. Any potential parallel implementation, which performs one of the standard iteration methods, such as Jacobi or Gauss Seidel, must synchronize all processors between two iterations. Hence, any sophisticated dynamic load balancing and communication reduction strategy of any potential parallel program cannot schedule tasks or communication from iteration k to another iteration $l \neq k$[1]. By analyzing the graph of a single iteration, we get something like a “lower bound” on the load imbalance and on the communication, which must be treated by any parallel program.

6.2 Graph Partitioners

We consider the partitioning problem as defined in Sect. 4.1.2. This problem applied to the task graph described in Sect. 5.1.2 allows us to assign tasks to processors, while minimizing the communication between the processors. Unfortunately, the partitioning problem does not consider all sources of overhead in a parallel computation. For instance, a balanced partition does equally distribute the computational load across the processors, but does not at all guarantee an evenly balanced distribution of communication load. In order to measure the balance of communication load we define the *connectivity* of one part $V_j \subseteq V$ of a partition:

$$conn(V_j) = \sum_{\{v,w\}\in E, \pi(v)\neq\pi(w), j\in\{\pi(v),\pi(w)\}} W(\{v,w\}) \ .$$

The *connectivity balance* of a partition is:

$$cb(\pi) = p \max_{0\leq j\leq p-1} \left\{ \frac{conn(V_j)}{2cs(\pi)} \right\} \ .$$

A perfectly connectivity-balanced partition satisfies $cb(\pi) = 1$.

[1] All statements remain valid for group-iterative methods (even though weakened), since in such methods scheduling is possible within a fixed range of iterations $\{k,\ldots,k+c\}$ but not beyond.

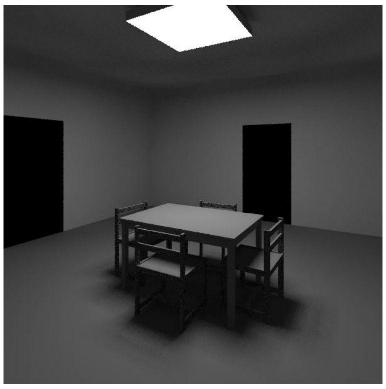


Figure 6.1: A simple scene.

Another source of overhead in a parallel implementation is *congestion*. Congestion may arise on low bandwidth, incomplete communication networks, when the communication load is not distributed equally across the available communication channels. We discuss congestion in more detail in Sect. 6.6 below.

In this work we will concentrate on three partitioning methods:

Metis Partitioner: The program Metis [64] is a publically available graph partitioner, which employs sophisticated multilevel algorithms producing high-quality partitions for a large variety of graphs. As mentioned above this partitioner cannot be used in a real implementation, since it requires a priori knowledge of the whole graph.

Naive Partitioner (NP): This partitioner randomly assigns vertices to partitions, without considering load balance and the reduction in cut size. The algorithm is simple: for each vertex, assign the vertex to a random partition $i \in \{0, \ldots, p-1\}$. For large graphs this results in a not too poor load balance. The expected cut size is $\frac{p-1}{p}W(E)$, if we assume a graph with uniformly distributed edges.

Spatial Partitioner (SP): This algorithm is also known as *coordinate sorting*. All vertices are assigned a location in $\mathbb{R}^k$.

The location of an element vertex is a point inside the 3D-scene to be rendered. It is defined as the center of the 3D bounding box of the geometric extent of the associated element.

The location of a link vertex is the 6D tuple composed of the two 3D locations of the associated elements.

The partitioner bisects the graph recursively with respect to alternating axis-orthogonal hyperplanes. Link and element vertices are partitioned independent of each other. A dynamic SP can be implemented with a worst case upper bound on the runtime (see Chap. 7).

NP is used as a “lower benchmark”. Metis is considered to produce partitions near the optimum – or near the best solution achievable in reasonable computation time – i. e. this solution is used as an “upper benchmark”. We will show that the difference between the lower and upper benchmarks is small, indicating that the HRA is poorly partitionable.

In a parallel DM-implementation of the HRA the book-keeping of remote objects is an important issue. A partitioning of the “spatial” type is completely described by a short directory representation that can be replicated on every processor. This fact was utilized before in parallel raytracing algorithms (e. g. [17]). A Metis partition may be very unorganized, such that there does not exist a short directory-like description of the partition. *Ghost objects* need to be held on many processors leading to increased book-keeping time and memory consumption. Hence, we would clearly prefer a simple SP, if its quality is nearly as high as Metis.

6.3 A First Experiment

In this section we want you to get a first feeling how the graphs of the HRA look like and how well they are partitionable. Let us consider a simple 3D scene depicted in Fig. 6.1, which consists of a room with four open doors and with a table and four chairs inside. The scene is lit by a single light source on the ceiling. All 410 primitives are rectangles.

	number	weight			
		min	avg	max	sum
links	13,546	(1|0)	(1|0)	(1|0)	(13,546|0)
elem's	561	(0|1)	(0|1)	(0|1)	(0|561)
vertices	14,107				
edges	604,541	1	1.01	26	612,913
vd(link)		2	45.1	443	
vd(elem)		10	1,097	3,827	
vd		2	86.9	3,827	

Table 6.2: Characteristic data of G_1^{room}.

We applied the HRA and counted all task to task accesses during runtime. We get different task access graphs G_k^{room} from iteration to iteration, where k denotes the iteration. First we have a closer look at the graph G_1^{room} during the first iteration (see Table 6.2).

The graph has got 14,107 vertices, where the portion of link vertices is an overwhelming 96 percent. The number of edges is 604,541. A complete graph would have about 99.5 million edges. Hence the adjacency matrix of our graph has only 0.6 percent nonzeroes, i. e. our graph is sparse.

Almost all edges have an edge weight of 1, but there are some edges with relatively large weights. The maximum edge weight is 26.

We defined the *vertex degree* of a vertex $v \in V$ as the weight-sum of those edges that are adjacent to v:

$$vd(v) = \sum_{\{v,w\} \in E} W(\{v,w\}) \quad .$$

The vertex degree in the graph G_1^{room} varies in a broad range from 2 to 3,827. The vertex with the largest degree of 3,827 is the root cluster. The next smaller vertex degree found in the graph is only 1,578 (not listed in the table). The average degree is 86.9. The relatively large degree of the root cluster element vertex means a communication hot spot in a potential parallel implementation.

The maximum degree of links is 443. This means, that there are link tasks, that need to access a large portion ($\frac{443}{561}$=79%) of the elements. Most of these accesses are for visibility computations. Of course this is mainly due to the "one-room-character" of the scene, where a surface sees almost all other surfaces.

In [41] we give data for the graph of the third iteration. This iteration is one of the computationally most complex iterations in terms of the number of processed links. In the graph we basically rediscover the same features as in the graph of the first iteration. The main difference is its increased size (27K vertices – 4M edges).

Now, we examine the partitionability of the graph. A run of Metis 4.0, that partitioned the graph G_1^{room} into 16 partitions, needed 41 seconds on an Intel Pentium II, 333 MHz, running Linux. The resulting partition (see Table 6.3) is well load balanced, and also the connectivity balance is not too bad[2].

The cutsize is expressed as the *relative cut size* in the table, that relates the cut size to the total edge weight:

$$rcs(\pi) = \frac{cs(\pi)}{W(E)} \quad \in [0,1] \quad .$$

The relative cut size of the Metis partition is large (85%), especially when compared to the relative cut size that can be achieved by NP (93.8%).

NP behaves as expected. Since vertices are assigned randomly to processors, the probability[3] that an edge runs across processor boundaries is $\frac{p-1}{p}$. The measured relative cut size in Table 6.3 meets this formula: $\frac{p-1}{p} = \frac{15}{16} = 93.8\%$. The load balance of the naive partition is worse than that of the Metis partition. The large second component (1.43) is due to the small number of element vertices compared to the number of processors, and due to the fact that NP does not care about load balance at all. Both SP and NP achieve better connectivity balance results than Metis, paid by a not too large a difference to the optimum cut size.

Things get worse, when looking at the 128-way-partitions (Table 6.3 bottom). The difference between the naive solution and Metis at the cut size is marginal.

[2] Metis 4.0 includes heuristics for achieving a good connectivity balance.

[3] Consider a task (vertex) v. Each neighbouring vertex u of v is assigned randomly to any processor by the naive partitioner. Hence, the probability, that u is mapped to the same processor as v, is $\frac{1}{p}$. We conclude that the edge between u and v runs between different processors with probability $1 - \frac{1}{p} = \frac{p-1}{p}$.

method	runtime	$lb(\pi)$	$cb(\pi)$	$rcs(\pi)$
Metis	41 sec.	(1.05, 1.03)	2.19	85.0%
NP	<1 sec.	(1.07, 1.43)	1.20	93.8%
SP	<1 sec.	(1.00, 1.03)	1.42	93.1%
Metis	74 sec.	(1.13, 1.14)	2.90	97.3%
NP	<1 sec.	(1.24, 2.51)	1.76	99.2%
SP	<1 sec.	(1.00, 1.14)	2.83	99.1%

Table 6.3: Partition of G_1^{room} into 16 (top) and 128 (bottom) parts.

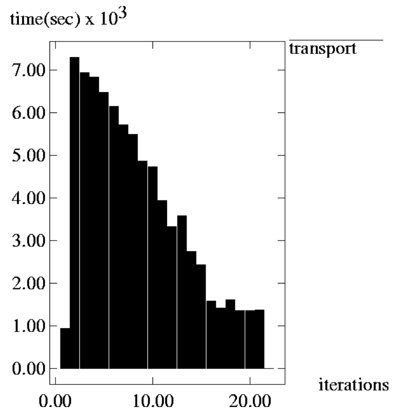


Figure 6.4: Runtime on a single processor per iteration in the refinement and transport stage for the nine room scene.

We may learn a first lesson from the above data. Using sophisticated graph partitioners for the HRA is overkill, since the task access graph is poorly partitionable. The difference between a naive solution and the one obtained by Metis is small. Metis consumes multiple more runtime than the simpler naive or spatial approaches. Hence, using one of the simpler strategies may be indicated.

6.4 Measuring Load Balance

In this section we will examine the load balance of the HRA. We do not really load balance the application. Instead we measure the maximum load imbalance that occurs, when we do not care about load balance at all. All measurements were done on an Intel Pentium II (333 Mhz) running Linux.

Since the one room scene is small with relatively little occlusion, we did experiments with a larger scene, which consists of 3×3 replicates of the one room scene. One can look from one room into another through open doors. Most of the results in this thesis are for the nine room scene. More data on the 1-room-scene is available in [41].

First, let us look at the total time consumption of the algorithm executed on a single processor from iteration to iteration. Figure 6.4 shows the time per iteration needed during link refinement and transports. The time for the push and pull stages is neglected, because it is less than one percent of the total time in every iteration. The runtime rapidly increases in the first two iterations and then decreases gradually until it gets nearly zero in the 22nd iteration. We may safely state, that the HRA is a very dynamic algorithm. But, fortunately, it is not really unpredictable, as is shown in the following.

We are going to analyze a run of the HRA on a hypothetical multi-processor system by simulation on a one-processor system. Each executed task contributes to the runtime of the processor that owns the task. We assume an initial, not necessarily balanced, partitioning of the vertices of the task access graph to p processors. In each iteration k we measure the total time t_i^k, spent by tasks that are owned by processor i.

Each processor i performs its own tasks one after another. Sometimes the processing of a task (vertex) v may involve the generation of a new task u, for instance when a link or an element is subdivided. The new task u may be seen as an increase of complexity of task v. The load of the processor, owning v is increased by the complexity of u. Hence, we should assign the new vertex u to this processor, in order not to bias the measurements t_i^k of processor i.

As the initial partitioning of the vertices we use a not necessarily balanced distribution of vertices, since we

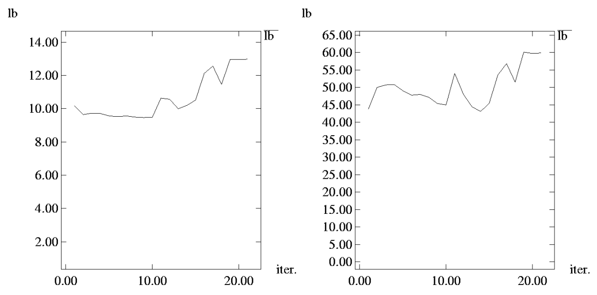


Figure 6.5: Load balance lb^k during transport calculations in the nine room scene for 16 (left) and 128 (right) partitions.

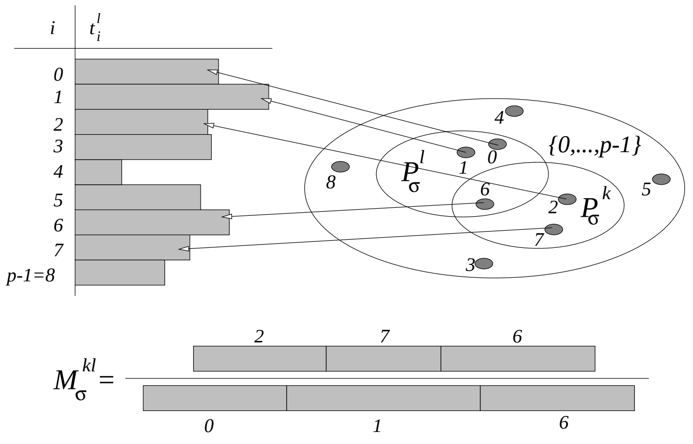


Figure 6.6: Definition of M_σ^{kl} for an example with 9 processors. $P_\sigma^l = \{0,1,6\}$ contains the processors with maximum runtimes t_i^l in the l-th iteration. $P_\sigma^k = \{2,6,7\}$ contains processors that are most busy in the k-th, but less busy in the l-th iteration. M_σ^{kl} is the quotient of accumulated runtimes in iteration l.

do not know the whole graph in advance. For instance at the beginning there exists only one self link at the root cluster element. All other link tasks are recursively generated from this single link. Also, elements are generated on the fly by subdivision. Of course, we could start from a balanced partitioning of those vertices, that are known from the beginning. But this would maybe falsify the load balance measurements during runtime.

We decided to assign the tasks to the processors once at the beginning based on their associated locations as they were defined for SP in Sect. 6.2. Each processor owns an equally sized region of all tasks. For the sake of measuring runtime without employing a load balancer – which is the aim of this section – these regions remain fixed for the whole run of the HRA. A newly generated task u (a sublink or subelement) has a predefined location, which is close to that of the generating task v. Hence it is likely, that the processor owning v also gets the additional load of task u.

The load balance of the algorithm during runtime is measured independently in each single iteration. This is necessary because in a parallel implementation after each iteration usually a synchronization between all processors takes place. We define the load balance lb^k in iteration k analogously to the load balance of a graph:

$$lb^k = p\frac{\max_{0\le i\le p-1} t_i^k}{\sum_{0\le i\le p-1} t_i^k} \quad .$$

Figure 6.5 shows the load balance for $p = 16$ and for $p = 128$ processors in the 9-room-scene. Load imbalance is high all the time (remember the ideal value of $lb = 1$). In this scene we absolutely need a load balancer. But, the balance changes are not dramatic (Remember, absolute load changes are dramatic at least in early iterations, see Fig. 6.4.)

A question arises whether the overloaded processors of the early iterations are overloaded in later iterations, too. In such a situation load mostly stays where it was put at the beginning. We characterize this fact by measuring the *load continuity*. When load continuity is high, then rebalancing the load in early iterations will amortize in later iterations. This is especially important, since rebalancing in early iterations is expensive, because work load is high and therefore large loads have to be moved.

In order to analyze load continuity we consider an upper σ-portion ($\sigma \in [0,1]$) of all processors, i. e. $P^k_\sigma \subseteq \{0,\dots,p-1\}$, $|P^k_\sigma| = \sigma p$, which contains the indices of the maximum loaded processors (the *hot spot processors*) in the k-th iteration (see Fig. 6.6 for an example). We want to examine, whether these processors are also overloaded in later iterations $l > k$. One possible measure for load continuity would be the size of the intersection set $P^k_\sigma \cap P^l_\sigma$. If this intersection set contains many processor indices, then there was only little change among the hot spot processors. There are cases however, where this measure is not exact enough. If for example the processors $P^k_\sigma - (P^k_\sigma \cap P^l_\sigma)$, which are not contained in the intersection set, are totally idle in iteration l, then we have a much worse load continuity than in the case, that these processors are relatively busy in iteration l (e. g. if they are contained in $P^l_{2\sigma}$). Hence, the measure should also include the load t^l_i for processors $i \in P^k_\sigma$.

To overcome these troubles we define the following *load continuity* matrix, which includes loads in iteration l:

$$M^{kl}_\sigma = \frac{\sum_{i\in P^k_\sigma} t^l_i}{\sum_{i\in P^l_\sigma} t^l_i} \quad \le 1 \ .$$

The quotient M^{kl}_σ is large if the hot spot processors of iteration k are also overloaded in iteration l. For experimental analysis, we set $\sigma = 0.1$, i. e. we are interested in the mostly overloaded 10 percent of all processors. Two resulting matrices are listed in Fig. 6.7.

On 16 processors the load situation can be divided into two phases. In the first iteration, those processors are overloaded, that are concerned with transporting light from the light sources to other surfaces. These processors are less busy in the second phase (starting at iteration 2). In the second phase there seems to be a relatively fixed set of processors, that are overloaded in each of the following iterations. The whole load continuity matrix looks uniform, which means that load balancing is likely to pay off in later iterations.

This basic prinicple can be found also in the matrix for 128 processors even though less clear especially in the last six iterations. For the one-room-scene the situation is similar as in Fig. 6.7 [41].

In [11] a parallel implementation of HRA is presented, that performs a simple demand-driven load mapping approach in the first iteration only. In all the remaining iterations the assignment of data to the processors is kept unchanged. This simple strategy is reported to work well, and the findings about load continuity here seem to justify that simple strategy subsequently.

6.5 Measuring Communication

In Sect. 6.3 we experimented with different partitioners on the task access graph of a single room scene. The large number of accesses to the root cluster element vertex is one important reason that all partitioners generated solutions with large cut size. Visibility tasks are responsible for most of the communication.

When calculating visibility on a distributed scene geometry naively by sending individual messages for each link, we cannot hope a parallel implementation of the HRA to show up scalable behaviour. Caching replicates of geometries in local memories for instance could be employed to reduce the communication requirements. Here we consider the extreme case, where all geometry data is cached in all local memories once, and these data will never be destroyed. This completely eliminates the need for communication for the visibility tasks. The replicated data is only a small fraction of the total data, since it only includes geometry of the top level surface elements. In the rest of this chapter we consider the task access graph without visibility accesses. We denote such graphs by the letter H. These graphs will allow a greater insight into the structure of the HRA beyond visibility computations.

In Table 6.8 the characteristic data of the task access graph H^{room}_1 is shown. The graph has got the same number of vertices as G^{room}_1 (see Table 6.2). The number of edges instead is greatly reduced from 604,541 to 27,507. Also the hot spot problem is alleviated, since the maximum vertex degree now only is 578.

When partitioning the graph H^{room}_1 we get much better results than for the graph G^{room}_1 with Metis and SP (see Table 6.9). Nevertheless, the cut size is high for both partitioners, especially when compared to cut sizes that could be achieved on graphs from other domains (see Sect. 6.7). The runtime of Metis is moderate for H^{room}_1, but it gets large for the nine room scene [41] (up to 358 seconds).

Because of the large amount of measured data, we present diagrams for the load balance, the communication balance and the cut size. We start with the relative cut size for the nine room scene in the first and in the third iteration for varying processor numbers. Data for the one room scene can be found in [41]. Figure 6.10 shows the results. Metis generates significantly better partitions than SP. When p is increased, all cutsizes increase. SP's cut sizes approach that of NP for large processor numbers.

The load balance of the link vertices[4] is good for small processor sets (Fig. 6.11). We did not include the load balance of element vertices, because the influence of element task processing on the total runtime is only

[4] In contrast to Sect. 6.4 now dynamic load balancing *is* performed

iteration l

iteration k

	1	2			...					10						...					21
1	1.00	0.93	0.92	0.92	0.92	0.91	0.91	0.90	0.90	0.90	0.91	0.90	0.91	0.88	0.89	0.92	0.91	0.93	0.91	0.91	0.91
2		1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00
			1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00
				1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00
⋮					1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00
						1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00
							1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00
								1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00
									1.00	1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00
10										1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00
											1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00
												1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00
													1.00	0.94	0.94	1.00	0.98	0.99	0.98	0.98	0.98
														1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00
⋮															1.00	0.99	1.00	1.00	1.00	1.00	1.00
																1.00	0.98	0.99	0.98	0.98	0.98
																	1.00	1.00	1.00	1.00	1.00
																		1.00	1.00	1.00	1.00
																			1.00	1.00	1.00
																				1.00	1.00
21																					1.00

iteration l

iteration k

	1	2			...					10						...					21
1	1.00	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.94	0.95	0.94	0.94	0.95	0.96	0.96	0.93	0.96	0.96	0.96
2		1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.99	1.00	0.99	0.99	0.96	0.95	0.96	0.94	0.94	0.94
			1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.99	1.00	0.99	0.99	0.96	0.95	0.96	0.94	0.94	0.94
				1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.99	1.00	0.99	0.99	0.96	0.95	0.96	0.94	0.94	0.94
⋮					1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.99	1.00	0.99	0.99	0.96	0.95	0.96	0.94	0.94	0.94
						1.00	1.00	1.00	1.00	1.00	0.99	0.99	1.00	0.99	0.99	0.96	0.95	0.96	0.94	0.94	0.94
							1.00	1.00	1.00	1.00	0.99	0.99	1.00	0.99	0.99	0.96	0.95	0.96	0.94	0.94	0.94
								1.00	1.00	1.00	0.99	0.99	1.00	0.99	0.99	0.96	0.95	0.96	0.94	0.94	0.94
									1.00	1.00	0.99	0.99	1.00	0.99	0.99	0.96	0.95	0.96	0.94	0.94	0.94
10										1.00	0.99	0.99	1.00	0.99	0.99	0.96	0.95	0.96	0.94	0.94	0.94
											1.00	0.99	0.96	0.99	0.97	0.94	0.96	0.97	0.94	0.94	0.94
												1.00	0.96	1.00	0.99	0.94	0.96	0.96	0.95	0.95	0.95
													1.00	1.00	0.99	0.96	0.96	0.96	0.95	0.95	0.95
														1.00	0.98	0.94	0.96	0.97	0.95	0.95	0.95
⋮															1.00	0.97	0.96	0.97	0.96	0.96	0.96
																1.00	1.00	0.98	1.00	1.00	1.00
																	1.00	0.96	1.00	1.00	1.00
																		1.00	0.97	0.97	0.97
																			1.00	1.00	1.00
																				1.00	1.00
21																					1.00

Figure 6.7: Load continuity $M^{kl}_{0.1}$ in the nine room scene for a 16-way-partition (top) and a 128-way-partition (bottom). Large entries are displayed in a lighter grey.

	number	weight			
		min	avg	max	sum
links	13,546	(1|0)	(1|0)	(1|0)	(13,546|0)
elem's	561	(0|1)	(0|1)	(0|1)	(0|561)
vert's	14,107				
edges	27,507	1	1.30	26	35,879
vd(link)		2	2.46	4	
vd(elem)		5	68.47	578	
vd		2	5.09	578	

Table 6.8: Characteristic data of H_1^{room}.

method	runtime	$lb(\pi)$	$cb(\pi)$	$rcs(\pi)$
Metis	5.1 sec.	(1.00, 1.03)	2.16	43.6%
NP	$<$ 1 sec.	(1.07, 1.43)	1.28	93.9%
SP	$<$ 1 sec.	(1.00, 1.03)	1.52	84.5%
Metis	9.3 sec.	(1.13, 1.14)	3.78	53.0%
NP	$<$ 1 sec.	(1.24, 2.51)	2.49	99.2%
SP	$<$ 1 sec.	(1.00, 1.14)	3.27	97.5%

Table 6.9: Partition of H_1^{room} into 16 (top) and 128 (bottom) parts.

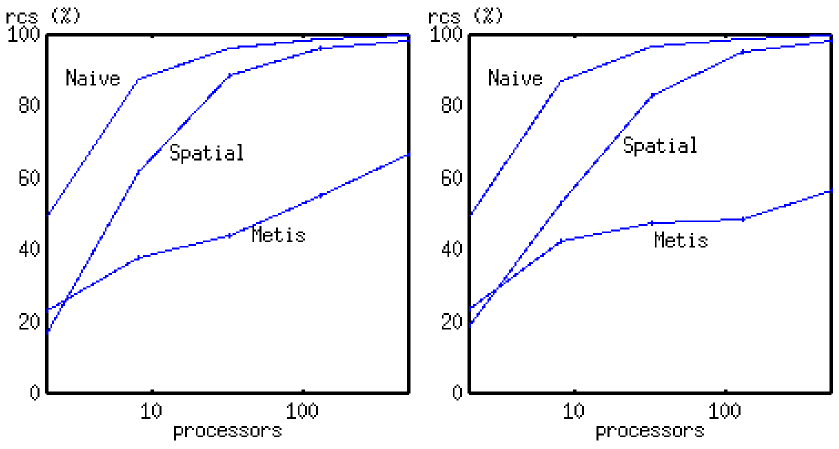


Figure 6.10: Relative cut size for nine room scene in the first (left) and third (right) iteration.

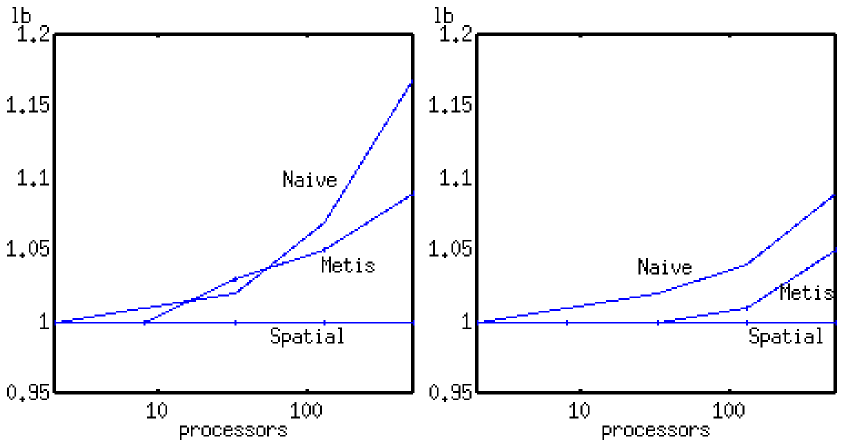


Figure 6.11: Load balance of link vertices for the nine room scene in the first (left) and third (right) iteration.

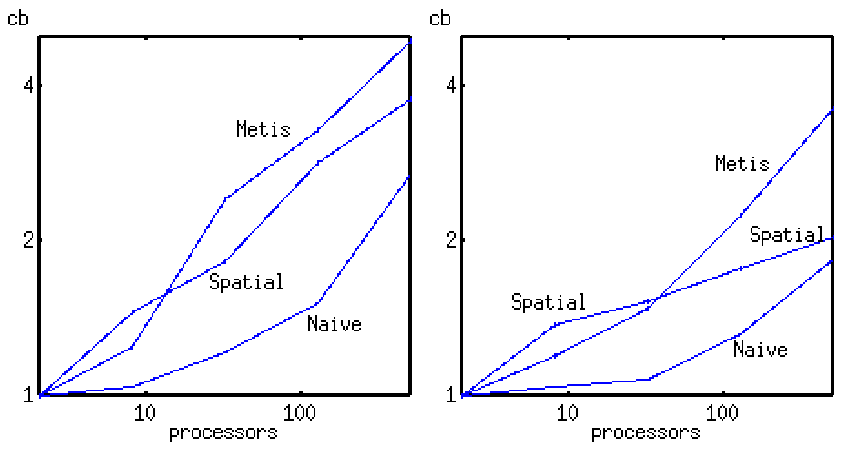


Figure 6.12: Connectivity balance for the nine room scene in the first (left) and third (right) iteration.

marginal. For large p the number of vertices per processor gets too small to achieve a good load balance with Metis. Nevertheless, the spatial partitions are very well load balanced even for large p.

The connectivity balance of the above scenes and iterations is compiled in Fig. 6.12. The connectivity balance in the first iteration is worse than in the third. Metis produces the worst balance, while SP achieves better balances. NP distributes partition connectivity evenly, too.

In summary, Metis produces fairly well load balanced partitions, whose communication balance is bad and whose cut size is best. SP produces excellent load balance and good communication balance, while the cut size is bad for large p.

We examine the combined effect of communication balance *and* cut size by considering the *maximum per vertex remote connectivity*, which is defined as

$$mvc(\pi) = \max_{0 \leq j \leq p-1} \{conn(V_j)\} \frac{p}{W(V)} \quad ,$$

where $W(V) = \sum_{0 \leq i \leq h-1} W_i(V)$ denotes the accumulated components of the weight tuples. The function *mvc* relates the maximum communication overhead on any processor to the average weight of vertices per processor. Hence, *mvc* is the number of remote data accesses per task during a given iteration. This of course is only an approximation of real communication overhead per task, because congestion may slow down data transport (see the following section).

Figure 6.13 shows the function *mvc* for various partitions. We see, that Metis achieves slightly better values than SP for midrange p. But for small and large p the difference gets small or the spatial solution even outperforms the Metis solution. All curves are close together indicating that the communication volume is high regardless of the partitioning method. Important to note, that the curves increase, when p increases. For large p the HRA imposes more communication on each single task.

The absolute value of *mvc* is large. On the average there is no task, that needs not communicate with other processors. In fact, many tasks communicate several times across processor boundaries.

Other applications, such as finite element methods defined on a planar graph, are often partitioned using a geometrically oriented *domain decomposition* method. Then only the “boundary vertices” need to communicate to remote processors, leading to *mvc* values that are significantly smaller than 1. Viewed this way, the HRA is worse partitionable than other applications. Of course, this is mainly due to the fact that the lines of sight between every pair of surfaces are potentially unblocked.

6.6 Measuring Congestion

Congestion occurs in a parallel environment, if the bandwidth of the network N connecting p processors is too low to process injected data just in time. In the previous section we demonstrated that the total communication volume of the HRA is large. Hence, we need a network N with high bandwidth.

Scalable supercomputers dispose of incomplete networks. Each processor can directly communicate with every other processor, but if all processors communicate at once, congestion will occur, because communication links are shared between processors.

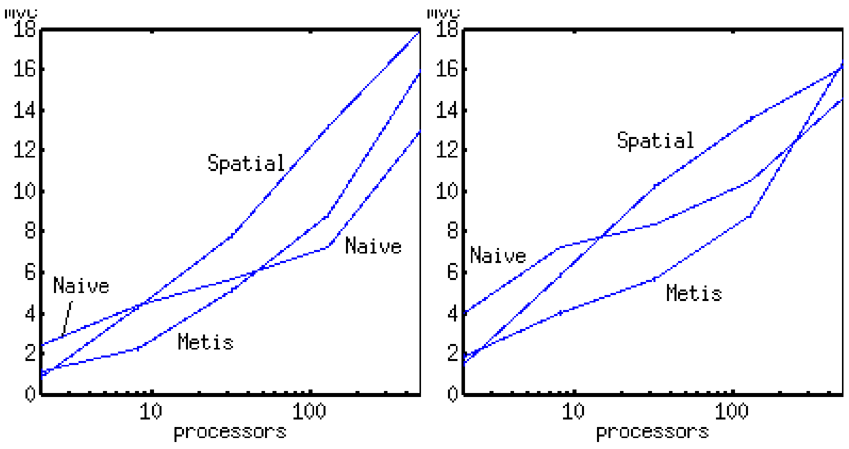


Figure 6.13: Maximum per vertex remote connectivity for nine room scene in the first (left) and third (right) iteration.

The chance of congestion depends on the specific network topology in use. For instance a network consisting of a single shared bus connecting all processors is highly prone to congestion. Conversely more expensive networks, such as a butterfly network, reduce the probability of congestion. Since we do not want to concentrate on a specific network topology, we simply characterize a network N by its maximum number of (hyper-)edges, denoted by $e(N)$.

A bus topology has only one single edge, i. e. only one pair of processors can communicate at a time without interfering any other communicating processors. A d-dimensional hypercube with $p = 2^d$ processors has $\frac{1}{2} p \log p$ edges. In a wrap-around-connected mesh each processor has four neighbouring processors; the total number of edges therefore is $2p$. A wrapped butterfly network has $p = d2^d$ nodes and $2p = d2^{d+1}$ edges [68]. Also other multistage networks, such as cube-connected-cycles, shuffle-exchange, and de-Brujin, have an edge number that is linear in the number of nodes.

Usually a processor executes tasks that communicate with tasks on different remote processors. In the worst case, a processor must communicate with *all* other $p-1$ processors. We will use the term *channel* to describe a pair of indices of potentially communicating processors $\{i, j\}$. A multicomputer of p processors has $\frac{1}{2}p(p-1)$ channels. Some of these channels may be unused.

An application that communicates over all $\frac{1}{2}p(p-1)$ channels is highly prone to congestion, since the interconnection network N provides significantly less independent channels. In a *locally communicating* application, every processor communicates to only a small subset of all processors. In such applications the number of used channels is only a small fraction of all channels.

We will count the number of different channels, that are used by a given partition π of a task access graph. The *channel cut size* of a given channel $\{i, j\}$, $i, j \in \{0, \ldots, p-1\}$, $i \neq j$, is defined as as:

$$ccs_{ij}(\pi) = \sum_{\{v,w\} \in E, \pi(v)=i, \pi(w)=j} W(\{v,w\}) \quad .$$

The channel cut size is the weight of all edges running between two given parts of a partition. If the channel cut size is zero, then the channel is not used by the task access graph at all. We define the *channel usage* of a partition as the number of channels with nonzero channel cut size:

$$cu(\pi) = \left| \left\{ \{i, j\} : ccs_{ij}(\pi) \neq 0 \right\} \right| \quad .$$

The number of channels $cu(\pi)$ is desired not significantly larger than the number of available edges $e(N)$ of a network. Otherwise, there is a high probability of congestion in the network N. Since we want to abstract from a particular network topology, we assume a network N, where the number of edges is linear in p, i. e. $e(N) = \gamma p$. This is a resonable assumption for a scalable network N.

Figure 6.14 plots the ratio $\frac{cu(\pi)}{p}$ against different numbers of processors, as measured in the graphs $H_{1/3}^{9-\text{room}}$. The curves for Metis and NP increase rapidly. This means that the probability of congestion is very high for large processor numbers.

SP produces partitions that exploit locality in the graph. Hence, the number of channels used per processor does not grow that dramatic as for the other partitioners. Nevertheless, the curves increase, which means an increased chance of congestion. When comparing the absolute values, we see that in the third iteration the channels are a bit more overused than in the first iteration.

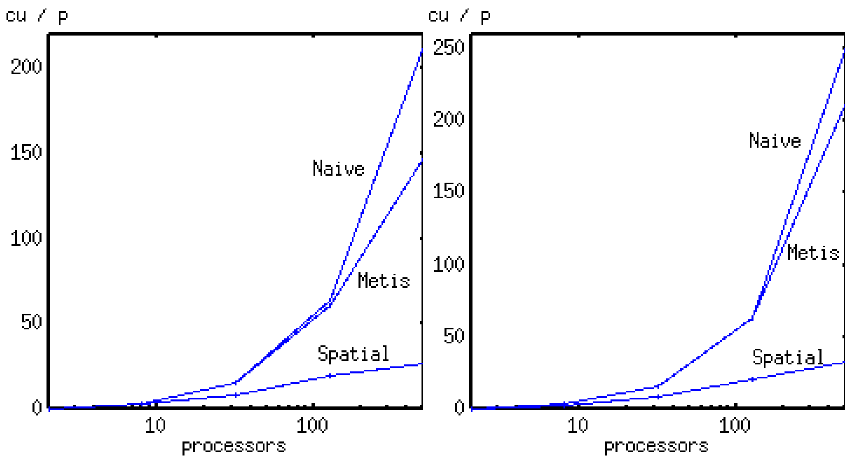


Figure 6.14: Channel usage $\frac{cu(\pi)}{p}$ for nine room scene in the first (left) and third (right) iteration.

graph	p	runtime	$lb(\pi)$	$rcs(\pi)$
AIRFOIL1	16	<1 sec.	1.03	4.5%
	128	<1 sec.	1.02	19.0%
BCSSTK30	16	1.5 sec.	1.03	8.1%
	128	2.8 sec.	1.04	28.0%

Table 6.15: Partition of graphs into 16 and 128 parts using Metis.

Finally we are going to analyze the bandwidth requirements of the HRA. We therefore relate the maximum per vertex remote connectivity to the average runtime spent on each task. Let $p = 512$ – the highest number of processors in the diagrams. We examine the Metis partition for the nine room scene in the third iteration in more detail. The maximum per vertex remote connectivity for this iteration is $mvc = 16.6$ (Fig. 6.13 right), i. e. every task handles 16.6 messages on the average in the third iteration. The total runtime is approx. 6,900 seconds (Fig. 6.4) in the third iteration. The total vertex weight is $W(V) = 428,317$ [41]. Hence, we have a runtime of $\frac{6{,}900\,\text{seconds}}{428{,}317} = 16.1$ milliseconds per unit vertex weight. Rating mvc by this runtime we get a number of $\frac{16.6\,\text{messages}}{16.1\,\text{milliseconds}}$, i. e. 1,031 messages per second.

Individually sending these 1,031 messages per second would result in large startup overhead. Consider for example the CM-5 machine with a startup time of $90\mu s$ [25]. Then about 93 milliseconds are spent on startup overheads per second of useful computations. This assumes zero latency and no congestion in the network. But, very probably there will be congestion.

Hence, message bundling is strictly necessary. A single processor communicates with 213 other processors on the average during the third iteration (Fig. 6.14 right). Every individual processor ($p = 512$) spends at least $\frac{6{,}900}{512} = 13.5$ seconds in the third iteration. Bundling all communicated data into 213 messages results in only $\frac{213}{13.5} = 15.8$ startup overheads (1.42 milliseconds on the CM-5) per second.

6.7 Graphs From Other Domains

In previous sections we discovered, that Metis generates partitions with relatively large cut sizes. In order to relate these cut sizes to those that could be achieved on other graphs, we consider here two graphs from other domains.

AIRFOIL1 : A 2D unstructured finite element mesh. 4,253 vertices – 12,289 edges [52] (see also Fig. 1.4 on page 3).

BCSSTK30 : Stiffness matrix from a statics model of an off-shore generator platform. 28,924 vertices – 1,007,284 edges [29].

Table 6.15 shows the partitioning results obtained with Metis for $p \in \{16, 128\}$. The runtime is very fast. The relative cut sizes are much better than for the HRA graphs in Table 6.9. Hence, we may conclude, that the HRA is worse partitionable than other "standard" problems.

6.8 Conclusions

The material in this chapter confirms in a quantitative approach that the HRA is poorly partitionable. We showed that a simple spatial partitioning method achieves results comparable to those of a costly graph partitioner like Metis.

The situation is very bad when incorporating visibility accesses from links to elements into the graph. Most results in this chapter hence concern the case, where visibility tasks are assumed to be performed locally without communication.

The HRA is a very dynamic algorithm. But, as shown by the load continuity matrices, it is worth spending time on load balancing, since it is likely, that this effort pays off in later iterations.

The total communication volume (cut size) of the HRA cannot be reduced arbitrarily, even if it were possible to use a costly graph partitioner such as Metis. A partitioner must take into account load balance and communication balance while reducing the cut size. The load balance is best for SP. The maximum per vertex remote connectivity plots (Fig. 6.13) summarize the communication overhead and show, that the Metis partitions are not much better than the spatial partitions. But calculation of Metis partitions would be much more time consuming. Also an important practical advantage of SP is that it avoids usage of *ghost objects* for book-keeping purposes (end of Sect. 6.2). The congestion plots ultimately suggest the use of SP because of a much smaller chance of congestion. In the following chapter we will describe a SP with provably small overhead.

Chapter 7

Locality Preserving Dynamic Load Balancing With Provably Small Overhead

Contents

In this chapter we describe a kind of an orthogonal recursive bisection clustering as it was published first in [38, 39, 40]. We show that the dynamic adaption of the clustering to changing load situations involves only small overhead. As a spatial clustering it is well suited to applications with local communication. Equally, hierarchical shooting was shown in Sect. 5.1.3 to benefit from such a clustering strategy.

We will describe a procedure that rebalances a database of geometric objects “on the fly” concurrently with an ongoing application. We describe procedures for imbalance detection and for locality preserving rebalancing. We analyze these procedures with respect to their computational time complexity. The estimated amortized computational time complexity of these procedures is shown to be small for every single dynamic task update. Such a theoretical result is useful, since it proves that the orthogonal recursive bisection strategy — which is used frequently in practice — also has a theoretically perfect background. The study in [71] with a very similar goal for dynamic load balancing without task interdependencies has given the initial impulse for our work.

A description of the load balancing algorithm and its concurrent execution with a user-defined application is treated in Sect. 7.1. In Sect. 7.2 a mathematical proof of the amortized complexity of each single dynamic update is given. Chapter 8 comprises the definition of a concrete simple application and experimental results.

7.1 Algorithms

We assume an application that can be described by a set of equally complex tasks. We gain load balance by clustering the tasks with respect to their associated geometric location. Dynamic updates are monitored by an imbalance detector. If an imbalance occurs, then a simple rebalancing takes place that relocates cluster boundaries thereby preserving locality.

The above technique is similar to *partial rebuilding* [81], which is used in keeping classical dynamic multidimensional databases balanced, and is itself derived from the general concept of dynamizing static data structures [73].

```
PQ = part of all initial tasks;
while (not finished)
  while (pending())
     receive(task);
     PQ <- task;
  endwhile
  perform first task of PQ locally;
  if (unbalanced()) then init-rebalance();
endwhile
```

Figure 7.1: Scheduling loop.

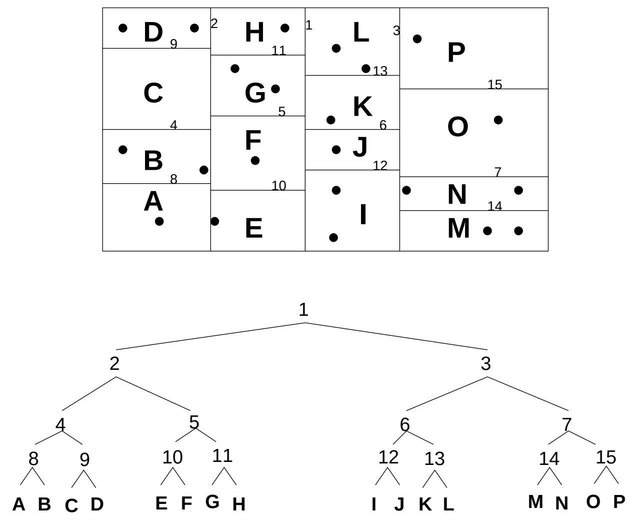


Figure 7.2: Tree clustering for $p = 16$ processors.

7.1.1 Embedding Of Load Balancing In An Application

On every processor a priority queue `PQ` stores all tasks that are to be executed on the processor. A scheduling loop that is executed asynchronously on every processor looks for incoming messages and executes the tasks (Fig. 7.1). Execution of a task may involve dynamical creation or deletion of tasks. This may cause imbalance of workload and maybe a rebalancing operation is initiated. The detection of imbalance (predicate `unbalanced`) is described in Sect. 7.1.4. The call to `init-rebalance` starts the rebalancing operation. It involves sending of messages to other processors and then returns immediately. The messages are processed by the destination processors within their regular scheduling loop. So, rebalancing does not synchronize any processors. It is performed besides the normal computations. This leads to the desirable effect that latency times are overlapped by computations.

7.1.2 The Clustering Method

The set of points in $\mathbb{R}^k$ representing tasks or objects[1] is stored in a binary search tree of limited depth. We will assume that there exists some integer l such that the number of processors is $p = 2^{kl}$. The real implementation is free of this restriction, but here it simplifies the presentation of the concepts.

We split the objects into 2^l subsets based on the value of the first coordinate (Fig. 7.2). These subsets are stored in the leaves of a binary tree of height l. At the next l levels we split the subsets with respect to the second coordinate. The last l levels of the tree are formed using the k-th coordinate. We arrive at a binary tree of height $h := kl$.

In theory it is often assumed that the points are *in general position*, because then there exists a perfectly balanced clustering. If this is not true in practice, but the maximum number of points with equal value in some coordinate is much smaller than N, then only small imbalances will occur. Hence, we will neglect the problem and assume that the points are in general position.

It is not a trivial task to construct an initial static multidimensional tree of points in a parallel manner. Possible solutions to this can be found e. g. in [3] and are not in the scope of this thesis.

[1] In realistic applications usually tasks are associated with some data stored in an “object”. This is why we use the terms task and object synonymically.

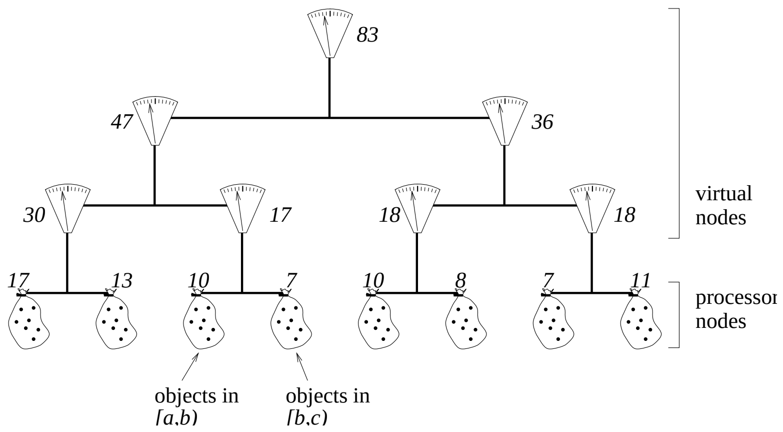


Figure 7.3: A database as a tree of scales. Each leaf node contains the objects of a processor. Inner nodes measure imbalance between children. The numbers are example values for the number of objects $E(\cdot)$ associated with the nodes.

```
diff($v,a,b$) := $\lfloor \frac{1}{2}\left((E(v_r)+b)-(E(v_l)+a)\right)\rfloor$

shift($v,a,b$):
  if ( $v$ is leaf )
   then execshift($v,a,b$)
   else $M$ :=diff($v,a,b$)
        shift($v_l,a,+M$)
        shift($v_r,-M,b$)
  endif
```

Figure 7.4: Strict shifting with respect to external shifts.

7.1.3 Rebalancing

Let us interpret the database of objects as a tree of scales (Fig. 7.3), where each inner node measures the imbalance between the two children and each leaf node contains the objects out of a subinterval of $\mathbb{R}^k$.

Definition 1 *Let $E(v)$ denote the number of objects that are associated with a node v. For inner nodes $E(v)$ is the sum of the $E(\cdot)$ values of the two children v_l and v_r. We call an inner node v perfectly balanced, if $|E(v_l)-E(v_r)| \leq 1$.*

The balance of an inner node v is defined as $B(v) := \frac{\min(E(v_l),E(v_r))}{\max(E(v_l),E(v_r))}$. We call an inner node v balanced, if $B(v) \geq 1-\beta$ for some constant $0<\beta\leq 1$.

If we could guarantee that at any time every inner node of the tree is balanced, then all processors get roughly the same amount of objects. Assume, that an inner node v of the tree has got out of balance. We can bring v back to perfect balance by shifting $M := \frac{1}{2}\left|E(v_r)-E(v_l)\right|$ objects from the overloaded child to the underloaded child. Since inner nodes are virtual nodes not containing real objects, the shift has to take place at the bottom level of the tree. The question, which leaf should deliver the objects to be shifted, is answered by the ordering inherent in our tree clustering, where the objects below v_l are smaller in some fixed coordinate than those below v_r. We treat the one-dimensional case first.

7.1.3.1 The One-Dimensional Case.

Let v denote an unbalanced inner node. E. g. let $\beta = 0.6$, then the node for which $E(v) = 47$ in Fig. 7.3 is unbalanced. v can be balanced perfectly by shifting $M := \frac{1}{2}|17-30| \approx 6$ objects from the left to the right subtree. The source of objects is the rightmost leaf below v_l, the target is the leftmost leaf below v_r. The resulting situation is: $E(v)=47$, $E(v_l)=30-M=24$, $E(v_r)=17+M=23$, $E(v_{ll})=17$, $E(v_{lr})=13-M=7$, $E(v_{rl})=10+M=16$, $E(v_{rr})=7$. Unfortunately, now the nodes v_l and v_r both are unbalanced. Rebalancing these two nodes can be done in parallel independently of each other in the next step.

We describe a procedure shift(v,a,b) that is applied recursively to all inner nodes v and is responsible for rebalancing v and all its descendants (Fig. 7.4). The arguments a and b represent two shifts that act from the outside to the leftmost or rightmost leaf below v, respectively. The rebalancing of v takes place under the

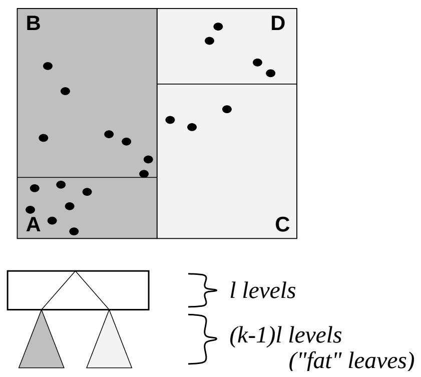


Figure 7.5: Higher dimensional rebalancing is reduced to k one-dimensional phases.

environment conditions that the leftmost leaf below v has received or will receive a objects from its left neighbour and the rightmost leaf below v gets b objects from its right neighbour. Negative values a, b mean, that the leftmost or rightmost leaves are delivering $|a|$ or $|b|$ objects to their neighbours.

If v is a leaf, the shifts a and b are executed directly (`execshift`(v, a, b)) by sending up to two messages containing objects to the neighbours. Of course for negative a or b no message is sent, but the leaf will receive objects from its neighbours in the near future. The leaf does not wait for message arrival, but will receive the message in its regular scheduling loop.

If v is an inner node, we calculate the number of objects M that need to be shifted internally between v_l and v_r and then pass M to the children of v in appropriate order. The initial call is `shift(`$r, 0, 0$`)`, where r denotes the root of the tree.

There are two open questions in how to realize `shift`. The first is, who executes the code of `shift` for the inner nodes v. This can be answered easily by mapping the `shift` calls for an inner node v to the leftmost leaf below v. Then every processor is responsible for not more than h inner nodes. The second question is, what has to be done, if at any leaf w the procedure `execshift`(w, a, b) cannot be performed immediately, because w has to deliver more objects than are available locally, e. g. $a < 0$ and $E(w) < |a|$? Then w has to postpone its `execshift` call until it gets objects from its immediate right neighbour. A detailed code of `execshift` is discussed in [38].

7.1.3.2 The k-Dimensional Case.

In $k > 1$ dimensions every inner node contains a hyperplane that is orthogonal to one of the k axes. The associated set of objects is divided into two subsets – one with locations above the hyperplane and one with locations below it.

As in the one-dimensional case we have to move objects between processors if any inner node v has got out of balance. The number of objects to be moved is the same as in the one-dimensional case but not the source of the objects. Consider a tree of height 2 storing points in $\mathbb{R}^2$ as sketched in Fig. 7.5. Assume that the root r is unbalanced. We can bring r to perfect balance by shifting $M = 3$ objects from $A \cup B$ to $C \cup D$. The objects to be shifted are those with largest value in the first coordinate. The problem with this is that it is not known which of the source processors contain the M largest objects with respect to the first coordinate. In order to determine the M largest objects processors A and B will have to communicate.

The clustering as described above is a tree of height kl with k groups of l levels each dividing the space into slabs with respect to a fixed coordinate. We will isolate the first group of l levels (Fig. 7.5) and regard the processor sets beneath consisting of subtrees of height $(k-1)l$ as ”fat” leaves. The algorithm of Fig. 7.4 is applied to this tree of height l. The only difference to the one-dimensional case is the implementation of `execshift`(v, a, b). Here each fat leaf has to determine its M largest or smallest objects in a communication operation before shifting the objects to a neighbouring fat leaf.

After the shifts with respect to the first coordinate have completed, then recursively the subtrees in the fat leaves are balanced independently in the same manner. We need k phases to balance the whole tree. In the k-th phase we will use the ordinary one-dimensional version of `shift`.

7.1.4 Detecting Imbalance

If once a balanced binary tree is given, how can we detect that any node of the tree gets out of balance? The problem with this is that imbalances result from unpredictable updates that are made independently by the leaves

```
diff($v,a,b$) :=
  if $\left(1-\frac{\beta}{2^{d(v)+1}} \le \frac{E(v_l)+a}{E(v_r)+b} \le \frac{1}{1-\frac{\beta}{2^{d(v)+1}}}\right)$
    then return 0
    else return $\lfloor \frac{1}{2}\left((E(v_r)+b)-(E(v_l)+a)\right)\rfloor$
  endif
```

Figure 7.6: Relaxed shifting. $d(v)$ denotes the length of the path from v to the leaves.

of the tree. The following lemma states a necessary condition for the situation that some inner node gets out of balance.

Lemma 2 *Assume that each node of the binary tree was balanced at time t_0. Let $E^0(v)$ denote the load at an inner node v at time t_0. Let r denote the root of the tree.*
If v is unbalanced at any time t after t_0 then there exists some leaf w below v for which

$$E(w) \notin \left[\frac{E^0(r)}{p}2\frac{1-\beta}{2-\beta}, \frac{E^0(r)}{p}2\frac{1}{2-\beta}\right] \quad .$$

Proof Assume, that the object number in all leaves below v is covered by the above interval at time t. For $u \in \{v_l, v_r\}$ let $d(u)$ denote the length of the path from u to the leaves. Then $E(u)$ is covered by $2^{d(u)}\,[\min E(w), \max E(w)]$, where the minimum and maximum covers all leaves w below v. Using definition 1 we can conclude that

$$B(v) \quad = \quad \frac{\min(E(v_l),E(v_r))}{\max(E(v_l),E(v_r))} \ge \frac{2^{d(u)}\min_w E(w)}{2^{d(u)}\max_w E(w)} \ge \frac{2^{d(u)}\frac{E^0(r)}{p}2\frac{1-\beta}{2-\beta}}{2^{d(u)}\frac{E^0(r)}{p}2\frac{1}{2-\beta}} = 1-\beta,$$

i. e. v is balanced.

□

The obvious strategy derived from this lemma to maintain balance in the whole tree is the following. Every leaf tracks whether its number $E(\cdot)$ is covered by the above narrowing interval. The tracking does not require any communication. We say a leaf *leaves* the narrowing interval, if its number $E(\cdot)$ is not covered by the interval. If any leaf leaves the interval, we apply the function `shift`$(r,0,0)$ to the root r in order to rebalance all nodes of the tree. Then we define a new narrowing interval by setting $E^0(r) := E(r)$.

7.1.5 Improving Average Efficiency

There are update patterns for which the above strategy is best. E. g. consider an update pattern that starts in perfect balance (load is $\frac{E^0(r)}{p}$ everywhere) and proceeds such that all leaves below the left son of the root create objects at the same rate and all leaves below the right son of the root remove objects at this rate. All leaves will leave the narrowing interval at the same time and then the root will be unbalanced. Now, rebalancing the tree perfectly is justified.

But, consider another update pattern where all leaves create objects at the same rate. All leaves will leave the narrowing interval at the same time, but there is no need for any shifts, since all inner nodes are balanced. We will slightly modify the `shift` procedure by replacing the `diff` function as shown in Fig. 7.6. Now, a small imbalance at node v is ignored. This prevents slightly unbalanced nodes being rebalanced perfectly which would involve expensive, but useless small messages. Below we will see that using this relaxed version does not worsen the worst case behaviour significantly. But we expect the average behaviour of the relaxed version to perform much better in realistic applications than the strict version. Therefore we used the relaxed version in our experiments in Chap. 8.

7.2 Complexity Analysis

We will examine the complexity of the above mechanisms for detecting and rebalancing an imbalanced distribution of objects. In Sect. 7.2.1 we calculate the cost of a complete rebalancing operation. Fortunately, we will see that, after rebalancing is complete, several updates are possible without any rebalancing being necessary. We quantify this number separately for the strict case (Sect. 7.2.3, Lemma 7) and the relaxed case (Sect. 7.2.4, Lemma 10). We will charge the communication cost of a rebalancing operation to the updates. In Sect. 7.2.2 we show that every single update has a small *amortized time complexity*. Theorem 8 in Sect. 7.2.3 and Theorem 11 in Sect. 7.2.4 specialize the result separately for the strict and the relaxed case.

7.2.1 Absolute Cost Of Rebalancing

First, we define the cost of different message types in our cost model (see Sect. 3.3).

Definition 3 *The above algorithm uses two types of messages: administrative messages containing small amounts of data and object messages containing objects. For the subsequent estimations we will assume a cost of* $C_{\text{adm}} = o + (A-1)g$ *for an administrative message where* A *denotes the maximum memory size of any administrative message. The cost of an object message is assumed to be* $C_{\text{obj}}(M) = o + (MS-1)g$ *where* S *denotes the memory size of an object and* M *is the number of objects contained in the message.*

These formulas assume that the cost of a message consists of a fixed overhead portion and a length-dependent bandwidth portion. We dropped the latency term, since we assume throughout our complexity considerations that latency can be hidden by computation (see discussion at the end of Sect. 3.3).

Lemma 4 *Let* M *denote the maximum number of objects contained in any shift between any two nodes during a complete rebalancing operation. Then the total cost of rebalancing the tree perfectly is not larger than*

$$(2k + (k-1)h)C_{\text{obj}}(M) + (k+2)hC_{\text{adm}} \quad .$$

Proof The first action in order to rebalance the tree perfectly is an administrative message that is cast to all processors, which informs them that the rebalancing starts. This takes hC_{adm} cycles per processor if collective communications on 2^h processors are simulated by h point-to-point communications.

Then a combine operation calculates the array of current numbers $E(\cdot)$ (cost hC_{adm}). After that recursive calls of `shift` pass the arguments a and b to all nodes of the tree. The cost of all recursive calls is less than hC_{adm}.

The cost $C_{\text{shift}}(d)$ of a single shift between two fat leaves v_{src} and v_{dest} depends on the height d of the involved fat leaves.

A shift consists of four phases:

i. extracting objects from the source leaves upwards to v_{src},

ii. moving objects to v_{dest},

iii. inserting below v_{dest} and finally

iv. removing below v_{src}.

Extraction (i.) and insertion (iii.) are performed independently on different processors and can be implemented as a filtering combine or a cast operation in $dC_{\text{obj}}(M)$ time. Moving (ii.) takes time for one message $C_{\text{obj}}(M)$. Removing (iv.) at the leaves below v_{src} involves dC_{adm} for casting a small message containing the actual split coordinate. The leaves are then able to remove the correct objects.[2]

The total per-processor-cost of the shift is

$$\begin{aligned} C_{\text{shift}}(d) &= \max\{dC_{\text{obj}}(M) + C_{\text{obj}}(M) + dC_{\text{adm}}, C_{\text{obj}}(M) + dC_{\text{obj}}(M)\} \\ &= (d+1)C_{\text{obj}}(M) + dC_{\text{adm}}. \end{aligned}$$

We sum over the heights of all fat leaves involved with shifts. The height of fat leaves varies from $(k-1)l$ in the first phase to 0 in the k-th phase. Every fat leaf is involved in up to two shifts (one to the left, one to the right). The sum of heights is:

$$\begin{aligned} 2\sum_{0\le j\le k-1} C_{\text{shift}}(jl) &= 2C_{\text{obj}}(M)\sum_{0\le j\le k-1}(jl+1) + 2C_{\text{adm}}\sum_{0\le j\le k-1}(jl) \\ &= (2k + (k-1)h)C_{\text{obj}}(M) + (k-1)hC_{\text{adm}} \quad . \end{aligned}$$

[2] The focus of this analysis is communication overhead. Hence we will not count the overhead of removing the objects locally on a processor.

Now, the proof is completed by summing up all above costs:

$$\begin{aligned}&3hC_{\text{adm}} + (2k + (k-1)h)C_{\text{obj}}(M) + (k-1)hC_{\text{adm}}\\&= (2k + (k-1)h)C_{\text{obj}}(M) + (k+2)hC_{\text{adm}} \ .\end{aligned}$$

□

7.2.2 Amortized Cost Of Rebalancing

Lemma 5 *Let t_0 denote a time, when the whole tree was perfectly balanced. Assume for some $t > t_0$ that a rebalancing operation is started and that U updates have happened in the leaves since t_0. Then the maximum number M of objects that have to be shifted between any two nodes during the rebalancing is bounded by $M \le U$.*

This result is easily seen by considering the border between any two neighbouring leaves. If not more than U updates happened to the left and to the right of the border, then not more than U objects have to move across the border to re-establish perfect balance.

Now we divide the cost of rebalancing (Lemma 4) by the number of updates U between two consecutive rebalancing operations.

Corollary 6 *The amortized time complexity of a single update is not larger than*

$$(2k + (k-1)h)Sg + \frac{((2k+1)(h+1)-1)C_{\text{adm}}}{U} \ .$$

Proof The amortized update time is the cost of a rebalancing operation (Lemma 4) divided by the number of updates U:

$$\begin{aligned}&\frac{(2k + (k-1)h)C_{\text{obj}}(M) + (k+2)hC_{\text{adm}}}{U}\\<\ &\frac{(2k + (k-1)h)(o + MSg) + (k+2)hC_{\text{adm}}}{U}\\\le\ &\frac{(2k + (k-1)h)(o + USg) + (k+2)hC_{\text{adm}}}{U}\\=\ &(2k + (k-1)h)Sg + \frac{(2k + (k-1)h)o + (k+2)hC_{\text{adm}}}{U}\\<\ &(2k + (k-1)h)Sg + \frac{((2k+1)(h+1)-1)C_{\text{adm}}}{U}\end{aligned}$$

Here we used Definition 3, Lemma 5 and the fact that $o < C_{\text{adm}}$ (Def. 3).

□

7.2.3 The Strict Case

Lemma 7 *Let t_0 denote a time, when the whole tree was perfectly balanced. When a leaf leaves the narrowing interval (Lemma 2) at time $t > t_0$, then at least $U \ge U^{\text{bound}}_{\text{strict}} := \frac{E^0(r)}{p}\frac{\beta}{2-\beta}$ updates have happened in the whole tree since t_0.*

Proof The minimum number of updates until a leaf may leave the narrowing interval is bounded by the difference of the borders of the narrowing interval (Lemma 2) and the initial number $\frac{E^0(r)}{p}$:

$$U \ge \min\left\{\frac{E^0(r)}{p}2\frac{1}{2-\beta} - \frac{E^0(r)}{p} \quad , \quad \frac{E^0(r)}{p} - \frac{E^0(r)}{p}2\frac{1-\beta}{2-\beta}\right\} = \frac{E^0(r)}{p}\frac{\beta}{2-\beta} .$$

□

Theorem 8 *Consider $p = 2^h$ processors containing a total number of E objects that are rebalanced by the function* `shift` *(Fig. 7.4). Then the amortized update time of a single update is not larger than*

$$(2k + (k-1)h)Sg + O\left(\frac{kph}{\beta E}\right)C_{\text{adm}} .$$

Proof We assume, that the tree is perfectly balanced at the beginning, i. e. $E(w) = Ep^{-1}$ for all leaves w. Combining Corollary 6 and Lemma 7 we get an amortized update time of

$$\begin{aligned}
& (2k+(k-1)h)Sg + \frac{((2k+1)(h+1)-1)C_{\text{adm}}}{U} \\
\le\ & (2k+(k-1)h)Sg + ((2k+1)(h+1)-1)\frac{2-\beta}{\beta}\frac{p}{E}C_{\text{adm}} \\
=\ & (2k+(k-1)h)Sg + O\left(\frac{kph}{\beta E}\right)C_{\text{adm}} \quad .
\end{aligned}$$

□

We interpret the Theorem. If $k = 1$, every update involves sending and receiving the updated object (cost $2Sg$) plus some overhead. If the dimension is increased by 1, additional $2+h$ send or receive operations are necessary in the worst case, because the object space has no canonical ordering. The overhead of sending and receiving messages is small if $\frac{kph}{\beta E}$ is small. We can reduce the overhead either by increasing β – which clearly increases load imbalances – or by storing many objects on few processors ($p << E$), which opposes to the demand of fast computation. Hence, we have a classical trade-off situation.

In Sect. 8.8 below we present experimental results of the above load balancer for a simple application. There we will see, that the bound of theorem 8 is relatively sharp for $k = 1$, but it gets more and more pessimistic, if k is increased. Hence, in a real application we fortunately will have much less overhead than predicted above.

7.2.4 The Relaxed Case

In order to improve the average behaviour of the balancing scheme we will use the relaxed version in Fig. 7.6. We will show, that then the number of updates between two successive rebalancing operations is still large.

Lemma 9 *Assume that all inner nodes v of the tree satisfy* $\texttt{diff}(v,0,0) = 0$. *Then the number of objects in every particular leaf is covered by the following interval:*

$$\left[\frac{E(r)}{p}\frac{4-3\beta}{2(2-\beta)}, \frac{E(r)}{p}\frac{4-\beta}{2(2-\beta)}\right] \quad ,$$

where $(h,\beta) \in \{1,\dots,20\} \times \left\{\frac{j}{10^6} : 0 \le j \le 10^6\right\}$.

Proof The condition $\texttt{diff}(v,0,0) = 0$ says that

$$1 - \frac{\beta}{2^{d(v)+1}} \le \frac{E(v_l)}{E(v_r)} \le \frac{1}{1-\frac{\beta}{2^{d(v)+1}}} .$$

By substituting $E(v_r) = E(v) - E(v_l)$ we get:

$$E(v)\frac{1-\frac{\beta}{2^{d(v)+1}}}{2-\frac{\beta}{2^{d(v)+1}}} \le E(v_l) \le E(v)\frac{1}{2-\frac{\beta}{2^{d(v)+1}}} .$$

Of course we get the same bounds for $E(v_r)$ by substituting $E(v_l) = E(v) - E(v_r)$.

Consider a fixed leaf w. The number of objects at w is bounded by

$$E(r)\prod_{2\le i\le h+1}\frac{1-\frac{\beta}{2^i}}{2-\frac{\beta}{2^i}} \le E(w) \le E(r)\prod_{2\le i\le h+1}\frac{1}{2-\frac{\beta}{2^i}} ,$$

where r denotes the root of the tree.

We have simulated the product formulas for the following set of parameters:

$$(h,\beta) \in \{1,\dots,20\} \times \left\{\frac{j}{10^6} : 0 \le j \le 10^6\right\} .$$

It turned out that for all these parameters

$$\begin{aligned}\prod_{2\le i\le h+1}\frac{1-\frac{\beta}{2^i}}{2-\frac{\beta}{2^i}} &\geq \frac{1}{2^{h+1}}\frac{4-3\beta}{2-\beta},\\ \prod_{2\le i\le h+1}\frac{1}{2-\frac{\beta}{2^i}} &\leq \frac{1}{2^{h+1}}\frac{4-\beta}{2-\beta}\end{aligned}$$

This proves the lemma.

□

This is now used for a bound on the number of updates for the relaxed case.

Lemma 10 *Let t_0 denote a time where the assumption of Lemma 9 is valid. When a leaf leaves the narrowing interval (Lemma 2) at time $t > t_0$, then at least $U \geq U^{\text{bound}}_{\text{relaxed}} := \frac{1}{2}\frac{E^0(r)}{p}\frac{\beta}{2-\beta}$ updates happened in the whole tree since t_0.*

Proof The minimum number of updates until a leaf may leaves the narrowing interval is bounded by the difference of the borders of the narrowing interval (Lemma 2) and the borders of the interval that was proved in Lemma 9.

$$\begin{aligned} U &\geq \min\left\{\begin{array}{c}\frac{E^0(r)}{p}2\frac{1}{2-\beta}-\frac{E^0(r)}{p}\frac{4-\beta}{2(2-\beta)},\\ \frac{E^0(r)}{p}\frac{4-3\beta}{2(2-\beta)}-\frac{E^0(r)}{p}2\frac{1-\beta}{2-\beta}\end{array}\right\}\\ &= \frac{1}{2}\frac{E^0(r)}{p}\frac{\beta}{2-\beta}.\end{aligned}$$

□

Theorem 11 *If the relaxed version of* `shift` *(Fig. 7.6) is used, then the worst case amortized update time is at most twice as large as the bound, which was stated in Theorem 8.*

Proof Let $C^{\text{bound}}_{\text{strict}}$ denote the bound on the cost of the whole rebalancing in the strict case (Lemma 4). Evidently, the relaxed version involves not more communication in the worst case than $C^{\text{bound}}_{\text{strict}}$, since the relaxed version simply omits some of the object shifts that are performed in the strict version.

The minimum number of updates in the relaxed case is $U \geq U^{\text{bound}}_{\text{relaxed}} = \frac{1}{2}U^{\text{bound}}_{\text{strict}}$ (see Lemma 7 and 10). The amortized update time is the cost C of a whole rebalancing operation divided by the number U of updates between two detected imbalance situations:

$$\frac{C}{U} \leq \frac{C^{\text{bound}}_{\text{strict}}}{U^{\text{bound}}_{\text{relaxed}}} = 2\frac{C^{\text{bound}}_{\text{strict}}}{U^{\text{bound}}_{\text{strict}}} \quad .$$

□

7.3 Conclusions

In this chapter we have described a very simple dynamic orthogonal recursive bisection rebalancing algorithm. The rebalancing can be performed asynchronously by point to point messages. Imbalance detection is done in a completely distributed manner, hence no global knowledge about the current load situation has to be distributed periodically.

Theorems 8 and 11 express the main theoretical results of this chapter. They say: every dynamic update is debited only by a small overhead. The overhead can be reduced by increasing the problem size E, by increasing the balancing parameter β, and by decreasing the number of processors p.

Fig. 7.7 shows the regions of overhead and parallel efficiency. In the lower left we have relatively large overhead, but a good parallel efficiency. In the upper right region the overhead is small, but the parallel efficiency is bad. Hence, at least from a theoretical point of view we have to choose β such that the overall speedup is best. As we will see from the experiments in the following chapter, many β-values lead to good speedups.

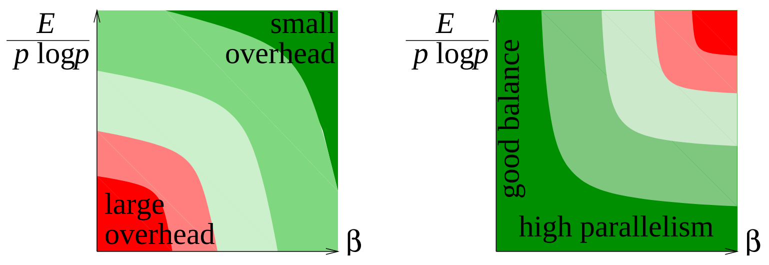


Figure 7.7: Tradeoff overhead against good balance and high parallelism.

Chapter 8

Experimental Results For A Simple Application

Contents

In this chapter we present experiments on a parallel computer, that have been published previously in [38, 39]. We have tested the relaxed version (Sect. 7.1.5) of the load balancing algorithm of the previous chapter for a simple application that treats a set of objects in k-dimensional space. We parameterized the application by realistic parameters, such as the load pattern, the number of objects, and the workload per task. In this chapter we show that for near-practice-parameterizations our load balancer achieves good speedups. Moreover in Chapter 9 we will present results on a real application — the hierarchical radiosity algorithm.

First in Sect. 8.1 we define the simple application. We will identify four different load patterns as representatives for real application's patterns[1]: "constant", "growing", "moderate", and "heavy". These classes are characterized by the minimum, maximum, average and median object density in different regions of k-dimensional space.

In the following Sections 8.3–8.8 we study the behaviour of the load balancer of Chapter 7 when various parameters are changed. Particularly as parameters we study the imbalance parameter β, the workload per object/task, the total number of objects/tasks, the load pattern, and the dimension. As a function of these parameters we mainly concentrate on the speedup, but also present results for the total runtime, the load balance, and the message traffic. We will see that the overhead, that was proved to be small in the worst case in the previous chapter, is even smaller in practice.

Finally in Sect. 8.9 we present the difference in speedup with and without rebalancing for the four above load patterns. We will see that our rebalancer improves poorly balanced applications a great deal, and it does not deteriorate the performance of an application, that is well balanced by itself.

8.1 A Simple Application

Our simple application treats a set of points (objects) in $[0,1]^k$. It comprises a few loops. Within each loop every object is treated once. The action for each object o consists of creating new objects close to o and/or deleting

[1] Later in Sect. 9.2 we will catch on to these patterns and classify the hierarchical radiosity algorithm.

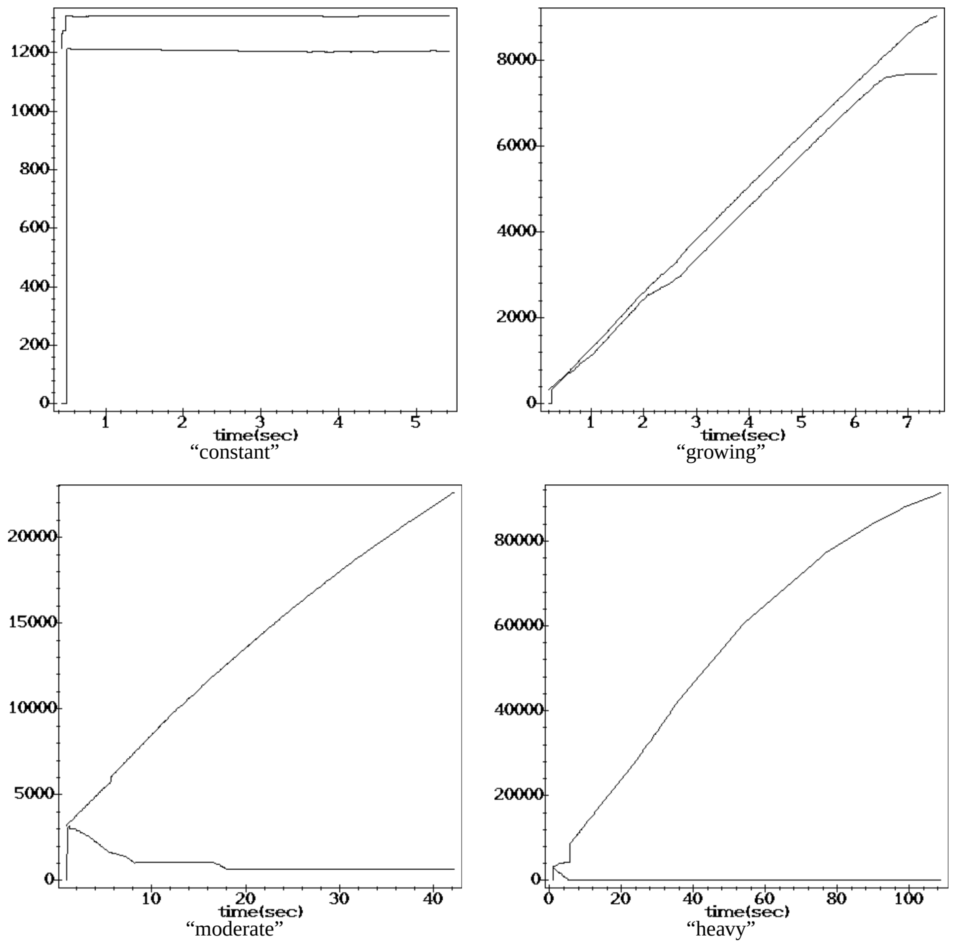


Figure 8.1: Minimum and maximum object density for different load patterns.

o. This simple application defines local dependencies between objects, because new objects are created *close* to the original object. Therefore clustering the objects on the processors leads to communication only between neighbouring clusters.

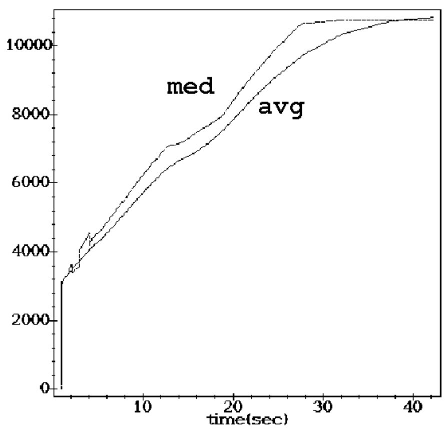
Several parameters affect the behaviour of the application. First, the number of objects affects the efficiency of the load balancing algorithm. Second, real applications impose tasks on objects that are much more complex than the task of creating a few objects near the original object. We will therefore provide a parameter that controls the amount of work to be done while treating a single object. Clearly, the speedup is expected to get better if more work is associated with each object.

Another important parameter is the load pattern generated by the application. We have examined various load patterns by assigning different *productivity* values to different regions of $[0,1]^k$. A productivity of 1 means, that after treating an object the expected number of objects is the same as before treating it. A productivity $prod > 1$ means that treating an object involves creating $prod - 1$ new objects on the average. If $prod < 1$ this means that the object under treatment is deleted with probability $1 - prod$.

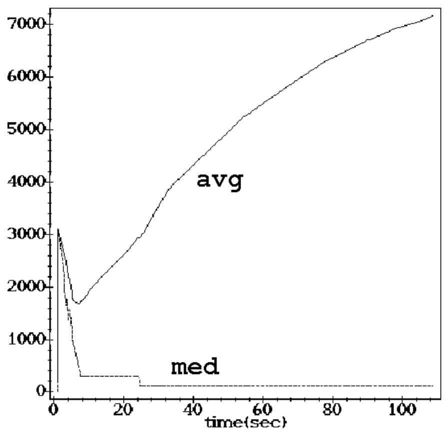
The effect of assigning productivity values to regions is a differently growing object density in different regions. We will characterize a load pattern by the minimum and maximum object density. Virtually we measured the number of objects contained in sixteen equally sized sub-hypercubes of $[0,1]^k$. Fig. 8.1 shows the minimum and maximum density vs. time. The first pattern labeled “constant” is characterized by a constant density in all regions. This was achieved by assigning an equal productivity of 1 to all objects. Thus, the number of objects is constant over time.

The second pattern labeled “growing” is characterized by an evenly growing density in all regions. This was achieved by assigning an equal productivity of 3 to all objects.

Both patterns do not really require any load balancing. For this reason we think of these patterns as *well natured* for parallel computing. But the following two patterns labeled “moderate” and “heavy” do require load balancing. These patterns show a great difference between the minimum and maximum object density. We generated the



“moderate”



“heavy”

Figure 8.2: Median and average density.

Processor Type	VR4400SC
Clock Rate	75 MHz
Memory	64 MB
Instructions (peak)	150 MIPS
Floating Point (peak)	50 MFLOPS
Transfer rate to network	40 MB/s
Network topology	Multi-Stage-Interconnection, 4x4 switches

Table 8.3: Data sheet of a node of NEC Cenju 3.

patterns by defining a productivity function of the following form:

$$prod(loc(o)) := prod_{max}\frac{decay^{(1-\frac{1}{k}\|loc(o)\|_1)}-1}{decay-1},$$

where $prod_{max} \geq 0$, $decay \geq 0$, k denotes the dimension, $loc(o) \in [0,1]^k$ denotes the location of object o and $\|x\|_1$ denotes the sum of elements of a vector x. For the “moderate” pattern we used $prod_{max} = 2.0$ and $decay = 0.1074$ and for the “heavy” pattern we used $prod_{max} = 5.6$ and $decay = 357.05$. It is not these formulas and parameters, but the shape of the minimum and maximum density curve in Fig. 8.1, that give the reader a clear idea of the underlying load pattern.

Since the lower two diagrams of Fig. 8.1 seem to be qualitatively similar, we introduce another *characteristic* of a load pattern: the average and median object density (shown in Fig. 8.2). For the “moderate” pattern the two curves are similar. For the “heavy” pattern the average density is much greater than the median density. We conclude that the “moderate” pattern is somehow smoother than the “heavy” pattern. Hence, we expect worse speedups for the latter.

Please note that the total number of objects is different for the four patterns. Hence, we cannot compare the absolute time and density values, but only the overall impression of the curves. All figures refer to two-dimensional databases ($k = 2$).

8.2 Environment

We implemented the application and the load balancing procedure in C++ in SPMD style using MPI on a NEC Cenju 3, a massively parallel system. The maximum number of processors available for this work was 64. A data sheet of the nodes of NEC Cenju 3 is shown in Table 8.3. The nodes are connected by a Multi-Stage-Interconnection Network that utilizes 4x4 switches. The transfer time between all nodes is constant.

8.3 Impact Of β On Runtime

We made experiments with varying values of β on 16 processors in $k = 2$ dimensions. The left part of Fig. 8.4 shows the impact of β on the runtime for an application of the “moderate” type. Let $T(\beta)$ denote the runtime

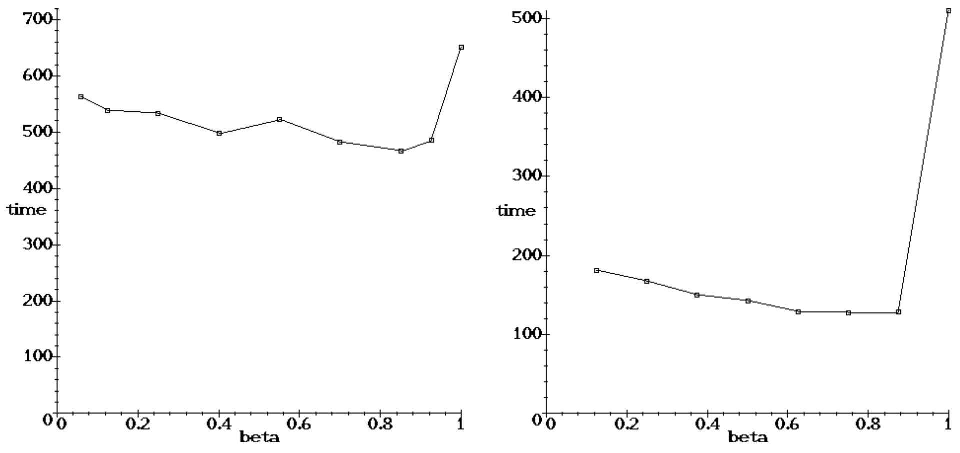


Figure 8.4: Runtime for different β's. (left: "moderate", right: "heavy").

as a function of β. The application started at 50,000 objects and finished at 173,000 objects meanwhile having performed 361,000 updates (deletions and insertions). Each object was loaded with a task that took 20 ms on a single processor.

For $\beta = 1$ we had no rebalancing at all and needed $T(1) = 650$ seconds. The best runtime was $T(0.85) = 466$ seconds. Hence, the runtime was reduced by 28 percent.

Assuming a perfect dynamic rebalancing procedure at no extra cost the best runtime achievable on 16 processors is $\frac{361{,}000 \times 20\text{ms}}{16} = 451$ seconds. Hence, our rebalancer balanced the application such, that it took only $\frac{466-451}{451} =$ 3 percent longer than the optimum.

We also made experiments with an application of the "heavy" type (20,000 objects at the beginning, 48,000 objects at the end, 94,000 updates). The workload per task was 20 ms. The runtime for various β is shown in Fig. 8.4 on the right. Without load balancing we needed $T(1) = 510$ seconds. Our load balancer achieved $T(0.75) = 127$ seconds or a reduction by 75 percent.

Assuming a perfect dynamic rebalancing procedure at no extra cost the best runtime achievable on 16 processors is $\frac{94{,}000 \times 20\text{ms}}{16} = 118$ seconds. Hence, our rebalancer balanced the application such, that it took only $\frac{127-118}{118} =$ 8 percent longer than the optimum.

From Fig. 8.4 we see that the impact of β on the runtime is not very critical. There is a broad range of β values leading to good runtimes.

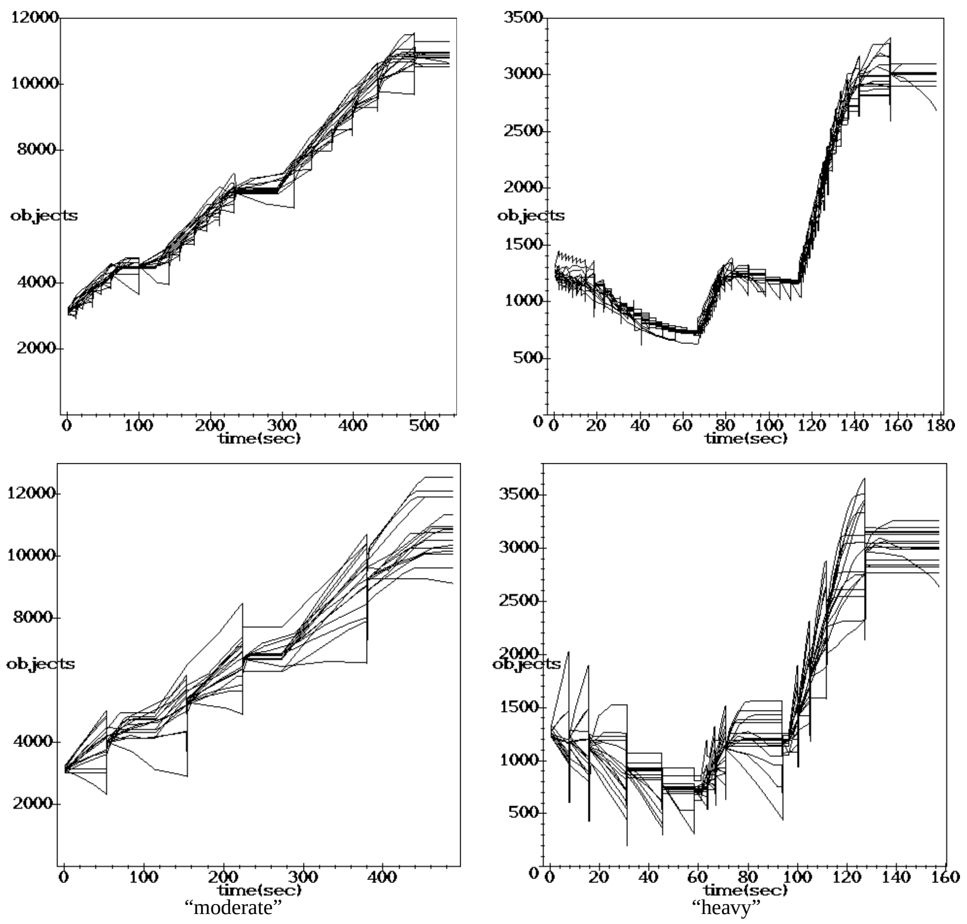
8.4 Impact Of β On Load Balance

For the two applications specified in Sect. 8.3 we studied in detail on 16 processors, how the load balancer distributes the objects across the processors. The task load was 20 ms, the parameter $\beta \in \{0.25, 0.75\}$. We measured the current number of objects on all 16 processors. Fig. 8.5 shows the results.

Evidently, our load balancer becomes active only, when the load balance gets bad (the peaks in the graphs). Then some or all nodes of the tree are rebalanced resulting in a convergence of the sixteen curves. We can see that a smaller value of β results in a better load balance. This is paid by a larger message traffic. To be specific, we counted the number of positive imbalance detection results and the number of shift messages and shifted objects during the application. Table 8.6 shows that a larger β as well as a smoother load pattern tend to reduce the traffic. The last column contains the average number of objects per shift. We can think of this number as an indicator of the efficiency of the load balancer. The less objects are transferred by a shift the larger is the communication overhead during load balancing.

8.5 Impact Of Task Load On Speedup

For a "moderate" application we measured the speedup for different loads per task. The free parameter β was fixed as a function of the number of processors $\beta(p) = (\log_2 p)^{-1}$.



“moderate” “heavy”

Figure 8.5: Influence of the load balancer (top: $\beta = 0.25$, bottom: $\beta = 0.75$) on the current load.

type	β	# positive imbalance detection results	# shifts	# shifted objects	avg. # obj. per shift
”mod.”	0.25	19	996	386687	388
”mod.”	0.75	5	242	308559	1275
“heavy”	0.25	58	3136	351255	112
“heavy”	0.75	14	734	293818	400

Table 8.6: Message traffic due to rebalancing.

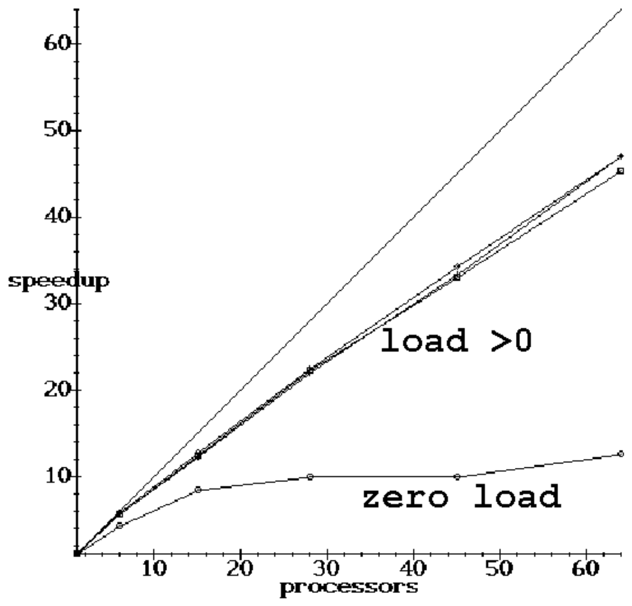


Figure 8.7: Speedup for different task loads (bottom up: 0, 5, 20, 50 ms) compared to ideal speedup.

Fig. 8.7 shows the speedup[2] for 50,000 initial objects, 361,000 updates and 173,000 final objects. We attached a load of 0, 5, 20 and 50 milliseconds (ms) to each task[3]. All nonzero loads performed well. The zero load was included because it gives an impression of the overhead of our load balancer. Even with zero loads we achieved a speedup of 9 on 15 processors.

8.6 Impact Of Object Number On Speedup

Again for a “moderate” application now we fixed the task load to 20 ms and set $\beta = \beta(p)$ as in Sect. 8.5. Fig. 8.8 shows the speedup for different numbers of objects. Comparable good linear speedups have been obtained for many objects (72,000, 361,000 and 1,081,000 updates).

We made two tests with few objects in order to study the behaviour of our load balancer on very lightweight applications. Concretely, the first one started with 100 objects, did 667 updates and finished at 289 objects. Fig. 8.8 shows the speedup for this application (the lower curve). On 64 processors the speedup is worse than on 45 processors. This could have been expected, since on 45 processors every processor only had to do 14.8 updates on the average and the runtime was low already (0.8 seconds). Clearly, such lightweight applications cannot benefit much from parallelization. At least, on 15 processors we achieved a speedup of 9.6.

The second lightweight application started with 1000 objects and arrived at 3400 objects after 7300 updates. Fig. 8.8 shows the speedup for this application (the slightly ”humpbacked” curve). Here the situation is better. The runtime varied from 138.7 to 3.2 seconds (1 to 64 processors). That is a speedup of 43.0 on 64 processors.

To get precise absolute data about the overhead of the load balancer, we started one application with no objects and no updates (not contained in Fig. 8.8). On 1 processor the runtime was 0.002 seconds, on 28 processors we needed 0.05 seconds and on 64 processors 0.15 seconds.

8.7 Impact Of Load Pattern On Speedup

We made a series of experiments for the four different load patterns described above. To get comparable results we parameterized the applications such that they performed an equal number of updates (about 360,000). The task load and β were set as in Sect. 8.6. Fig. 8.9 shows the speedups. As expected the speedup gets better for smoother load patterns. This is a reasonable and desireable feature of a load balancer.

[2] The speedup is defined as the ratio of $\frac{T_1}{T_p}$, where T_1 denotes the runtime needed on a single processor and T_p denotes the runtime needed on p processors. Ideally this ratio equals p.

[3] E. g. calculating 370 square roots and storing them in dynamic memory consumes 1 ms.

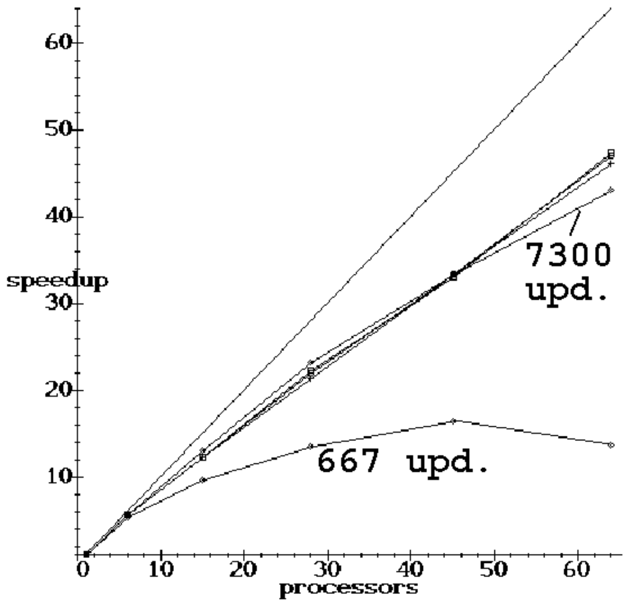


Figure 8.8: Speedup for different numbers of updates (667 … 1,081,000) compared to ideal speedup.

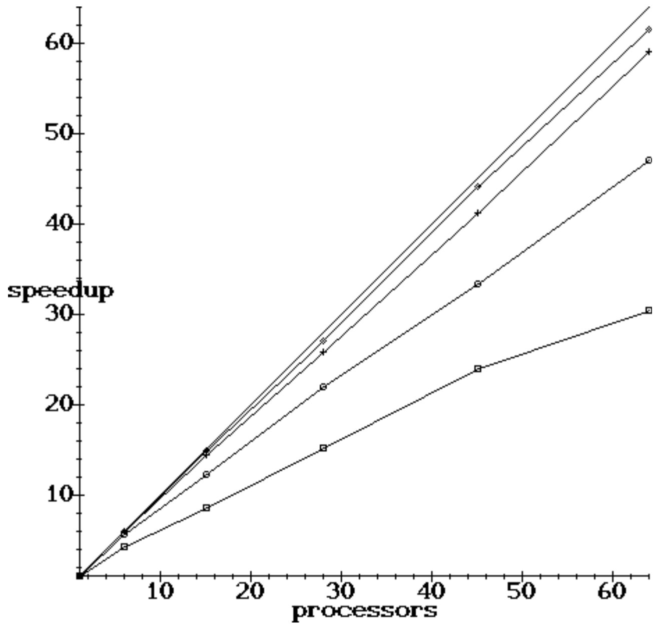


Figure 8.9: Speedup for different load patterns (bottom up: “heavy”, “moderate”, “growing”, “constant”, ideal).

		k		
		1	2	3
p	1	0.0	0.0	0.0
	6	0.75	0.18	0.11
	15	0.90	0.35	0.11
	28	0.95	0.24	0.15
	45	0.97	0.31	0.13
	64	0.98	0.26	0.13

Table 8.10: Ratio of real message traffic compared to the bound.

8.8 Impact Of Dimension On Message Traffic

There is a problem in designing an experiment that measures the efficiency decrease of the load balancer for an increasing dimension parameter k. We cannot simply compare the runtimes or speedups of applications, since we do not know how to extend a load pattern in one dimension to a ”similar” load pattern in higher dimensions. Also the amount of communication of the application itself surely varies for varying k. Hence, we do not know how to define applications in different dimensions that are equally complex to treat by a load balancer.

In fact, we already analyzed the impact of the dimension on the efficiency of the load balancer. Sect. 7.2 gives an upper bound on the amortized message traffic for a single update (Theorem 8). The bound increases linearly with increasing k. In this section we will measure the impact of k on the message traffic in a real application.

Consider the following simple application. At the beginning all processors are empty; then only the first processor performs $U = 150,000$ local insertions; after that a rebalancing happens. This load pattern can be regarded as a worst case, since the load imbalance is maximum for a given number of updates. We measured the message traffic on every processor during the rebalancing and compared it to the bound of Theorem 8. Here we restrict ourselves to the *gap*-term $traffic_{bound} := (2k+(k-1)h)S$. Let $traffic_{real}$ denote the maximum number of bytes divided by U, that have been communicated (sent or received) by any processor during the rebalancing operation. Table 8.10 shows the ratio $traffic_{real}/traffic_{bound}$ for different numbers of processors and different dimensions. We see that the bound is relatively sharp for $k = 1$. For increasing k the bound gets more and more pessimistic. Hence, in realistic applications, the efficiency of our load balancer depends much less than in a linear way on the dimension parameter.

8.9 When Is Load Balancing Reasonable?

A natural question is, when generally using a load balancer is reasonable. We don’t want that an application that actually is balanced by itself is retarded by a load balancer. We experimented with all four load patterns and compared the speedup of the application with and without load balancing. The parameter β and the task load were set as in Sect. 8.6. The object number was set as follows:

pattern	initial number	final number	updates
“constant”	20,000	20,000	24,000
“growing”	5,000	135,000	156,300
“moderate”	50,000	173,400	361,000
“heavy”	20,000	47,800	94,000

Figs. 8.11 to 8.14 show the results. For the “constant” load pattern, the speedup with and without load balancing are approximately equal. The “growing” pattern can gain a little from load balancing. On 64 processors the speedup is 51.3 without load balancing and 57.0 with load balancing. For the “moderate” pattern the situation is similar. On 64 processors the speedup gain is 45.4 compared to 41.5. The “heavy” pattern performs very bad without load balancing (speedup of 6.8 on 64 processors). With load balancing we achieve satisfying speedups (33.3 on 64 processors).

We conclude that by using our load balancer much better performance is achievable. Applications that are balanced already are not worsened seriously by the load balancer. Hence, performance does not require that the load balancer is suspended for specific applications.

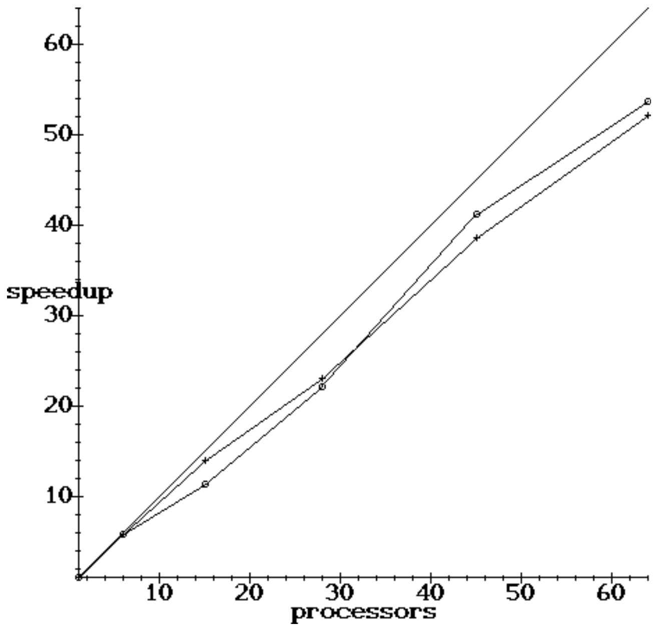


Figure 8.11: Speedup for “constant” pattern with (symbol ∘) and without (symbol +) load balancing.

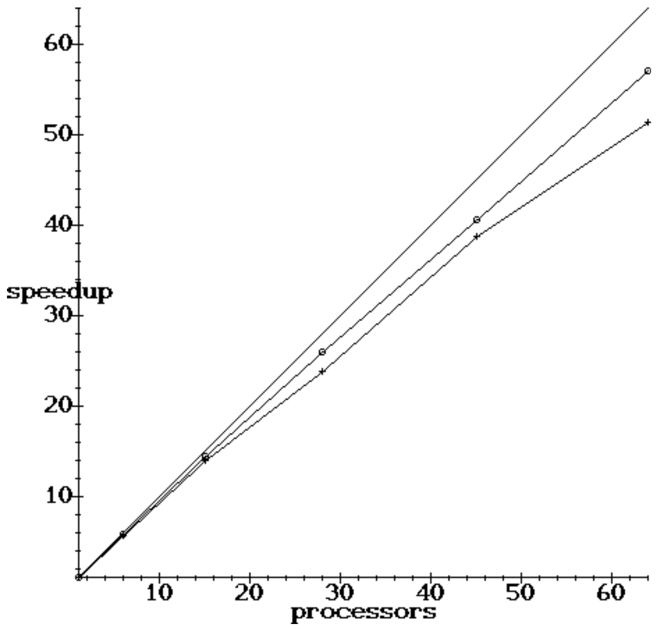


Figure 8.12: Speedup for “growing” pattern with (symbol ∘) and without (symbol +) load balancing.

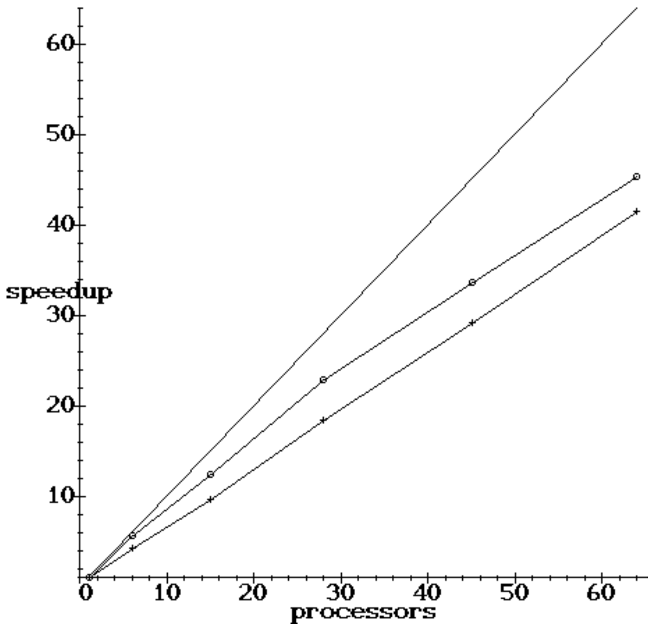


Figure 8.13: Speedup for “moderate” pattern with (symbol ∘) and without (symbol +) load balancing.

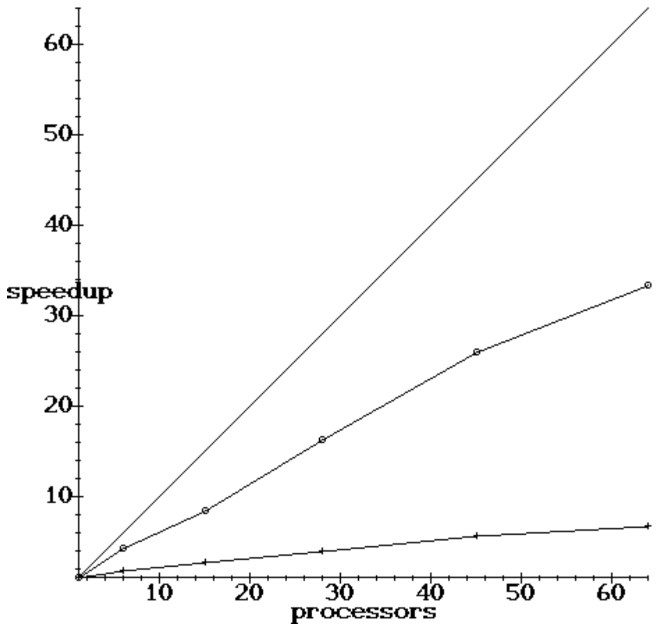


Figure 8.14: Speedup for “heavy” pattern with (symbol ∘) and without (symbol +) load balancing.

8.10 Conclusions

We have shown by experiments on a parallel computer that the spatial load balancer introduced in the previous chapter performs well and reasonable on a simple synthetic application.

Choosing a value for the balancing parameter β is not a difficult task, since a broad range of β-values leads to good results. The overhead of our load balancer compared to an ideal load balancer is only 3 to 8 percent. Speedups are good for a number of objects and a workload per task that usually appear in practical problems. The small overhead of the theoretical worst-case, which was proved in the previous chapter, has been shown here to get even smaller in practice, when the dimension is larger than 1.

Chapter 9

Reference Implementation: Hierarchical Radiosity

Contents

Let us look back a while and summarize the knowledge we have gathered so far in the previous chapters. We analyzed the Hierarchical Radiosity Algorithm using a graph partitioning technique and realized that spatial partitioning is a technique promising well scalability. We then defined a dynamic spatial partitioner and showed that it works well theoretically in the worst case and experimentally for a simple application. These results may be valuable not only for the HRA, but also for other scientific applications.

In this chapter we apply the spatial partitioner to the HRA problem. This material was published first in [43, 44]. Our basis is the mapping of elements to 3D space and links to 6D space as described in Section 5.1.3. We will see, that there are still problems to be solved, namely

Asynchronous Processing. An asynchronous formulation of the HRA is important to avoid unnecessarily expensive barrier synchronizations (Sect. 9.3.1).

Grouping. It is absolutely necessary to group elements and links to *containers* in order to reduce administration overhead (Sect. 9.3.2).

Data Reference Locality. Elements that once have been transferred to remote caches should be reused there as often and soon as possible. This can be achieved by choosing the right links for refinement (Sect. 9.3.3).

Before starting to describe these concepts in detail, we investigate the *characteristic* of the HRA much like we did for the simple application in Sect. 8.1. We classify the link and element production behaviour of the HRA by observing the minimum and maximum or the median and average load over time, respectively. The main result of Sect. 9.2 is that the element distribution is well natured, while the link distribution is severely “heavy”.

Sections 9.4 and 9.5 present programming concepts that have proved very useful during the implementation of the spatial partitioner and the asynchronous HRA. Little has been published on the implementation of HRA on DM architectures despite the fact that people regard this a difficult task (e. g. [94]). These sections contain the stuff that an interested reader will find most helpful during implementation. We think these *results* are of equal importance as the runtime measurements in Sect. 9.6. The result can be seen as a framework for dynamic spatial partitioning for scientific computing.

The performance of our program is documented in Sect. 9.6. We show plots that illustrate how the two key problems *balance* and *overhead* are defeated by our program. Speedup plots give an impression of scalability. On a Cray T3E the speedup curve is almost linear up to 64 processors. This is better than previously published attempts on massively parallel distributed memory computers. The program was also measured on a network of 8 Linux PCs with satisfying results.

9.1 Example Scenes

We made hierarchical radiosity experiments with a *hall*-scene as depicted in Fig. 9.1. The hall is illuminated by 16 pendants, 16 candles and 4 ceiling lamps. There are four complex plant geometries in the corners of the hall, which should be well clusterizable. The scene consists of 13,664 primitives (5,376 triangles, 6,744 rectangles, 856 spheres, 400 cylinders, 128 cones and 160 rings). The number of cluster elements on top of these primitives is 3,398. The total number of elements (clusters and primitives) is 17,062. During the first three iterations[1] we had about 14,000,000 link processings. The final number of leaf elements after three iterations is about 90,000. The scene model was processed using the MGF library [105].

A simpler scene consists of the same hall as above, but the pendants, the candles, the plants, and a bit detail on the tables are missing. A single ceil lamp illuminates this scene, which contains 4,873 primitives (4,053 rectangles, 296 spheres, 332 cylinders, 96 cones, 96 rings), and 1,220 clusters. After three iterations we had about 1,400,000 link processings and about 20,000 final leaf elements.

Of coarse there are many more types of scene geometries that could have been tested, e. g. an architectural scene with many large occluding walls, or an open air scene with a single dominating light source (the sun). The principle structure of the parallel algorithm of this chapter does not greatly depend on the geometry. Different geometries affect the number of links and the regions where links are dynamically generated. The effect of different numbers of links is well studied by our two test scenes (one more complex than the other). The effect of links concentrating in particular regions is covered by our large hall scene, where links aggregate greatly inside the complex plant geometries. Hence, we think of the hall scene as a difficult test case for a parallel HRA implementation. For these reasons, and for the desire to save time, we limit ourselves to the above two example scenes.

9.2 Characterizing Hierarchical Radiosity

In Sect. 8.1 the load production behaviour of a simple application was characterized by observing the minimum and maximum or the median and average load over time, respectively. We will follow this approach and characterize the HRA the same way.

We partition the 3D and 6D space into 64 equally sized fixed grid cells and observe over runtime, how many elements and links live in each separate cell. The cell sizes remain constant during the whole run, but not the contents.

[1] We limit ourselves to three iterations, since the resulting images after four, five, six iterations were visually undistinguishable from the image after three iterations. A hurried practitioner would proceed equally.

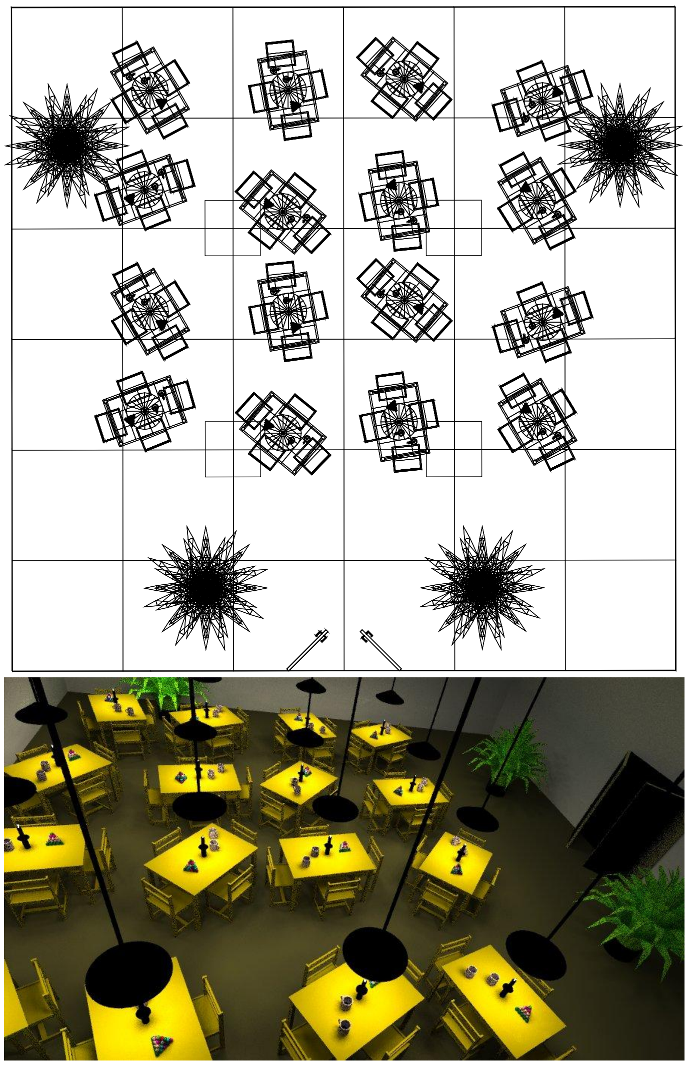


Figure 9.1: Top and bird view of *hall*-scene.

9.2.1 Element Distribution

The left part of Fig. 9.2 shows two curves that express the minimum and maximum number of elements per cell. We see that at least one of the 64 cells is empty everytime and one cell contains about 4000 elements, which is about 10 percent of all elements. In an evenly balanced situation every cell would get about 1.5 percent. The curves have been obtained by the first two iterations of an HRA run on the simple hall scene.

The right part of Fig. 9.2 shows the average and the median number of elements per cell. The median curve is much lower than the average curve. This means that at least half of the cells are relatively empty all the time. Regarding the classifications “constant”, “growing”, “moderate”, and “heavy” in Sect. 8.1 the element distribution of the HRA is “growing” during the first about 300 seconds and then something like “constant”. The initial element balance is poor. In a first phase many elements are dynamically subdivided resulting in a quickly growing element load. After that, in a second phase the number of all elements does not change dramatically. A few rebalancing operations in the second phase should suffice to reach an enduring balance. We will see that this is true in the experiments in Sect. 9.6.2.

As a general rule of thumb we think of the element distribution as *well natured*, since load balancing is only rarely necessary in practice.

9.2.2 Link Distribution

The hierarchical shooting algorithm of Sect. 2.2.6 is a very dynamic application. Links are created and removed one after another rapidly. This manifests itself in Fig. 9.3. On the left we see the minimum and maximum number of links per cell over time. At every time there is at least one empty cell. Some cells get a highly fluctuating set

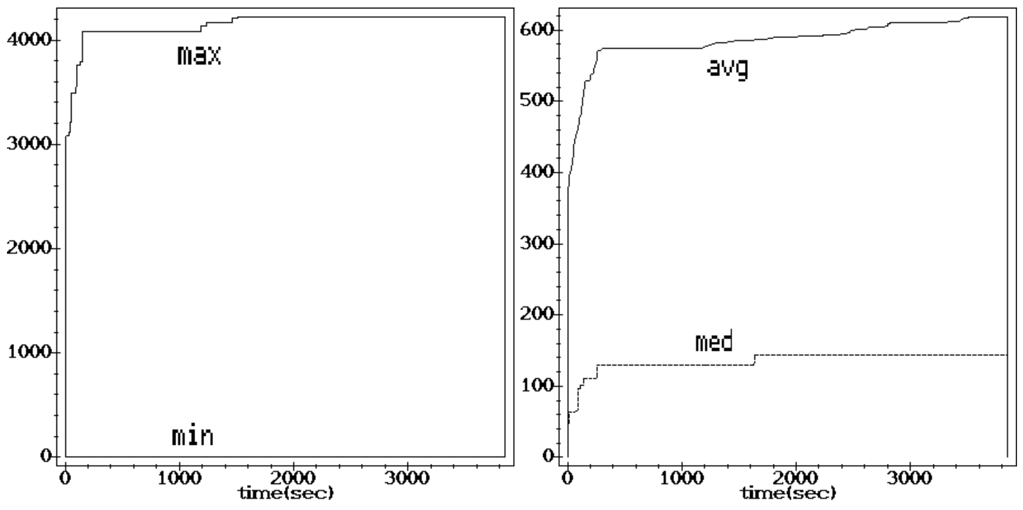


Figure 9.2: Characterizing the distribution of elements in the simple hall scene.

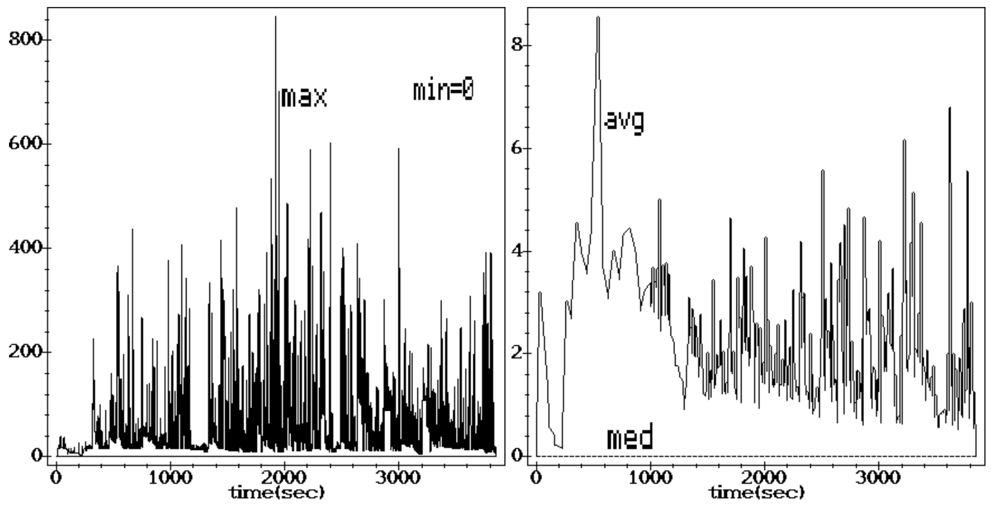


Figure 9.3: Characterizing the distribution of links in the simple hall scene.

of links. On the right the median curve is zero, which means that at least half of the cells are empty all the time. The average curve is similar in shape to the maximum curve. Regarding the classifications in Sect. 8.1 the link distribution of the HRA is absolutely "heavy".

It seems, that we have experienced two contradicting facts about the links in the hierarchical shooting algorithm. Section 6.4 states that the load continuity among the set of links is high, which suggests a classification of the distribution of links as something like "moderate". In this section we just saw that the link distribution in fact is "heavy". Actually, these findings do not contradict.

The original hierarchical radiosity algorithm [53] stores all links in memory and reuses them in later iterations. This strategy consumes lots of memory, but it leads to a smoother link distribution than experienced here — at least at first sight. At second sight we see that the highly dynamic distribution resulting from our shooting algorithm is only dynamic within each iteration, but not between iterations.

In Sect. 6.4 we studied the total load from iteration to iteration. Here we studied the load from millisecond to millisecond. Despite the highly dynamic appearance of the curves in Fig. 9.3 there exists some long term continuity. A good parallel distribution of links should exploit this continuity, simultaneously coping with the problem of highly varying load. A first step towards that goal is described in Sect. 9.3.2 below, where a good initial distribution and grouping of elements is described, that probably will reduce the need for rebalancing operations at the beginning.

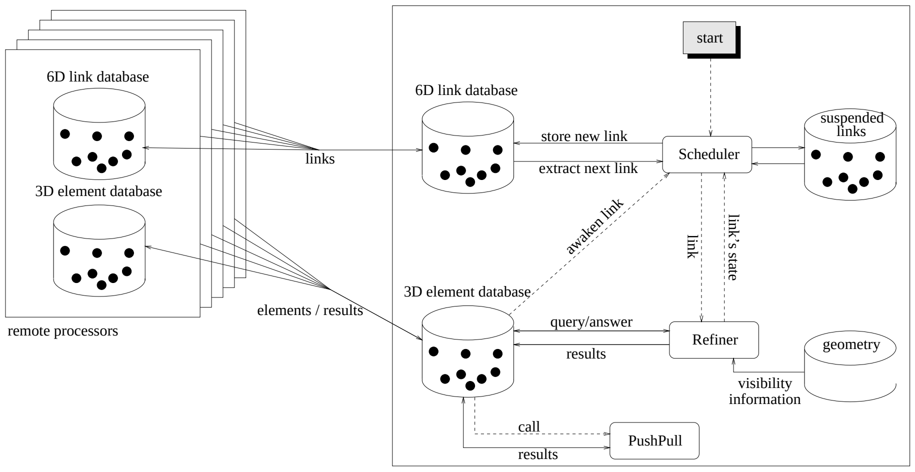


Figure 9.4: Overview of the parallel algorithm for hierarchical shooting.

9.3 Parallel Algorithm

In this section we first give an overview of the parallel hierarchical shooting algorithm. Our basis is the mapping of elements to 3D space and links to 6D space as described in Section 5.1.3. We will deepen the aspects of asynchronous processing, initial grouping, and element-reference-locality through link-ordering.

Every processor manages two databases, an element and a link database (cf. Fig. 9.4). The union of all databases comprises the whole set of links and elements that would be treated by a sequential algorithm. At the beginning there exists only a single root cluster self link. Hence, at the beginning only one processor has a non-empty local link database The union of all element databases is the set of initial elements at the beginning.

When the algorithm starts, a *Scheduler* process is called, which searches for a link in the local link database. If found one, the link is passed to a *Refiner* process. This process starts by searching the two associated elements in the element database. To be more specific, the search is performed autonomously by the database itself. If the queried elements are not stored locally, a query message is sent to another processor. Immediately after that the Refiner gets active again. If at least one element was not found locally, the current link is passed back to the Scheduler. There the link is put to a pool of suspended links.

If the Refiner was lucky and both elements were found locally, then the actual transport computation for the link is performed. For visibility calculations a separate geometry structure is used that is replicated on all processors. Afterwards the link is discarded and results for both elements (updates of radiance or new children) are passed to the element database.

When elements were not local, at some time element copies will arrive from remote processors. The copies are stored in a cache internally in the element database. The corresponding suspended link(s)[2] are awakened automatically. The results of a Refiner process, which operates on element copies, are communicated back to the element's home processor autonomously by the element database. Once there, the results are stored in the receiver element.

The Refiner may create sublinks from a given link. These are passed to the scheduler which stores them in the link database. The link database uses a directory to decide, whether the link is to be stored locally or not, and possibly sends the link to another processor.

The databases itself perform load balancing. Data items and administrative messages are exchanged automatically between the corresponding database instances. The databases get control at regular time intervals to process incoming data or rebalancing requests.

9.3.1 Asynchronous Processing

Usually, in parallel hierarchical radiosity algorithms a global barrier synchronisation happens immediately before the first root cluster self link is treated and immediately before the first push calculation at the root cluster is performed. Our asynchronous program design eliminates explicit time consuming synchronisations. The computation of an HRA iteration is allowed to be in a different state in different regions of the element hierarchy. For example some elements may have pushed downwards already, while others are involved in link refinements, yet.

[2] There may be several links waiting for the same element. The element database itself retains references to all requesters (links).

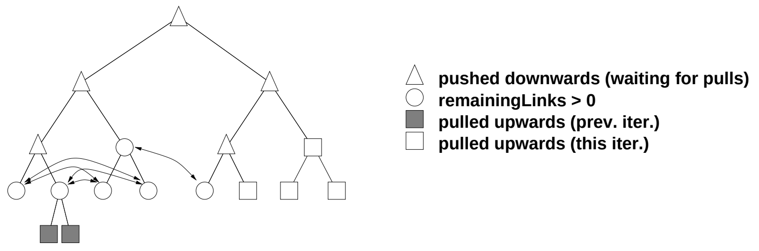

pushed downwards (waiting for pulls)
remainingLinks > 0
pulled upwards (prev. iter.)
pulled upwards (this iter.)

Figure 9.5: A situation where some element containers have pushed downwards and some have pulled upwards already.

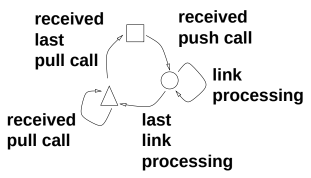


Figure 9.6: States and transitions of elements (meaning of symbols as in Fig. 9.5).

Fig. 9.5 shows a complete element hierarchy together with all links that are distributed across the local databases. We see that there are no links anymore in the upper region of the hierarchy (these have been refined and discarded), and that there are no links anymore internally in the right subtree (these have been finally established and discarded). If an element is not referenced by any link and will not be referenced by any link in the current iteration, then the element is allowed to immediately push downwards. It is not necessary to defer the push calculation until all links are done. The triangular elements in Fig. 9.5 have pushed downwards. The white boxed elements pushed downwards, too, but also pulled upwards already. An element may pull upwards, if all child elements have pulled upwards. The circular elements received a push from their parent element but are not allowed to push downwards, because they are referenced by existing links (remaining links > 0). The grey boxed elements also are not allowed to push downwards, because they did not receive a push from their parent and potentially will be referenced by links, that result from link refining. The last thing that happened with these elements is that they pulled upwards to their parent in the previous HRA iteration.

We consider things that happen to elements (cf. Fig. 9.6). At the beginning all elements are in the box-state, i. e. they did not participate in any calucation of the current HRA iteration. Each element will receive a push from its parent, which turns its state to a circle. Links may be created that refer to an element, which does not change the state of the referenced elements. Links may be processed and discarded. At some time the last[3] link that refers to a given element is discarded. Then the element pushes downwards and mutates to a triangle. Later the element will get pull data from the children. When all children have pulled upwards, then the element itself pulls upwards and changes its state to a box.

In the parallel algorithm we always want to drive an element's state as far as possible. Therefore each link processing is followed by trials to push from the two associated elements downwards. For both elements a *PushPull* process tries to push and pull through the hierarchy as long as the conditions of Fig. 9.6 are satisfied. This process is called by the database on the element's home processor. The PushPull process may propagate results in cooperation with the database automatically across processor boundaries to further descendants/ancestors.

9.3.2 The Initial Element Grouping Strategy

The initial distribution of links and elements should be such that all processors get busy as quick as possible. Once a processor has got a link to process, we should try to keep this processor busy as long as possible in order to prevent load balancing operations at the beginning, when only few links are present at all.

An ideal initial mapping would assign a small bunch of links to each processor, which results in exactly the same runtime on each processor when the links are refined recursively to the bottom. A fundamental problem is that we cannot confidently estimate the computational complexity of a given link in advance. Even less we know about the complexity of all the sublinks resulting from a given link's recursive refinement. We start with a single root cluster self link, which can be processed by a single processor only, meanwhile all other processors sitting

[3] In Fig. 9.6 "last" link processing and "last" pull call are meant as a local per element condition. When the condition gets true, of course there might exist unprocessed links / non-pulled children in other regions of the hierarchy.

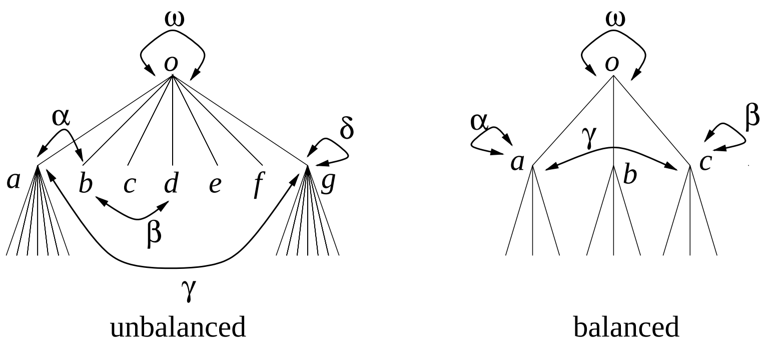


Figure 9.7: On the left probably γ is more complex than e. g. β. On the right all sublinks are assumed equally complex.

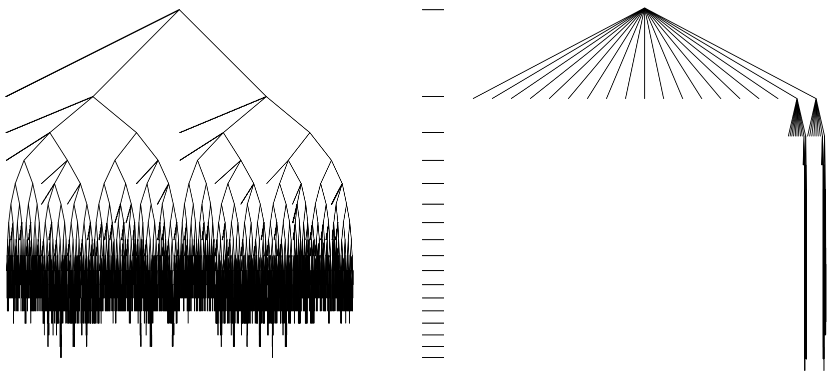


Figure 9.8: Element hierarchy of the hall scene. Both pictures depict the same tree with a different distribution of the leaves along the horizontal axis. The levels are marked in the middle.

idle. The root link must be refined into several sublinks before these can be distributed to other processors.

Fig. 9.7 shows an imaginary situation on the left, where the root cluster self link $\omega = \{o,o\}$ is refined. Some of the sublinks are shown in the figure. E. g. the sublink β runs between two elements b,d, both being leaves in the element hierarchy. Hence, the link β is likely not to be refined into sublinks, which means that the recursive computational complexity of this link is supposably small. The sublink $\gamma = \{a,g\}$ runs between inner nodes. We classify this link as a presumably more complex link, since we expect that it is refined into lots of sublinks.

On the right of Fig. 9.7 we see a balanced element hierarchy. Here we may assume that processing every link α,β,γ including all recursive refinements is equally complex. Of course, this is not true, but it is a good guess until we know the real complexity. From this we may conclude, that it would be advantageous to have a balanced element hierarchy.

Unfortunately, the initial element hierarchy may represent a totally unbalanced tree. We consider the initial element hierarchy of the hall scene, which has 17 levels. On the left of Fig. 9.8 we show a picture of the element hierarchy, where each leaf — regardless on which level the leaf is situated — has got an equal space on the horizontal axis (about 0.003 millimeter per leaf). On the right we drew the same tree, but now on each level every subtree got an equal horizontal space. We see that the root cluster element has got two clusters and seventeen surfaces as children. On the following levels the situation is qualitatively similar. The many primitives stored at high levels lead to a very unbalanced tree.

The imbalance of the element hierarchy is mainly due to the fact that in our implementation cluster siblings are disallowed to overlap each other. If we allowed overlaps, we could put each of the seventeen surfaces at the second level into one of the two clusters at this level, leading to a node degree of two at all inner nodes of the hierarchy.

Our strategy now is as follows. We group nodes of the element hierarchy into super nodes, called *element containers*. The same way that elements form a tree, also the element containers are organized in a tree. The container tree will be better balanced than the original element tree. Also using element containers instead of single elements, the administration overhead for remote element accesses will be reduced.

Fig. 9.9 shows an element hierarchy with clusters and surfaces as nodes. The nodes are grouped into element containers, shown as shaded regions. Every container has its own *miniroot*. For example the miniroot of container F is the white cluster in Fig. 9.9. The resulting hierarchy of containers is shown in Fig. 9.10.

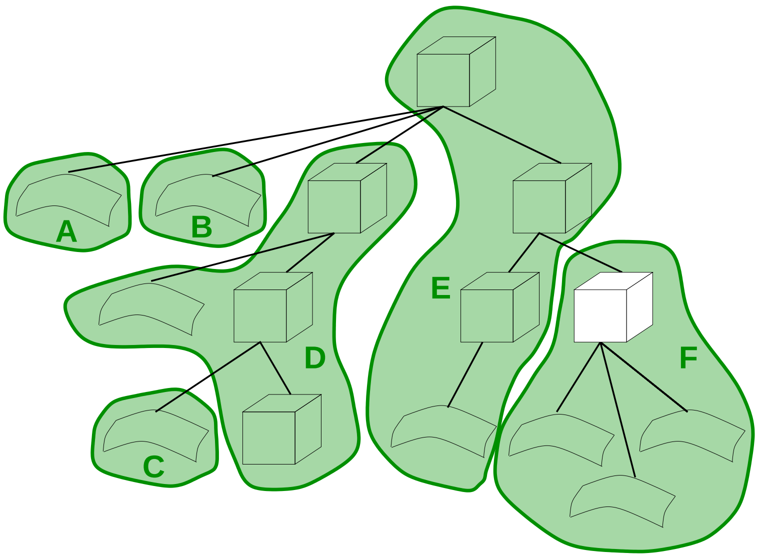


Figure 9.9: Grouping nodes of the element hierarchy.

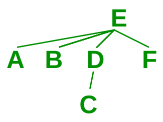


Figure 9.10: A tree of element containers.

In order to automatically group n elements into containers, we pursue the following simple strategy. We start with n element containers, each container containing a single element. Iteratively we select a container leaf and merge it to its parent container. This reduces the number of containers by one. We terminate when the number of containers is as small as desired. At the beginning we preferably select leaf containers with a single element in it. Later we select leaf containers which contain the fewest elements. This results in a tree of containers, where the degree of inner nodes is reduced, since primitives on high levels are merged to their parent. As a second effect the number of elements per container tends to be larger at lower levels of the container tree. The upper level containers contain only few elements. This feature is important for a quick distribution of link tasks at the beginning of an iteration, as is described in more detail in Sect. 9.3.3.

As an example the *hall*-scene has been processed into 346 element containers. This is an average of 46.315 initial elements per container. The above simple strategy formed the container tree shown in Fig. 9.11. This tree has 11 levels and was drawn the same way as the tree on the right of Fig. 9.8. It is much better balanced than the original element hierarchy.

The number of initial elements per container ranges from 1 to 140. Leaves of the container tree contained between 71 and 140 elements. Hence, the leaves tend to be fatter than the inner nodes.

We can also see the balance of the container tree by looking at the number of containers per level, which is shown in Fig. 9.12. In the level-range $[0,7]$ the curve looks exactly like an exponential function (2^{level}). Below a certain level the curve falls off.

The number of initial elements that are contained in containers on a given level of the container tree is shown in Fig. 9.13. We show this plot since it documents, that elements are rare on high levels, which leads to link tasks of low complexity on higher levels.

9.3.3 The Processing Of Links

We define a non-interruptable *link task* as to compute all links between the elements contained in two element containers. The larger the number of elements in the two containers, the higher the complexity of this task.

Fig. 9.14 shows two element containers A and B. Children of elements of A or B that are not contained in A or B are shown in grey. A first link (link no. 1) running between the two miniroots of A and B represents a single link task T. The link is processed and found to need refinement. Four sublinks are created (no. 2, . . . , 5). One of them (no. 4) runs between container B and another container, that is a child container of container A. Hence, link no. 4 is not part of this task but instead comprises a new task. All the other links (no. 2, 3, 5) are part of T. Link 2 and

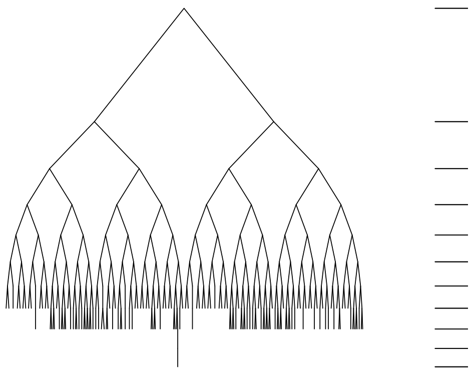


Figure 9.11: The element container tree for the hall scene.

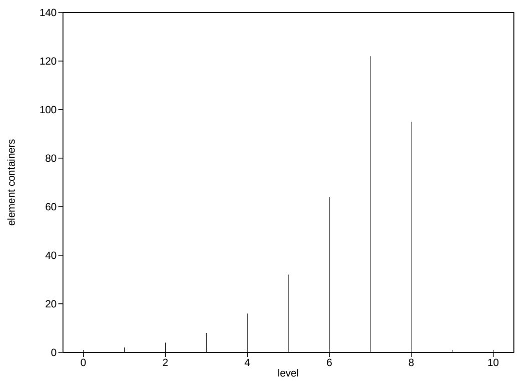


Figure 9.12: Containers per container tree level.

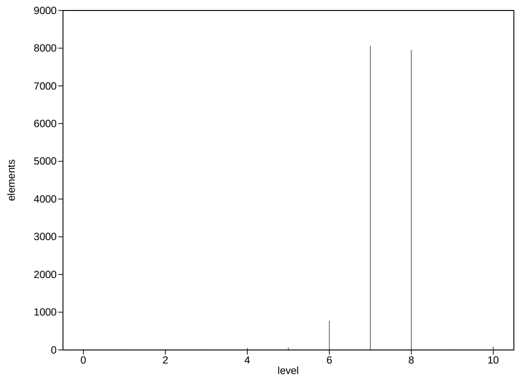


Figure 9.13: Initial elements per container tree level.

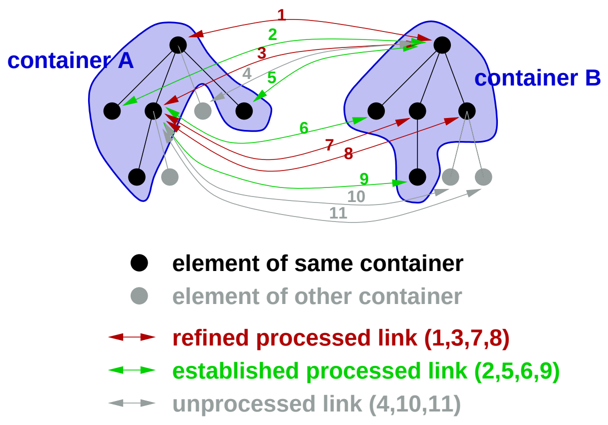

container A
container B
1
2
3
4
5
6
7
8
9
10
11
● element of same container
● element of other container
refined processed link (1,3,7,8)
established processed link (2,5,6,9)
unprocessed link (4,10,11)

Figure 9.14: Processed links of a link task.

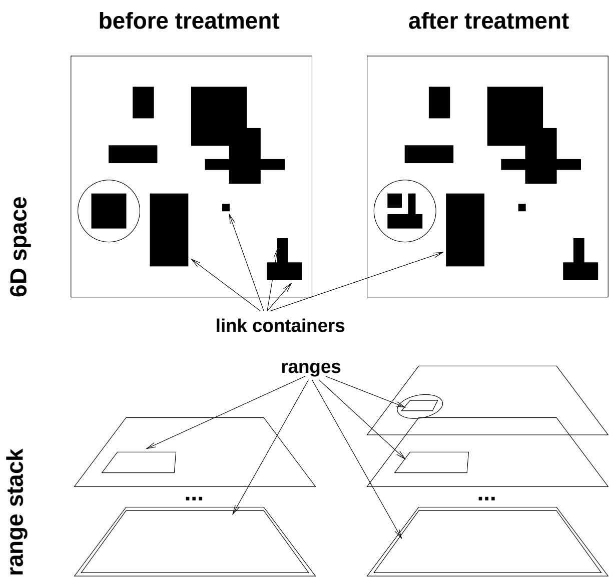


Figure 9.15: The encircled link container on the left has been treated and created some new link containers (encircled on the right). After that the range of the left link container is pushed on top of the range stack.

5 are found to be allowed to interact (established), link 3 is refined. All sublinks of no. 3 (no. 6, 7, 8) fall into T's responsibility. 6 is established, 7 and 8 are refined again. At the end of task T we have processed all links except 4, 10, 11. The latter each define a new link task.

Thanks to the above initial element grouping method, we are facing a more or less balanced element container tree that is sparsely settled at the upper levels. Since the upper level containers are relatively light, the first link-computation tasks have a low computational complexity. As mentioned at the beginning of Sect. 9.3.2 this is desirable in order to distribute work at the beginning of the algorithm as quick as possible.

We use the term *link container* for the link task just described. A link container contains references to the two miniroots of the element containers. The name link *container* suggests that several individual links are processed when a single link container is treated.

In a parallel program the processing of a link task T involves requests for the two associated element containers. Copies of the element containers are stored on arrival in a local cache. It is important for a good cache coherence that the sub tasks that result during the processing of T (i. e. link tasks 4, 10, 11 in the above example) are processed soon after the processing of T finished.

In an asynchronous parallel program it is not possible to use the standard program stack mechanisms to ensure that e. g. task 4 is processed soon after T. This is because task 4 may involve remote requests that prevent an immediate execution. Therefore we provide our own self made *6D range stack* (cf. Fig. 9.15). The stack contains 6D ranges. When a link container is treated, its 6D range is pushed on top of the stack. When searching for a next link container to execute, we first look for link containers whose range is contained in the range that lies on the top of the stack. If not found, we pop the uppermost range from the stack and recurse.

Another important effect of using the range stack is a saving of memory. The total number of link containers to be stored at one time is much smaller than when we simply took a random link container for the next execution.

9.4 General Implementation Concepts

In this section we describe the basic implementation concepts that are more or less independent of the hierarchical radiosity application. Later in Sect. 9.5 we discuss concepts specific to the HRA.

Since our implementation comprises more than 200 classes and about 100K code lines, we focus here on concepts, not on individual classes. In order to explain the basic principles, at certain passages for clarity reasons we will "lie" a bit, since an authentic presentation of our code would be too confusing.

As an example for such a "lie" consider a collection of instances of class `B`, which is stored *k*-d-tree-like inside a class `C`. We denote this simply as "`C` aggregates multiple instances of `B`". Classes that are needed only for the *k*-d-tree are omitted from the diagrams.

The diagrams are in UML (unified modelling language) notation [15]. Fig. 9.16 shows an overview of all classes discussed in this and the following section.

9.4.1 Identifiers

In a distributed memory environment it is essential not to refer to data items by pointers, since pointers are valid only inside a single local memory module. We use system wide unique identifiers (class `ID`, see Fig. 9.17). An identifier is represented by an integer (attribute `ID::id`). The constructor `ID::ID` creates a new identifier based on the last identifier, which is stored in the static attribute `ID::lastid`. In order to get system-wide unique identifiers, a processor with rank r creates only identifiers that suffice $r = \mathtt{id}$ modulo p.

Every object that needs to be identified across processor boundaries is specialized from the class `Identi-fiable`. The corresponding identifier is contained in this class and can be queried using `Identifiable:-:getID`. On construction of a new `Identifiable`-instance a new unique `ID` is constructed. A global hash map `IDMap`, which maps identifiers to objects, is maintained during `Identifiable`-constructors and -destructor. Every new object is inserted into the hash map. Every deletion of an object involves removing the object from the hash map. From the outside, a simple query (`IDMap::operator[]`) retrieves the corresponding object for a given identifier. Even simpler, this query can be delegated to `ID::getObject`, hence the user code for retrieving an object from a given identifier is

```
ID i= 0815;
MyClass* obj= dynamic_cast<MyClass*>(i.getObject());
```

where the calling code should know, of which type the referenced object is. In this example, `MyClass` is assumed to be a subclass of `Identifiable`.

9.4.2 Points And Ranges

As probably every object-oriented program that treats geometric data structures, also our program has a class `PointD<dim>`, which represents points and vectors, and a class `RangeD<dim>`, which represents an axis-aligned box (see Fig. 9.18). A template parameter expresses the dimension. All classes in our program that are dimension dependent are template classes. `Axis` is a type definition for a small integer type.

9.4.3 Addressing

In a distributed environment it is important to address remote objects efficiently such that they can be found quickly on demand. It would be waste of time if we would have to scan all processors when searching for a specific object with a given identifier.

In Chapter 7 a spatial mapping was described, which can act as a directory of objects. This directory can be used to efficiently search for remote objects. For this we need to address a remote object not only by its identifier but also by its location in space. In our program we use `RangeD<dim>` to describe a set of locations which contains the desired object's location. The class `ReferenceD<dim>` is used to bundle the identifier and the range of an object (cf. Fig. 9.25).

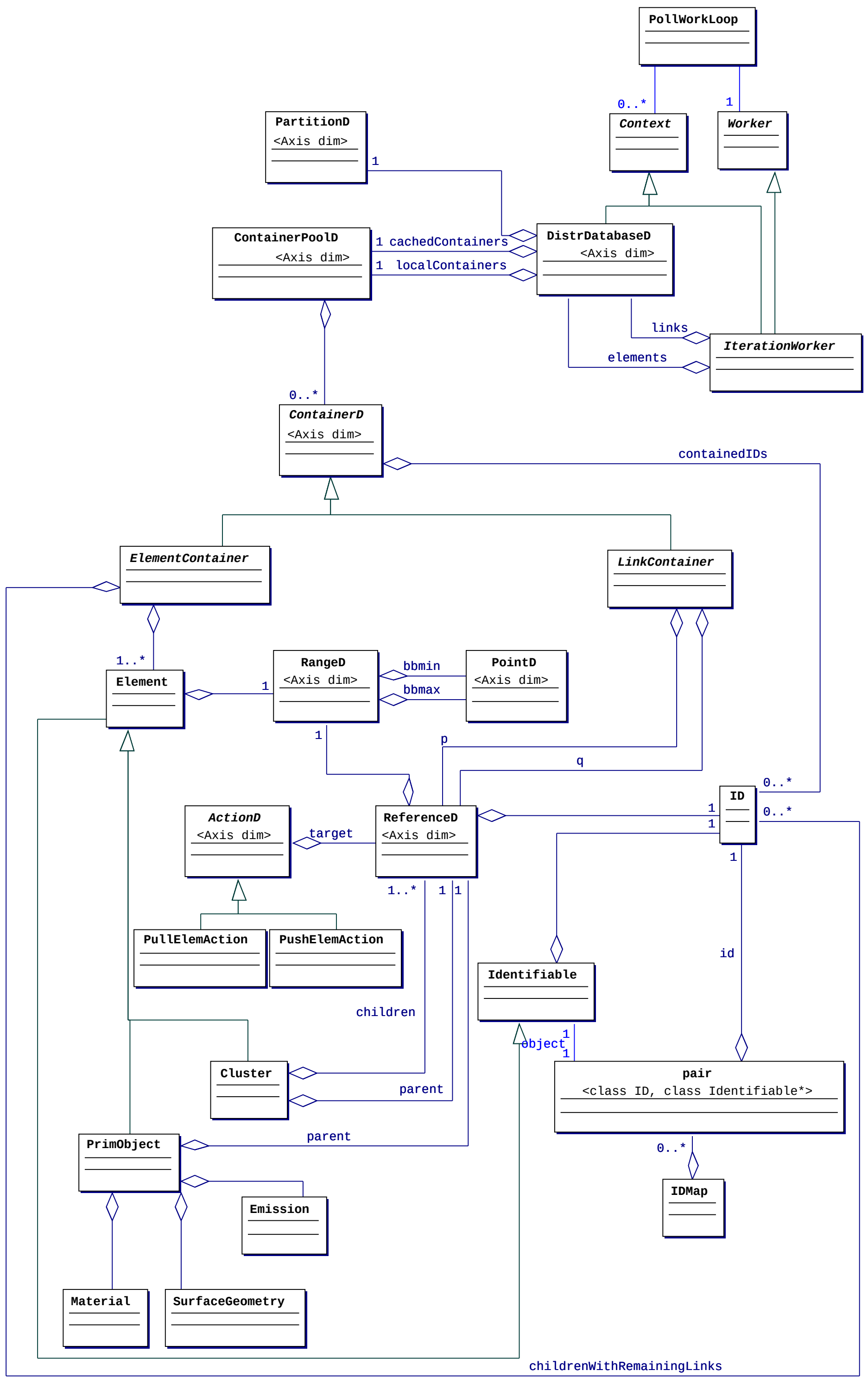


Figure 9.16: Some of the basic implementation concepts.

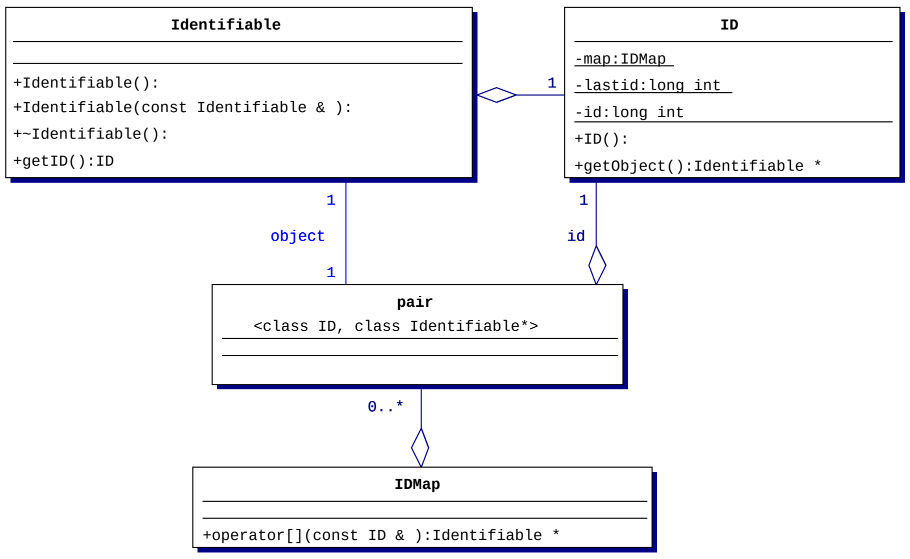

Identifiable
+Identifiable():
+Identifiable(const Identifiable &):
+~Identifiable():
+getID():ID
ID
-map:IDMap
-lastid:long int
-id:long int
+ID():
+getObject():Identifiable *
1
1
object
1
1
id
pair
<class ID, class Identifiable*>
0..*
IDMap
+operator[](const ID &):Identifiable *

Figure 9.17: Identifiers and global hash map.

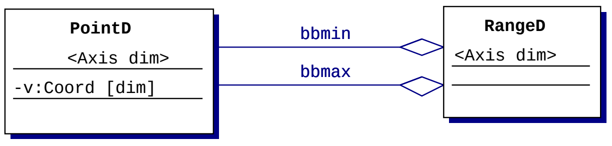


Figure 9.18: Point and Range classes.

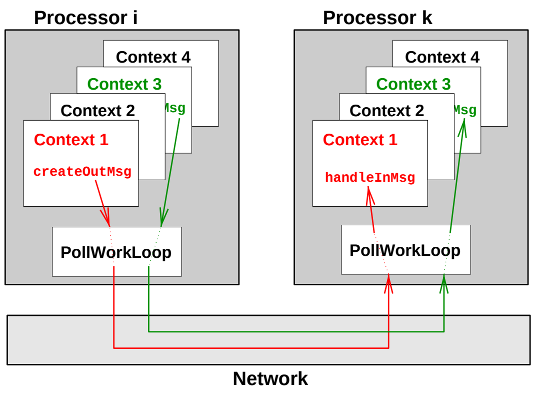


Figure 9.19: Scheduling of messages to contexts.

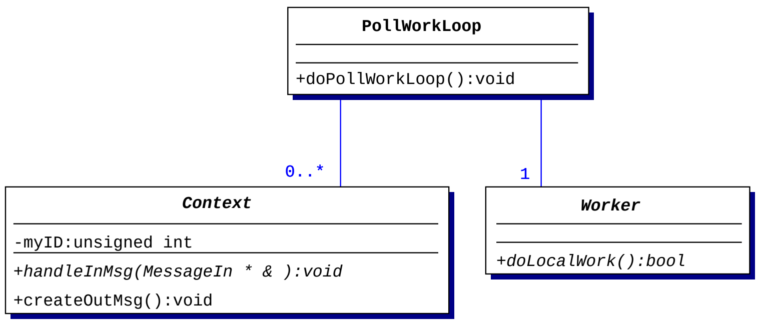


Figure 9.20: Messaging and scheduling.

9.4.4 Messaging

On every processor there are different messaging contexts (see Figs. 9.19, 9.20). Each context is identified by a unique identifier. A message sent from processor *i* to processor *k* is handled at the destination inside the same context in that it was sent at the source. In order to achieve this, the method `createOutMsg` simply writes the context identifier into the message. At the receiving processor a scheduler (class `PollWorkLoop`) reads the identifier and passes the message to the corresponding context. This concept makes message transfers very comfortable.

The class `Context` provides a pure virtual method `handleInMsg`, which must be implemented by user contexts. The scheduler calls this method, when it receives a messages. Outgoing messages are bundled to larger bunches by `PollWorkLoop` in order to save startup time-overhead for small messages.

Fig. 9.20 shows another `Worker` class. An user application derives a class from this class and implements the method `doLocalWork`. Inside this method the user application does a small portion of local work and perhaps sends a few messages. It usually returns before all work has been done in order to give the scheduler the chance to handle incoming messages. After that `doLocalWork` is called again.

In our HRA implementation for instance there exists a class `IterationWorker` (see Fig. 9.16), which is derived from both `Context` and `Worker`. The method `IterationWorker::doLocalWork` defines a portion of local work as the execution of one link container (cf. Sect. 9.3.3).

Fig. 9.21 shows, how the line of control switches from `PollWorkLoop` to `Worker::doLocalWork`, when local work is performed, to `Context::createOutMsg`, when a message is sent, and to `Context:-:handleInMsg`, when a message is received. When `doLocalWork` returns `false`, this signals the scheduler to terminate the computation. For distributed termination detection we use a ring-like token send technique as described in [50].

9.4.5 Containers

A `ContainerD<dim>` represents a collection of objects (see Fig. 9.22). We use a container to coarsen the granularity of remote data accesses. E. g. when we need to access a remote element during the refinement of a link in the HRA, then we request not only a single element but a container of elements (an `ElementContainer`, see Fig. 9.26) and then try to use all contained elements on the requesting processor before we report the results back to the source.

A `ContainerD<dim>` has got a range in space, which can be queried using the method `getRange`, and a weight, which is available by `getWeight` and which represents something like the number or the size of objects contained in the container. The identifiers of objects contained in the container is available by `getContained-IDs`.

A container is either an *original* or a *copy*. Copies may be created from an original using the method `cre-ateCopy`. A copy is a container that has been requested from remote (e. g. the `ElementContainer` in the above example). After a copy has been used on a remote processor, changes inside the copy are reported back to its home processor, where the original container lives. For this purpose we have provided the methods `backReport`, which is called on the copy and writes data to a message buffer, and `receiveBackReport`, which is called on the original when the message is received.

A Container keeps in mind, how many copies are whirring around (attribute `count`, see Fig. 9.23). This is important, since an algorithm's progress may depend on the fact that all possibly existing copies have reported back to their original.

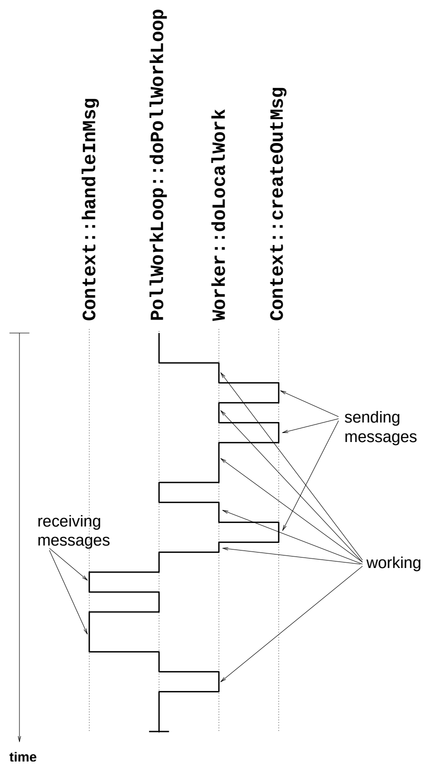


Figure 9.21: Illustration of the line of control.

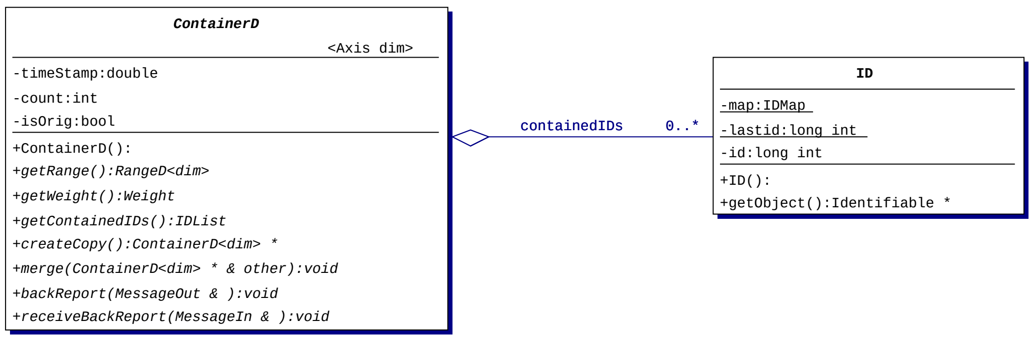

ContainerD
<Axis dim>
-timeStamp:double
-count:int
-isOrig:bool
+ContainerD():
+getRange():RangeD<dim>
+getWeight():Weight
+getContainedIDs():IDList
+createCopy():ContainerD<dim> *
+merge(ContainerD<dim> * & other):void
+backReport(MessageOut &):void
+receiveBackReport(MessageIn &):void
containedIDs
0..*
ID
-map:IDMap
-lastid:long int
-id:long int
+ID():
+getObject():Identifiable *

Figure 9.22: The container class.

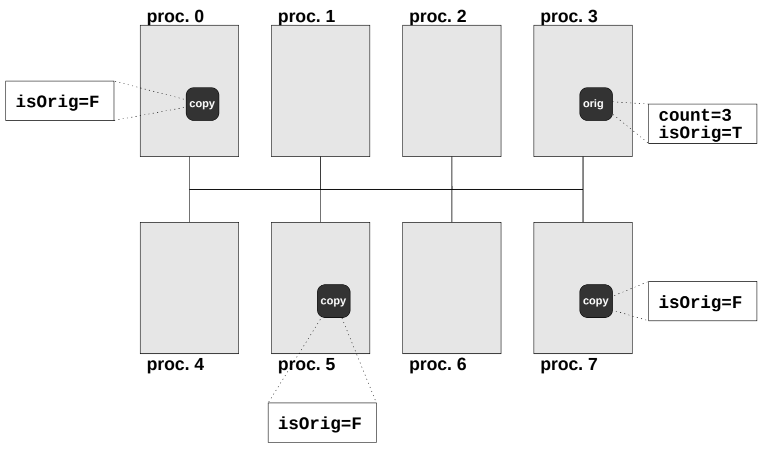


Figure 9.23: An original container with three copies.

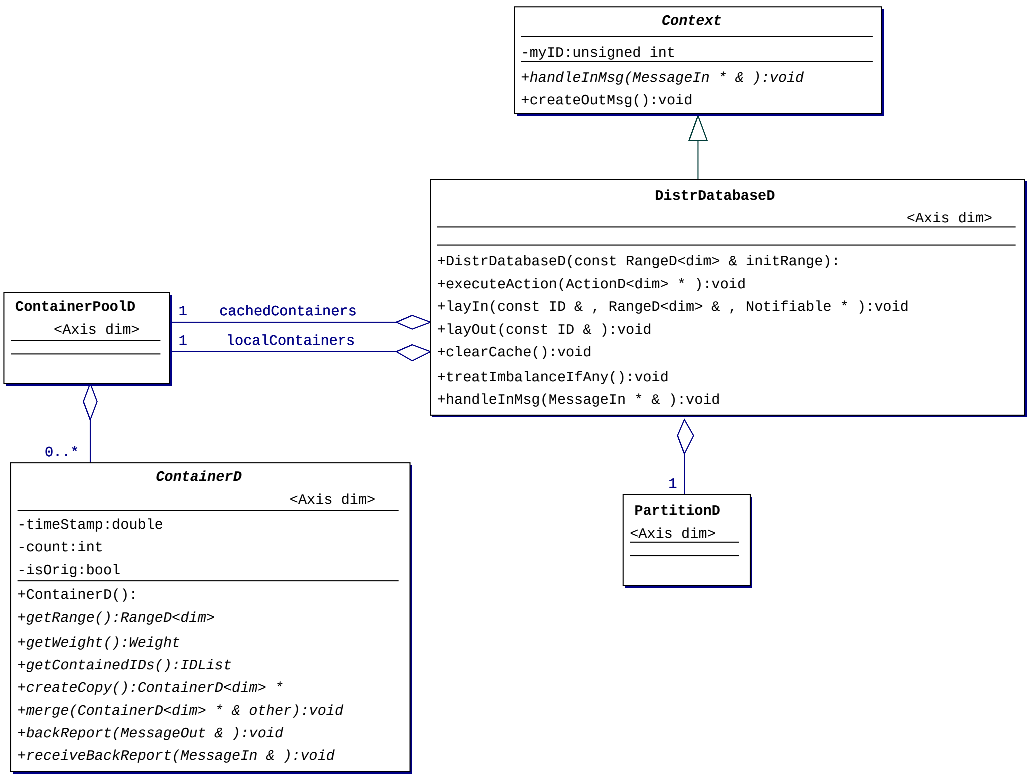

Context
-myID:unsigned int
+handleInMsg(MessageIn * &):void
+createOutMsg():void
DistrDatabaseD
<Axis dim>
+DistrDatabaseD(const RangeD<dim> & initRange):
+executeAction(ActionD<dim> *):void
+layIn(const ID & , RangeD<dim> & , Notifiable *):void
+layOut(const ID &):void
+clearCache():void
+treatImbalanceIfAny():void
+handleInMsg(MessageIn * &):void
ContainerPoolD
<Axis dim>
1 cachedContainers
1 localContainers
0..*
ContainerD
<Axis dim>
-timeStamp:double
-count:int
-isOrig:bool
+ContainerD():
+getRange():RangeD<dim>
+getWeight():Weight
+getContainedIDs():IDList
+createCopy():ContainerD<dim> *
+merge(ContainerD<dim> * & other):void
+backReport(MessageOut &):void
+receiveBackReport(MessageIn &):void
1
PartitionD
<Axis dim>

Figure 9.24: The classes of the distributed database of containers.

9.4.6 Database Of Containers

Containers are distributed across processors, and copies of containers may be created and propagated on demand. The class `DistrDatabaseD<dim>` (see Fig. 9.24) comprises all intelligence that is needed to find remote containers and to rebalance the containers dynamically according to the rebalancer of Chapter 7. `PartitionD<dim>` contains the current assignment of regions to processors and acts as a directory, which tells the processor index for a given location in space.

The containers are stored in a *k*-d-tree in class `ContainerPoolD<dim>`. We have two instances of this class, one containing the original containers and one containing the copies, which are also labeled *cached* containers.

A user application requests remote data by calling the method `DistrDatabaseD<dim>::layIn`. The arguments of this method specify the object's identifier and the range, where the object is located (cf. Sect. 9.4.3). The range may be any range including the objects extent. Supplying a range as small as possible leads to the desirable effect that less processors are involved in the request. A call to `DistrDatabaseD<dim>::layIn` returns immediately after sending a message to a remote processor. Later, when an answer message is processed by `DistrDatabaseD<dim>::handleInMsg`, then the requesting part of the user program needs to be notified, that the object is available. For this purpose, now a user-supplied `Notifiable` is informed.

A cached container can be triggered by a user program to immediately report back to its original counterpart by calling `DistrDatabaseD<dim>::layOut`. The method `DistrDatabaseD<dim>::clearCache` results in an explicit back-report of all cached containers. A third way of triggering back-reports is when the total weight of all cached containers exceeds a predefined limit. Then some cached containers are layed out automatically without user-intervention. We employ a *least recently used (LRU)* strategy, which preferably keeps those containers in the cache, which have recently been used. The `ContainerD<dim>` class's attribute `timeStamp` (see Fig. 9.22) keeps the last date, when a container was used.

We already mentioned that all dynamic rebalancing is performed inside the class `DistrDatabaseD<dim>`. A rebalancing operation can be initiated from the user program by calling `treatImbalanceIfAny`. This method detects a potential imbalance like in Sect. 7.1.4, sends out messages, if necessary, and returns immediately. From now on the rebalancing progresses without user intervention. All actions are handled automatically inside `handleInMsg`, until the rebalancing is complete.

Interesting to note is the fact, that during rebalancing two "equal" containers could meet on the same processor (see also [26]). For instance consider a situation where an original has lived before on processor *i* and is now moved

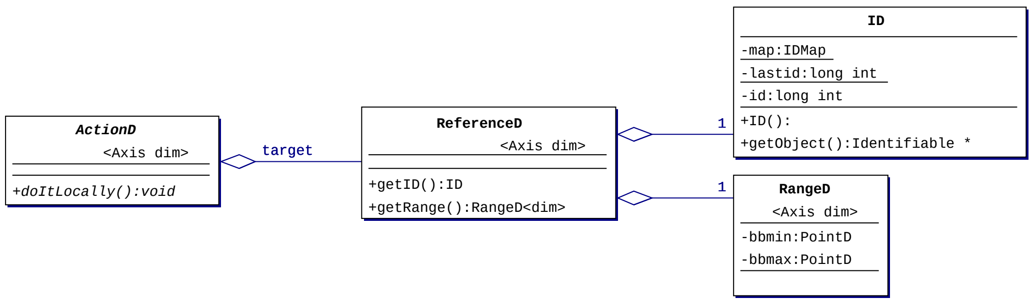


Figure 9.25: The Action class.

during rebalancing to processor *k*. On processor *k* we might already have a copy of that container. Now at *k* we need to merge the copy and the original and store the resulting original. Let's consider another example. Processor *i* queries processor *k* for a copy of an original container living on processor *k*. Processor *k* packs an answer message to *i* containing the queried data. After that let us assume that (before the answer arrives) a rebalancing operation moves exactly the requested container original from *k* to *i*. A few milliseconds later the requested copy arrives, which now has to be merged to the original.

One more problem arises when copies meet copies, which may happen when a user application queries the same container twice. The two copies are merged, but the original cannot recognize this merger in its attribute `ContainerD<dim>::count`. Hence, for copies we use the `count` attribute to count the number of mergers. When the copy finally reports back to the original, this number is evaluated in order to filter out the real number of remaining copies.

Of course there exist methods in `DistrDatabaseD<dim>` to insert and remove containers (not shown in Fig. 9.24). Insertions and removals are performed using the *Action*-concept, which is described in the following section.

9.4.7 Actions

An *action* is a small package of work that is executed on a *target*. The target's address (cf. Section 9.4.3) is given as the target object's identifier and it's range (class `ReferenceD<dim>`, Fig. 9.25). The package of work is given in the method `ActionD<dim>::doItLocally`, which must be implemented by subclasses.

Actions are designed to be executed only on originals, not on copies. An action is issued on any processor by calling the method `DistrDatabaseD<dim>::executeAction`. Then the action hops from processor to processor until the original was found. This hopping of course is efficiently guided by the information of the target's range. In a static environment usually a single hop should suffice. In a dynamic setting, the action may have to run after objects, which are moved by an ongoing rebalancing operation. Action-hopping is performed inside `DistrDatabaseD<dim>` automatically. When the target is reached, then the action's `doItLocally`-method is called. We can think of `ActionD<dim>` together with `PartitionD<dim>` as implementing a redundant nameserver as it was described in [106].

The action concept is a very powerful and useful concept in asynchronously communicating SPMD programs. We have found during implementation that this general principle can help avoiding serious errors, which show up non-deterministically and are hard to debug. It would have been a great relief in our previous projects [36, 26], if we had invented this concept before.

9.5 Specific Concepts For Asynchronous Parallel Hierarchical Shooting

In this section we discuss concepts of our implementation that have shown to be useful in an asynchronous SPMD program for hierarchical radiosity.

9.5.1 Elements

Fig. 9.16 shows on the left the class `Element` and its two descendants `Cluster` and `PrimObject`. These classes contain the coefficients needed to represent a part of the radiance and importance functions on a limited support. A `Cluster` additionally contains information about the cluster's extent, the visibility of the cluster with respect to its siblings, and the addresses (class `ReferenceD<dim>`) of the parent cluster and of the children. A

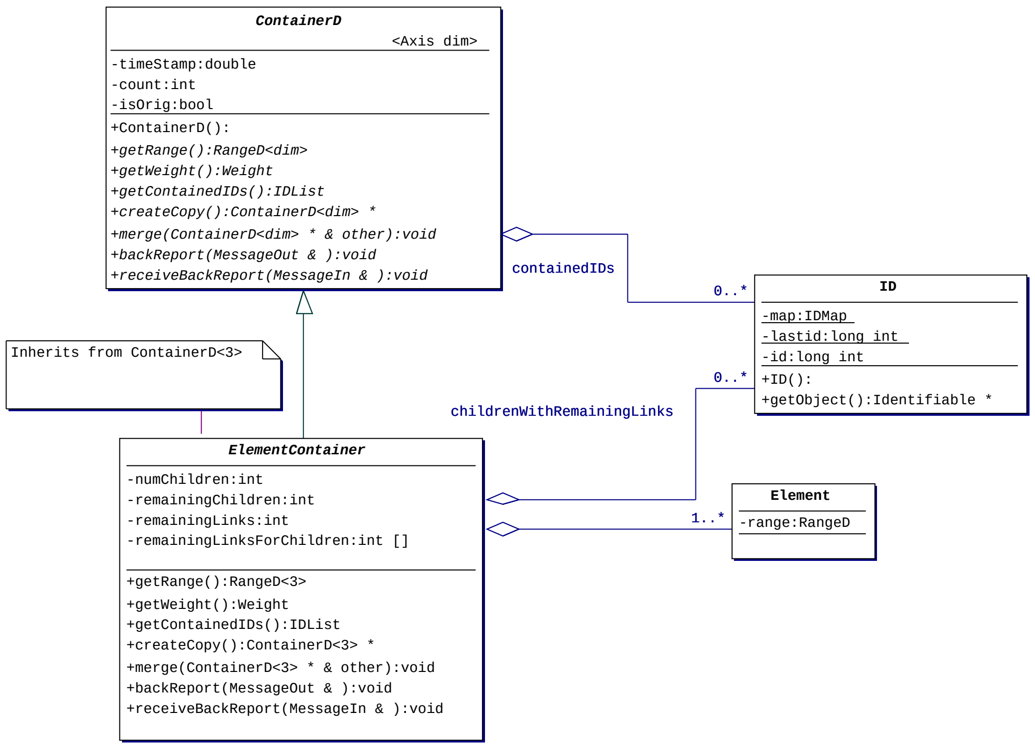

ContainerD
<Axis dim>
-timeStamp:double
-count:int
-isOrig:bool
+ContainerD():
+getRange():RangeD<dim>
+getWeight():Weight
+getContainedIDs():IDList
+createCopy():ContainerD<dim> *
+merge(ContainerD<dim> * & other):void
+backReport(MessageOut &):void
+receiveBackReport(MessageIn &):void
containedIDs
0..*
ID
-map:IDMap
-lastid:long int
-id:long int
+ID():
+getObject():Identifiable *
Inherits from ContainerD<3>
0..*
childrenWithRemainingLinks
ElementContainer
-numChildren:int
-remainingChildren:int
-remainingLinks:int
-remainingLinksForChildren:int []
+getRange():RangeD<3>
+getWeight():Weight
+getContainedIDs():IDList
+createCopy():ContainerD<3> *
+merge(ContainerD<3> * & other):void
+backReport(MessageOut &):void
+receiveBackReport(MessageIn &):void
Element
-range:RangeD
1..*

Figure 9.26: The element container class.

PrimObject additionally contains the address of the parent cluster and the boundary data, i. e. the geometry in SurfaceGeometry, the reflection properties in Material and the emissive behaviour in Emission.

Element is derived from Identifiable, thus it knows about its identifier and its range. The particular element-ranges are used to construct the whole range of a container of elements, an ElementContainer.

9.5.2 ElementContainer

An ElementContainer is a collection of instances of clusters or primitives, that are aggregated as Elements (see Fig. 9.26). As a subclass of ContainerD<3> an element container implements the abstract methods of ContainerD<3>. We store all ElementContainer-instances in an instance of class DistrDatabaseD<3>. There may be local and cached element containers, as described elsewhere (Sect. 9.4.6).

During the pull operation (see Sect. 2.2.6), we need to note in an element container, how many children have pulled upwards to a container. If the attribute remainingChildren gets zero, then the container received all pulls and now itself may pull to the parent container. remainingChildren is initialized during the push operation to numChildren, which stores the number of child containers below an element container.

The attribute remainingLinks is used to count the number of unfinished existing link containers. As long as this number is greater than zero, no push operation downwards to the children is allowed (cf. Sect. 9.3.1). We need to store this number in an element container, because the existing link containers may be unknown locally, since they are distributed among remote processors.

Consider a link container L between two element containers A and B (Fig. 9.27). After processing of L the attribute remainingLinks is decreased by 1 in both A and B. For every new link container (here M), that resulted from the processing of L, the remainingLinks-attribute of the corresponding element containers (here B and C) should be increased by 1. The problem in a distributed environment is, that the processor that processes L has access only to L, and to (maybe cached versions of) A and B, but not to container C. We work around that problem by storing a proxy value in container A, that will be propagated to C "as soon as necessary". We need a proxy for each child container (one for D and one for C). The attribute remainingLinksForChildren contains these proxies, while the aggregation childrenWithRemainingLinks contains the identifiers of those children with a non-zero proxy-value. The phrase "as soon as necessary" can be resolved as "when pushing from A to C". During the push operation the parent's proxy value is added to the child's remainingLinks-attribute. After all let us assume that A is cached locally. We have to induce a back-report of A to its home processor, where the modified remainingLinks-attribute is merged to the original element container. The original of A is not al-

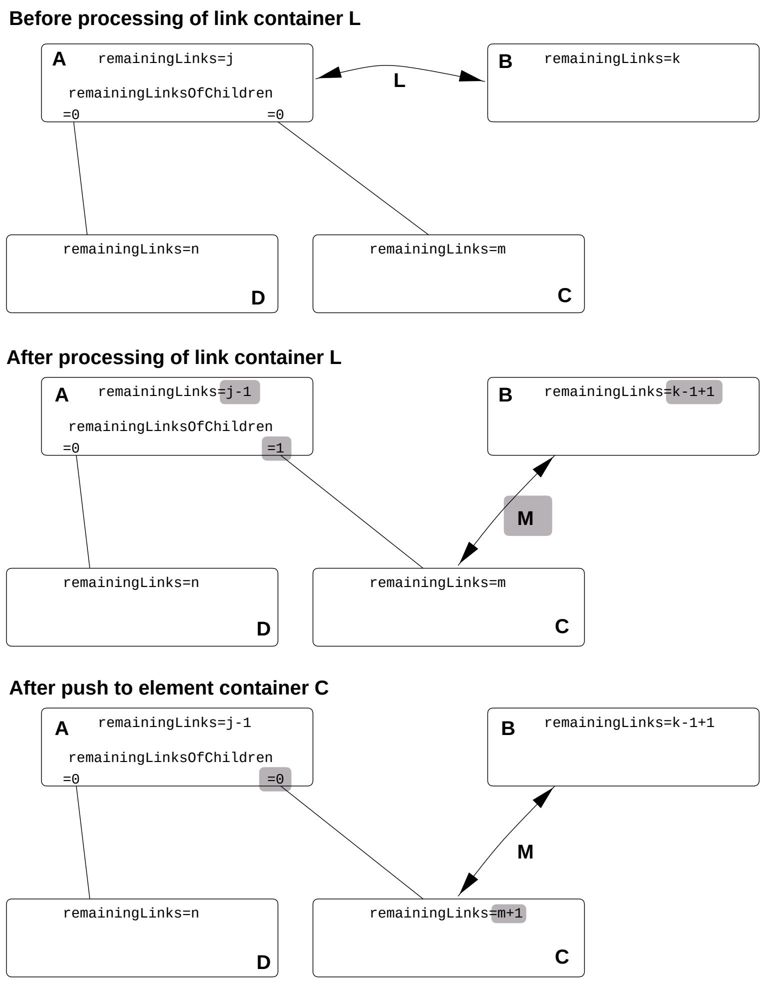


Figure 9.27: Explanation of the attributes `ElementContainer::remainingLinks` and `ElementContainer::remainingLinksForChildren`.

lowed to push downwards, until all existing copies of A (counted in `ContainerD<3>::count`, see Sect. 9.4.5) have reported back. So, when an `ElementContainer` receives a back-report, then it always tries to issue a push operation on itself, since this back-report could have been the last missing back-report to this container.

9.5.3 LinkContainer

The class `LinkContainer` (Fig. 9.28) is a very simple class, which is derived from `ContainerD<6>`. As a subclass a link container implements the abstract methods of `ContainerD<6>`. We store all `LinkContainer`-instances in an instance of class `DistrDatabaseD<6>`. There may be only local, but no cached link containers.

A link container aggregates two `ReferenceD<3>` instances as “pointers” to the two miniroots of the two associated element containers. Processing a link container involves the refinement of several links between two element containers as described above in Sect. 9.3.3. The decision, which link container is processed next is aided by a 6D range stack as explained in Sect. 9.3.3.

9.5.4 Push- And PullElemActions

Fig. 9.29 shows two classes derived from `ActionD<3>`. These actions are used to perform the push and pull operations of the hierarchical radiosity algorithm. Both, a push and a pull operation have a `target`, which specifies a `ReferenceD<3>` of an element. For a `PushElemAction` the attribute `target` describes the miniroot of a child container. For a `PullElemAction` this attribute describes any element of the parent container.

Since a push or pull operation may take place across processor boundaries, both actions carry the data of the source of the operation. For a push the action knows about the parent’s data in the attribute `PushElemAction::parentData`, for a pull the attribute `PullElemAction::childData` contains the child container’s data.

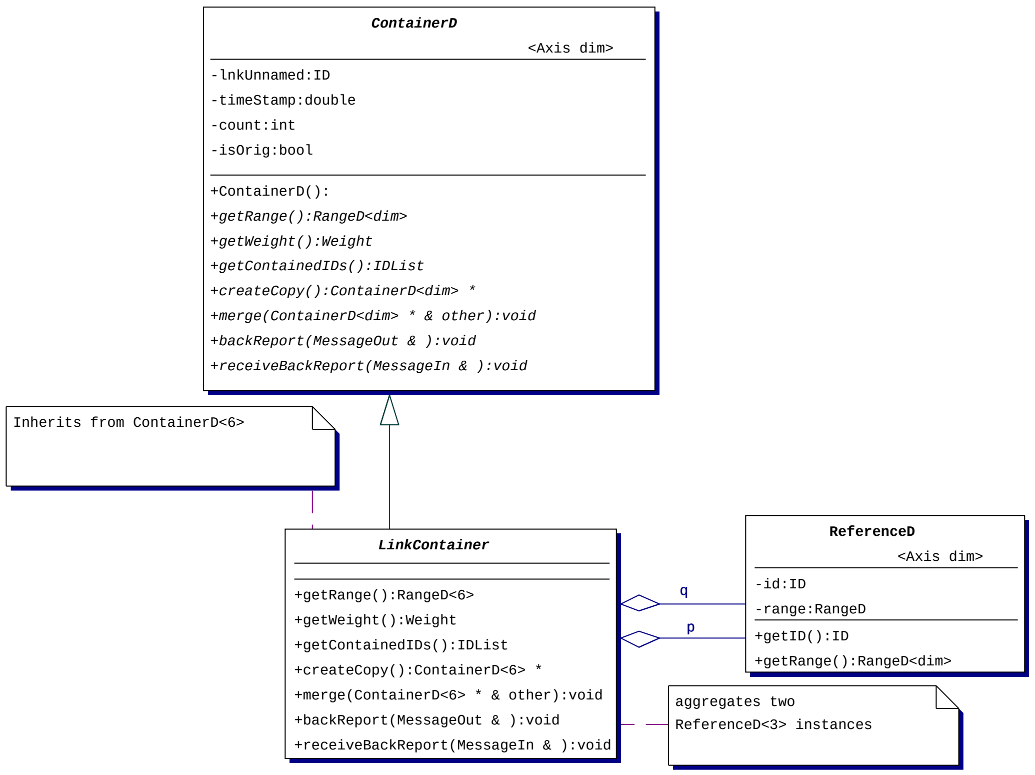

ContainerD
<Axis dim>
-lnkUnnamed:ID
-timeStamp:double
-count:int
-isOrig:bool
+ContainerD():
+getRange():RangeD<dim>
+getWeight():Weight
+getContainedIDs():IDList
+createCopy():ContainerD<dim> *
+merge(ContainerD<dim> * & other):void
+backReport(MessageOut &):void
+receiveBackReport(MessageIn &):void
Inherits from ContainerD<6>
LinkContainer
+getRange():RangeD<6>
+getWeight():Weight
+getContainedIDs():IDList
+createCopy():ContainerD<6> *
+merge(ContainerD<6> * & other):void
+backReport(MessageOut &):void
+receiveBackReport(MessageIn &):void
q
p
ReferenceD
<Axis dim>
-id:ID
-range:RangeD
+getID():ID
+getRange():RangeD<dim>
aggregates two
ReferenceD<3> instances

Figure 9.28: The link container class.

A push action additionally contains `remainingLinksChange`, a value that acts as a proxy for a corresponding child value, as it was described in Sect. 9.5.2.

Push actions are issued during the HRA run everytime when a link container was processed. A push action's `doItLocally`-method is executed on the original element container and checks, whether it is allowed to push downwards (allowed, if `remainingLinks` of the target equals zero, and if there are no existing copies of the target).

Pull actions are issued, when a push action reaches a leaf element container. At inner nodes of the container hierarchy pulling upwards is allowed, if the attribute `ElementContainer::remainingChildren` gets zero (cf. Sect. 9.5.2). If a pull action reaches the root element container, then a new root link container is created and processed.

9.6 Runtime Results

In this section we present some time measuring results of our program. The test platforms were a Cray T3E and a network of workstations. The application and the load balancing procedure are implemented in C++ in SPMD style using MPI.

The load balancer implemented in `DistrDatabaseD<dim>` needs to know the weight of each individual object. The weight should be chosen such that both memory usage and computation time is distributed evenly. Estimating the computation time of the task associated with a given link is difficult. Also the complexity of push and pull operations that are associated with the elements may vary from element to element. As a simple heuristic we assume that the memory size of an object is proportional to the complexity of the associated computations. This simple assumption leads to satisfying results, as is shown below.

In this section we will present plots that were generated over time during the run of our program. These plots are meant to show, how the rebalancing affects the behaviour of the program. They give hints on the key problems (namely *Balance* and *Overhead*) that have to be solved by anyone who re-implements our methods. The runtimes were measured using the MPI-library's wall clock timer functions.

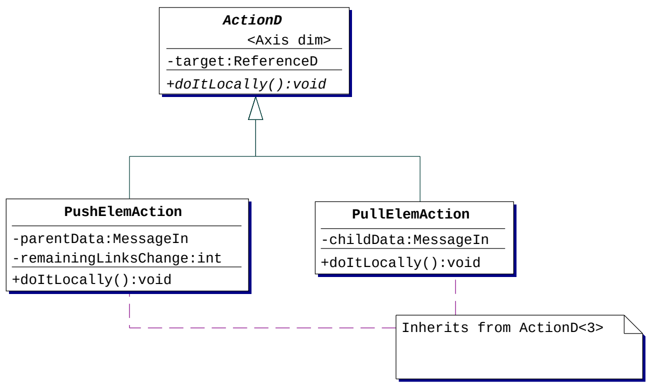


Figure 9.29: The classes for push and pull.

Processor Type	DEC Alpha EV5-21164
Clock Rate	450 MHz
Memory	128 MB
Instructions (peak)	1800 MIPS
Floating Point (peak)	900 MFLOPS
Transfer rate to network	6 $\times$ 500 MB/s
Network topology	3D torus

Table 9.30: Data sheet of a node of Cray T3E.

9.6.1 Environment

The Cray T3E is a massively parallel supercomputer (see factsheet in Table 9.30). The nodes are connected by a 3D torus. Basically every node consists of a memory module, a CPU, and a network router. The latter are responsible for message routing, hence, despite of the incomplete network topology the CPUs are not involved in any routing. The maximum number of processors available for this work was 128.

As an example we consider the calculation of three iterations for the hall scene.

9.6.2 Balance

Fig. 9.31 shows the size of the element database on a 32 processor run. As can be seen, the elements are distributed unevenly at the beginning[4]. Shortly after start the 32 curves approach each other meaning a fairly balanced element distribution. The curves do not match exactly because we performed the rebalancing not on the actual elements but on an approximate load representation.

After an initial growth phase, the element database does not change notably (which has been seen already in Sect. 9.2.1). Hence there is only low need to improve the distribution of elements. The element database sizes remain nearly constant until the end of the program.

In Fig. 9.32 we see a plot that expresses, when rebalancing operations have been performed during the run. The plot's function value is 1 during rebalancing and zero otherwise. Only six short rebalancing operations were needed to keep the element database balanced. The effect of rebalancing can be seen at the small "steps" in Fig. 9.31, when elements were moved between processors.

A similar plot is shown in Fig. 9.33 for the link database. Here the situation is very different. We had a total of 349 rebalancing operations, and almost always a new rebalancing operation was started immediately after a previous operation finished. The "holes" around 410 seconds and 1805 seconds mark the end of an iteration, where no links are present. Instead during these time intervals pushpull calculations have been performed, which are executed independent of any links.

We know from Sect. 9.2 that the number of links in the shooting HRA is changing very dynamically. Our load balancer did the best efforts to distribute the link containers evenly. Fig. 9.34 shows the number of link containers on 32 processors. The number is changing rapidly and the rebalancer does not reach a really balanced situation during the first iteration (25–410 seconds). In the second (410–1805 seconds) and third (1805–3265 second) iteration the situation is better, but not perfectly balanced.

[4] The first 25 seconds in the plots have been consumed during I/O and some preprocessing for time measurements. We have focused in this thesis on the execution of the HRA-iteration's calculations and did not try optimizations on the preprocessing. Hence this time prefix is ignored

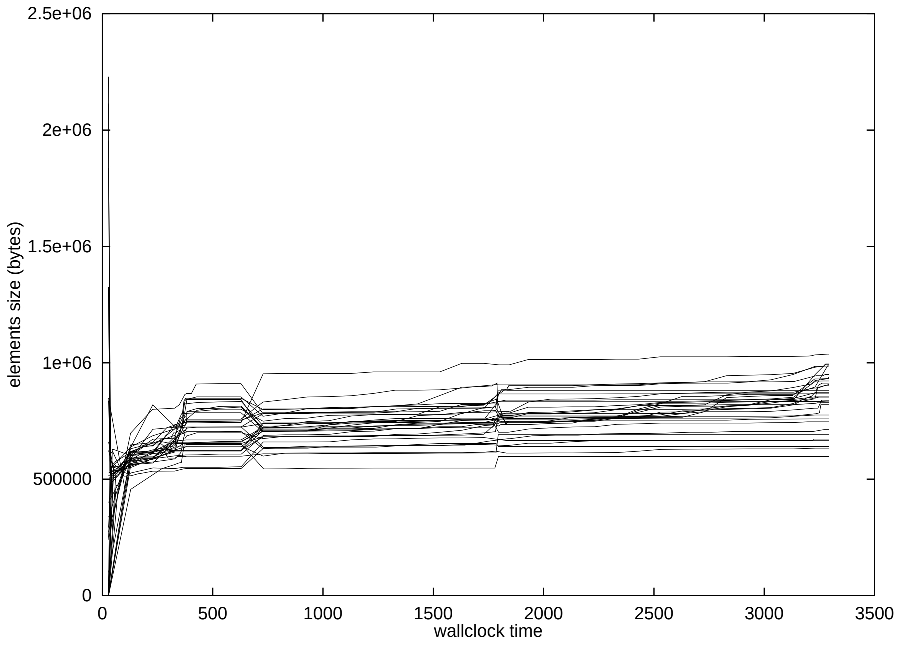


Figure 9.31: Element database size over runtime of the program on 32 processors.

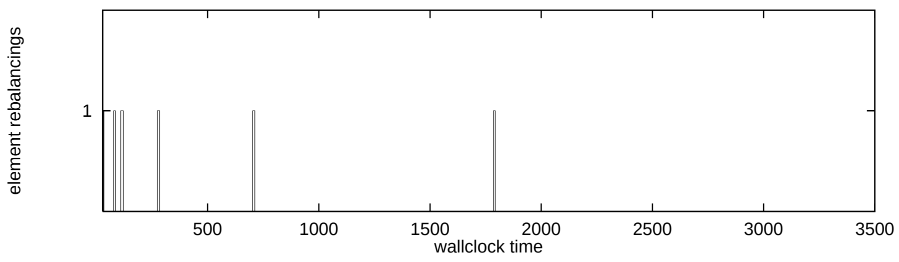


Figure 9.32: Rebalancing operations of the element database.

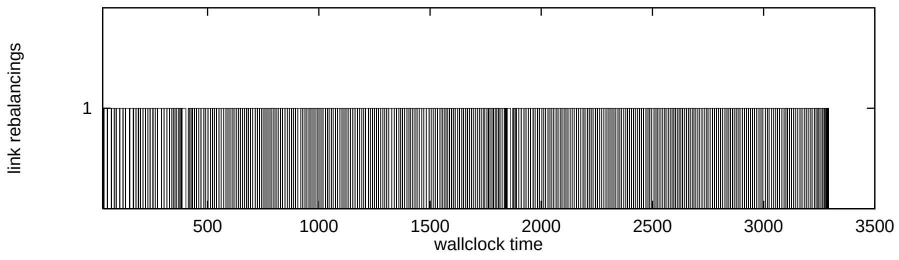


Figure 9.33: Rebalancing operations of the link database.

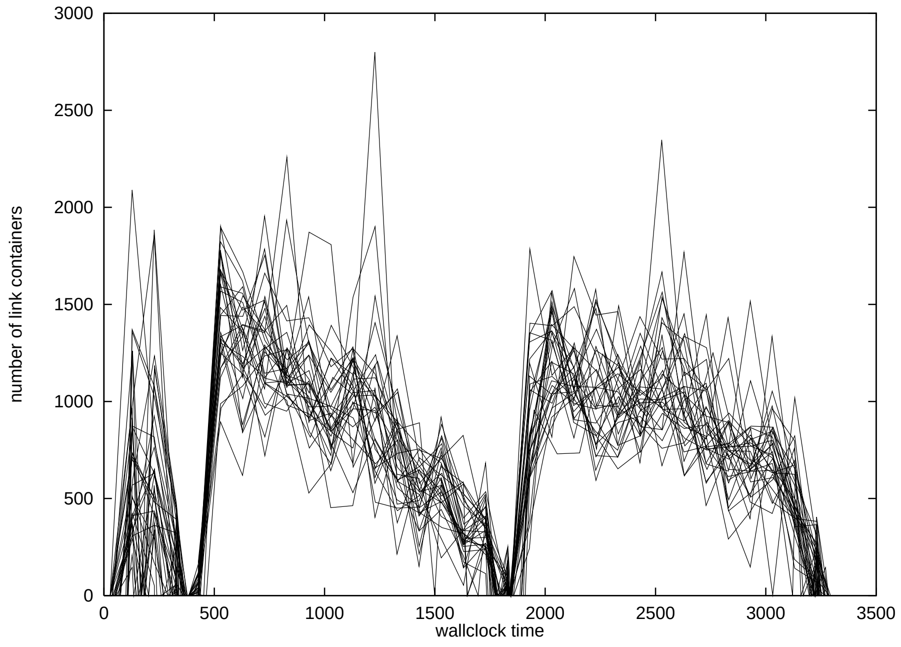


Figure 9.34: Link container number over runtime of the program.

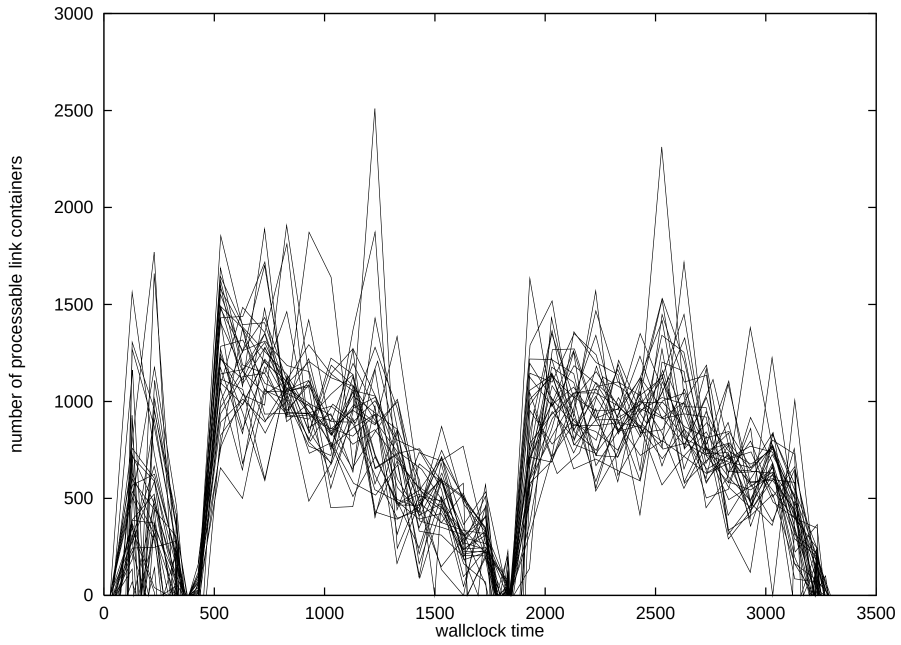


Figure 9.35: Number of processable link containers over runtime of the program.

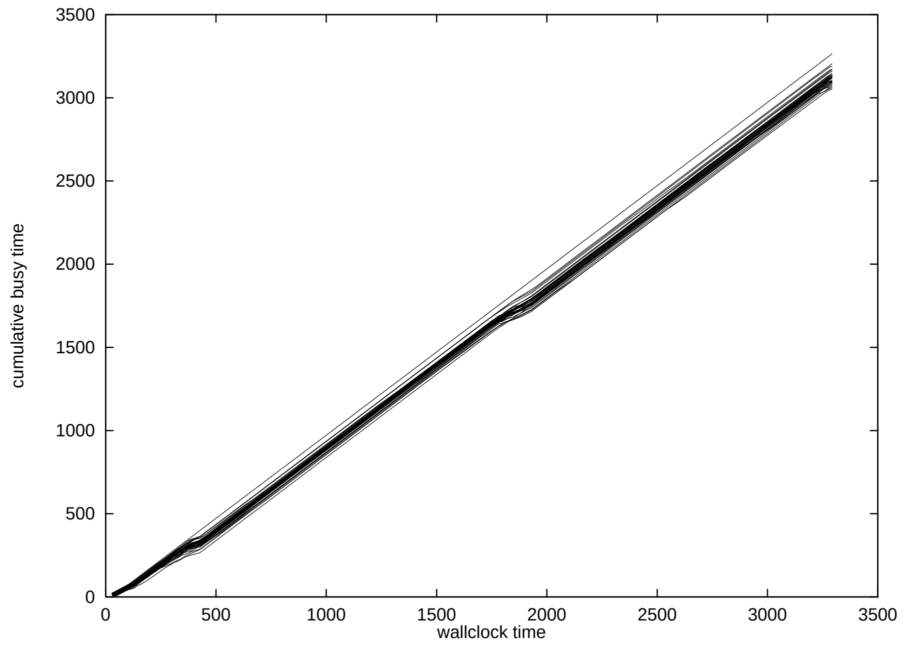


Figure 9.36: Cumulative busy time over runtime of the program.

The imbalance of the link database is not a severe problem. During the parallel computation it is important that at any time every processor has at least one link container locally stored that is *processable*. A processable link container is a link container whose two associated element containers are accessible locally or in the cache. Fig. 9.35 shows the number of processable link containers per processor. Except in the first iteration almost everytime the 32 curves are greater than zero, meaning that all processors are busy.

We measured explicitly the busy time of our program on each processor. Fig. 9.36 shows the cumulative busy time on each processor. The uppermost curve is the ideal busy time (slope 1). The 32 busy-time curves below are very close to the ideal curve. Hence, our rebalancer seems to be able to keep all processor relatively busy. The curves have narrow regions with a flat slope at the end of each HRA iteration. This is nearly unavoidable, since at these points a kind of global synchronisation[5] happens, where all processors wait for the root cluster link to be processed. The average busy time per processor is 3119 seconds of a possible 3265 seconds, i. e. the processors are busy during 96 percent of wallclock time.

9.6.3 Overhead

Of course, keeping all processors busy is only half of the goal. The processors should be doing useful computations most of the time. Fig. 9.37 shows the time per processor spent on useful computations. Again the uppermost curve is the ideal time (slope 1). The difference between the 32 curves of Fig. 9.37 and the busy-time curves of Fig. 9.36 expresses the overhead due to communication and parallel book-keeping. At the end an average of 2720 seconds have been spent on useful computations, i. e. about 87 percent of the busy time has been spent on “useful” communication.

We analyzed the behaviour of the element cache. The cache size limit was set to 4MB. At the end of each iteration the cache was flushed in order to propagate element data back to the home processors. Fig. 9.38 shows the element cache size for each processor. The cache limit was reached by many processors.

The final average *element cache hit ratio* was 88 percent, i. e. 88 percent of all elements searched in the local database or in cache have been found immediately. The hit ratio of 88 percent shows that either many accessed elements are locally stored or that the elements in the cache are reused frequently. Since there are no arrangements in our parallel algorithm that ensures that the elements needed during a link processing are local, we may assume that the latter is true. Fig. 9.39 shows, that the total cumulative hit ratio increases rapidly at the beginning of the run and remains high until finish. Around 410 and 1805 seconds, i. e. at the beginning of a new iteration when the caches just have been flushed, there are only shallow dales in the curves of Fig. 9.39, meaning that the cache is filled quickly again with the most used elements.

in the analysis of runtimes.

[5] The processors are not explicitly synchronized as explained in Sect. 9.3.

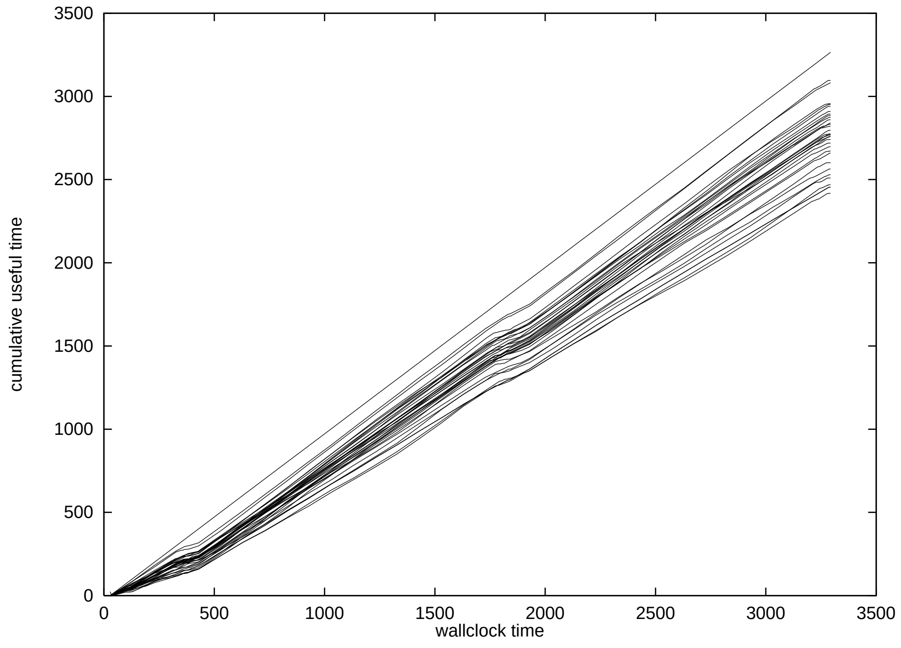


Figure 9.37: Cumulative “useful” time over runtime of the program.

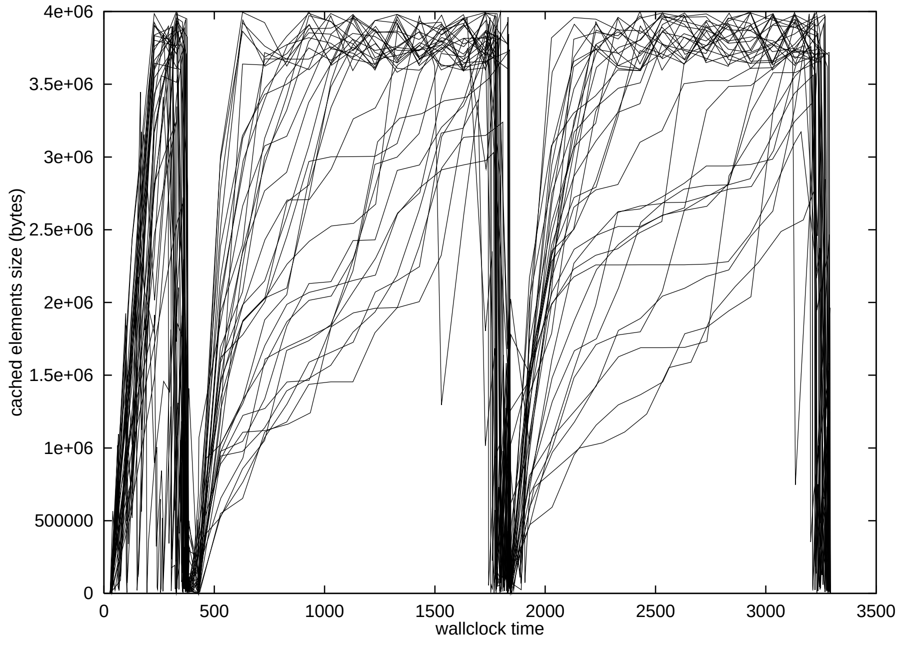


Figure 9.38: Element cache size over runtime of the program.

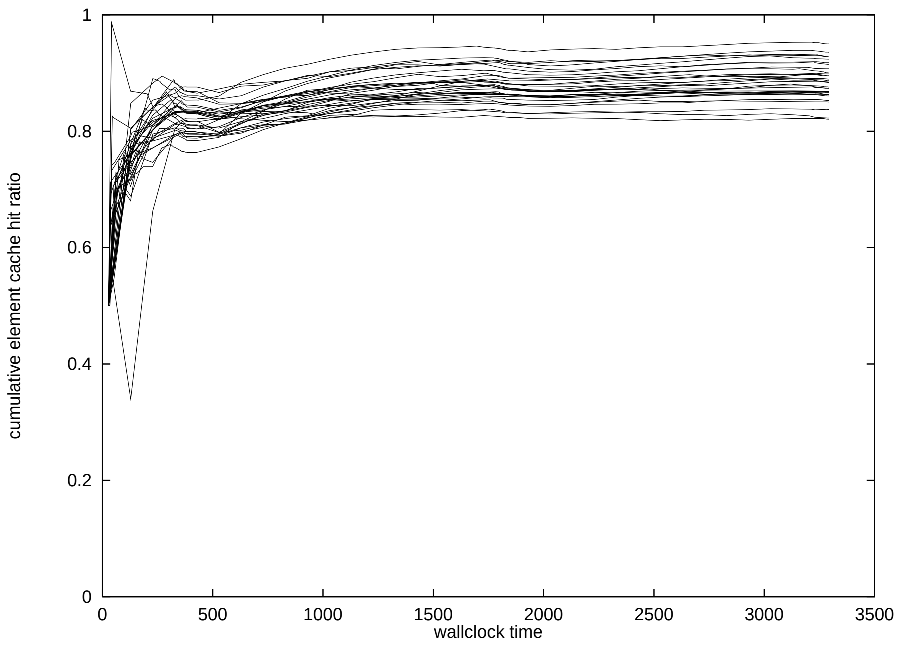


Figure 9.39: Cumulative element cache hit ratio over runtime of the program.

There is another interesting hit ratio to be considered. A link container can be processed immediately, when both associated element containers are accessible. The final portion of link containers that were processable immediately was 79 percent.

9.6.4 Speedup

Finally, we evaluated the runtime of the three HRA iterations on different processor numbers. Fig. 9.40 shows the speedup obtained on the Cray T3E. Because of job runtime limitations on the parallel computer we could not run the program on a single processor. Instead — as often seen in literature — we estimated the runtime on a single processor based on a three processor run, assuming that the speedup is 3 on three processors. As a result on 64 processors we were 53 times faster than on a single processor. The curve is fairly linear for up to 64 processors. For larger processor numbers the curve increases only slowly. For 128 processors the speedup is 61.

In Sect. 5.1.1.1 we discussed existing implementations of the HRA on distributed memory architectures. For some of these implementations ([16, 94, 13, 11]) speedups have been reported for tests on some massively parallel super computer. We are going to compare those results with Fig. 9.40.

Actually, one cannot immediately compare the speedups of the above implementations. The reasons are quite evident. First the size of test scenes differs and also the character of the scenes. Second, some approaches employed clustering, some not. Some do shooting, some gathering. Some used flat polygons, some allowed curved surfaces. Also the computer's computation and network performance is very different. We should remember these caveats when looking at Fig. 9.41, where we have plotted all speedup curves in a single diagram. We see that our approach seems at least competitive.

Let us consider as a second example the simple hall scene. The reduced scene complexity leads to smaller total runtimes (7.5 minutes on 32 processor for the simple hall scene, 55 minutes for the hall scene). The ratio of computation time to communication overhead worsens for smaller scenes. This can be seen in the speedup diagram 9.42. The shape of the curve is similar to Fig. 9.40, but not the scale. This is a well known and nearly unavoidable effect for communication intensive applications.

Besides the Cray T3E we have tested our program on eight identical Linux PCs (Intel Celeron 400 MHz, 128 MB, 100MBit ethernet). Since we use MPI for communication the same program could be used without severe portability problems.

The simple hall scene was observed during the first three iterations. Fig. 9.43 shows the almost perfectly linear speedup. The overhead due to communication in the PC network is larger than on the supercomputer. Hence, the speedup is only 6.3 on eight processors.

As above we are going to compare the speedup with previously published speedups on a network of workstations ([74, 35, 31]). Please remember the warnings above when considering Fig. 9.44, showing the scalability

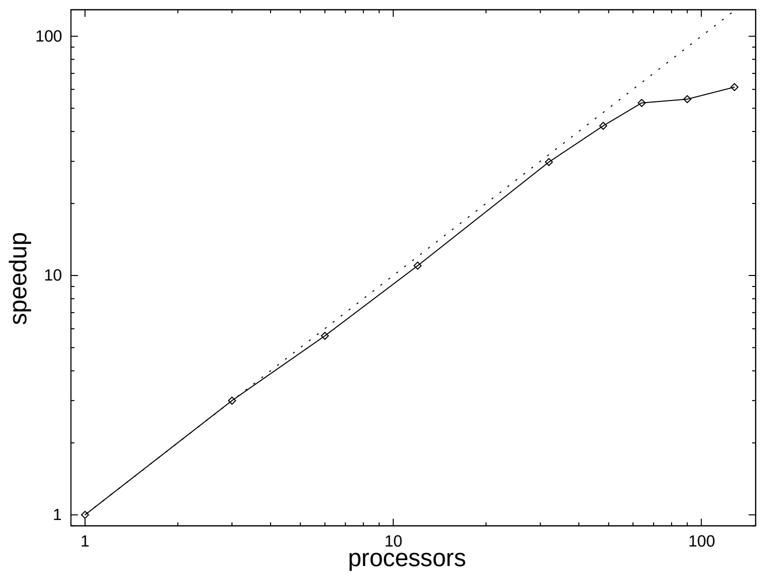


Figure 9.40: Speedup on a Cray T3E for the hall scene compared to ideal (dotted) speedup.

of all these approaches in a single diagram. The super linear speedup of Meneveaux/Bouatouch [74] is due to insufficient memory on a single processor, which leads to more frequent disk accesses, which in turn leads to a high number of disk cache misses. Fellner et. al. [31] and Funkhouser [35] follow a master-slave approach. Their publications document a speedup based on the number of slaves. In order to make those results comparable to our SPMD approach, on 2 processors (master plus slave) we defined the speedup as 1.

In academic research the speedup of a parallel program mostly is measured as we have done it here:

$$\text{Speedup} = \frac{T_1}{T_p}.$$

The times T_i are measured for a fixed problem size on different numbers of processors i. In virtually every scientific computing algorithm there is a real positive fraction of serial, non-parallelizable work $s \in [0,1]$. *Amdahl's Law* of 1967 [5] says that then the speedup obtainable from even an infinite parallel processors is only $\frac{1}{s}$. It seems, that for the hall scene the speedup is limited to about 60, meaning that for this application s supposably is about $s \approx \frac{1}{60}$. Amdahl's argument is a main reason for early skepticism about the usefulness of massive parallelism.

The above way of measuring speedup is not very useful for practitioners. Gustafson [51] states in 1988, that Amdahl's arguments contain the implicit assumption that $1-s$ (i. e. the parallelizable fraction of work) is independent of the number of processors, which is *virtually never the case*. One does not take a fixed-sized problem and run it on various numbers of processors; in practice, *the problem size scales with the number of processors*. As a first approximation, Gustafson found, that usually it is the *parallel* or *vector* part of a program that scales with the problem size.

Regarding this, the objectionable fact that the speedup curve gets flat for $p > 64$, appears less awkward than at first sight. In practice, for a relatively simple task as the simple hall scene, usually we would use only few processors. Complex tasks instead may get access to larger systems, and these tasks benefit most from parallelism. Consider the hall scene, which shows a good speedup of 53 on 64 processors. If we would have another, equally complex scene, that must be calculated simultaneously, we could assign another 64-processor-partition to this scene. The resulting overall speedup for the calculation of both scenes on 128 processors will be $2 \times 53 = 106$. The second scene could for example come naturally from an *incremental radiosity* problem, where two successive frames of a changing scene geometry are to be calculated.

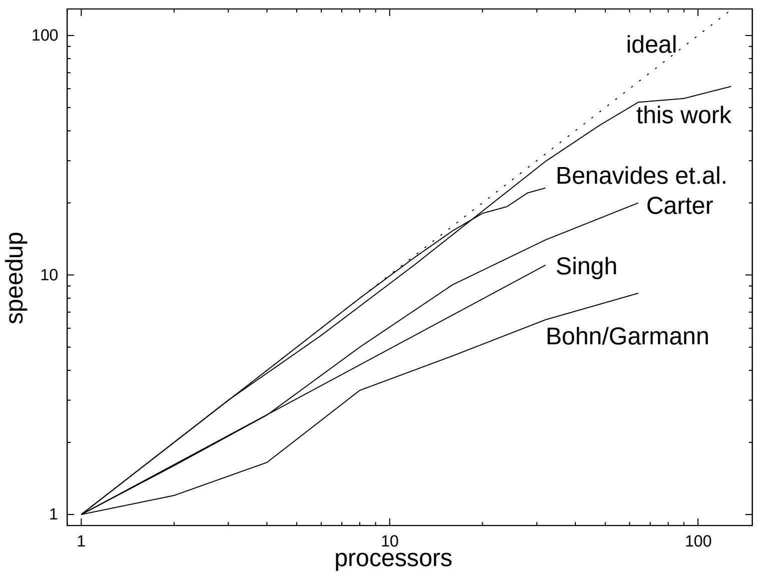


Figure 9.41: Speedup of different HRA-MPP implementations. See caveats in the text.

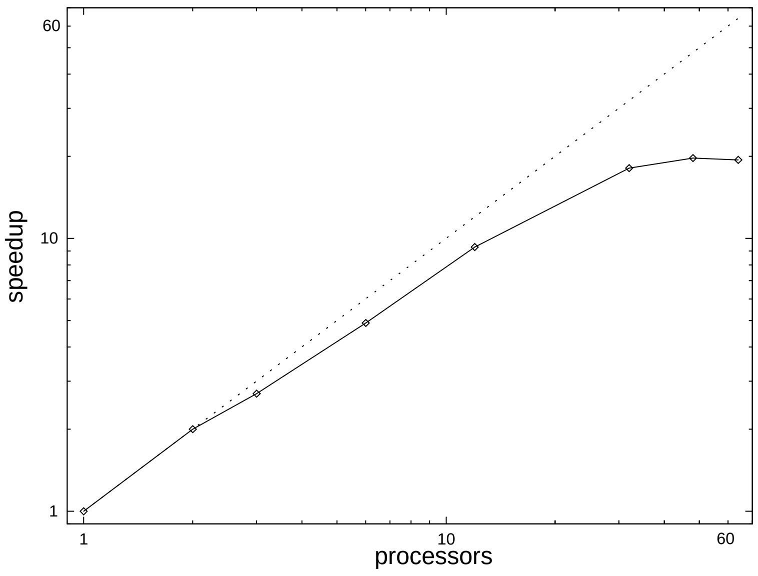


Figure 9.42: Speedup on a Cray T3E for the simple hall scene compared to ideal (dotted) speedup.

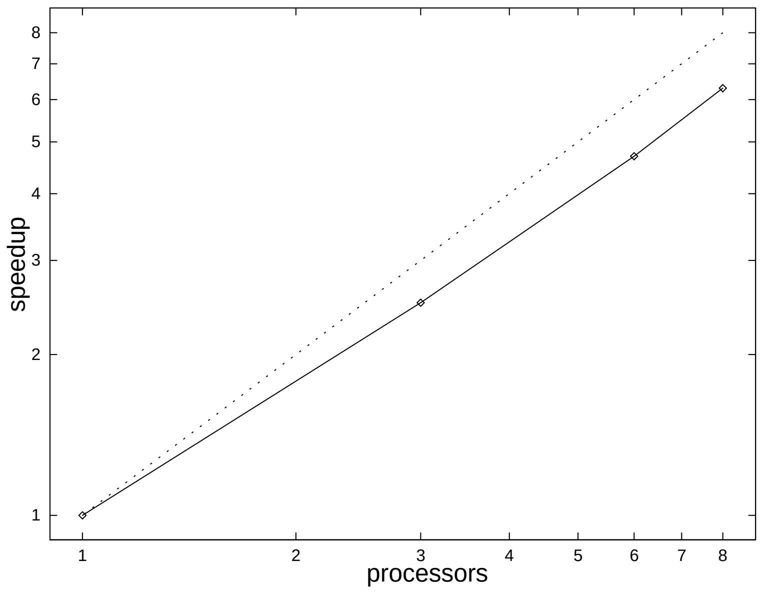


Figure 9.43: Speedup on a Linux PC network for the simple hall scene compared to ideal (dotted) speedup.

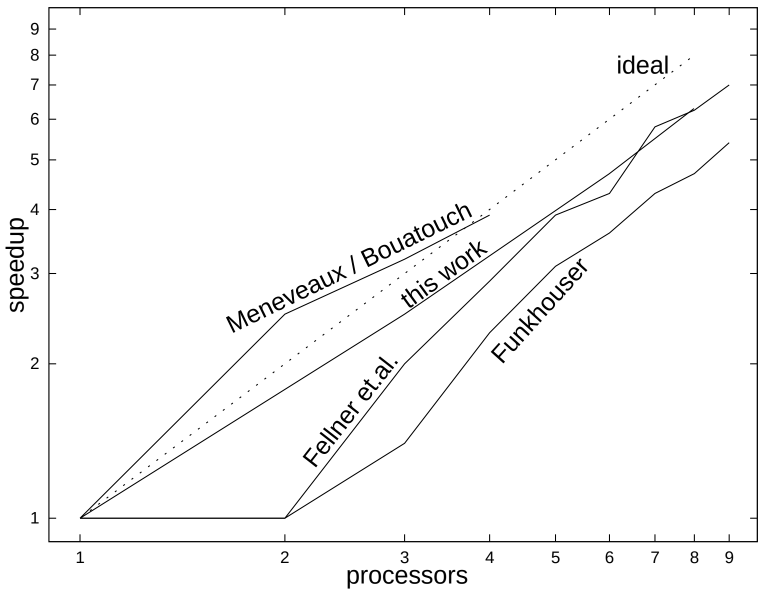


Figure 9.44: Speedup of different HRA-NOW implementations. See caveats in the text.

Chapter 10

Conclusions

This dissertation has focused on two topics: spatial partitioning and parallel global illumination. In this chapter we summarize the results and point out our original contributions. Finally we discuss a few directions of future research.

10.1 Summary

Currently known global illumination algorithms can be classified into two major classes: particle-based and element-based. Both approaches are very time and space consuming. Hence, parallel strategies have been devised by several researchers in the past. Little work has been done on parallelizing the sequentially most efficient hierarchical finite element algorithms. Little to none has happened on parallelizing hybrid approaches.

The hierarchical finite element algorithms have been experimentally shown in this thesis to be very dynamic in nature, very communication demanding, and very prone to congestion. Dynamic load balancing using a spatial partitioner is the best compromise among three representative load balancers, when comparing the combined effect of load imbalance, communication imbalance, and congestion. A reference implementation of hierarchical radiosity using spatial partitioning on a Cray T3E proves the usefulness and scalability of this strategy.

Spatial partitioning has been used earlier by researchers in many domains. It is a well established, robust, and simple strategy of load balancing. Mostly spatial partitioning is used for static load balancing only once at the beginning of a parallel program's run. Nevertheless it can be used to dynamically adapt a partition over runtime. Based on the widely accepted LogP cost model now it has been shown, that dynamic spatial partitioning can be implemented with a worst case overhead that is related to the number of dynamic load updates. The overhead is small in theory and also in practice, as has been shown on a simple synthetic application managing objects in space.

10.2 Original Contributions

The research done in the context of this thesis has led to the following original results:

- A thorough experimental analysis of the dynamics of load and the structure of communication of the hierarchical radiosity algorithm (Chapter 6). Using graph partitioning for this analysis is a new, originally contributed examination technique. New and useful measures have been defined for the continuity of load and the congestion.

- The finding that remote accesses for visibility calculations in hierarchical radiosity potentially could foil any success in parallel hierarchical element implementations (Sect. 6.3, 6.5).

- The observation that load changes very dynamically during a single iteration of shooting hierarchical radiosity (Sect. 9.2.2), but shows relatively high continuity from iteration to iteration (Sect. 6.4).

- The discovery that high quality, expensive graph partitioners do not reduce the communication of hierarchical radiosity essentially better than simple spatial partitioning (Sect. 6.5, 6.6). Spatial partitioning even seems preferable, because it is fast, and it largely reduces congestion.

- A short experimental comparison of the partitionability of standard finite element problems and hierarchical radiosity, showing that the former are much easier to partition than the latter (Sect. 6.7).
- The definition of a dynamic spatial load balancer (Sect. 7.1.3, 7.1.5) with a distributed imbalance detector (Sect. 7.1.4).
- A proof that this spatial load balancer has a small worst case amortized time complexity (Sect. 7.2) in the LogP cost model.
- A new classification scheme of the load dynamics of spatially mappable applications as “constant”, “growing”, “moderate”, and “heavy” based on the minimum/maximum and median/average load density curves over time (Sect. 8.1).
- A thorough experimental analysis of the key influencing parameters (task load, max. allowed imbalance, object number, load pattern, dimension) on the *real world* behaviour of the dynamic spatial load balancer on a simple, spatially mapped application (Sect. 8.3–8.8). The results show good scalability for near-practice parameters, making this load balancer usable for other scientific computing domains.
- The characterization of the link creation procedure of the hierarchical shooting algorithm as “heavy” (Sect. 9.2.2) and that of the elements as something between “constant” and “growing” (Sect. 9.2.1).
- A formulation of hierarchical shooting algorithms for global illumination that abstracts from the representation of light at the elements (Sect. 2.2.5, 2.2.6). This formulation could be useful for a generic description of hybrid algorithms, where light may be represented at the source and at the receiver of an interaction either by particles or by elements.
- A mapping of hierarchical finite element methods to 3D and 6D space (Sect. 5.1.3) putting out a new locality property that greatly reduces the chance of congestion in a parallel implementation. Also this mapping is jointly applicable with a particle based algorithm, making a parallelization of hybrid methods possible (Sect. 5.3).
- Showing, how global barrier synchronizations in a parallel implementation of hierarchical radiosity can be avoided, by allowing the computation to proceed differently in different areas of the element hierarchy (Sect. 9.3.1).
- Introducing grouping of elements and links to achieve a coarser task granularity (Sect. 9.3.2, 9.3.3).
- A parallel implementation of hierarchical shooting radiosity with dynamic spatial partitioning that proves well scalable on a workstation network and on a massively parallel supercomputer (Sect. 9.6).
- An explanation of the object-oriented design of that implementation, showing key concepts that make spatial partitioning in general and spatially partitioned hierarchical shooting specifically manageable for the programmer (Sect. 9.4, 9.5).

10.3 Directions For Future Research

Requirements on the quality of computer generated images are growing since their invention. Modelling surfaces as diffusely reflecting is sufficient for many architectural and outdoor scenes. Nevertheless there are a lot of applications that need higher degrees of realism. Using directional reflection in a finite element algorithm has been shown to be manageable [20, 89]. It is an interesting question, how these directional approaches affect the parallel efficiency of spatial mapping strategies. Supposeably there will be no dramatic influence, because of our specific mapping of links to 6-dimensional space. This mapping can be used also for directional reflection. Thus, locality issues do not seem to be affected.

Higher order bases instead of the Haar basis may be used in finite element algorithms. Commonly used bases have bounded support and can be used immediately in our framework. Actually the parallel performance is likely to increase when using such bases, because the complexity per link task increases due to the higher order of basis functions. Communication is not affected, hence the ratio of computation to communication gets better.

As mentioned in the text, parallelizing hybrid particle / finite element approaches is an interesting and promising direction of future research.

Throughout this thesis we assumed a non participating medium between the surfaces. It is possible to model a scattering, absorpting, and emitting medium as small volumes that can be integrated in a unified hierarchical algorithm [92]. Since the overall structure of the unified algorithm is the same as for the basic hierarchical radiosity algorithm, it should be a simple but interesting task to study the efficiency of spatial partitioning for that algorithm.

In this thesis our main focus was a massively parallel supercomputer with a homogeneous network of identical processors. Considering heterogeneous networks could be interesting, when using a network of different workstations. Our dynamic spatial partitioning approach is capable of modeling different processor speeds. For each node of the tree of Fig. 7.3, page 57, we could store a performance value that expresses the speed of all leaves below the node. During rebalancing these values may be used to distribute load unevenly across the processors. If a processor's speed may vary over time, then we would have to broadcast current speeds during every rebalancing operation — at only low extra cost. Since one of our basic principles was to abstract from the specific network topology, it will be difficult to include modeling of different and/or varying bandwidth on the channels between different processor pairs in our approach.

Appendix A

Calculating Transports

In this chapter we show as an example, how a transport between two surfaces can be calculated. Other transports (cluster to surface, surface to cluster or cluster to cluster) and the push and pull calculations are slightly different but can be derived much the same way.

At the sending surface we have the following element representation of unshot irradiance in some basis $\{N_j^s\}_j$:

$$E^{\text{unshot},s}(\vec{x}', \vec{\omega}') = \sum_j e_j^{\text{unshot},s} N_j^s(\vec{x}', \vec{\omega}').$$

The coefficients are transported to the receiver, whose next iteration unshot irradiance is represented in the basis $\{N_j^r\}_j$ by

$$E^{\text{next},r}(\vec{x}, \vec{\omega}) = \sum_j e_j^{\text{next},r} N_j^r(\vec{x}, \vec{\omega}).$$

The transport of irradiance is performed (see Sect. 2.2.5) by the statement $\Lambda_r^{\text{next}} += \mathscr{T}\Lambda_s^{\text{unshot}}$, which means the following in our situation:

$$E^{\text{next},r}(\vec{x}, \vec{\omega}_{\vec{x}\to\vec{x}'}) += \cos(\vec{\omega}_{\vec{x}\to\vec{x}'}, \vec{n}_{\vec{x}}) v(\vec{x}, \vec{x}') \int_{H^2} f_r(\vec{x}', \vec{\omega}' \to \vec{\omega}_{\vec{x}'\to\vec{x}}) dE^{\text{unshot},s}(\vec{x}', d\vec{\omega}').$$

We substitute the above element representations and get:

$$\begin{aligned}&\sum_j e_j^{\text{next},r} N_j^r(\vec{x}, \vec{\omega}_{\vec{x}\to\vec{x}'})\\ &+= \cos(\vec{\omega}_{\vec{x}\to\vec{x}'}, \vec{n}_{\vec{x}}) v(\vec{x}, \vec{x}') \int_{H^2} f_r(\vec{x}', \vec{\omega}' \to \vec{\omega}_{\vec{x}'\to\vec{x}}) \sum_j e_j^{\text{unshot},s} N_j^s(\vec{x}', \vec{\omega}') d\vec{\omega}'.\end{aligned}$$

In order to incorporate the transported light into the representation at the receiver we project the whole equation into the dual basis $\{\tilde{N}_i^r\}$:

$$\begin{aligned}&\left\langle \tilde{N}_i^r, \sum_j e_j^{\text{next},r} N_j^r(\vec{x}, \vec{\omega}_{\vec{x}\to\vec{x}'}) \right\rangle\\ &+= \left\langle \tilde{N}_i^r, \cos(\vec{\omega}_{\vec{x}\to\vec{x}'}, \vec{n}_{\vec{x}}) v(\vec{x}, \vec{x}') \int_{H^2} f_r(\vec{x}', \vec{\omega}' \to \vec{\omega}_{\vec{x}'\to\vec{x}}) \sum_j e_j^{\text{unshot},s} N_j^s(\vec{x}', \vec{\omega}') d\vec{\omega}' \right\rangle.\end{aligned}$$

Using linearity and duality, we arrive at the following update prescription:

$$e_i^{\text{next},r} += \sum_j e_j^{\text{unshot},s} T_{ij}^{rs}$$

where

$$\begin{aligned}T_{ij}^{rs} = \iint\limits_{\text{support}(r)} &\tilde{N}_i^r(\vec{x}, \vec{\omega}_{\vec{x}\to\vec{x}'}) \cos(\vec{\omega}_{\vec{x}\to\vec{x}'}, \vec{n}_{\vec{x}}) v(\vec{x}, \vec{x}') \int_{H^2} f_r(\vec{x}', \vec{\omega}' \to \vec{\omega}_{\vec{x}'\to\vec{x}}) N_j^s(\vec{x}', \vec{\omega}') d\vec{\omega}'\\ &d\vec{x} d\vec{\omega}_{\vec{x}\to\vec{x}'}\end{aligned}$$

denotes the *transport coefficient* – a coupling coefficient between the basis functions N_i^r and N_j^s. If the integration is done numerically by sampling and if the samples for $\vec{\omega}_{\vec{x}\to\vec{x}'}$ are generated naively over the complete hemisphere, most of the samples will contribute zero, because the samples do not match the spatial support of the sender. Instead a spatial parameterization over the sending surface would be advantageous:

$$T_{ij}^{rs} = \int\limits_{\text{spatial}-\text{support}(r)} \int\limits_{\text{spatial}-\text{support}(s)} \tilde{N}_i^r(\vec{x},\vec{\omega}_{\vec{x}\to\vec{x}'})G(\vec{x},\vec{x}') \int_{H^2} f_r(\vec{x}',\vec{\omega}' \to \vec{\omega}_{\vec{x}'\to\vec{x}}) \\ N_j^s(\vec{x}',\vec{\omega}')d\vec{\omega}'d\vec{x}d\vec{x}'.$$

Appendix B

Notations And Abbreviations

B.1 General Abbreviations

HRA Hierarchical Radiosity Algorithm

SP Spatial Partitioner

NP Naive Partitioner

DM Distributed Memory

ORB Orthogonal Recursive Bisection

MPP massively parallel processing/processors

NOW network of workstations

SPMD single program multiple data

B.2 Mathematics And Rendering

M^2 The set of 2D locations on a (set of) surface(s)

B^3 The set of 3D locations inside a cluster box

$\vec{x}$ a location on a surface

H^2 the set of directions to the upper hemisphere

S^2 the set of directions from the center to the surface of a sphere

$\vec{\omega}$ a direction vector. Depending on the context $\vec{\omega}$ is either 3D (as a vector in 3D space) or 2D (as a function parameter defining a point on the sphere or hemi-sphere).

$\vec{\omega}_{\vec{x}\to\vec{x}'}$ a direction from point $\vec{x}$ to point $\vec{x}'$

μ a tuple of location and direction $\mu = (\vec{x}, \vec{\omega})$.

h_i a direction in H^2 and a function argument for a pdf (see Sect. 2.3.2)

$p(h)$ a probability density function (pdf), also: $p =$ number of processors

$\vec{n}_{\vec{x}}$ surface normal vector at a surface location $\vec{x}$

$v(\vec{x}, \vec{x}')$ visibility term (0 or 1, see Sect. 2.1.1)

$||\vec{x} - \vec{x}'||$ euclidean distance between two spatial locations

$G(\vec{x}, \vec{x}')$ geometric term (see Sect. 2.1.1)

$L(\vec{x}, \vec{\omega})$ radiance as a function of point and direction

$L^{\mathrm{e}}(\vec{x}, \vec{\omega})$ emitted radiance

$f_r(\vec{x}, \vec{\omega}_{\mathrm{in}} \rightarrow \vec{\omega}_{\mathrm{out}})$ bidirectional reflectance distribution function (brdf). Describes, how light is reflected off a surface given a point on the surface and an incoming and outgoing direction.

$B(\vec{x})$ radiosity as a function of location

$B^e(\vec{x})$ emitted radiosity

$\rho(\vec{x})$ diffuse reflectance. Describes, how light is reflected off a diffuse surface.

$\mathscr{T}$ linear transport operator (see Sect. 2.1.3)

$\mathscr{G}$ propagation operator (see Sect. 2.1.4)

$\mathscr{R}$ reflection operator (see Sect. 2.1.4)

I radiant intensity (see Sect. 2.1.4), also: an interval in multdimensional space (see Sect. 2.2.4.2)

L^{in} incident radiance (see Sect. 2.1.4)

E irradiance (see Sect. 2.1.4), also: the number of objects in a subtree (see Sect. 7.1.3), and: the set of edges of a graph (see Sect. 4.1.2)

$E^{\perp}$ perpendicular irradiance (see Sect. 2.1.4)

Φ radiant flux (see Sect. 2.1.4)

$\Delta\Phi$ radiant flux of a particle (see Sect. 2.3.4)

$\mathscr{L}^2(V)$ space of square integrable functions on V

$\langle f, g \rangle$ inner product on a function space

N_i, N_j basis function of a function space (see Sect. 2.2.1)

$\tilde{N}_i, \tilde{N}_j$ dual of a basis function (see Sect. 2.2.1)

δ_{ij} Kronecker's delta, $\delta_{ij} \in \{0,1\}$ and $\delta_{ij} = 1$ if and only if $i = j$.

$\phi_t^s(u)$ Haar scaling function (see Sect. 2.2.2)

$\psi_t^s(u)$ Haar detail function, wavelet (see Sect. 2.2.2)

$L^{\mathrm{imp}}(\vec{x}, \vec{\omega})$ importance as a function of point and direction (see Sect. 2.1.5)

$L^{\mathrm{imp,e}}(\vec{x}, \vec{\omega})$ emitted importance

I^{imp} importance intensity (see Sect. 2.1.5)

$L^{\mathrm{imp,in}}$ incident importance (see Sect. 2.1.5)

E^{imp} projected incident importance (see Sect. 2.1.5)

$E^{\perp,\mathrm{imp}}$ perpendicular incident importance (see Sect. 2.1.5)

Λ_m representation of radiance on a given support m (see Sect. 2.2.5)

$\Lambda_m^{\mathrm{unshot}}$ representation of the unshot radiance on a given support m (see Sect. 2.2.5)

$\Lambda_m^{\mathrm{next}}$ representation of the next iteration's unshot radiance on a given support m (see Sect. 2.2.5)

s sending element (see Sect. 2.2.5)

r receiving element (see Sect. 2.2.5), also: root of tree clustering (see Sect. 7.1.4)

c child element (see Sect. 2.2.6)

B.3 Parallel Computing And Graph Partitioning

p number of processors, also: $p(\cdot)$ = probability density function

λ latency (see Sect. 3.3)

L latency (see Sect. 3.3), also: radiance function (see Sect. 2.1.1)

o overhead for injecting a message into the network (see Sect. 3.3)

g gap between consecutive messages measured in processor cycles (see Sect. 3.3)

G a guest graph (see Sect. 4.1.1). Also gap for long messages (see Sect. 3.3)

G_k^* graph of the *-scene in the k-th iteration (see Sect. 6.3)

H_k^* graph of the *-scene in the k-th iteration without visibility accesses (see Sect. 6.5)

H a host graph (see Sect. 4.1.1)

V set of vertices of a graph (see Sect. 4.1.2)

V_i set of vertices of a portion of a graph (see Sect. 4.1.2)

E set of edges of a graph (see Sect. 4.1.2), also: irradiance (see Sect. 2.1.4), and: the number of objects in a subtree (see Sect. 7.1.3)

v_j a vertex of a graph (see Sect. 4.1.2)

e_j an undirected edge of a graph (see Sect. 4.1.2)

$W_i(v)$ the i-th weight of vertex v (see Sect. 4.1.2)

$W(e)$ the weight of edge e (see Sect. 4.1.2)

π a partitioning of a graph (see Sect. 4.1.2)

$lb(\pi)$ load balance of a partitioning (see Sect. 4.1.2)

$cs(\pi)$ cut size of a partitioning (see Sect. 4.1.2)

$rcs(\pi)$ relative cut size of a partitioning (see Sect. 6.3)

$mvc(\pi)$ maximum per vertex remote connectivity of a partitioning (see Sect. 6.5)

$cu(\pi)$ channel usage of a partitioning (see Sect. 6.6)

$conn(V_j)$ connectivity of a subset of vertices (see Sect. 6.2)

$cb(\pi)$ connectivity balance of a partitioning (see Sect. 6.2)

$vd(v)$ vertex degree of vertex v (see Sect. 6.3)

M_σ^{kl} load continuity matrix (see Sect. 6.4)

B.4 Locality Preserving Load Balancing

k dimension of a k-dimensional space (see Sect. 7.1.2)

h the height of the clustering tree above p processors, i. e. $h = \log_2(p)$ (see Sect. 7.1.2)

l $= \frac{h}{k}$ (see Sect. 7.1.2)

v a node in the tree clustering of Sect. 7.1.3

v_l left son of v (see Sect. 7.1.3)

v_r right son of v (see Sect. 7.1.3)

w a leaf in the tree clustering (see Sects. 7.1.3, 7.1.4)

r the root of the tree clustering (see Sect. 7.1.4), also: receiving element (see Sect. 2.2.5)

$E, E(v)$ number of objects/tasks below v (see Sect. 7.1.3), also: set of edges of a graph (see Sect. 4.1.2), and: irradiance (see Sect. 2.1.4)

$E^0(r)$ number of objects/tasks below r at some time t_0 when the tree clustering was balanced (see Sect. 7.1.4)

$d(v)$ length of the path from v to the leaves in the tree clustering of Sect. 7.1.3

$B(v)$ balance value at node v (see Sect. 7.1.3)

β a user-specified bound on $B(v)$ ($0 < \beta \leq 1$, see Sect. 7.1.3)

M number of objects/tasks to be shifted between v_l and v_r (see Sect. 7.1.3)

S memory size of an object/task in bytes (see Sect. 7.2.1)

C_{adm} cost of an administrative message (see Sect. 7.2.1)

$C_{\text{obj}}(M)$ cost of a message containing M objects/tasks (see Sect. 7.2.1)

U number of updates between two rebalancing operations (see Sect. 7.2.2)

B.5 Implementation

`Identifiable` an object having an `ID` (see Sect. 9.4.1)

`ID` identifies an `Identifiable` (see Sect. 9.4.1)

`PointD<dim>` a point/vector in space (see Sect. 9.4.2)

`RangeD<dim>` a axis aligned box in space (see Sect. 9.4.2)

`ReferenceD<dim>` the address of a remote object (see Sect. 9.4.3)

`Context` context of messages (see Sect. 9.4.4)

`Worker` provides packages of work (see Sect. 9.4.4)

`ContainerD<dim>` (see Sect. 9.4.5)

`DistrDatabaseD<dim>` a collection of containers (see Sect. 9.4.6)

`ActionD<dim>` is performed on local containers (see Sect. 9.4.7)

`Element` represents light by contant basis functions (see Sect. 9.5.1)

`Cluster` contains `PrimObjects` (see Sect. 9.5.1)

`PrimObject` a primitive surface object (see Sect. 9.5.1)

`Material` reflection characteristics of a primitive (see Sect. 9.5.1)

`Emission` emission characteristics of a primitive (see Sect. 9.5.1)

`ElementContainer` a collection of elements (see Sect. 9.5.2)

`LinkContainer` a coupling of two element containers (see Sect. 9.5.3)

`PushElemAction` (see Sect. 9.5.4)

`PullElemAction` (see Sect. 9.5.4)

Bibliography

[1] M. Adler, S. Chakrabarti, M. Mitzenbacher, and L. Rasmussen. Parallel randomized load balancing. In *ACM-SIGACT Symposium on the Theory of Computing (STOC)*, pages 238–247, 1995.

[2] W. Aiello, B. Awerbuch, B. Maggs, and S. Rao. Approximate load balancing on dynamic and asynchronous networks. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 632–641, May 1993.

[3] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka. Parallel construction of multidimensional binary search trees. Technical Report SCCS-753, Northeast Parallel Architectures Center (NPAC), at Syracuse University in Syracuse, New York, March 1996.

[4] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation. Technical Report TRCS95-09, Computer Science Department, University of California, Santa Barbara, April 1995.

[5] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS conference proceedings (Atlantic City, N. J., Apr. 18–20)*, volume 30, pages 483–485, Reston, Va., 1967. AFIPS press.

[6] B. Arnaldi, X. Pueyo, and J. Vilaplana. On the division of environments by virtual walls for radiosity computation. In *2nd Eurographics Workshop on Rendering, Barcelona*, 1991.

[7] J. Arvo and D. Kirk. Particle transport and image synthesis. In *SIGGRAPH '90, Dallas*, 1990.

[8] J. Arvo, K. Torrance, and B. Smits. A framework for the analysis of error in global illumination algorithms. In *Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH '94*, pages 75–84, 1994.

[9] L. Aupperle and P. Hanrahan. A hierarchical illumination algorithm for surfaces with glossy reflection. *Computer Graphics (Proceedings '93)*, pages 155–162, 1993.

[10] A. Barnoy and S. Kipnis. Designing algorithms in the postal model for message passing systems. In *Proc. of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 13–22, 1992.

[11] J. I. Benavides, G. Cerruela, P. P. Trabado, and E. L. Zapata. Fast scalable solution for the parallel hierachical radiosity in distributed memory architecture. In *Proceedings of the Second Eurographics Workshop on Parallel Graphics and Visualisation*, pages 49–60, Rennes, France, September 1998.

[12] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Comp.*, C-36(5), May 1987.

[13] C.-A. Bohn and R. Garmann. A Parallel Approach to Hierarchical Radiosity. In V. Skala, editor, *Proceedings of the Winter School of Computer Graphics and CAD Systems '95*, pages 26–35, Plzen, Czech Republic, February 1995. University of West Bohemia.

[14] S. H. Bokhari. On the mapping problem. *IEEE Transactions on computers*, C-30(3), 1981.

[15] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.

[16] M. B. Carter. *Parallel Hierarchical Radiosity Rendering.* PhD thesis, Iowa State University, Department of Electrical Engineering and Computer Engineering, Ames, Iowa, November 1993.

[17] E. Caspary and I. D. Scherson. A self-balanced parallel ray tracing algorithm. In Heywood Dew, Earnshaw, editor, *Parallel processing for computer vision and display*. Addison Wesley, 1989.

[18] S. E. Chen, H. E. Rushmeier, G. Miller, and D. Turner. A progressive multi-pass method for global illumination. In *SIGGRAPH '91, Las Vegas*, August 1991.

[19] P. H. Christensen, E. J. Stollnitz, D. H. Salesin, and T. DeRose. Wavelet radiance. In *Fifth EUROGRAPHICS Workshop on Rendering, Darmstadt, Germany*, pages 287–302, 1994.

[20] P. H. Christensen, E. J. Stollnitz, D. H. Salesin, and T. D. DeRose. Global illumination of glossy environments using wavelets and importance. *ACM Transactions on Graphics*, 15(1):37–51, January 1996.

[21] P. H. Christensen. *Hierarchical Techniques for Glossy Global Illumination.* PhD thesis, University of Washington, 1995.

[22] M. F. Cohen and D. P. Greenberg. The hemi-cube: A radiosity solution for complex environments. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):31–40, August 1985.

[23] M. F. Cohen and J. R. Wallace. *Radiosity and Realistic Image synthesis.* Academic Press Professional, Cambridge, 1993.

[24] T. W. Crockett. An introduction to parallel rendering. *Parallel Computing*, 23:819–843, 1997.

[25] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, and T. van Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[26] N. Czeranka. Paralleles Hierarchisches Radiosity aufbauend auf dem DGC-Verfahren. Diplomarbeit, Fachbereich Informatik, Universität Dortmund, D-44221 Dortmund, Germany, 1998.

[27] P. Diniz, S. Plimpton, B. Hendrickson, and R. Leland. Parallel algorithms for dynamically partitioning unstructured grids. In *Proc. 7th SIAM Conf. Parallel Proc. Sci. Comput.*, February 1995.

[28] M. Dippe and J. Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18(3):149–158, July 1984.

[29] I. S. Duff, R. G. Grimes, and J. G. Lewis. User's Guide for the Harwell-Boeing Sparse Matrix Collection (Release I). Technical report, CERFACS, Toulouse Cedex, France, October 1992.

[30] P. Dutre. *Mathematical Frameworks And Monte Carlo Algorithms For Global Illumination In Computer Graphics.* PhD thesis, Katholieke Universiteit Leuven, September 1996.

[31] D. Fellner, S. Schäfer, and M. Zens. Photorealistic rendering in heterogeneous networks. In *Proc. ParCo '97, Advances in Parallel Computing*. North-Holland, Amsterdam, 1997.

[32] C.-C. Feng and S.-N. Yang. A parallel hierarchical radiosity algorithm for complex scenes. In *Symposium on Parallel Rendering, Phoenix, AZ*, pages 71–77, October 1997.

[33] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th IEEE Design Automation Conference*, pages 175–181, 1982.

[34] I. Foster. *Designing and Building Parallel Programs.* Addison Wesley, 1995.

[35] T. A. Funkhouser. Coarse-grained parallelism for hierarchical radiosity using group iterative methods. In *SIGGRAPH '96, New Orleans, LA*, August 1996.

[36] R. Garmann. Paralleles Hierarchisches Radiosity auf der CM-5. Diplomarbeit, Fachbereich Informatik, Universität Dortmund, Germany, D-44221 Dortmund, November 1994.

[37] R. Garmann. Hierarchical Radiosity - an analysis of computational complexity. Technical Report 584, Fachbereich Informatik, Universität Dortmund, Germany, D-44221 Dortmund, August 1995.

[38] R. Garmann. Maintaining a dynamic set of geometric objects on parallel processors. Technical Report 646, FB Informatik, Universität Dortmund, D-44221 Dortmund, Germany, May 1997.

[39] R. Garmann. Maintaining Dynamic Geometric Objects On Parallel Processors. In *IEEE Symposium on Parallel Rendering, Phoenix, Arizona*, pages 31–38, October 1997.

[40] R. Garmann. Locality Preserving Load Balancing with Provably Small Overhead. In A. Ferreira, J. Rolim, H. Simon, and S.-H. Teng, editors, *5th International Symposium on Solving Irregularly Structured Problems in Parallel, IRREGULAR'98, Berkeley, California, August 9–11*, LNCS 1457. Springer, 1998.

[41] R. Garmann. On The Partitionability Of Hierarchical Radiosity. Technical Report 702, Fachbereich Informatik, Universität Dortmund, D-44221 Dortmund, Germany, January 1999.

[42] R. Garmann. On The Partitionability Of Hierarchical Radiosity. In *IEEE Parallel Visualization And Graphics Symposium, San Francisco, California*, October 1999.

[43] R. Garmann. Spatial partitioning for parallel hierarchical radiosity on distributed memory architectures. Technical Report 734, Fachbereich Informatik, Universität Dortmund, D-44221 Dortmund, Germany, April 2000.

[44] R. Garmann. Spatial partitioning for parallel hierarchical radiosity on distributed memory architectures. In *Third Eurographics Workshop On Parallel Graphics & Visualisation, Girona (E), 28-29 September*, 2000.

[45] R. Garmann, C.-A. Bohn, and H. Müller. Parallel Hierarchical Radiosity on the CM-5. Technical Report 557, Fachbereich Informatik, Universität Dortmund, Germany, D-44221 Dortmund, December 1994.

[46] A. S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufman, San Francisco, California, 1995.

[47] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modelling the interaction of light between diffuse surfaces. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18(3):213–222, July 1984.

[48] S. J. Gortler, P. Schröder, M. F. Cohen, and P. Hanrahan. Wavelet radiosity. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 221–230, August 1993.

[49] W. D. Gropp and E. Lusk. User's guide for `mpich`, a portable implementation of MPI. Manual ANL/MSC-TM-ANL-96/6, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[50] P. Guitton, J. Roman, and G. Subrenat. Implementation results and analysis of a parallel progressive radiosity. In *IEEE Parallel Rendering Symposium*, 1995.

[51] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, May 1988.

[52] S. W. Hammond. Mapping unstructured grid computations to massively parallel computers. Technical Report 92.14, RIACS, NASA Ames, 1992.

[53] P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics*, 25(4):197–206, July 1991.

[54] A. Heirich and S. Taylor. A parabolic load balancing method. In *Proc. 24th Int. Conf. Par. Proc.*, volume III, pages 192–202, New York, 1995. CRC Press.

[55] B. Hendrickson and R. Leland. Chaco. `http://www.cs.sandia.gov/CRF/chac.html`.

[56] B. Hendrickson and R. Leland. The Chaco User's Guide Version 2.0. Technical Report SAND95-2344, Sandia National Laboratories, Albuquerque, NM 87185-1110, July 1995.

[57] B. Hendrickson, R. Leland, and S. Plimpton. An efficient parallel algorithm for matrix-vector multiplication. *Intl. J. High Speed Comput.*, 7(1):73–88, 1995.

[58] HPC++ Consortium. HPC++-homepage. `http://www.extreme.indiana.edu/hpc++/`.

[59] HPF-Forum. HPF-homepage. `http://www.crpc.rice.edu/HPFF/`.

[60] V. 'Isler, C. Aykanat, and B. Özgüc. Subdivision of 3d space based on the graph partitioning for parallel ray tracing. In *Proc. of 2nd Eurographics Workshop on Rendering, Barcelona, Spain*, May 1991.

[61] J. Jaja. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

[62] H. W. Jensen. Global illumination using photon maps. In *Proc. Seventh Eurographics Wirkshop on Rendering, Porto*, pages 21–30, 1996.

[63] M. H. Kalos and P. A. Whitlock. *Monte Carlo Methods, volume I. Basics.* Wiley, 1986.

[64] G. Karypis and V. Kumar. Metis – multilevel partitioning. `http://www.cs.umn.edu/~ karypis/metis`.

[65] G. Karypis and V. Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, University of Minnesota, Dept. of CS, Minneapolis, MN 55455, August 1995.

[66] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49:291–307, February 1970.

[67] E. Lafortune. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering.* PhD thesis, Department of Computer Science, Faculty of Engineering, Katholieke Universiteit Leuven, February 1996.

[68] F. T. Leighton. *Introduction To Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufman Publishers, 1992.

[69] D. Lischinski, B. Smits, and D. P. Greenberg. Bounds and error estimates for radiosity. In *Proceedings of SIGGRAPH 94*, pages 67–74, 1994.

[70] P. Liu. The parallel implementation of n-body algorithms. Technical Report 94-27, DIMACS center, Rutgers University, Piscataway, New Jersey 08855-1179, May 1994.

[71] R. Lüling and B. Monien. A dynamic distributed load balancing algorithm with provable good performance. In *Proc. of the 5th ACM Symposium on Parallel Algorithms and Architectures (SPAA '93)*, pages 164–173, 1993.

[72] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: A survey and synthesis. In *Proc. 28th Hawaii Int. Conf. on System Sciences (HICSS-28)*, volume 2, pages 61–70, 1995.

[73] K. Mehlhorn and M. H. Overmars. Optimal dynamization of decomposable searching problems. *Information processing letters*, 12(2):93–98, April 1981.

[74] D. Meneveaux and K. Bouatouch. Synchronisation and load balancing for parallel hierarchical radiosity of complex scenes on a heterogeneous computer network. *Computer Graphics Forum*, 18(4):201–212, 1999.

[75] K. Menzel. *Parallele 3D-Bildgenerierung.* Dissertation, Universität-Gesamthochschule Paderborn, Fachbereich Mathematik-Informatik, D-33095 Paderborn, Germany, April 1995.

[76] Message-Passing-Interface-Forum. `http://www.mpi-forum.org`.

[77] Message-Passing-Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 1994.

[78] B. Monien, R. Diekmann, R. Feldmann, R. Klasing, R. Lüling, K. Menzel, T. Römke, and U.-P. Schroeder. Efficient use of parallel & distributed systems: From theory to practice. In J. van Leeuwen, editor, *Trends in Computer Science*, Lecture Notes in Computer Science. Springer, Berlin, 1995.

[79] MPICH — a portable implementation of MPI. `http://www.mcs.anl.gov/mpi/mpich`.

[80] A. T. Ogielski and W. Aiello. Sparse matrix computations on parallel processor arrays. *SIAM J. Sci. Comput.*, 14(3):519–530, May 1993.

[81] M. H. Overmars. *The Design of Dynamic Data Structures.* Lecture Notes in Computer Science; 156. Springer-Verlag, Berlin, 1983.

[82] D. Paddon and A. Chalmers. Parallel processing of the radiosity method. *Computer-Aided Design*, 26(12):917–927, December 1994.

[83] S. N. Pattanaik and S. P. Mudur. Computation of global illumination by monte carlo simulation of the particle model of light. In *Third Eurographics Workshop on Rendering*, pages 71–83, May 1992.

[84] I. Peter and G. Pietrek. Importance driven construction of photon maps. In *9th Eurographics Workshop on Rendering, Vienna*, 1998.

[85] R. Preis. Efficient partitioning of very large graphs with the new and powerful helpful-set heuristic. Diplomarbeit, University of Paderborn, 1994.

[86] R. Preis and R. Diekmann. Party partitioning library. `http://www.uni-paderborn.de/cs/robsy/party.html`.

[87] E. Reinhard, A. G. Chalmers, and F. W. Jansen. Overview of parallel photo-realistic graphics. In *EUROGRAPHICS '98, STAR – State of the Art Reports*, 1998.

[88] E. Reinhard, A. J. F. Kok, and F. W. Jansen. Cost prediction in ray tracing. In X. Pueyo and P. Schröder, editors, *Rendering Techniques '96 (Proc. of the Eurographics Workshop in Porto, Portugal, June 17–19, 1996)*, pages 41–50, Wien, 1996. Springer.

[89] P. Schröder and P. Hanrahan. Wavelet methods for radiance computations. In *Proc. 5th Eurographics Workshop on Rendering*. Eurographics, June 1994.

[90] P. Shirley. *Physically Based Lighting Calculations for Computer Graphics*. PhD thesis, University of Illinois, November 1990.

[91] P. Shirley, B. Wade, P. M. Hubbard, D. Zareski, B. Walter, and D. P. Greenberg. Global illumination via density-estimation. In *Proceedings of the Sixth Eurographics Workshop on Rendering*, pages 187–199, June 1995.

[92] F. Sillion. A unified hierarchical algorithm for global illumination with scattering volumes and object clusters. *IEEE Transactions on Visualization and Computer Graphics*, 1(3), September 1995.

[93] F. Sillion and C. Puech. A general two-pass method integrating specular and diffuse reflection. *Computer Graphics*, 23(3):335–344, July 1989.

[94] J. Pal Singh. *Parallel Hierarchical N-Body Methods and their implications for multiprocessors*. PhD thesis, Stanford University, February 1993.

[95] J. Pal Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer Graphics & Applications*, pages 45–55, July 1994.

[96] P. Slusallek, M. Stamminger, W. Heidrich, J.-C. Popp, and H.-P. Seidel. Composite lighting simulations with lighting networks. *IEEE Computer Graphics & Applications*, March/April 1998.

[97] M. Stamminger, H. Schirmacher, and P. Slusallek. Getting rid of links in hierarchical radiosity. In *Computer Graphics Forum (Proceedings EUROGRAPHICS '98)*, 1998.

[98] R. Troutman and N. L. Max. Radiosity algorithms using higher order finite element methods. In *Computer Graphics Proceedings, Annual Conference Series*, 1993.

[99] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.

[100] R. van Driessche and D. Roose. An improved spectral bisection algorithm and its application to dynamic load balancing. *Parallel Computing*, 21:29–48, 1995.

[101] R. van Liere. Divide and conquer radiosity. In *Proc. of 2nd Eurographics Workshop on Rendering, Barcelona, Spain*, May 1991.

[102] C. Walshaw. Jostle – mesh partitioning software. `http://www.gre.ac.uk/jostle`.

[103] G. Ward. Real pixels. In James Arvo, editor, *Graphics Gems II*, pages 80–83. Academic Press, 1991.

[104] G. Ward and P. S. Heckbert. Irradiance gradients. In *Third Eurographics Workshop on Rendering, Bristol*, 1992.

[105] G. Ward, R. Shakespeare, I. Ashdown, and H. Rushmeier. Materials and geometry format. `http://radsite.lbl.gov/mgf/`.

[106] R. Williams. Dime distributed irregular mesh environment. Technical report, California Institute of Technology, February 1990.

[107] R. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency*, 3:457–481, 1991.

[108] C.-Z. Xu and F. C. M. Lau. The generalized dimension exchange method for load balancing in k-ary n-cubes and variants. *J. Par. Distr. Comp.*, 24(1):72–85, January 1995.

[109] J. Zara, A. Holecek, J. Prikryl, J. Burianek, and K. Menzel. Load balancing for parallel raytracer on virtual walls. In *3rd Int. Conf. in Central Europe on Comp. Graph. and Vis. (WSCG95)*, pages 439–447, 1995.

[110] D. M. Zareski. Parallel decomposition of view-independent global illumination algorithms. Master's thesis, Cornell University, January 1996.

[111] H. R. Zatz. Galerkin radiosity: A higher order solution method for global illumination. In *Computer Graphics Proceedings, Annual Conference Series*, 1993.