Hierarchical Parallelism for a Global Illumination Algorithm

George Drettakis

Department of Computer Science
University of Toronto
Toronto, Ontario, Canada

A Thesis submitted in conformity with the requirements
for the Degree of Master of Science in the
University of Toronto

Abstract

Global illumination algorithms can create highly-realistic computer generated images. FIAT is a global illumination algorithm developed at the University of Toronto that uses an adaptive space subdivision technique, and relies on discrete approximation to simulate the distribution of light through the environment. The algorithm is very expensive in both computation and memory. To improve the execution speed of FIAT and to meet its high memory demands we introduce a novel algorithm that utilises *hierarchical parallelism*.

In the approach presented, a two-level hierarchy of parallelism is used. The first level assigns groups of subvolumes, created by space subdivision, to each station available in a loosely-coupled environment. Multiprocessor stations provide an opportunity for exploiting a second level of parallelism by splitting the computation inside each subvolume into tasks which are assigned to individual processors within the workstation. For loosely-coupled distribution across several stations, numerous problems resulting from concurrent execution are addressed. The solutions presented include an algorithm to assign volumes to stations, an asynchronous communication scheme, and synchronisation protocols. For the tightly-coupled parallelism occurring within a single multiprocessor, significant modifications to the serial algorithm are required to achieve good load balancing among the processors and an attendant improvement in execution speed. The hierarchical scheme developed allows computation to occur on several multiprocessors in parallel.

The design was implemented and tested on both multiprocessor and uniprocessor workstations connected by a local area network. The results demonstrate that execution speed is substantially improved, and by distributing memory across stations it is possible to satisfy very large memory requirements. Significant benefit is achieved at both levels, demonstrating that hierarchical parallelism is a feasible and advantageous approach for FIAT.

Acknowledgements

Thanks first and foremost to my supervisor Prof. Eugene Fiume, who dedicated far more time than I deserved, for both the research and the writing process of this thesis. Eugene is an incredible source of inspiration and energy; working with him is a privilege. Thanks also to my second reader Prof. Michael Stumm, who found the time to make many useful comments on both the structure and the content of this thesis; his mark is more than visible in the pages that follow. Thanks are due to Ian Small whose professional reading and editing of this thesis greatly improved its style and language. The members of the FIAT group, Prof. Alain Fournier, Chuan Chee and Marc Ouellette, all contributed with their suggestions and comments concerning FIAT and my work. On the systems side, Mark Moraes helped make the design and implementation of this thesis much cleaner and of higher quality.

Many thanks to my parents and my sister, to whom this thesis is dedicated, for their continuous support, encouragement and love.

But where would I be without my friends? Even though an ocean separates me from most of them, their letters, electronic and paper, and their phone calls were a continuous source of strength. To Christos, Mania and Lily in Athens, Dimitris, Vali and Efi in Crete, and also Dionisis, now in Wisconsin, and to all my friends around the world who find it more difficult to cope with the distance, a big thank you.

Adapting to Canada was not easy. I am indebted to all the people in the Dynamic Graphics Project and the CS Department, who have put up with my continual grumbling about the weather and my philosophical problems with the structure of this society, without complaint. Thanks also to the University of Toronto for financial support.

Much credit is due to my professors and all the other people at the University of Crete and the Institute of Computer Science, for giving me a sound undergraduate education, and preparing me for research.

Thanks to Massimo's Pizza and the cinemas of Toronto for keeping me alive and happy.

Table of Contents

Chapter 1

Introduction

1.1. Visual Realism in Computer Graphics

Have you ever noticed the bright reflections on the tablecloth when light shines through a glass of water? Or the subtle difference in the colour of a wall when a brightly coloured object is placed next to it? These are just two examples of *secondary illumination*, visual effects resulting from the multiple reflections and refractions of light. Although these may seem unimportant visual cues, they contribute significantly to the way we perceive the environment around us. In a computer generated scene these subtleties have considerable effect in how realistic the image looks.

All such secondary lighting phenomena are the results of complex interactions of light with objects in space. Simulating this process to produce a high-quality computer graphics image requires the use of significant computational resources and storage.

Many computer graphics researchers have devoted their efforts to creating realistic images. The techniques developed have evolved and changed over time, but as algorithms are designed to allow the reproduction of even more realistic lighting phenomena, the practical impact is the same: more computation time and more storage is required. To achieve secondary illumination effects such as the concentration of light from reflections and refractions, or the effect of 'colour bleeding' on diffuse surfaces, the computation times of methods previously proposed are in the order of many days [Imme86] and the required storage in the hundred of megabytes or even gigabyte range.

We will be dealing with FIAT, a new computer graphics image synthesis algorithm that can simulate many of the effects of secondary illumination that other algorithms cannot. The cost in memory and computation is, however, very large. As a result it is a pressing need to provide ways to allow pictures to be rendered faster, in this way validating the algorithm.

One appealing solution is to parallelise the computation. As cheaper parallel workstations become available, all connected on a local area network, a hierarchical solution becomes appropriate. The solution of the rendering problem can be segmented into tasks that are distributed among the parallel workstations. These tasks are then split into smaller subtasks executed on each one of the processors on the parallel workstation. This has the double benefit of speeding up computation and distributing memory requirements over a number of computers.

1.2. Visual Realism and Hierarchical Parallelism

The effort in this thesis is concentrated on a parallel algorithm for FIAT, a paradigm that attempts to create images of high realism by simulating the propagation of light in the environment. A discrete approximation of light propagation is utilised to achieve this goal. Although it is still an ongoing project, the first indications are good. Perhaps the most crucial obstacle in the development of FIAT, and most probably of all physically-based models, is the enormous computational expense involved. Small pictures take hours to produce, and testing becomes very time-consuming and difficult. As a result, testing, and consequently development, are impeded.

A remedy to these problems can be found in utilising parallelism. The concept of *hierarchical parallelism* is especially appealing. We believe that this approach is necessary, advantageous and viable:

- It is necessary, because it is the way to overcome the limitations of technology, which restrict the number of powerful processors that can be placed in parallel computers. This is especially true if one does not wish to adopt the alternative topologies of hypercubes or mesh-connected computers. This approach is also dictated by the evolution of technology that is providing affordable hierarchical environments.
- It is advantageous, since it can offer speedup of computation for complex problems, and simultaneously expand the storage space available to the algorithm. The speedup is achieved by decomposing the algorithm at several levels, each of which can be solved in parallel. Memory expansion is facilitated by combining the memories of multiple machines. The advantage of the hierarchical structure is that the extra memory is available to multiple processors at each node.

- It is viable, as we shall show by the design and implementation presented in this thesis. Although the attempt we will present was challenging, the overall benefits fully justify the effort involved.

1.3. Achievements

The basic idea of the FIAT algorithm (described in greater detail in Chapter 5) is the subdivision of the scene into small volumes, in which the propagation of light is simulated. Based on the the results of light propagation, the corresponding parts of the screen are shaded.

The parallel model we adopted in this thesis is a two-level hierarchy. Initially the hierarchical structure of the volumes in space is segmented, and the parts of this structure are distributed among the (possibly parallel) workstations connected with a network in a loosely-coupled fashion. The computation inside each subvolume of space is in turn decomposed in a manner suitable for tightly-coupled parallelism, implemented on the shared-memory multiprocessor workstations.

Parallelising a large application such as FIAT is not easy. To achieve a successful task decomposition suitable for tightly-coupled parallelism, a significant reorganisation of the serial algorithm is required. Two main reasons make this necessary. The first is the fact that even though there are tasks that can be performed in parallel inside the space subvolumes created by the algorithm, the high-bandwidth communication required between the potentially parallel threads can result in significant delays from synchronisation and locking. The second reason is that it is necessary to modify the way in which the computation is performed so that each processor can be assigned comparable load, thereby achieving satisfactory load balancing. The tightly-coupled approach we develop overcomes these problems and shows that FIAT is suitable for parallelism.

In the presentation of the loosely-coupled algorithm we discuss a number of more complex issues. This part of the solution involves difficulties that stem from the significant differences between distributed and traditional uniprocessor computation. The problems dealt with for this parallel approach are mainly those of synchronisation, due to the asynchronous nature of communication between the stations. We also address issues of maintaining consistency with the approach in the original algorithm, since the distribution of the computation alters several facets of the solution. Another major issue in the loosely-coupled design is that of minimising network traffic. We design and implement a solution that deals with the synchronisation issues using a simple, clean protocol, and allows network traffic to be reduced using a buffering scheme.

Overall, pleasing results were achieved by the parallel algorithm developed for FIAT:

- The speedup achieved by both levels of this parallel attempt is satisfactory. Though not linear, the tighly-coupled solution, depending on the scene rendered, results in speedup of up to 3.5 using 4 processors. The distributed case, although inherently limited by the speed of the network used, achieves a speedup of up to 6 using a collection of one 4 processor, one 2 processor and two uniprocessor computers, working together on the same problem.
- The memory requirements of FIAT are distributed across workstations in the distributed solution. As a result, higher quality pictures that require more storage can be rendered.

The whole effort described in this thesis was an interesting experiment, both in parallelism and in computer graphics. In summary there are two main achievements in this thesis:

- It is shown that FIAT can be successfully parallelised, and as a result FIAT becomes a much more attractive solution for image synthesis. The rendering capabilities of FIAT which surpass those of previous computer graphics algorithms, can now be explored and developed at much lower cost in computation time.
- It is shown how hierarchical algorithms are a viable solution to the computational and storage requirements of expensive computer graphics algorithms. By presenting a full, working solution, we indicate some of the capabilities and the advantages of hierarchical parallelism.

1.4. Thesis Structure

This thesis relates to two major areas in computer science, graphics and parallel computation. Before presenting the new material of this thesis, we must first establish the background and previous work in both areas. As a result, the background material presented is somewhat extensive.

Chapter 2 is an overview of the evolution of rendering in computer graphics. An introduction to the basic concepts of illumination models is presented. We then continue with the actual rendering algorithms, beginning with the first scan-conversion methods, and then presenting ray tracing, radiosity-based techniques and finally several hybrid methods that are the most successful attempts to achieve realistic computer imagery to date. We close Chapter 2 with a brief introduction to FIAT, so that the reader can see how it extends other rendering algorithms, and where it fits in the history of rendering algorithms. The reader familiar with computer graphics research may find it preferable to quickly skim through this chapter.

In Chapter 3 we present a brief overview of available parallel architectures. We discuss shared-memory single-bus architectures in slighly more detail, since these are the computers we use in the tightly-coupled parallel implementation. We also discuss the loosely-coupled environment we use, and introduce some evaluation measures that will be utilised for determining the success of the methods developed. Those familiar with the research in parallel computation may wish to omit this chapter, since it covers well known architectures and concepts in parallelism.

Chapter 4 is an overview of the various attempts that have been made by other researchers in computer graphics to utilise parallelism and its capabilities. We have summarised these attempts in Table 4.1 on page 35. A reader who wishes to get a quick overview of the work in the area may wish to consult this table. The last section of this chapter compares previous attempts with the work developed for this thesis.

In Chapter 5 we present FIAT in some detail, to allow the reader to understand how the algorithm functions, and what the problems and challenges are for a parallel approach.

Chapters 6 and 7 are the description of the hierarchical algorithm we developed to parallelise FIAT. In Chapter 6 we discuss the tightly-coupled solution, suitable for shared memory multiprocessors. We present the design issues and the solutions proposed and implemented, together with measurements from the implementation.

In Chapter 7 we describe the problems of loosely-coupled distribution that are more complex than those in the tightly-coupled case, since the model of computation is significantly different from the serial approach. We present the solutions given to these problems, and we also discuss results of the implementation.

Finally, in Chapter 8 we summarise the research in this thesis, with a condensed presentation and analysis of the results of the implementation. We close with a discussion of future research and a retrospective of our effort.

Chapter 2

The Evolution of Computer Graphics Rendering Algorithms

2.1. Introduction

In this chapter we will present an overview of computer graphics algorithms developed to generate realistic images. We introduce the concept of an illumination model, and discuss the distinction between local and global illumination models. An overview of existing local illumination models is presented. The main categories of rendering algorithms are then briefly introduced:

- Scan-conversion algorithms.
- Ray tracing and related approaches.
- Radiosity-based techniques, including the algorithms developed to refine the original approach.
- Combined methods that attempt to merge the benefits of ray tracing and radiosity-based approaches.

We close the chapter with a brief introduction to FIAT, the target algorithm of the parallel solution in this thesis. A more detailed description is presented in Chapter 5, that will allow the reader to comprehend the needs and the problems for parallelism.

Since this discussion is intended as an introduction to image-synthesis techniques, the reader familiar with the research in this area should feel free to skim through this chapter.

Figure 2.1: Rendering

2.2. Illumination Models

In computer graphics systems the goal is to produce an image from a scene that is specified by the user. A scene $S \subseteq R^3$ is a collection of modelling primitives such as curved surfaces, polyhedra or algebraic surfaces, but can sometimes include simpler primitives such as polygons. An image is an array $I \subseteq Z^2$ of pixels†. Each pixel has a colour assigned to it. An illumination model, given the scene S, the viewing information that determines from where the scene is perceived by the observer, and information about lighting in the scene, determines a colour for any point in S. The process of mapping a scene to an image is called *rendering* (Figure 2.1). A rendering algorithm samples the illumination model to determine the colour of the pixels.

Broadly speaking there are two categories of illumination models, *local* and *global*. A local illumination model accepts as input the scene, viewing parameters as defined above, and a set of light sources. The output of the model is outgoing light intensity (Figure 2.2(a)). Global illumination models differ in that outgoing light information can also affect incoming illumination (Figure 2.2(b)).

Local illumination models are used by rendering algorithms to shade visible pixels in the final raster image (especially in scan-conversion techniques), or to determine shade values at a set of points in three dimensional space (for example in ray tracing). In a similar fashion, global illumination models are used to determine intensity values in the scene by modelling the interaction of light with the environment in some fashion.

An algorithm that implements a global illumination model (a global illumination algorithm) attempts to simulate the distribution of light through the environment from the light sources to the objects and then to the observer. To this end Kajiya [Kaj86], formulated the *rendering equation* (Equation (1)). This equation defines the intensity $I(x,x')$ of light passing from a surface x to a surface x' in a scene. We assume that the scene S

† We will be considering only raster images in this thesis.

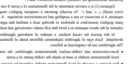


Figure 2.2: Local and Global Illumination Models

consists of the union of small surface elements x''. The equation states:

$$I(x,x') = g(x,x')[\varepsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''] \qquad (1)$$

The function ε defines the amount of light emitted from x' to x and ρ is the intensity of light reaching x' after being scattered from surface x''. The function g is a geometric predicate determining the visibility relation between x and x'.

Equation (1) is a concise statement of the illumination of a scene if one considers all (very small) x, x' and x''. To actually generate a computer graphics image using this equation, it is necessary to use a sampling and reconstruction technique. A goal of computer graphics researchers is therefore to develop such a method that approximates the solution of the above equation to a level that will render convincing and realistic images.

In this pursuit for visual realism, a number of rendering algorithms have been developed. Each type of algorithm implements different kinds of illumination models. The algorithms can be categorised as follows:

- Scan-conversion and visible-surface determination algorithms: the interpolation of local illumination models is used to shade the visible points of a scene.
- Ray tracing: a cascade of local illumination models achieved by tracing rays into the scene, and creating the corresponding ray tree.
- Radiosity-based techniques: global illumination resulting from the application of heat-transfer theory.
- Combined methods: the use of ray tracing and radiosity-based approaches in an attempt to achieve an approximation to global illumination.

We will be discussing each category separately. As local illumination models are used in the first two approaches, and even in the radiosity-based techniques for final image generation, we first present an overview of their evolution.

2.2.1. Local Illumination Models

Local illumination models are methods for producing outgoing light intensity information for a point in the scene, given a set of light sources and viewing information, and using scene information that is local to that point. Before discussing the various local illumination models, we present the parameters common to all. These are depicted in Figure 2.3. For a point P of a surface we define the *surface normal* N, the vector perpendicular to the surface at point P, and ϕ, the angle of incidence between N and the direction of impinging light L. R is the direction of ideally reflected light, which also forms an angle ϕ with N, while E is the direction of the viewer, or eye direction, that has an angle of θ with the reflection direction R. We assume that for a light source h the intensity of light emitted is I_h.

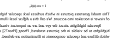


Figure 2.3: Local Illumination Parameters

The relationship between the angle of incidence of the light impinging on a surface and the brightness of light at that point is well known. Early in the evolution of computer graphics this fact was used in the models developed [Bouk70]. In these models the intensity value is given from the equation:

$$I = \cos(\phi) I_h$$

This model generates pictures in which surfaces lack specular highlights since the angle of viewer is not taken into account. We call this a *diffuse* local illumination model.

Specular highlights attract the eye and are an important visual cue. To allow these highlights to be visible in the pictures rendered, Phong [Phon75] developed a model in which diffuse and specular components to illumination are defined.

The Phong local illumination model is defined as:

$$I = k_a I_a + \sum_{all\ l_i} (k_d \cos(\phi) I_{l_i} + k_s \cos^n(\theta) I_{l_i})$$

The product $k_a I_a$ is an ambient illumination term that defines the overall light existing in the scene, and k_s, k_d are respectively the specular and diffuse reflectivity of the surface. It must be noted that I is a vector function for light, which is sampled for various light wavelengths depending on the colour system being used†. The function $\cos^n(\theta)$ is a decreasing function with respect to n, if θ is constant and $0 \le \theta \le \pi/2$. As a result, for a specular surface a suitably selected large value for n will make the specular intensity higher in a smaller region, producing the effect of a highlight.

The above approaches are to a large extent arbitrary, and do not include an attempt to actually model the physical interaction of light with a surface. The pictures that can be generated using the previous models are limited by this fact in the degree of realism they can reproduce. Blinn [Blin77] first attempted to overcome this limitation by developing a model of the structure of each surface. In this model, a surface consists of micro-facets that can be oriented in various directions. A probability distribution is used to describe these orientations. The geometric relationships between these micro-facets are derived (occlusion, blocking etc.), and with the addition of results from thermal engineering, a model with some physical credibility is formulated. One of the interesting results of this approach is correct highly specular reflection when light impinges at a grazing angle on a surface.

Cook and Torrance [Cook82] extend Blinn's model even further by incorporating measured surface reflectance properties using thermal engineering techniques and a different distribution for micro-facets. This allows different surface materials to be rendered convincingly, albeit at the cost of increased computation and the availability of each material's reflectance property information.

2.3. Scan Conversion and Visible-Surface Determination

The first approaches to rendering were scan-conversion algorithms. These methods interpolate samples of a local illumination algorithm to generate a shade for the pixels in the image, and employ a visible-surface determination method to identify the visible points.

---

† Both k_s and k_d are vector valued, while in Phong's model k_s is a scalar, and hence specular reflected light has the colour of the light source. A well known colour system used is RGB, in which colour is sampled for the red, green and blue components.

2.3.1. Shading Techniques

The simplest shading technique is *flat* shading, which simply assigns a single value to all pixels of a surface element.

Shading techniques that use interpolation are particularly useful when shading a polygon that has resulted from the subdivision of a curved surface, as shading will make the edges of the polygons less noticeable, resulting in a smoother-looking surface. In such approaches, each vertex of the polygon maintains the normal vector of the curved surface at that point. Gouraud [Gour71] uses these values to calculate the shade at the vertices and then interpolates the shade across the polygons as they are scan-converted. The method due to Phong [Phon75] interpolates the normals at the vertices thus generating a value for the normal at every pixel. The local illumination model can then be accurately applied for every pixel, resulting in even smoother looking surfaces at a significantly higher cost.

2.3.2. Visible-Surface Determination

The simplest visible-surface technique is the z-buffer [Fole82]. A buffer of depths is maintained, and when a pixel is chosen it is displayed only when its depth is smaller than that in the buffer, in which case the buffer value is updated. More involved techniques, such as that developed by Watkins [Fole82], use edge-coherence of polygons and scan-line coherence, and manipulate a set of complicated data-structures to achieve faster image rendering. Warnock [Fole82] developed an area-subdivision method in which the screen is subdivided when a decision cannot be easily made about visibility by means other than depth comparison (e.g. a polygon containing or surrounding another etc). Subdivision continues until indivisible areas, typically the size of pixel, are created. Depth comparisons are then made for these areas.

2.3.3. Computational Cost

If one takes into account the state of technology at the time these algorithms were developed, they are relatively expensive. Visible-surface determination by the z-buffer technique has a complexity of $O(nM)$ where n is the number of objects and M is the resolution of the screen. This is somewhat costly, but once more involved local illumination models are used that require normal interpolations, cosine and power calculations, they overwhelm the execution time. On a typical 10 MIPS, 1 MFLOP workstation of today, with 8 MB of main memory, a scene with a few hundred polygons takes on the order of a minute to compute. The largest part of the storage requirements is the object database, that for the same example, would by no means exceed the hundreds of KB range, thus easily fitting in main memory. If a z-buffer technique is used on a machine that lacks special hardware, the storage required for the z-buffer can be large (several megabytes), but this problem can be alleviated by using lower precision. Scan-conversion methods, however, still comprise the only feasible way to produce animated sequences of computer generated imagery in a reasonable amount of time. With the advent of affordable special purpose hardware it is now possible to achieve real-time generation of such animations. The more complex methods described in the following sections are still too expensive for such endeavours.

2.4. Ray Tracing

If rays are shot from the light sources into the environment, and the laws of optics applied whenever an object is hit until light reaches the observer, multiple specular interreflections and refractions would be accurately modelled. Tracing rays from the light sources, however, would be exceptionally inefficient as there would be an enormous amount of wasted compuation involved (many rays would never reach the eye). As an alternative, the ray tracing method was proposed for computer graphics images initially by Whitted [Whit80].

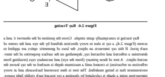


Figure 2.4: Ray Tracing

Ray tracing is conceptually quite simple. Given the position of the observer E and a screen (Figure 2.4), a ray is shot in every direction defined by the eye and the centre of each pixel. If this ray intersects an object, the laws of geometric optics are applied to determine a reflected and refracted ray, depending on the surface properties of the intersected object. A tree of these primary (from the eye) and secondary rays (resulting from reflection or refraction) is created until a maximum depth is reached or the ray leaves the environment that is being modelled. The tree is then traversed backwards and at every intersection point a shade is calculated by shooting a ray toward each visible light source, and applying a local illumination model. The shading information is propagated up the tree to determine the final shade for the pixel (see Figure 2.4).

This method produces very realistic pictures, especially for highly specular (shiny, mirror-like) environments. Nevertheless, it suffers from some problems that are inherent to the algorithm itself. It does not model any secondary illumination effects (for example a mirror that illuminates an area that is in shadow from the primary light sources). It cannot handle diffuse interreflections, since it only samples reflections across the perfect reflection direction. Ray tracing also suffers from sampling and aliasing artifacts, since it is essentially a point-sampling technique. Whitted noted this in the original paper [Whit80] and proposed super-sampling as a solution (for example shooting through the four corners of a pixel). He also noted that the largest amount of time was spent performing intersections of rays with objects.

It is interesting to note the rather lax approach that ray tracers typically take toward energy and its conservation. The local illumination models used for shading when ascending through the ray tree do not have any strong relation to physical reality. It is usually the case that if the model is used to integrate over the sphere of directions about a reflection point, the incoming power can be arbitrarily increased or decreased.

2.4.1. Enhancements to Ray Tracing

Alternative solutions have been developed that attain more accurate sampling or that compute the image more efficiently. Cook [Cook84] proposed distributed ray tracing, a method which attempts to sample in directions proportionally to the intensity of lights. This technique can be used to model soft shadows, motion blur and depth of field effects. Heckbert and Hanrahan [Heck84] used beams instead of rays thereby significantly reducing the amount of computation at the cost of not handling refraction correctly. Amanatides [Aman84] used cones which involve propagating an origin and diameter information through the scene. This allows more accurate approximation of solid angles. As the light source spread can be determined, fuzzy shadows can be computed, dull reflections are produced and anti-aliasing becomes more efficient, since the technique inherently samples area rather than points. Pencil tracing is proposed in [Shin87]. This approach traces a group of rays, called pencils, through the environment, using a set of system matrices. The result is the generation of caustics† and more accurate shadows. A modification to the rendering equation [Kaji86] is also presented that results in a formulation that preserves energy. Glassner [Glas84] proposes a space subdivision technique that allows much more efficient computation as the number of intersections is reduced.

† Caustics are bright spots or areas in a scene, resulting from concentration of light due to multiple interreflections or refractions. According to Webster's dictionary: "caustic ... relating to or being the envelope of rays emanating from a point and reflected or refracted by a curved surface". An example of caustics are the bright areas created from light refracted through a glass of water.

2.4.2. Computational Cost

The computational cost of ray tracing is relatively high. As was evident from very early in the evolution of ray tracing programs, the cost of intersections is very large, since they involve many floating point operations. Various suggestions have been made, including the use of dedicated intersection processors and parallel methods (which will be reviewing in Section 4.3). However, for images that take on the order of a few minutes to generate using a visible-surface technique such as z-buffer or Watkins, a typical ray tracing program will take tens of minutes, or even hours. This renders the use of ray tracers for long animations dubious, especially for low-budget projects that lack expensive hardware. Ray tracing and related methods are nevertheless used commercially by those who are prepared to pay the price.

2.5. Radiosity-Based Approaches

One of the drawbacks of ray tracing and its related approaches is their inability to render diffuse environments convincingly. Although environments containing highly-specular objects catch the eye and are useful for advertising, many scenes in real life are predominantly diffuse. The more recent solutions of the radiosity-based algorithm essentially solve the global illumination problem for totally diffuse polygonal environments.

The first radiosity-based approach borrowed several techniques from thermal engineering [Gora84]. The objects in the three-dimensional scene are subdivided into small patches. All surfaces are assumed to be perfectly diffuse reflectors, or in the case of light sources, diffuse emitters. The radiosity of a surface is the light energy radiating from the surface. For an arbitrary point on the surface, light is assumed to radiate in all directions, giving rise to a hemisphere of light distribution. For a patch j of a surface, the radiosity B_j is given by the expression:

$$B_j = E_j + \rho_j \sum_{i=1}^{N} B_i F_{ij} \qquad (2)$$

The number of patches in the environment is N, E_j is the light emitted from surface j, ρ_j is the reflectivity of surface j and represents the fraction of incident light that is reflected, and F_{ij} is the form factor and represents the fraction of the energy that leaves surface j and impinges on surface i. This equation assumes perfectly diffuse surfaces that reflect equally in every direction.

After subdividing the scene into patches, the algorithm finds the matrix of *form factors* and then solves the matrix of equations to determine the radiosity B_i, of all patches. The radiosity is calculated in terms of intensity rather than energy, and at the end of the calculation it is interpolated across the screen projection of the patches to generate a smoothly shaded image (Figure 2.5).

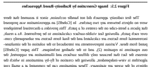


Figure 2.5: Image Generation by Radiosity-Based Approaches

The first radiosity approach did not handle occlusion, since it assumed that there were no objects between any two patches. In [Cohe85] an approximation was proposed that places a hemi-cube on the centres of a patch. This provides a discretised hemisphere over each patch, allowing fast visible-surface calculations to be performed. As a result, incremental form factors are calculated and the final form factors are subsequently computed from these. Various improvements are introduced to the solution of the simultaneous equations in equation (2), and to the shading techniques. This paper [Cohe85] stresses the fact that because only diffuse surfaces and illumination are supported, the algorithm is view-independent, allowing the creation of fly-by animations in which the objects of the scene are stationary. These animations can be computed at a small cost once the initial form factor and radiosity calculations have been performed. However, the use of such animations is questionable, as in reality interesting animations consist of objects moving; all form factors must be recomputed even if only one object moves†.

Several methods based on radiosity have been proposed more recently. Some of these will be discussed in the following section on combined methods. Of the methods that are purely radiosity-based perhaps the most interesting is the progressive refinement approach described in [Cohe88]. This approach reverses the logic of 'collecting' light and instead 'shoots' light through hemi-cubes. Each patch keeps track of the amount of 'unshot' radiosity. The picture is created by initially using an approximation based on the total energy in the environment and is then progressively refined as radiosity is shot from the most imbalanced to the less imbalanced hemi-cubes. This allows rows of the matrix

† It is only fair to note, however, that radiosity was initially developed for architectural applications. In such animations most objects are, more often than not, stationary (buildings don't walk).

of form factors to be computed, stored, used and then discarded, one row at a time, significantly lowering the storage requirements of the radiosity-based algorithm. The significant contribution is that pictures that would have otherwise been impossible to compute due to the storage requirements of the form factors now become computable. To see how important this can be, imagine a scene with 5000 polygons each subdivided into 100 patches. We immediately see that we have to deal with $500000^2 = 25 \times 10^{10}$ form factors leading to impractical storage requirements. On the other hand a single row of this matrix is only $O(10^5)$ which is feasible.

The convincing images rendered by radiosity-based approaches makes them very attractive. With the progressive refinement approach it becomes a practical algorithm for rendering diffuse environments. Special purpose hardware [Fuch89] will be able to render scenes in real-time using a radiosity-based approach. Such a system has yet to be built, however. There are also a number of limitations of the algorithm based on the fact that subdivision is patch-based. Much unnecessary computation can occur if little light reaches an area in which there are many patches requiring many form factor calculations. Related to this is the problem of choosing how to subdivide the surfaces into patches to avoid unnecessary computation for areas that due to perspective projection cover a very small portion of screen space. Storage requirements for form factors grow as the complexity of the scene increases, although this problem is alleviated in the progressive refinement solution. The pure radiosity-based algorithm's overwhelming problem still remains its inherent inability to handle specular surfaces.

2.6. Combined Methods

Ray tracing handles purely specular scenes convincingly, while radiosity-based approaches render exceptionally realistic images of diffuse environments. After the introduction of the two methods, several designs were proposed to utilise the capabilities of both to create better pictures. Although it is known that the split into specular and diffuse reflection is artificial, several approaches that attempt to combine the two techniques have been put forward.

There are a number of 'two-pass' solutions to the global illumination problem. In [Wall87] an algorithm is presented that initially performs a standard radiosity algorithm with the addition of 'virtual worlds' to model mirrors and filters. A second pass is performed that determines whether a surface has a specular component. If it does, a reflection frustum is placed on the surface and a recursive ray tracing technique is applied to the significant directions of this frustum as defined by the bidirectional reflectance function. This results in convincing specular and diffuse effects in a single scene. A second approach is presented in [Shao88]. Specular energy is computed in the perfect reflection directions of the hemi-cubes, giving accurate specular energy at the patch

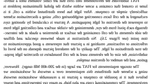


Figure 2.6: Collecting Light for Shading in FIAT

FIAT subdivides the environment into cells that are cubic volumes until a subvolume is created that contains a scene within which the global illumination problem is sufficiently simple to compute. After light has been distributed within a cell, it is exchanged at the cell faces connecting neighbouring cells, using a discretisation scheme to represent the direction of light propagation. A picture is calculated by shooting rays through each pixel from the eye, determining the visible surface and collecting light from the side elements of the cell containing the surface to determine a shade at the intersection point (see Figure 2.6). No distinction is made between specular and diffuse reflections or refractions, resulting in a picture that represents a good approximation to the true distribution of light. The quality of the rendered picture depends on the level of subdivision, the size of the elements used at the cell boundaries to discretise light propagation, and the number of direction samples.

The storage requirements for FIAT are large (in the 200–400 MB range). However, using a scheme that distributes this requirement over a network of workstations we believe that it is possible to compute a picture in a reasonable workstation network environment. The computational costs are also high, as many ray intersections and power distribution calculations have to be performed. However, we shall see in this thesis that it is possible to parallelise these calculations on tightly-coupled multiprocessors, allowing more reasonable throughput for picture computation.

Chapter 3

Hierarchical Parallelism: Architectures and Environments

3.1. Introduction

Hierarchical parallelism is the segmentation of a computational problem at several levels, with the computation for each level occurring in parallel. An example of such a hierarchically parallel environment is a network of multiprocessors. A large problem can be split into tasks, each of which is assigned to one of the multiprocessor stations. Within each station these tasks are then split into even finer pieces that are assigned to each processor. In this way a two-level hierarchy of parallelism is achieved.

To better understand the environment of hierarchical parallelism used in this thesis, we present a brief overview of parallel computer architectures. Since our target environment is a network of single-bus shared-memory multiprocessors, we discuss this specific architecture in some detail. We also present the actual machines used in the implementation. The loosely-coupled environment for parallelism is then introduced with a description of the network used to connect the multiprocessors followed by a discussion of some software issues. Additionally, we introduce the metrics used to evaluate the parallel solutions developed. We close the chapter with a first look at how hierarchical parallelism can be effectively utilised in the case of FIAT and other space-subdivision algorithms.

This chapter is an introduction to some of the basic parallel architectures and issues concerning their use. Readers familiar with current research in parallelism should not hesitate to skip this chapter.

3.2. Overview of Parallel Architectures

Various approaches to parallel computer hardware architectures exist. The most commonly used classification is that of instruction and data streams. By this classification a uniprocessor system is a Single Instruction Single Data architecture (SISD). An architecture that incorporates many processing elements that execute the same machine instruction in lockstep is classified as Single Instruction Multiple Data (SIMD). When there are many processing elements that compute independently of one another we have a Multiple Instruction Multiple Data (MIMD) configuration.

3.2.1. SIMD Architectures

One common SIMD architecture is the vector processor. Such systems contain a number of identical processing units that perform the same operation on different elements of an array or vector. A typical configuration for such a system is for the vector unit to be attached to a host uniprocessor that performs normal serial computation on the array data and at discrete points in time sends data to the vector processor.

The important concept in vector processing is that each processing unit executes the same instruction during each machine cycle. This means that the implementation of code that contains conditional statements may actually require more overall work on the vector processor than the single processor since usually both sides of a conditional must be executed.

Since the processing typically occurs on independent elements of the array in each separate processor, in most systems each unit has its own local memory which is flushed to main memory at the end of the vector operations.

A different type of SIMD architecture is the processor array model. One such system is the Connection Machine [Hill86], an array of thousands of processors executing in lockstep and connected in a special type of network configuration. Although it is difficult to map conventional serial algorithms to such architectures there have been several successful attempts to parallelise algorithms on them, resulting in significant speedup [Hill86].

Related to this type of parallelism, is the concept of pipelining. Pipelines in microprocessors consist of s stages that each execute a certain task, such as instruction fetch, instruction decode, ALU operations etc. Once the pipeline is filled with instructions, there are s tasks occurring in parallel, each stage operating on a different instruction. This allows throughput to be increased significantly, in the best case by a factor of s.

3.2.2. MIMD Systems

In MIMD systems multiple processors execute independently. This immediately raises the question of how the threads of a parallel program communicate with each other and how they synchronise. Several solutions to these problems exist, and the method applied is heavily dependent on the memory organisation of the parallel system and the way that processors communicate with each other. There are two main kinds of memory configurations in MIMD architectures. In the first, memory is distributed in some fashion among various processors, that typically each have local storage and are connected via some form of interconnection network. In the second case, memory is shared among processors and as a result requests to the same location cause problems of contention. Although we will not be discussing them here, there are several architectures that employ a combination of both kinds of memory configurations.

3.2.2.1. Distributed-Memory Systems

Distributed-memory systems employ various types of network configurations to connect processors, such as hypercubes, rings etc. In these architectures messages are the only form of data-exchange between processing elements and are therefore also used for synchronisation. Each processing element has access to local memory and some set of neighbouring processors depending on the connection network. Access to non-local memory can often be slow as it may require messages to be passed across a path of neighbouring processors to the destination processor that has the data element. For many algorithms this places a natural limit on the complexity of the solution. However, there are problems that adapt nicely to this configuration and can achieve large speedup using such a system†.

3.2.2.2. Shared-Memory Systems

Shared-memory architectures are conceptually closer in design and organisation to traditional uniprocessors than are distributed-memory multiprocessors. The main reason for this is the way that memory is manipulated from the programmer's point of view. In shared-memory architectures many processors are connected to the same block of memory, allowing access to it in the same fashion as in traditional uniprocessors. In shared memory there obviously must be some mechanism to serialise accesses to the same memory location in hardware. This is typically achieved by some arbitration mechanism that is hardware dependent.

There exist various configurations for shared-memory architectures. One common solution is to place processors and memory on the same shared bus. This is the parallel architecture used in this thesis, and therefore we will be addressing details of such

---

† Such cases are documented in the series of Tech. Reports from Caltech on the Cosmic Cube system, collected in [Fox88].

systems in the following section. Alternative configurations include crossbar switches in which each processor can simultaneously access a memory element without the saturation problems of a common bus (Figure 3.1(a)) or multistage switch networks that are cheaper than crossbar switch configurations. An example multistage network from the BBN Butterfly is shown in Figure 3.1(b) (adapted from [Rett86]). The solid boxes represent a 2 processor to 4 memory module configuration and the total picture a fully configured 8 processor to 8 memory module system. The boxes in the diagram represent switches. The memory access time is constant for crossbar switches and logarithmic for the multistage interconnect.

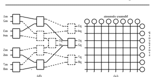


Figure 3.1: Crossbar Switch and Multistage Interconnect

3.3. Shared-Memory Single-Bus Architectures

Shared-memory single-bus multiprocessors are becoming widely available. These machines are cheaper to produce than distributed-memory architectures or shared memory designs with complex switching networks. It is also the case that they adapt easily to traditional uniprocessor environments, since in the worst case they can be used as compute servers, with serial jobs being run on separate processors.

3.3.1. Hardware Overview

Although it is difficult to present a single general architecture that covers the characteristics of all the machines in this category, it is possible to identify certain properties common to all that must be understood to allow their use in an efficient manner. The basic block diagram of such machines is presented in Figure 3.2. A number of processors are connected to a common bus to which memory modules are also attached. The processors share a common address space that is accessible to all. Because such an organisation ultimately creates increased traffic on the common bus, cache memories are typically attached to the processor modules. A second reason for using caches, shared by uniprocessor systems design, is to achieve higher speeds for memory accesses. The use of caches in multiprocessors, however, introduces the problem of maintaining a consistent view of the data shared among various processors. Hardware cache consistency algorithms are generally employed.

In addition to memory, a number of I/O subsystems may either be associated with only one processor, or exist as intelligent controllers on the bus that can handle transactions with any other unit.

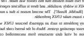


Figure 3.2: Single Bus Shared Memory Parallel System Structure

3.3.2. Software Issues

In many existing multiprocessor systems, derivatives or extended versions of UNIX are used. There are two main reasons that this choice was made. The first is the that the source code for UNIX is widely available, and there is sufficient experience in adapting it to multiprocessors [Bach84, Russ87]. The second reason is that a number of marketing and standardisation issues make UNIX preferable.

There are many problems to such an approach because UNIX was designed as an inherently uniprocessor operating system. Much of the kernel code had to be modified to avoid the corruption of vital data structures from uncontrolled concurrent accesses

[Bach84]. The largest problem, however, was overcoming the limitation of the UNIX concept of a process. UNIX processes are 'heavy-weight' in the sense that they are expensive to create and destroy. They also maintain a lot of state information that makes migration costly in a multiprocessor environment.

As an alternative, 'light-weight' threads of control are used. Threads typically share a common virtual address space and are cheap to create, destroy and reschedule. The operating system can be aware of the differences between light- and heavy-weight processes for the purposes of resource sharing [Bart88].

Programs written for shared-memory multiprocessor architectures require mutual exclusion primitives to protect critical sections of code that can potentially result in data corruption due to concurrent access. Such protection is provided by the locking mechanisms that these architectures employ. Locks can be spin-locks, in which case the processor that fails to acquire a lock busy-waits until it is available. Alternatively, a queuing system can be associated with each lock so that a process that fails to acquire a lock is placed on a queue and blocked to be awakened when the lock is free. Semaphores are often implemented this way.

Since locking occurs frequently, both in the operating system and in user programs, it is often the case that hardware support exists for synchronisation primitives. Many architectures use a separate bus for locks (SGI [Akel89], Balance [Thak88]). With such support it is often faster to use spin-locks than semaphores since the overhead of the context switch that may result after a semaphore operation can potentially be larger than the 'waste' resulting from the busy-wait. It is usually the case that spin locks are better for protecting small segments of code that are a few instructions long. Many such situations occur in operating system code.

3.3.3. Target Hardware: the Power Series and the Firefly

Two existing multiprocessors that actually implement some of the designs described above are the SGI Power Series multiprocessor and the DEC Firefly. The Power Series is a workstation product produced by SGI, while the Firefly is a research machine not commercially available. These are the two machines that we will be using in the implementation, and it is therefore interesting to acquire some knowledge of their design.

3.3.3.1. SGI Power Series

The Power Series [Akel89, Bart88] was initially intended as a faster front-end machine for the graphics pipeline hardware provided by SGI, but it has since become one of the cheaper commercial multiprocessor solutions available. The Power Series comes in a variety of configurations from 1 to 8 processors.

The system consists of two buses to which all processors and memory are connected, the Sync bus and the MPLink bus shown in Figure 3.3 (adapted from [Akel89]). The Sync bus is used to achieve hardware locking among processors and peripherals,

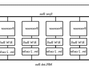


Figure 3.3: SGI Power Series Hardware Overview

while the MPLink bus is the data transfer bus on which all read\write operations take place. Each processor effectively has three levels of cache: the two internal 64K instruction and data caches, a read\write buffer and a second-level cache that attaches to the MPLink bus. The internal caches are isolated from the MPLink bus and improve system performance within the processor board. The Sync bus is very important, since it allows a large number (64K) of hardware test-and-set variables. Hardware spin-locks can be directly used by the applications programmer with system provided library routines. This means that fine grain parallelism can be achieved through the use of spin-locks with minimal system overhead.

The operating system is IRIX, a UNIX variant with enhancements for multiprocessing. In contrast with other thread approaches such as the one described below for the Firefly, IRIX maintains a relatively large overhead for task creation and deletion. After creation, the tasks are 'light-weight' as far as scheduling and migration is concerned. This overhead can be avoided by initially creating a pool of threads that are subsequently activated when necessary [Bart88]. Given this assumption, both models are essentially equivalent.

3.3.3.2. The DEC Firefly

The Firefly [Thac87] was built as a research project at DEC Systems Research Centre to prove that it was feasible to develop a multiprocessor system in a short period of time. The system consists of 4-7 VAX processors connected to a single bus with a memory module and an I/O subsystem attached to one processor, shown in Figure 3.4 (adapted from [Thac87]). The caches associated with each processor board are of about the same speed as main memory and are not intended as a speed enhancement to main-

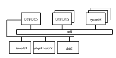


Figure 3.4: The Firefly Architecture

memory accesses, but rather as a way to offload the bus traffic from multiple requests due to the multiple processors.

The operating system is a UNIX derivative called Topaz, which is written in Modula 2+ [Rovn85], an extension of Modula 2 [Wirt85]. The system is designed to allow hundreds of light-weight threads to be active at a given time, thus requiring the creation, deletion and context switching overhead to be very small.

The report on this system [Thac87] makes a point of noting that there were many compromises in the design to achieve the goal of producing a working, usable system in a short period of time. In light of this decision, such things as more complicated designs for the I/O subsystem were knowingly left out since they would have been too time consuming to implement.

3.4. Loosely-Coupled Parallelism: Distributed Environments

Loosely-coupled parallelism is a broad term used to describe systems ranging from simple general-purpose network configurations to sophisticated distributed-memory machines with specialised processor interconnection networks. The hardware configuration and software issues we discuss below are those of the environment that we shall be using for the implementation for this thesis. We will note other possible solutions where appropriate.

3.4.1. Hardware

Loosely-coupled environments are groups of workstations connected through a local area network (LAN). These workstations can either be parallel processors or simple uniprocessor systems (Figure 3.5). The LAN used can vary in speed and technology. A common type of LAN is the Ethernet that we will be using in the implementation of this thesis. Ethernet is an old, slow network standard with a total bandwidth of 10 Mbits\sec and a maximum point-to-point bandwidth of 2-3 Mbits\sec on a network with average load. It is clear from the performance figures that any algorithm that uses such a network for large data transfers is limited by the network speed and by the inevitability of saturation.

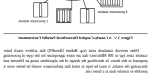


Figure 3.5: A Loosely-Coupled Hierarchical Parallel Environment

Other network standards exist (e.g. Hubnet [Ikem84]) that achieve much faster transfer rates (up to 100 Mbits\sec) that are more appropriate for the type of processing proposed in this thesis. In describing the design of the algorithms using an Ethernet and discussing the results, it must be kept in mind that performance would be better were it possible to transfer data at a faster rate.

3.4.2. Software Issues

The first layer of software support for the Ethernet can be one of the many network standards. The software used in this thesis is an implementation of the TCP/IP protocol. It is not necessary to examine this protocol's peculiarities in detail. All that is necessary to comprehend the loosely-coupled design that will be presented later (Chapter 7) is that any station can connect to any other station in a full-duplex manner. The operating system allows two modes of read/writes on these connections. Input/output can be blocking, in which case if a read or write operation cannot complete (because the data for the read is not yet available or the consumer of the write data is not yet ready), the process initiating the I/O operation is blocked. Alternatively, I/O can be non-blocking, in which case if one of the above situations occurs, the operation fails, returning an error code indicating the condition allowing the application to take appropriate action. The select system call allows processes to maintain several active read or write requests on communications channels. A process blocks on the select call and is awakened when one or more of a set of specified channels contains data to be read or that has been written to it.

Programming in this model can be performed using a message-based paradigm. These messages can be used to exchange data, or initiate an operation on a remote station. Synchronisation is critical, since relative speeds of stations and network delays are unpredictable. In Chapter 7 we will introduce a model that will allow us to develop simple solutions to the synchronisation problems that arise.

3.5. Measuring the Success of Parallel Solutions

Many ways of evaluating parallel solutions exist. Unfortunately, most of these deal with systems in which there is no hierarchy and which assume the existence of n homogeneous processors. In this section we present the metrics that we will use to evaluate the tightly-coupled part of the proposed design. These can also be applied to a loosely-coupled scheme, but differences in communications speed at each level should be taken into account when evaluating the results.

3.5.1. Evaluation Measures for Tightly Coupled Parallelism

The goal of parallel computation is speedup. If the serial running time of the program is T_1 and the parallel running time of the algorithm on n processors is T_n then speedup can be defined as:

$$S = \frac{T_1}{T_n}$$

This is a good measure of the gain obtained from parallelism.

The ideal goal for parallelising a program with serial running time T, using an architecture with n processors is to achieve a running time of T/n. However, there are some inherent limitations related both to the specific application in question and the architecture of parallel systems that make this ideal unachievable.

A key concept is that of the *degree of parallelism* for a given problem. This is the maximum number of separable components of an algorithm that can be computed in parallel, which is the limit of the benefit one can expect from parallelising a given algorithm.

Apart from speedup, another measure of the performance of a parallel program is processor utilisation. Processor utilisation, sometimes called efficiency, is defined as:

$$PU = \frac{S}{n} \times 100 = \frac{T_1}{nT_n} \times 100$$

This number expresses the percentage of time each processor is busy, on the average.

What is ignored from this model is the structure of the parallel solution. When a serial algorithm is parallelised, it is normally the case that part of the computation is still executed serially. This is usually due to the fact that some parts of the algorithm are not parallelisable. If we call this portion of the execution time T_{sp}, we can use the following modified definition of speedup:

$$S' = \frac{T_1 - T_{sp}}{T_n - T_{sp}}$$

This, of course, is not 'true' speedup, but it does convey a measure of how successful the specific implementation of a parallel algorithm is compared to the theoretically maximum speedup obtainable by the scheme in question. This number gives an indication of the cost of the overhead (synchronisation, locking, waiting) involved in the parallel solution.

Analogously we can define scaled processor utilisation PU' as

$$PU' = \frac{S'}{n} \times 100$$

This gives a rough approximation of the percentage of the execution time wasted in the overhead of synchronisation, locking and waiting.

3.6. Hierarchical Parallelism for Space-Subdivision and FIAT

Space-subdivision algorithms are good candidates for hierarchical segmentation, since the subdivision technique naturally suggests a form of hierarchy. An obvious way to segment such a problem is to split the hierarchical data structure into elements (subvolumes) that are distributed to the stations at the highest level of the hierarchy. On each station the subvolumes can then be dealt with in parallel, if such parallelism is possible. Two issues arise in such a segmentation. The first is that the assignment algorithm must minimise the communication between stations. The second issue is that of parallelism on a station. Although application specific, the problem being solved is more amenable to parallelism if the computations inside a subvolume do not have to be executed in strict order. Many algorithms exhibit such properties, such as pattern recognition, most rendering algorithms, and some simulation techniques.

For the case of global illumination and FIAT in particular, the unit of volume that can be dealt with in a tightly-coupled fashion is the cell. Groups of cells can then be sent to various stations, and light update information can be exchanged via the network between the stations containing the neighbouring cells.

A successful tightly-coupled parallel solution for a cell in FIAT utilises all the incoming light information that exists in local shared memory to perform light distribution within the cell. Shading of the pixels affected by the interaction of light with objects in the cell can also be performed in parallel. As we shall see in Chapter 6, although the nature of light as a linear operator allows reorganisation of computation and therefore parallelism, such an attempt requires certain modifications of the original serial algorithm.

For the full hierarchical approach, the goal is dual: speedup of the computation and segmentation of the memory, so that more storage can used to render higher-quality pictures. Many synchronisation issues arise that are satisfactorily solved by the design presented in Chapter 7.

Chapter 4

Parallelism in Computer Graphics

4.1. Introduction

Parallel approaches to computer graphics algorithms have been considered by researchers as viable solutions to the high cost of computing realistic images. For visible-surface determination or scan-conversion techniques, parallelism is used effectively in the context of pipelining the various activities. Several successful hardware designs allow animations of complex scenes to be rendered in real time using these methods. In ray tracing, the overwhelming cost of ray intersections was immediately identified, thus making the intersection calculations prime candidates for parallelism. For radiosity-based approaches, parallelism is used to speed up the computation of form factors.

There are various ways in which we can categorise the efforts for parallelism in graphics. The first is by class of rendering algorithm. The second distinction is whether the approach is a software or a hardware solution. The third distinction that is of interest to research in this area is whether the solution attempts to split the problem in image space or object space or both. By image space we mean the domain defined by a two-dimensional discrete coordinate system analogous to physical device coordinates such as pixel addresses. By object space we mean a three-dimensional domain defined by a continuous world coordinate system. One final observation that is relevant to this discussion is whether the specific approach actually resulted in an implementation. Apart from the specialised graphics hardware constructions, few of the results reported actually describe an existing system. Many are simply paper designs, or report simulation results. These simulation results are very vague in most cases, and most projects did not actually end in implementing their simulated systems. The concepts are important but it seems that the effort required to actually implement a parallel design is significant, and although some of the real systems referred to here may seem to be trivial, the amount of work required to actually produce a working parallel solution deserves significant credit. We have attempted to present a collection of various approaches in Table 4.1. The tabulation includes the reference and an indicator for all of the above categories. This table is by no means complete, although an effort has been made to cover several different types of techniques.

There have been previous surveys categorising parallelism in computer graphics. Two interesting reviews are by Crow [Crow88b] and by Whitman [Whit89]. The former investigates several approaches including many of the hardware designs that have been constructed, but concentrates more on the potential for speedup for scan-conversion techniques. The latter concentrates on several efforts for visible-surface determination techniques and points out weaknesses in each approach. Both are useful sources of previous work in the area.

For the rest of this chapter we will review some of the previous computer graphics attempts for parallelism. We first examine methods developed for scan-conversion/visible-surface algorithms, then overview designs for ray tracing and radiosity-based approaches. The final two sections examine real hardware designs separated into research or very specialised hardware and more widely available commercial systems.

Categorisation of Parallel Computer Graphics Solutions				
Algorithm	Reference	Subdivision	Status	Description
Scan Conversion	[Appa88]	Image Space	H/W Impl.	The Stellar Graphics Supercomputer.
	[Chan81]	Object Space	Proposal	Static subdivision of 3d space
	[Clar82]	NA	H/W Impl.	Geometry Engine, h/w graphics pipeline, real-time scan-conversion.
	[Crow88a]	Object Space	S/W Impl.	A triangle/processor scan-converter on Connection Machine.
	[Deme85]	Image Space	H/W Impl.	SLAM processor; VLSI design.
	[Dyer87]	Image Space	S/W Impl.	Z-buffer vectorised for interpolation and shading.
	[Fium83]	Image Space	S/W & H/W Simulation	Rapid instruction for depth-comparison. Parallel anti-aliasing.
	[Fuch77]	Image Space	H/W Proposal	Image segments/processor assignment
	[Fuch82]	Image Space	H/W Impl.	Initial pixel-planes, processor per pixel design.
	[Fuss82]	Image Space	H/W Impl. (inc.†)	Multiple triangle processors.
	[Gura88]	Image Space	H/W Impl.	SAGE systolic array VLSI design.
	[Kapl79]	Image Space	Proposal	Analysis of parallelism for z-buffer and scan-conversion.
	[Levi84]	Colour Space	H/W Impl.	Chap PIXAR design for parallel RGBα.
	[Niim84]	Image Space	H/W Simulation	Two level parallelism scan-lines/pixels.
	[Park80]	Image Space	Proposal	Splitter trees for z-buffer.
	[Torb87]	Image Space	H/W impl	Parallel configurable VLSI design.
	[Wein81]	Object Space	Proposal	"Object-processors" scan-convert in parallel.

Table 4.1: Parallel Approaches in Computer Graphics

† (inc.) indicates that the implementation was incomplete at publication time or that it is unknown whether or not the system was actually implemented.

Table 4.1. (continued)

Categorisation of Parallel Computer Graphics Solutions				
Algorithm	Reference	Subdivision	Status	Description
Ray Tracing	[Clea86]	Object Space	S/W Impl	Faster ray intersections due to object space subdivision.
	[Dela88]	Object Space	S/W Impl	Ray tracing with sub-space traversal on the Connection machine.
	[Dipp84]	Object Space	Simulation	Adaptive space subdivision, dynamic adjustment to load.
	[Gaud88]	Ray & Image Space	H/W Impl (inc.)	3-stage Pipeline, multi-pipeline configuration.
	[Koba87]	Object Space	Simulation	Intersection processor array interconnect with shading processors. Adaptive octree.
	[Koba88]	Object Space	Simulation	Hierarchical system. Clusters of intersection processors load balanced statically, dispatch work locally.
	[Nish83]	Image Space	H/W Impl.	The LINKS-1 64 node parallel processor with configurable pipelines.
	[Plun85]	Ray Space	S/W Impl.	Vectorised ray tracing.
	[Sche88]	Object Space	S/W Simulation	Dynamic load balancing for tree representation of object space.
Radiosity	[Chen89]	Object Space	S/W Impl.	Form factor calculations on workstation network as part of progressive refinement.
	[Sill89a]	Object Space	S/W Impl.	Form factor calculations for extended radiosity over workstation network.
	[Reck90]	Object Space	S/W Impl.	Form factor calculations over workstation network. Evaluation.
Multiple Algorithms	[Fuch89]	Image & Object Space	H/W Impl. (inc.)	Pixel Planes 5. Processor/pixel area with parallel graphics processors on a ring.
	[Potm89]	Image Space	H/W Impl.	4-connected CPUS. Pipeline of pipe nodes. Fast DSP chips.

4.2. Parallelism for Scan-Conversion Algorithms

Some of the first proposals for parallelism in computer graphics were paper designs for hardware architectures. Examples of these are described in [Fuch77], [Park80] and [Wein81]. All three segment image space and split the work among processing elements in some fashion. The first paper [Fuch77] is interesting for historical reasons, since it is a very early proposal for segmentation of image space so that each processing element deals with a set of pixels. The approach described in [Park80] is a parallel z-buffer using a tree splitting mechanism that does not balance the load among processors very well, and the design in [Wein81] uses each processing element as an 'object processor' that performs scan-conversion in parallel, but limits the system to one object per processor. Analyses of expected results are presented. Another interesting paper hardware design is presented in [Niim84], which introduces a two level hierarchy of processing elements that are assigned scan-lines and pixel spans within scan-lines respectively. Microcode simulation suggests that this technique, if implemented, could produce real time animation sequences.

An interesting analysis investigating the parameters that affect parallelising scan-conversion algorithms is presented in [Kapl79]. Memory references and depth-comparisons are measured with respect to the method selected to segment the image. It is observed that loss of coherence limits the gains of parallelism as the granularity of segmentation becomes finer.

In [Fium83] an extension to the NYU Ultracomputer is proposed that takes advantage of instructions that propagate operations directed at the same memory location through the switching network. The *RepMin* instruction is proposed that replaces a shared memory location with the minimum of those accessing it concurrently. This is used to implement a parallel z-buffer. Scan line spans are assigned to different processing elements. This was a paper design that would require architectural enhancements to be implemented. A software simulation was used to generate pictures.

All of the above designs are image-space subdivision techniques. Chan [Chan81] proposes an object-space subdivision approach to scan conversion. Simulation results presented in the paper also demonstrate that the loss of coherence from splitting three-dimensional space results in a performance penalty.

Two more recent designs attempt parallelism for a specific architecture. In [Crow88b] a parallel scan-conversion algorithm is proposed for the Connection Machine architecture. Although no real timing results are reported, it is apparent that performance gains are achieved by using the connection network for depth-sorting and by assigning a polygon per node. In [Dyer87] a vectorised z-buffer is presented, and results from an actual implementation on a Convex C-1 are reported. Interpolation for the various values (colour, position, etc.) of edge pairs is vectorised by reorganising loops. Shading calculations and deferred depth computation are also vectorised.

4.3. Parallel Ray Tracing Algorithms

Ray tracing is simple to parallelise in image space. Assuming a common object data base one can simply assign a pixel to each processor and allow the computation to proceed independently. Nevertheless, several schemes have been reported in the literature that describe mainly object space parallel designs.

An interesting adaptive subdivision scheme is described in [Dipp84]. This algorithm uses generalised cubes to subdivide object space. The subdivision scheme in itself results in a significant speedup of the ray tracing algorithm, since it allows a solution to be calculated with fewer intersections. The most attractive element of this approach is that the generalised cube areas dynamically adapt depending on the load in the area for which they are responsible. This is achieved by moving the corners of the cubes. Simulation test results are presented in [Dipp84] predicting linear speedup, although an implementation of this technique was never reported.

Hierarchical data structures are often used to represent a subdivision of space so that more intelligent methods can be used to intersect rays with objects. In [Sche88] such a data structure is proposed. Once the structure is available, it is split into two levels. The section of the structure in the upper level is available to all processors, while the section below is distributed to the different processors. A priority queuing system is used to achieve load balancing in the system. The simulation results are promising and show that the load is balanced in the system satisfactorily. Similar to the above approach but using less ambitious load balancing schemes are the systems described in [Koba87] and [Koba88]. The simulations show that it is possible to gain significant speedup using a network of intersection processors even if the network bandwidth is low.

A vector processor is used to process rays in the design presented in [Plun85]. The scheme used is that rays are queued until there is enough work to fill the pipeline of the vector processor. The intersection calculations are then executed by the vector processor in parallel. The timing results presented show significant speedup. It is interesting to note that the intersection calculations no longer dominate the total execution time when the vector processor is used, since a significant amount of time is spent traversing a CSG tree structure. By vectorising this part of the code an even better speedup was achieved, compared to the approach that simply parallelised the ray intersections.

Other ray tracing schemes listed in Table 4.1 are space-subdivision techniques described in [Clea86], and [Dela88], and also a 3-stage pipeline hardware scheme presented in [Gaud88].

4.4. Parallelism in Radiosity-Based Algorithms

An overwhelming portion of the compuation for radiosity-based approaches lies in the form factor calculations that are performed using either analytic methods or the incremental hemi-cube technique. Two methods developed independently ([Chen89, Reck90], and [Sill89a]) utilise a network of computers to compute form factors in parallel for the progressive refinement method. Reportedly the largest problem in these techniques is network saturation when the number of patches is large.

4.5. Research or Specialised Parallel Architectures

The architectures discussed here are mainly results of research efforts. However, the Pixel Machine and the PIXAR Image Computer are both commercial products, but not widely distributed.

Pixel planes ([Fuch82], [Fuch89]) is a SIMD custom VLSI system. The basic concept behind the design is the assignment of one processing element per pixel. Each PE has local memory and a multiplier tree that allows it to perform linear interpolation. The reported timing results are tens of thousands of Gouraud shaded polygons per second making this system very attractive for generating animated sequences of moderate quality. The latest version of this system [Fuch89] is enhanced with more memory per pixel, quadratic evaluators and a set of graphics processors that allow front end processing. These are all connected on a ring network allowing the components to cooperate effectively. The system is split up into renderers that are each assigned to a 128x128 area of the screen. This allows for better processor utilisation than in the initial processor-per-pixel design†. This system, though still not completed, will reportedly allow 30 msec frame generation for reasonably complex scenes, and allow even more sophisticated algorithms such as radiosity to produce animations in real time. Other VLSI efforts listed in Table 4.1 are the SLAM processor [Deme85] that is built from chips that can perform scan-conversion for n scan lines in $2n + 2$ cycles, the SAGE [Gara88] system that is a systolic array that handles a scan line at a time and a design presented in [Fuss82] that consists of parallel pipelines of specialised triangle processors. Table 4.1 also contains a design by Torborg that can expand arbitrarily and adapt to various graphics configurations [Torb87].

The Pixel Machine [Potm89] is a multiprocessor with two components. The first is a pipeline of pipe nodes and the second is an array of pixel nodes that are organised in a 4-connected manner. The nodes are very fast digital signal processor chips which feature

† If there are areas of the screen that are empty, the processors assigned to these pixels are idle.

special support for floating-point operations and some mathematical functions commonly found in graphics algorithms such as square roots and dot products. The frame buffer can be interleaved among the pixel nodes allowing a natural image segmentation. Scan-conversion, volume rendering and ray tracing algorithms have all been implemented in the same architecture and programming environment with marked success.

The LINKS-1 [Nish83] architecture is a 64-processor system that connects to a host computer. Neighbouring processing units share a local memory that can be switched between them. This allows for a setup of variable length pipelines that can be dynamically changed in time. The initial hardware layout includes a frame buffer system that can collect information from each processing element. The original paper proposes the use of this architecture for high-performance ray tracing graphics.

Chap [Levi84] is a 4-channel SIMD processor that allows parallel processing for the RGB and α (transparency) elements of a pixel. There are special instructions and different memory-access modes that allow higher-level conventional programming constructs such as if-then-else to be implemented. This specialised architecture was used in the PIXAR Image Computer.

4.6. Widely Available Commercial Graphics Hardware

Several manufacturers produce special-purpose hardware for computer graphics, many of which incorporate some degree of parallelism in their design. These include Silicon Graphics' Geometry Engine and Ardent that has introduced the Titan graphics architecture. We will briefly describe these two systems since they present different approaches to providing graphics functionality in hardware. Other vendors that have produced graphics support in hardware include Stellar [Apga88], Hewlett-Packard and Apollo.

The Geometry Engine System [Clar82] is a pipeline of twelve identical chips called geometry engines. This pipeline is separated into 4 chips for the matrix manipulation subsystem, 6 for clipping and 2 for scaling operations. These systems were reasonably priced, and allowed real-time generation of thousands of polygons per second on a desktop workstation. The evolution of the Geometry Engine is the GTX series of Iris workstations [Akel89]. Enhanced Geometry Engines are responsible for the various parts of the graphics pipeline. At the end of the pipeline each of five span processors has responsibility for a section of the screen and performs updates in parallel. Z-buffering is supported in hardware with special memory. This system achieves a display rate of 100,000 Gouraud shaded quadrilaterals per second.

The Ardent Titan [Died88] takes a somewhat different approach, essentially to the same task. This system performs the transformation and clipping on a parallel host processor that has been enhanced with vector processing units to allow these operations to be faster. Scan-conversion and interpolation are performed in the specialised graphics hardware that also supports a z-buffer. The performance of the Titan is 200,000 Gouraud shaded triangles per second, or approximately half that for quadrilaterals, comparable to the IRIS.

4.7. Discussion

The techniques for parallelism in computer graphics are heavily dependent on the specific algorithm that they attempt to parallelise. Different portions of the computation are expensive for each rendering algorithm. As a result, different parts of the computation are suitable for parallelism in different algorithms. Nevertheless, there are some useful general concepts in some of the approaches presented.

For image space parallel schemes, it is usually the case that subregions of the pixels are assigned to different processors either by segmentation of the screen [Fuch77], or by task decomposition based on the object structures [Fuss82, Torb87, Fium83]. The complexities of the visible-surface determination algorithms allow such techniques to reap significant benefit from parallelism, as these computations are mostly independent for each pixel. For FIAT, we will be utilising the concept of image-space parallelism in its simplest form, by distributing the pixels among the processors in the final shading phase.

The object space approaches are closer to the scheme developed in this thesis. A key concept in most object space techniques is space subdivision. The subdivision of the three-dimensional scene can be static [Chan81] or adaptive [Dipp84, Koba87]. In FIAT we use an adaptive space subdivision technique by assigning sub-volumes to (possibly parallel) stations. The concepts presented in [Dipp84], in which the sub-volumes adapt according to the load, can be applied to the methods we will be developing.

One final concept that is central in our approach is that of *hierarchical task decomposition*. One such approach is the idea of splitting the data structure representing space in a hierarchical fashion [Sche88], allowing some of the structure to be accessible to all processors and some only to specific nodes. The design in [Niim84] also has a two-level hierarchy of scan-lines and pixel spans. As introduced briefly in Section 4.6, the scheme developed for FIAT creates a two-level hierarchy. Groups of cells are assigned to parallel stations which in turn segment the computation inside the cell executing each segment on a different processor.

The computationally expensive nature of computer graphics algorithms has given rise to the large number of techniques discussed in this chapter. As we show in Chapter 6 where the tightly-coupled parallel solution is developed, and in Chapter 7 where the full hierarchical model is presented, FIAT is a good candidate for successful parallelism.

Chapter 5

FIAT: A Light Driven Global Illumination Algorithm

5.1. Introduction

The creation of realistic computer graphics imagery requires simulation of the effects of global illumination. In simple terms, global illumination effects are the visual cues resulting from light that is *everywhere*, a consequence of the complex and diverse interaction between light and the environment. A natural method to generate such pictures is to compute the way that light, starting from the light sources, is reflected off or refracted through various objects and disseminates into the environment. After computing this distribution of light, one can specify a viewpoint and a window, and take a snapshot of the light intensity captured in this view, as if taking a photograph with a camera. FIAT strives to produce this result by intelligent approximation. Using adaptive subdivision techniques, and simulating light propagation with a discrete approximation, a solution that closely resembles global illumination becomes feasible.

In this chapter we will review the basic FIAT algorithm, the adaptive space subdivision technique employed, the method used to propagate and exchange light between the subvolumes resulting from the subdivision, and the computation that models the interaction of light with objects. We then discuss a progressive rendering technique using a light update buffer. Throughout this discussion we stress the parts of the algorithm that are relevant to parallelism and we briefly discuss storage and computation requirements. This chapter is meant as an overview of the algorithm, not a thorough examination of it; the algorithm is introduced in [Four90], and discussed in more detail in [Ouel89].

5.2. The Basic Algorithm

For a given closed volume in space, we can deal with the distribution of light inside it independently of the rest of the environment, if we know how much light is emitted inside that volume and how much light is entering it. The only assumption needed for this to be true is that all light operations are linear:

$$O(aP_1 + bP_2) = aO(P_1) + bO(P_2)$$

where $O(P)$ is a light operator operating on light power P. The consequences of this are that light propagation and distribution in the environment can occur in any order, with the same cumulative result.

This paradigm requires that it be simple to calculate the distribution of light in a given volume. To facilitate such a requirement an adaptive subdivision technique is used that splits volumes recursively until they become simple enough to be dealt with. The light power pertinent to a volume is modelled as the power P_E emitted inside the cell, the power P_A absorbed inside the cell, the power P_I incoming to the cell and finally the power P_O outgoing from the cell. Based on these quantities we define the power imbalance ΔP for a cell:

$$\Delta P = P_E + P_I - P_A - P_O$$

A positive ΔP represents power that either is emitted inside the cell or has reached the subvolume from its neighbours. When a subvolume is processed, incoming and emitted light is either absorbed or converted into outgoing power, and as a result ΔP becomes equal to 0, since all the power has been accounted for. The only case in which ΔP will become non-zero again is if power from a neighbouring subvolume is propagated back into this volume. After balancing a subvolume, the light information can be used to shade a view of this environment defined by a viewpoint and a window.

After dealing with all subvolumes so that the highest ΔP is less than a certain threshold and the total power in the system is less than a second, global, threshold, an approximation to the global illumination of the environment modelled will have been achieved.

To determine when the total power in the system is below the predefined global threshold, the sum of all the power existing in the system is maintained at all times. Initially it is equal to all the power emitted from the light sources. At a given stage in the execution of FIAT it represents the remaining power that has yet to be 'dealt with'. The sum of this total plus the absorbed and discarded power is equal to the initial power value. We will call this number the global power sum $GPower$.

5.3. Discrete Representation of Light and Propagation

To approximate global illumination, FIAT uses a discrete representation for light and its transfer through space. By using such an approach, analytical form factor calculations between surface elements can be avoided, and provided that enough information is maintained in this discrete representation a satisfactory approximation to global illumination can be achieved.

The discrete representation used in FIAT has two components. The first is an adaptive subdivision technique which creates spatial volumes that are sufficiently small to allow simple computation of the interaction of light with the environment, and the second is the directional representation of light flow.

5.3.1. An Adaptive Space Subdivision Technique

Space in FIAT is split into cubic cells. Initially the environment is enclosed in a cell encompassing all the objects in the scene. When light reaches the cell, FIAT determines whether the cell can be dealt with, and, if not, subdivides it into eight equal-sized subcubes. Subdivision is performed until a predefined maximum level is reached. Cells considered simple enough to be dealt with, without further subdivision, are the following: empty cells that contain no objects, cells that contain either a single object or a single light source, and cells that can no longer be subdivided as they have reached the maximum subdivision level.

After the initial cell is subdivided, some method must be employed to choose which cell will be treated next. The heuristic used chooses the cell with the highest power imbalance ΔP as defined previously. To find this cell efficiently a *heap of cells* is maintained, ordered by decreasing power imbalance.

This scheme has the following two appealing properties. First, only space that light reaches is considered, since cells that receive no light power are ignored. Secondly the most 'difficult' regions are likely to be dealt with first, so that when a progressive rendering method is used, the most significant secondary lighting effects appear early in the evolution of the final image.

An important final point concerning subdivision is the fact that the maximum subdivision level is a user tunable parameter. By choosing this parameter appropriately a first approximation that will give a coarse indication of global illumination in a scene can be computed quickly using less storage. When the final, high-quality image is desired the subdivision level can be increased, allowing for more accurate approximation of light flow.

5.3.2. A Discrete Representation of Light Flow

To represent the flow of light between two cells, a buffer of the flux crossing the adjoining faces is maintained. A more accurate representation of light traversing a boundary is achieved by splitting each face into small side elements, which we term sexels. Sexels are of fixed size. To determine this fixed size, three parameters are used: the maximum level of subdivision n, a user defined variable x such that x^2 sexels cover a face of the smallest possible cell, and the length of an edge of the initial cell L. Then the length of the edge of the smallest possible cell is $L/2^n$. The fixed sexel size is:

$$FSS = \frac{L}{x \cdot 2^n}$$

If, for example, $n = 1$, $x = 2$ and is $L = 8$, then the fixed sexel size will be $FSS = 8/(2 \cdot 2^1) = 2$. This is illustrated in Figure 5.1 (adapted from [Ouel89]).

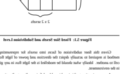


Figure 5.1: Fixed Size Sexels and Subdivision Level

Given this finer subdivision of faces into sexels for representing light flow, a method is required to actually depict the direction and power of light flowing from one cell to another. Ideally what should be modelled is the light flux in any given solid angle in the environment.

An approximation to this is achieved by defining a discrete direction space that covers the sphere of directions around the centre of a sexel. A cube is placed around a given sexel S, so that the plane defined by the sexel bisects the cube parallel to one face (see Figure 5.2). On each face of this cube a grid of size GS by GS is placed. From the centre of this cube, which is also the centre of the sexel, we can draw lines to the centres of the squares defined by the grid placed on the cube faces. The directions of these lines define the set of representative directions. This set of directions is used to sample the flow of light across the sexel face.

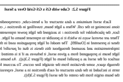


Figure 5.2: Cube with GS x GS Grid Over a Sexel

Each sexel maintains a data structure of a hemi-cube, corresponding to the directions incoming to the cell. When a light beam, travelling in direction *i*, crosses the sexel face, the placeholder for direction *i* is assigned the light power represented by the beam. There are several ways in which it is possible to represent beams of light. A discussion of these is presented in [Ouel89]. The model of light propagation used in the current FIAT implementation and assumed throughout this thesis is that of *parallel beams*. When a light beam is propagated from a given sexel, it behaves as if it is emitted from the whole surface of the sexel as a parallel beam of light (Figure 5.3(a)). As beams are propagated through space, some deviation from the straight line occurs. To account for this, an error term is included in the data structure of each direction in a sexel, expressing the offset of the beam origin from the centre of the sexel (Figure 5.3(b)).

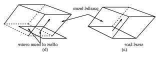


Figure 5.3: Parallel Beams and Offsets

5.4. Light Sources and Light Propagation

FIAT supports several light source types, including point-light, extended-area diffuse and directional-light sources. Light flux emitted from these is shone onto the initial cell. The continuum of light from the sources is transformed into power samples assigned to the discrete beams on the sexels of the outer cells. For extended area light sources the solid angle of light is projected onto the cell faces and subsequently assigned to 'preferred directions', that is the directions closest to the directions of the flux of light from the source, in the sexel beam data structures.

Once light is assigned to the sexels, it is propagated through space. The simplest cases are when either the cell is empty or, for cells that contain objects, the directions of light in which the corresponding beams do not intersect any objects. For these cases, straight-line propagation is used. When a beam starting from one sexel crosses an empty space and gets assigned to another sexel in a corresponding face opposite to that of its origin, it is assigned to the same direction as the one it started from. As it is usually the case that the beam centre will not coincide with the centre of the destination sexel, the error term is used to ensure that the beam does not stray far from a straight line.

5.5. Interaction with Objects in Space

When a light beam emanating from a sexel face hits an object, it can be reflected or refracted in some manner, depending on the surface properties of the object. In contrast to previous rendering algorithms, FIAT takes a unified view of reflection phenomena. Thus surfaces with either specular, diffuse or a mixture of both reflective behaviours are treated uniformly. We will discuss how one light beam impinging on a surface is dealt with, a process that is repeated for every beam coming into the cell.

For a given beam, a ray is shot in the representative direction to which it is assigned. If it hits a surface, light must be distributed around this intersection point. To achieve this distribution, a *local illumination model* is used to model the behaviour of light as it interacts with this surface. A variation of the Blinn [Blin77] model has been adapted to achieve this goal. A cube of directions is centred at the intersection point. For simplicity, this cube always has the same orientation in space. For every direction defined by the cube, a power calculation is performed based on the surface properties of the object in question in order to determine how much light is reflected or refracted in that direction. In a manner similar to propagation through empty space, light is propagated from the intersection point to the closest face, and assigned to the outgoing sexel beams of the neighbouring cell (see Figure 5.4). This scheme results in a satisfactory approximation of the distribution of light in a complex environment, containing objects with different surface characteristics.

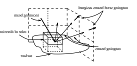


Figure 5.4: Distribution of Light Around an Intersection Point

The memory requirements for such a distribution are large, since it is necessary to store a large number of beams especially if we are dealing with area light sources and diffuse surfaces. As we shall see below, however, a large amount of memory is discarded in the process of balancing the power of the cells in a given scene. The amount of processing required to deal with a cell depends on the number of incoming beams and on the surface properties. Many intersection calculations must be performed to determine the interaction of light with the objects in the scene. A significant amount of processing time is also spent calculating the appropriate face, sexel, direction and error offset of the destination beam that is reflected or refracted from a surface.

The approach described above is that of the current FIAT implementation and the one we will be dealing with in this thesis. There are, however, some problems of under-sampling that result from applying this technique. At the time of writing, new methods that include the concept of spread of a given beam are being developed that will solve many of the sampling problems. Also in consideration are better reconstruction techniques for recovering light information from the discrete sampling space.

5.6. Generating a Picture from Sexel Information

From a purely theoretical point of view, when the FIAT algorithm has finished computing a scene, light power has been absorbed or has escaped into the environment outside the scene, and the distribution of light through the scene has been correctly calculated. Generating a picture from this information is simple. For every pixel a visible point is determined and the light that shines on this point from the incoming beams is collected. Using a local illumination model, the shade of the pixel can then be determined (see Figure 5.5).

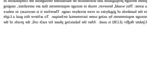


Figure 5.5: Collecting Light to Shade a Pixel

The solution just described would require that all the sexel beam information computed through propagation and distribution be maintained throughout the computation of a scene. This would, however, result in storage requirements that are unrealistic, ranging in the hundreds of gigabytes or even terabyte range. Therefore it is necessary to reduce storage requirements by using some incremental technique. To achieve this goal a Light Update Buffer (LUB) is used. After the balancing phase for each cell, the pixels of the

LUB that are associated with visible points within that cell are shaded. After this, all the sexel information of this cell can be discarded since it is no longer needed. This cell will only be dealt with again as a result of light information reentering the cell from its neighbours. In this case the additional shading information is added to the LUB.

5.7. Conclusion

FIAT is still an ongoing project and therefore it is difficult to pass judgement on its success. Pictures generated to date show that effects previously unattainable by computer graphics rendering algorithms can be produced by FIAT. These include light caustics created from the concentration of light due to multiple reflections and colour bleeding from diffuse surfaces in the same scene as secondary lighting effects from specular reflectors. These results are encouraging and show that a decent approximation to global illumination can be successfully produced.

The storage requirements of FIAT are large, as are the running times both for distribution and shading. This fact is the primary motivation for the parallel solutions developed in this thesis. By tapping the power of parallel and distributed environments we will be able to significantly reduce the computation time spent to generate FIAT images. By using the large memory space existing across a network of workstations higher subdivision levels will become feasible and the creation of pictures of previously unattainable quality will be possible.

Chapter 6

A Tightly-Coupled Parallel Solution in a Cell

6.1. The Goals for a Tightly-Coupled Parallel Solution

The bulk of execution time in FIAT is consumed by the processing related to dealing with a cell. The light beams coming into the chosen cell are shot into the environment to be either propagated through empty space and assigned to sexels in neighbouring cells, or to be distributed in various directions as a result of interaction with objects inside the cell. Subsequently, the section of the Light Update Buffer (LUB) related to this cell is shaded, thus refining the image that is being incrementally created. This computation is a prime candidate for shared-memory, tightly-coupled parallelism, due to the inherent locality of the computation in the cell, and its large expense.

In this chapter we present the parallel algorithms developed for the computations inside a cell. In this effort we strive to meet the following four goals:

- Maximum parallelism: to perform as much computation as possible in parallel.
- Load balancing: to distribute the computation evenly among all processors.
- Contention-free parallelism: to find a solution that minimises locking, and allows the processors to compute as independently as possible.
- Independence from the number of processors: develop a solution that can adapt to different numbers of tightly-coupled processors. The range of processors in which we are interested is that of existing common-bus shared-memory architectures, typically from 2 to 10.

In this chapter we first analyse the serial algorithm (using three test scenes) to determine the computationally expensive sections. We investigate the structure of the algorithm and the data dependencies involved to decide how much of the computation in these sections can be parallelised, thus maximising the work performed in parallel. To successfully balance the load, we investigate the various ways in which the computation can be segmented into independent components so that the work is evenly distributed. Private buffering and reorganisation of the order of computation proved to be necessary, both to balance the load evenly and to avoid locking. We close with the results of the implementation and a summary of the approach.

6.2. Analysis of the Serial Algorithm

The computation in the cell can be split into two components. The first, which we will call balancing, involves the distribution of incoming light as a result of interaction with objects in the cell and its propagation from the beams on the incoming sexels of one face to the outgoing sexels on another face. The second component, shading, involves updating the section of the LUB associated with the cell. This is achieved by determining the visible surface for each pixel and shading the pixel based on the incoming light that impinges on this visible point. These two processes are independent and can occur in any order, the only requirement is that the incoming light information be maintained throughout shading.

6.2.1. The Serial Algorithm

Figure 6.1 depicts the high-level pseudo-code of the balancing (a) and shading algorithms (b) for the serial version of FIAT.

```
/* Balancing routine */

BalanceCell (cell)
{
    for each face
      for each sexel
        for each beam
          /* does the beam hit anything ? */
          P = RayIntersect
          if something is intersected at point P
            DistribLight(P)
          else
            PropLight(beam)
}

DistribLight(hit_point)
{
    for cube of directions around hit_point
      /* calculate power reflected/refracted in direction */
      Power = PowerCalc()
      Sexel = FindDestSexel()
      assign Power to Sexel
}

PropagateLight(beam)
{
    Sexel = FindDestSexel
    assign the power of beam to same direction on Sexel
}
```

Figure 6.1: (a) The serial algorithm for balancing

In the balancing routine, every beam on each sexel is shot into the cell. If something is intersected, light is distributed, in which case power calculations are performed for the cube of directions. Otherwise, light is propagated to the opposite face.

```
/* shading code */
ShadeCell (cell)
{
    determine xmin,xmax ymin,ymax of LUB corresponding to cell
    for x = xmin to xmax
      for y = ymin to ymax
        /* intersect ray from eye through centre of pixel (x,y) */
        P = RayIntersect()
        if P is on an object inside cell
          /* find sexels whose beams impinge on P */
          while (sexel = FindHitSexel())
            /* calculate power sexel shines on P, shading (x,y) */
            ShineBeam(sexel, P, x, y)
}
```

Figure 6.1: (b) The serial algorithm for shading

For shading, the algorithm loops through the pixels corresponding to the cell, collecting power from the incoming beams to determine the shade.

6.2.2. Analysing the Behaviour of FIAT in a Cell

From the pseudo-code presented above, the inner loops that are potentially the most time consuming parts of the computation can be identified. A first inspection shows that for balancing much time is spent shooting rays into the cell to determine if the beam in question intersects anything. When distribution occurs, complex calculations are required to determine the outgoing power for every outgoing direction around this intersection point. Locating the outgoing sexel that this power will be assigned to also contributes significantly to the total execution time. Since these calculations are expensive and the number of directions sampled is large, it can immediately be seen that a considerable amount of time will be spent performing distribution. Most of the time in shading is spent in the power calculations to determine the light impinging on the visible point. The determination of the contributing beams is also costly.

These are the intuitive estimations that can be made simply by examining the structure of the serial algorithm. However, as the interaction of light inside a cell is an extremely complicated process, it is necessary to confirm this intuition by using standard profiling tools. To produce such profiling data we ran FIAT on a few test scenes, selected to exhibit a variety of lighting and surface conditions. The computation for these scenes results in execution patterns that are representative of many aspects of FIAT, making the scenes suitable for evaluating the parallel solutions developed.

figure rle/s1.halft space .3i figure rle/s2.halft

Figure 6.2(a) Figure 6.2(b)

figure rle/s3.halft

Figure 6.2(c)

6.2.3. Selecting Test Cases

There are various modalities to FIAT with respect to the type of light sources and the type of surfaces that exist in each scene, each resulting in different behaviour of the algorithm. It is therefore desirable to test as many of these modalities as possible. We chose the following three scenes:

Scene 1

A shiny ashtray (60% specular, 40% diffuse) lit by a directional light source that is at a 45 degree angle in front of the ashtray. The model consists of 600 triangles, so that many cells contain more than one object. The bright spots in the middle of the ashtray are caustics due to secondary illumination (Figure 6.2(a)).

Scene 2

A room that has 3 walls, a red left wall, a blue right wall, a grey back wall, and a grey ceiling and floor. There are 12 triangles in this scene. The surfaces in this scene are mostly diffuse, and absorb some of light falling on them. The scene is lit by a extended area light source that is in front of the room. This scene shows colour bleeding, similar to that shown by radiosity, on the ceiling from the two side walls (Figure 6.2(b)).

Scene 3

The same as Scene 2, except that the floor is now shiny (60% specular, 40% diffuse) and acts as a mirror. This scene shows the (relatively) sharp reflection of the two walls in the floor mirror in a scene containing perfectly diffuse objects (Figure 6.2(c)).

For scenes 2 and 3 the FIAT parameters used are a maximum subdivision level of 2 and the number of sexels on the edge of the smallest cell is 2. Because the memory requirements for these scenes are excessive, it is necessary to use subsets for test runs on machines that do not have enough real memory. This is to allow measurements to be made of the execution time without the overhead of paging. For scenes 2 and 3 the subset scenes have the floor and the back wall removed, and for scene 3 the ceiling is shiny instead of the floor that is no longer present. We refer to these scenes as scenes 2a and 3a respectively.

6.2.4. Interpreting the Profiling Results

Tables 6.1-6.3 present the time spent in the most expensive functions of FIAT for the three test scenes broken down into balancing and shading. We have used scenes 2a and 3a as the machines with the appropriate profiling software have limited memory. These tables include all functions with a significant contribution to the total execution time. The number associated with each function is the percentage of the total execution time spent in the function itself and its descendants. Therefore, the first entry is the parent function that calls all the other (children) functions listed below. The 'Self' entry is the time spent in the parent function itself while the 'Other' entry is the percentage spent by other functions called by the parent in question.

Profile for Scene 1.			
Balancing		Shading	
Function Name	% of time	Function Name	% of time
BalanceCell	53.6	ShadeCell	8.6
DistribLight	49.2	FindHitSexel	3.8
RayIntersect	3.0	Self	2.5
Other	1.4	ShineBeam	1.9
		Other	.4
DistribLight	49.2		
Self	23.5		
PowerCalc	15.5		
FindBestSexel	9.9		
Other	.3		

Table 6.1: Profiling Results for Scene 1

Profile For Scene 2a.			
Balancing		Shading	
Function Name	% of time	Function Name	% of time
BalanceCell	57.5	ShadeCell	36.7
DistribLight	54.7	FindHitSexel	14.6
PropLight	1.1	Self	9.6
Other	1.0	ShineBeam	8.8
RayIntersect	.7	Other	3.5
		RayIntersect	.2
DistribLight	54.7	ShineBeam	8.8
Self	25.3	Self	3.3
FindBestSexel	26.3	PowerCalc	3.5
PowerCalc	2.9	Other	2.0
Other	0.2		

Table 6.2: Profiling Results for Scene 2a

Profile For Scene 3a.			
Balancing		Shading	
Function Name	% of time	Function Name	% of time
BalanceCell	64.3	ShadeCell	26.1
DistribLight	62.1	FindHitSexel	11.0
PropLight	1.0	Self	7.2
Other	1.2	ShineBeam	7.4
		Other	.5
DistribLight	62.1	ShineBeam	7.4
FindBestSexel	27.6	Self	2.7
Self	24.3	PowerCalc	3.2
PowerCalc	9.9	Other	1.5
Other	.3		

Table 6.3: Profiling Results for Scene 3a

6.2.5. Analysing the Profiling Results

What is immediately evident from the profiling results is the fact that overall more time is spent balancing than shading†. In scene 1, less shading occurs since light tends to concentrate, affecting a smaller number of pixels. This is due to the specular property of the surfaces and the fact that the light source is directional. The differences between 2a and 3a are due to the fact that in 2a all surfaces are diffuse, while in 3a a specular surface also exists. In a totally diffuse environment, every beam intersecting a surface results in a calculation for every direction around the intersection point. For a specular surface, there are many beams in which some of the calculations can be avoided, since it can be

† We have omitted in these results the time spent in occlusion calculations inside a single cell containing more than one object, since they are not relevant to the solution. It is evident from the tables that a large proportion of the total time is spent on these for scene 1 (more than 35%) as it includes more such cells, while the percentage is much less for scenes 2a and 3a.

determined that the specific direction will not have sufficient outgoing power.

For balancing, the 'Self' component of `DistribLight` is effectively spent performing power calculations of various kinds, making the total time spent for power calculations in balancing to be 39%, 28.2% and 34.2% of the total execution time respectively for each scene. A significant amount of time (9.9%, 26.3%, 27.6%) is spent determining which sexel has to be assigned the outgoing power for a given direction when distributing over an intersection point. For shading, the time is split almost evenly between finding the sexels whose beams impinge on the visible surface point (3.8%, 14.6%, 11.0%) performed by `FindHitSexel` and the power calculations to determine the shade (4.4%, 18.4%, 14.6%), performed in `ShineBeam` and the 'Self' portion of `ShadeCell`.

6.2.6. Linearity of Light and Reorganising the Computation

An important factor affecting all attempts to parallelise the computation in a cell is the linearity assumption for light effects (see Chapter 5 and [Four90]). This is a reasonable assumption for most lighting phenomena. In most cases the perceptual effect of two lighting effects is independent of the order in which they are applied. Nevertheless, there are some cases in which the power absorbed by a material does actually affect reflective or refractive properties, but we will not be concerning ourselves with these.

The linearity assumption guarantees that the order in which we distribute light is irrelevant. Therefore, we can perform the propagation or distribution of light beams in any arbitrary sequence, yielding the same results. As we shall see this is very important since it gives us the ability to reorganise a large amount of the computation in ways that are more convenient for parallelism.

6.3. Selecting an Approach to Parallelism

To select an approach to parallelism for an existing, complex serial algorithm, it is necessary first to identify the *grain of parallelism*, that is which subtasks of the algorithm can be performed in parallel; second to propose an *assignment method* which will distribute tasks among processors; finally to determine a *synchronisation mechanism* that will allow these parallel tasks to communicate and synchronise so that correct execution is guaranteed.

To find the subtasks that can be executed in parallel, an abstract examination of the algorithm must be undertaken that effectively determines the degree of parallelism inherent to each part of the solution. For this purpose it is necessary to understand all data dependencies that arise due to the task segmentation. In shared memory environments, it is easy to overlook the fact that certain data are shared by various processes, and, therefore, can be potentially updated concurrently, leading to unpredictable results. To deal with such data dependencies once they are identified, several solutions exist. Locks may be employed to protect the shared variable, requiring that all updates be performed only after a lock has been acquired. Alternatively the computation may be reorganised to allow independent computation, possibly involving a data-merging stage at the end.

Depending on the specific application in question, it may be possible to achieve good load-balancing by employing an intelligent initial allocation of tasks. Alternatively, adaptive techniques can be used to change the distribution of load during the execution. It must be noted, however, that there is always a performance penalty to be paid if such adaptive strategies are used, and hence they must be approached with caution and evaluated carefully.

Synchronisation in this environment is relatively complex. Various methods exist, including semaphores, locks and barriers. The development of involved and complicated synchronisation techniques relates to the issues of load balancing; careful evaluation is required, and the final decision must be based on performance evaluation and timing results.

6.3.1. Parallel Balancing of Cells

For each incoming beam in a sexel, a ray intersection must be performed with the objects in the cell to determine whether the beam will distribute its power because it hit some object, or whether it will simply propagate though space. The intersection operation of an incoming beam is independent of any other intersection of the same form, since no data is written apart from a structure which contains the hit point information if such a point exists. This information is typically kept in a variable that exists privately in the process's stack therefore causing no problem of memory contention. As a result there is no obstacle to performing these intersections in parallel.

Distribution involves finding the destination sexel and calculating the power for each direction. As seen from the profiling results this is a large proportion of the execution time, and therefore parallel distribution will result in significant performance improvement. There are two tasks that can be used as the grain of parallelism: the computation around an intersection point and the computation for one direction in the cube around the point.

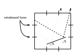


Figure 6.3: Different Intersection Points Update Same Beam

In the first scheme different intersection points are assigned to different processors, and each processor performs all the computation for the full set of directions. This suffers from a problem of possible contention since two different intersection points can result in the concurrent update of the same sexel in the same direction. This is shown in Figure 6.3, which demonstrates distribution around two different intersection points, P_1 and P_2.

The second possible approach is to process the directions around a single intersection point in parallel, by assigning different directions to different processors. At first glance this seems like a reasonable strategy. However there is one hidden detail that is not shown in the pseudo-code of `DistribLight` (Figure 6.1(a)) that leads to a subtle problem. To guarantee termination, and to be as close to physical reality as possible, FIAT requires that the total power leaving an intersection point is no larger than the power entering it. In other words if the power impinging at point P from a given direction was P_{inp} and the local illumination model assigns P_{out_i} power to be outgoing in direction i (Figure 6.4), we wish the total outgoing power P_{out} to be less than or equal to P_{inp}:

$$P_{out} = \sum_{i=0}^{d-1} P_{out_i} \le P_{inp}$$

where d is the number of directions in the sphere around P. The inequality accounts for the power absorption. To ensure this, the initial serial implementation of FIAT multiplied P_{out_i} by a scaling factor S defined as follows

$$S = \frac{P_{inp}}{\sum_{i=0}^{d-1} P_{out_i}} \quad (1)$$

and, as a result :

$$P_{out} = \sum_{i=0}^{n-1} SP_{out_i} = S\sum_{i=0}^{n-1} P_{out_i} = \frac{P_{inp}}{\sum_{i=0}^{n-1} P_{out_i}} \sum_{i=0}^{n-1} P_{out_i} = P_{inp}$$

thus exactly equating P_{inp} to P_{out} (we have purposely omitted the details that relate to the fractions of diffuse and specular reflection and of transmission, and also the fraction corresponding to absorption, as the cumulative effect is the same). This normalisation function guarantees conservation of energy, albeit in a somewhat arbitrary manner. The final value assigned to the destination sexel is SP_{out_i}.

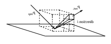


Figure 6.4: Outgoing Power in Direction i

The problem with parallelism is that since we compute all P_{out_i} in parallel, we cannot get the final value for each direction that has to be scaled by S, until all processors have finished computing P_{out_i} for all directions assigned to them. This requires barrier synchronisation with attendant reduction in speedup, because the time to process all directions is not the same and because it is difficult to predict for a single intersection point which directions will involve more work than others. Hence, if this scheme for parallelism is to be used, an alternative method for dealing with normalisation should be developed.

6.3.2. Initial Parallel Balancing Attempts

Before presenting a solution that satisfies the goals put forward in Section 6.1, we will briefly discuss some first-order approaches to parallelism in the cell.

Without changing the structure of the initial algorithm, we can statically divide the processing of either faces, sexels or beams among processors. In such a scheme, the main loop of `BalanceCell()` is executed on every processor, but only a subset of the faces, sexels or beams respectively are dealt with on each. Processing of faces and sexels are too coarse to be suitable for tightly coupled parallelism. Such a scheme would not scale well (there are 6 faces, and in the worst case there are 6 sexels in a cell), and also does not provide good load balancing since it is conceivable that only one face or only a small number of sexels contain all the incoming light. The beam (direction) level, is more promising as there are usually many directions per sexel†. Light is distributed relatively evenly among the directions in all the cubes above intersection points and all the sexels in a scene. However, dealing with beams in parallel suffers from the contention problem of concurrent updates to the same directions on the same sexels, described in the previous section. Three ways of addressing this problem are possible without significantly altering the overall structure of the initial serial algorithm.

Locking. This solution requires that locks be acquired before performing a write on a beam in a destination sexel. There can be one global lock for all sexels, in which case computation degrades as the processors spend most of their time waiting for locks to be free. Alternatively a lock per face or sexel could be used. Unfortunately, the memory requirements for allocating a lock per sexel are prohibitive and render the solution useless in the case of high subdivision. A lock per face will also cause waiting because a large part of the compuation is serialised.

Private buffers. A second approach is to perform all sexel updates in a private buffer, to acquire a lock when the buffer is full and to write the updates to the real sexels. This again suffers from the problem of lock contention, as the lock will have to be retained for comparatively long periods of time, when the copying is done. Because the buffer and the real sexel information occupy distant locations in memory, this solution can strain the memory hardware.

Pooled Buffers. A third solution is to employ a two pass approach. Each sexel is preassigned to a processor in some static manner. If there are n processors, each processor maintains n buffers. In the first pass each processor propagates and distributes light, and when a sexel must be updated, it inserts the update information in the buffer corresponding to the processor assigned to this sexel (Figure 6.5 (a)). In the second pass, each processor runs through the buffers containing updates for the sexels assigned to it on all the processors, performing the real updates (Figure 6.5(b)). This solution avoids the problem of frequently accessing distant memory, since the set of sexels that each processor accesses is smaller. During the first pass only local accesses are required. This solution yields relatively good results but is limited by the overhead of maintaining the separate buffers and by the need to synchronise often, since the number of sexel update operations is very large (hundreds of thousands per cell balancing operation), thus causing buffers to fill relatively often.

From the above discussion we see that simple solutions which do not alter the original structure of the serial algorithm are limited due to synchronisation, memory access or locking constraints. The solution finally adopted allows these problems to be circumvented so that satisfactory speedup is achieved.

---

† A typical instance of FIAT has 192 directions per sexel.

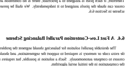


Figure 6.5: Pooled Buffer Scheme

6.3.3. Parallel Shading

Shading presents much less difficulty to parallelise successfully than balancing. This is because the only data being written in the shading process are the LUB pixels. Therefore, if each pixel is assigned to a processor, there is no contention and each processor can shade the pixels assigned to it independently. We shall be presenting such a scheme in Section 6.6.

6.4. A Fast Low-Contention Parallel Balancing Scheme

The desirable parallel solution for balancing should segment the problem so that little extra code or memory is required to support the segmentation, and should allow processors to compute autonomously. Such a solution is possible, but requires reorganising the computation in the initial serial algorithm.

Both propagation and distribution of light around an intersection point are based on a set of directions as discussed in Chapter 5. In the case of propagation, a beam starting from a source sexel updates a beam of a destination sexel on a different face that is associated with the same direction as the original beam (see Figure 6.6(a)). In the case of distribution around an intersection point P, the power outgoing in direction i is assigned to a sexel in direction i (Figure 6.6(b)). These two observations lead to a natural task decomposition scheme for parallelism:

> *Each direction is assigned to a different processor, making this processor responsible for all updates in this direction on any sexel, as a result of either propagation or distribution.*

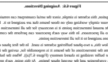


Figure 6.6: Assigning Directions.

For propagation, this scheme is simple, since the serial counterpart can remain intact with each processor simply working only on those beams that are assigned to it. For distribution of light around intersection points it is necessary for the all intersection points to be available to all processors. In this way each processor can perform the computation for the directions assigned to it for all intersection points.

To allow this, a two-phase buffering scheme is used. In the first phase each processor performs the intersections of the beams it is responsible for, saving the intersections in a private buffer residing in shared memory (Figure 6.7(a)). When the buffers fill, the processors synchronise and the second phase begins. At this point, each processor runs through the buffers on all the processors and for every intersection performs the power calculations only for the directions assigned to it and updates these directions on the relevant sexels (Figure 6.7(b)). The line thickness in Figure 6.7 indicates which processor is accessing each buffer.

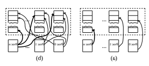


Figure 6.7: (a) Phase 1 of the Parallel Solution (b) Phase 2

This scheme suffers from the normalisation problem described earlier. If the same normalisation scheme is used as in the serial case, all processors will have to synchronise at every intersection point with a barrier to use the total outgoing power in the calculation of the scaling factor S, thus resulting in severe performance degradation. To achieve the same goal of conserving energy, without having to compute the sum of outgoing power for each intersection point, we use an upper bound to the amount of power that can possibly leave a point P on a given surface if incoming power P_{inp} impinges on P. This upper bound was developed for other purposes in [Ouel89], but is sufficient to overcome this problem, since it provides correct normalisation while eliminating the need to synchronise. Assuming a Blinn-like reflection model (see Chapter 2) for a surface with constant $n > 0$, the upper bound of outgoing power due to specular reflection over a hemisphere is:

$$P_u = \int_0^{2\pi}\int_0^{\pi/2} \cos^n(\theta)\,d\theta\,d\phi = 2\pi\int_0^{\pi/2} \cos^n(\theta)\,d\theta$$

We can use this in place of $\sum_{i=1}^{k-1} P_{out_i}$ in equation (1), giving us a revised value of S:

$$S = \frac{P_{inp}}{P_u} \qquad (2)$$

P_u can be precomputed once for each surface in the scene using numerical integration, and stored with the surface description. To perform normalisation each process needs only to apply equation (2), eliminating the need for calculating P_{out_i}, and therefore the need for synchronisation.

This scheme is a satisfactory solution, but as with all approximations it does have some drawbacks. The first is the fact that we are normalising with an upper bound, and in most cases we will therefore be unnecessarily dampening the outgoing power at an intersection point, creating a dimmer picture. The second problem is that due to the discrete nature of the direction samples around an intersection point, if n is very large it can be the case that all the power distributed from this highly reflective surface is reflected in only one sampled direction. Since the algorithm may use just one sample for the whole solid angle represented by this direction, this high power may appear to be emanating from the whole angle while in reality it should be emitted only from a fraction of the real angle represented by the discrete angle. This can create a slight infringement of the power conservation rule, since the approximation 'creates' energy. Although the amount of energy 'creation' is negligible in most practical situations, a remedy such as finer sampling of the discrete angles for large n may be desirable. An alternative is to clamp based on the sampling rate, in a way similar to that described in [Aman88].

Figure 6.8 depicts the code that is executed on each processor.

```
ParallelBalanceCell()
{
    for each face
        for each sexel
            for beams in directions assigned to this processor
                intersect beam with objects
                if something is intersected at point P {
                    BufferIntersection(P)
                    if ( BufferFull ) {
                        WaitForOthers()
                        BufDistribute()
                    }
                }
                else
                    PropagateLight(beam)
}

BufDistribute()
{
    for each processor's buffer
        for each intersection in buffer
            for each direction assigned to this processor
                calculate power reflected/refracted in direction
                normalise using P_s
                assign power to beam in corresponding sexel
}
```

Figure 6.8: Parallel Balancing Scheme

The function `WaitForOthers()` is the routine that implements the synchronisation at the barrier. To determine which beams have been assigned to the specific processor, we use the processor id pid, $1 \le pid \le n$, and assign all directions that have index i, $1 \le i \le d$ to this processor, such that with $k = 1, 2, \ldots,\ i = k \times pid$.

6.4.1. Goals Achieved by the Parallel Approach

The solution just described achieves the goals set out at the beginning of this chapter in a satisfactory manner.

- All expensive portions of the computation are performed in parallel (ray intersections with the objects, the power calculations and the calculations to find the destination sexels used to distribute around an intersection point).

- The scheme has a satisfactory 'natural load balancing' property, since typically light is distributed in a large number of directions. This results in all processors having a comparable amount of work to do, since the load distribution is tied to the distribution of directions.
- There is very little contention in this solution. A beam in a sexel can only be accessed by one specific processor, making locking unnecessary. However, since sexels are allocated on demand, it is necessary to lock in the case that the sexel is not yet allocated. To avoid this locking a preallocation scheme can be used for the sexel structure. However, this results in a penalty of extra memory requirements. The solution also has the benefit that the only synchronisation point occurs when the intersection buffer fills. At this point, the processors have to synchronise to ensure reading of correct data by all processors, since the buffers are overwritten after distribution.
- The solution is theoretically independent of the number of processors, with an upper bound equal to the number of directions. However the benefit of parallelism would quickly be overcome by the overhead of maintaining and synchronising a large number of processors. Due to the memory-intensive nature of the application, bus saturation from memory requests quickly affects performance, as we shall see from the timing results (Section 6.7).

From the above we see that from the point of view of design this scheme has the desirable properties that are required from the parallel balancing scheme. As we shall see in the section on timing results (Section 6.7) the solution performs satisfactorily on a real parallel system.

6.5. A Parallel Approach to Shading

From Figure 6.1(b) we can see that parallelising shading is a straightforward task. Each pixel can be assigned to a different processor that then performs the ray intersection and the power calculations when collecting light from the relevant sexels in parallel. If we have n processors the parallel shading algorithm executed on processor i in parallel is shown in Figure 6.9.

```
/* shading code */
ParShadeCell ( i )
processor-id i
{
    for y = ymin to ymax
        x = xmin + i
        while ( x < xmax ) {
            intersect ray from eye through centre of pixel (x,y)
            if object hit inside cell {
                collect light from all relevant sexel beams
                shade pixel (x,y)
            }
            x += n
        }
}
```

Figure 6.9: Parallel Shading Scheme

For every scan line each processor with processor-id i shades pixels $i + jn$ where n is the number of processors and $a \leq j \leq b$ such that $a = \min k : xmin \leq (i + kn)$ and $b = \max k : (i + kn) \leq xmax$.

6.5.1. Goals Achieved by the Parallel Shading Scheme

Although the shading scheme is relatively simple, it achieves the goals put forward:

- The computationally expensive part of the shading lies in the collection of light, which requires a power calculation and a determination of the relevant sexels and in the intersection of the ray from the eye. All are performed in parallel.
- Load balancing depends on the values of $xmin, xmax, ymin, ymax$. The best case for the assignment described above, is when all scan lines have a length that is divisible by the number of processors. For this case, assuming that each pixel requires equal amount of time t to be processed, each processor does $((xmax - xmin) \bmod n) \times t \times (ymax - ymin)/n$ work. The worst case will occur if n is large and the projected cell covers a vertical line one pixel thick. Then each of the $n-1$ processors is idle for $(ymax - ymin) \times t$ time resulting in a total waste of $(n-1) \times (ymax - ymin) \times t$.
- This scheme requires no locking or synchronisation for the duration of the shading computation. Each processor is statically preassigned its work, and does not need to write any memory that another processor will potentially access.
- The parallel shading approach we have proposed can be used theoretically with as many processors as there are pixels in a region of the LUB related to a cell. However, at high subdivision levels, the number of pixels may be small therefore resulting in many idle processors. If the number of scan lines is not divisible by the number of processors, a performance penalty is paid. For small numbers of processors (2-8) these problems are not very important, since the waiting or idle time is small.

The timing results in Section 6.7 confirm the above claims, since they show significant speedup on real multiprocessor systems.

6.6. Software Engineering Issues in the Tightly-Coupled Solution

Designing and implementing a parallel scheme for a program as large and complex as FIAT is not an easy task. Apart from the problems related to selecting a suitable and effective parallel approach, there are several problems of seemingly smaller importance that have to be resolved before the parallel solution can actually be run successfully on a real parallel machine. In some situations the serial implementation benefits from the strategies employed for parallelism.

6.6.1. Problems with Parallelising a Large Serial Application

The first problem encountered when parallelising a large complex program built for a uniprocessor is that many functions that one wishes to execute in parallel write global variables. In this case these accesses must be put in a critical region. A similar problem results from the use of module-wide ('static' in C) global data structures that suffer from the same problem as several instances of the same module may be executing concurrently.

An example of the former that occurred in the case of FIAT were the global variables that kept track of the total power discarded and absorbed. When more than one instance of the balancing routine ran concurrently there was no guarantee of the atomicity of the update operation. The solution chosen was to maintain subtotals, one per processor of these sums, and combine them at the end of each cell operation.

The latter type of problem occurred in the ray intersection code which relies on a voxel-traversal grid. This grid was implemented as a global variable for the relevant module. To allow for concurrent grid-traversal, separate copies of the were made for each processor.

In both these cases the extra memory required is bound by the (presumably small) number of processors in the system. Moreover, the expense of the combining computation is also small, while the gain from avoiding the overhead of locking is worthwhile.

6.6.2. Serial Algorithm Improvement from Parallelism

The parallel algorithm was tested initially for various test cases that fit in the real memory of the systems used. As soon as subdivision levels in FIAT are raised, the memory requirement quickly becomes much larger than the available real memory, therefore incurring the overhead of paging. Surprisingly, comparisons with the original algorithm appeared to be giving the parallel version speedup 2-2.5 times the number of processors; for example, on a 4 processor system the speedup was 9.8. It quickly became clear that the reorganisation of the computation by buffering intersection points and separating the direction space into sections equal to the number of processors provided better memory access patterns. By simulating the parallelism in the uniprocessor case the serial version achieves much better performance in the case when paging overhead is significant.

6.7. Timing Results

Tables 6.4-6.8 show the timing results for the three selected test scenes. Also shown is speedup and processor utilisation (efficiency), and scaled speedup and scaled processor utilisation as defined in Chapter 3. These timing results are from runs on an IRIS 4D/120S, using up to 4 processors and 32 MB of memory.

Timing results for Scene 1							
# Procs	Tot. Time	Ser. Time	S	PU	S'	PU'	Size (MB)
1	1:07:54.8	-	1.00	100%	1.00	100%	8.4
2	39:06.3	32.4	1.74	87.0%	1.75	87.5%	8.5
3	26:25.9	32.4	2.57	85.7%	2.60	86.7%	8.6
4	22:27.6	32.4	3.02	75.5%	3.07	76.8%	8.7

Table 6.4: Timing for Scene 1

Timing results for Scene 2							
# Procs	Tot. Time	Ser. Time	S	PU	S'	PU'	Size (MB)
1	1:37:13.8	-	1.00	100%	1.00	100%	22.2
2	53:29.9	14.9	1.82	91.0%	1.82	91.0%	22.3
3	36:12.5	14.9	2.69	89.7%	2.70	90.0%	22.4
4	30:15.1	14.9	3.21	80.3%	3.23	80.8%	22.6

Table 6.5: Timing for Scene 2.

Timing results for Scene 3							
# Procs	Tot. Time	Ser. Time	S	PU	S'	PU'	Size (MB)
1	2:00:11.8	-	1.00	100%	1.00	100%	22.4
2	1:05:51.1	13.7	1.83	91.5%	1.83	91.5%	22.5
3	45:50.6	13.7	2.62	87.3%	2.63	87.7%	22.3
4	37:54.9	13.7	3.17	79.3%	3.18	79.5%	22.7

Table 6.6: Timing for Scene 3.

Timing results for Scene 2a							
# Procs	Tot. Time	Ser. Time	S	PU	S'	PU'	Size (MB)
1	24:58.3	-	1.00	100%	1.00	100%	14.2
2	13:38.9	9.5	1.83	91.5%	1.84	92.0%	14.3
3	9:33.3	9.5	2.61	87.0%	2.64	88.0%	14.4
4	7:42.9	9.5	3.24	81.0%	3.28	82.0%	14.5

Table 6.7: Timing for Scene 2a.

Timing results for Scene 3a							
# Procs	Tot. Time	Ser. Time	S	PU	S'	PU'	Size (MB)
1	34:17.4	-	1.00	100%	1.00	100%	14.7
2	18:37.9	9.7	1.84	92.0%	1.85	92.5%	14.8
3	12:48.8	9.7	2.68	89.3%	2.70	90.0%	14.9
4	10:19.7	9.7	3.32	83.0%	3.36	84.0%	15.0

Table 6.8: Timing for Scene 3a.

The serial time† reported in the above results is the time spent reading in the scene, which is obviously larger for the 600 triangles in scene 1. Also included in this figure is the time spent in the octree related functions and the time spent propagating through empty cells. The serial time is the average of the reported times spent in serial mode for each run that differed by a maximum of 0.5 seconds. The speedup and efficiency achieved are at satisfactory levels, as they offer significant benefit in reducing the execution time. There are two factors affecting speedup that can be noted from the results.

Light source and surface types

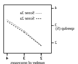
The performance of the solution depends on the type of light sources and the type of surfaces existing in the scene. The results for scene 1 are somewhat worse overall, since the light source is directional and the surfaces are specular. This means that a small number of directions have more power than others, resulting in uneven load among processors. More time is spent waiting at the barriers and processor utilisation decreases.

Large memory requirements and hardware

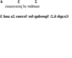
We see that for runs with larger memory, such as scenes 2 and 3, processor utilisation degrades faster than the cases with smaller memory requirements (scenes 2a and 3a). Since FIAT is very memory intensive, having more processors accessing the single

† The 'non-parallelisable' portion of the computation defined as T_{np} in Chapter 3.

The following graphs present speedup for 2 to 4 processors. Graph 6.1 shows the speedup for the scenes 1, 2 and 3 and Graph 6.2 for scenes 2a and 3a. We see from these that the speedup for the small scenes (2a, 3a) is better and that there is little degradation as the number of processors grow.

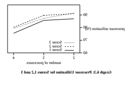


Graph 6.1: Speedup for Scenes 1, 2 and 3

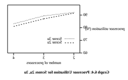


Graph 6.2: Speedup for Scenes 2a and 3a

The following two graphs show processor utilisation for the two sets of scenes. Again scenes 1,2 and 3 have larger memory requirements (Graph 6.3) than scenes 2a and 3a in Graph 6.4. We notice from these graphs, which use the same scale for easier comparison, that the penalty for each extra processor in the large memory case is 6-10%, while for the scenes with low memory requirements the penalty is less, varying from 2-5%. This means that if the memory requirements are small, the solution can scale to a larger number of processors.



Graph 6.3: Processor Utilisation for Scenes 1,2 and 3



Graph 6.4: Processor Utilisation for Scenes 2a, 3a

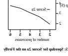
6.7.1. Results on the DEC Firefly

The implementation of FIAT is portable across UNIX systems since the machine-dependent functions are all in one source file. This design made it possible to transfer the parallel implementation to the Fireflies with relative ease, allowing the measurement of execution times using 1 to 6 processors. Table 6.9 show the timing for scene 2a.

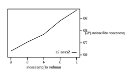
Timing results for Scene 2a			
# Procs	Tot. Time	S	PU
1	2:45:51	-	-
2	1:25:39	1.94	97.0
3	1:02:33	2.65	88.3
4	53:52	3.08	77.0
5	46:57	3.53	70.6
6	43:49	3.79	63.2

Table 6.9: Timing for Scene 2a on the Firefly

In the graphs 6.5 and 6.6 we plot speedup and processor utilisation for the above runs. For runs of 1 to 4 processors the speedup is close to that of the IRIS. However we see that speedup degrades as we further increase the number of processors. It is not clear whether this result would be similar on an IRIS, since the memory architecture is different and since the excessive memory usage is the primary cause of degradation.



Graph 6.5: Speedup for Scene 2a on the Firefly



Graph 6.6: Processor Utilisation for Scene 2a on the Firefly

6.8. Alternative Schemes

The paradigm of parallelism that we have used is a barrier-synchronisation model. Although in our implementation the parallel solution actually does give good speedup, it is the case that by design all barrier synchronisation schemes suffer from waiting since in real parallel MIMD systems there is no guarantee on how long a task will actually take on a given processor. This problem is even more evident in the multi-tasking/multi-user environments employed by the parallel machines we used, since the load on a given processor can change because of other jobs running at the same time.

An alternative model for parallelism is the use of dispatching queues that contain work to be done. These tasks are dispatched by the queue handler to workers that become idle. We describe two ways to implement this paradigm in a parallel system.

The first is to use a locking (or monitor-based) queue mechanism. In this case, a producer/consumer model is developed with access to the queue of work being in a critical section. If it is possible to produce sufficiently large tasks that can be queued and then consume these at a fast rate, with only a small portion of each process's time spent in the dispatch monitor, this scheme can be successful. For an application as complex as FIAT, in which a significant amount of memory is required for these tasks to be queued, and for which much extra code would have to be included to allow such a solution, it is questionable if the effort required for such a scheme would actually prove worthwhile.

The second approach (due to [Boot89]) is to have a dedicated processor (administrator [Gent81]) that maintains a queue of work and has the sole responsibility of maintaining the queue pointers. The dedicated processor busy waits while waiting for incoming work or dispatch. In this scheme one processor is effectively 'wasted' and the overhead of maintaining the queues would add some delay also. In comparison to our scheme, this approach becomes attractive when the number of processors becomes larger than 6 or 7, as in this case wasting one processor would result in approximately the same speedup. This approach becomes much more attractive for systems with around 30 processors or more.

6.9. Summary

In the context of a hierarchical parallel scheme, the best choice for tightly-coupled parallelism in FIAT is the computation in a cell. The information for balancing and shading exists in shared memory, and the computation required consists of many expensive, separable components that are independent. To design, implement and evaluate the tightly-coupled parallel solution described in this chapter, the following steps were taken:

- Initially the algorithm was analysed to identify the expensive portions of the computation, for both balancing and shading. Results of profiling tools were used for this task.
- Once the potential candidates for parallelism were selected based on the analysis, various alternative task subdivision schemes were considered. Data dependencies, and computation ordering that ensured correct execution had to be considered in this evaluation process. As a result of this process it was seen that straightforward solutions to balancing were inadequate, due to the complexity and data interdependencies involved.
- After reaching the above conclusion, a two-phase, low-contention solution for balancing was developed. This scheme assigns the full responsibility of subsets of the direction space to each processor, thus guaranteeing independence between processes. In the first phase, intersection points of the incoming beams with the environment are buffered. In the second phase, each processor accesses all these buffered points dealing only with those directions that are its responsibility. For shading, each pixel is assigned to a different processor so that it can then compute the shade independently with no locking, since the only data written is the pixel.
- To evaluate the proposed solution, three test scenes were used that tested several modalities in the behaviour of FIAT. The timing results as well as the speedup and processor utilisation were reported and graphed. Processor utilisation for the IRIS multiprocessor machine varies from 77%-84% using 4 processors and from 87-92% using 2 processors. For the Firefly the results were somewhat worse as the processor utilisation varied from 63.2% for 6 processors and reached 97% for 2 processors, indicating that the solution suffers from bus saturation when the number of processors increases.

Overall, the solution proposed yields satisfactory results in terms of speedup and allows time-consuming FIAT pictures to be rendered significantly faster. It allows existing resources to be utilised effectively, and is an exploration of the applicability of parallelism to solving large problems, especially in computer graphics. Together with the loosely-coupled distributed scheme described in the next chapter, this tightly-coupled parallel solution allows global illumination to be a more practical approach to computer graphics image synthesis.

Chapter 7

Loosely-Coupled Parallelism for Groups of Cells

7.1. Loosely-Coupled Parallelism and the Hierarchical Solution

To achieve maximum benefit from a network of parallel and serial computers, a design must be developed that will allow FIAT to be segmented in a loosely-coupled fashion. In such a configuration, multiple instances of the tightly-coupled parallel solution can execute concurrently on various stations, thus constituting a two-level hierarchy of parallel computation.

Both levels of parallelism can offer benefits. Parallel execution at the station level results in speedup of the computation, as was shown in Chapter 6. A successful segmentation technique that distributes cells across stations will allow FIAT to use a larger virtual address space, permitting higher levels of subdivision and therefore higher quality pictures, as well as more real memory, reducing the overhead of paging. An additional goal for the loosely-coupled scheme is speedup of the computation, as different stations will be working on separate segments of the scene. All sexel updates occurring between cells that are on different stations have to be performed through the network. The bandwidth necessary for such communication is high. The cost of communication is a prime factor in such an environment, and therefore has to be carefully considered in the design of the distributed algorithm.

In the following sections, we present a high-level description of the algorithm developed for distributing FIAT across a network of workstations, some of which are multiprocessors. We then discuss in some detail techniques employed to solve some of the problems encountered. A pseudo-code description of the algorithm is used to summarise the solutions adopted. A migration algorithm that allows adaptation to the varying memory capabilities of the stations is discussed, and the results of the implementation are shown. We present future enhancements and close with a summary and discussion.

7.2. A First Approach to a Loosely-Coupled Algorithm

To distribute FIAT across a loosely-coupled network of stations, one host is selected as the initiator or master. The master has the responsibility of spawning FIAT on the other stations and establishing communication channels. The processes running on the other stations are called workers, as they run FIAT on the segment of the scene assigned to them by the master. The master also behaves as a worker, computing FIAT for the parts of the scene not distributed to other stations, in addition to its responsibilities as initiator and arbitrator of the distributed computation.

In this configuration for the distributed execution of FIAT (Figure 7.1), the core of the serial algorithm described in Chapter 5 remains unchanged. Subsets of the octree of cells are assigned to the stations that perform balancing and shading using the beam information associated with them. The only significant difference from the original algorithm is the fact that there are certain cells that have neighbours on remote stations. For any light propagation or distribution that occurs through the faces connecting these cells, the sexel update information must be transmitted across the network.

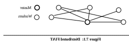


Figure 7.1: Distributed FIAT

7.2.1. The Algorithm on the Master Station

The master initially reads in a description of the stations on the network and spawns corresponding worker processes. The scene description is then read in, and the object database and octree are created. Lights external to the scene are shone and the resulting light is propagated through all empty cells. The lights existing inside the initial cell are also shone, so that all the power emitted by the light sources has been disseminated. This process results in the initial cell being split, since a significant number of cells that have more than one object are encountered.

This initialization of the scene will result in a relatively static octree structure, since many cells that will be subdivided have already reached the maximum possible level of subdivision. The master then begins distributing groups of cells to the worker processes previously spawned on the remote stations. The power and beam information associated with the sexels on the faces of these cells are also transmitted. The stations also receive the whereabouts of each cell's neighbours, allowing sexel update information to be transmitted directly between neighbouring cells on remote stations.

After this assignment has been performed, the master process begins executing the normal FIAT algorithm on the portion of the octree remaining on the master station. These cells are balanced and the corresponding pixels in the LUB are shaded, with updates to sexels on remote cells being transmitted across the network. Eventually the termination conditions for FIAT will be reached on all stations, since the power will have either escaped into the environment or will have been absorbed. The master then requests the transmission of the portions of the LUB associated with the cells assigned to each station. When these pieces of the LUB have been collected, the partial images are combined and the full picture is dumped to the display device or file.

7.2.2. Loosely-Coupled Algorithm on the Worker Stations

Once a worker has been spawned by the master, it reads in the scene description and initializes the object database, the initial cell and the LUB. Cell splits that occur on the master are broadcast to all workers, resulting in the cells being split locally on every station. As a result the octree is replicated on each node. To maintain a consistent view of the octree across the network, cells are assigned integer identification numbers that are the same on all stations. This permits fast and consistent reference to cells when updates are sent from station to station. On workers, the cell-splitting operation is not associated with power distribution. The octree built on the workers is a simple 'skeleton' that carries no power or sexel information. The existence of the same octree on all stations allows the distribution of cells to stations to be simple, since the structure and the neighbourhood relationships are the same everywhere.

After the initialisation phase, the worker waits for a group of cells to be assigned to it by the master. As a result of this assignment, a section of the LUB is implicitly assigned to the worker because each pixel is associated with one cell depending on the screen projection of the cell and the occlusion relationships among objects in the scene (Figure 7.2). Because of occlusion, these regions can be highly disconnected. After the octree has been segmented among all stations, each worker proceeds to balance the cells it has been assigned and shade the corresponding pixels in the LUB. When the termination condition is reached locally, the worker will wait until the master requests the section of the LUB it is responsible for. After transmitting the LUB, the worker terminates.

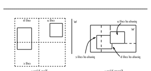


Figure 7.2: Associating Pixels to Cells

7.3. Design Issues in the Loosely-Coupled Solution

The description of the loosely-coupled approach we have given in the previous section has been at a high level, in order to convey the gist of the approach. However, there are a number of interesting problems that had to be solved to construct a functional hierarchical system. We will present some of these issues, concerning the following problems:

- The initial subdivision of the octree to a relatively stable level at an early stage in the execution of FIAT. When this subdivision has been achieved, algorithms that distribute the cells to stations can be more effective, since they have more information to work on.
- The segmentation and distribution of the octree of cells across the network. The method selected to perform this assignment is of crucial importance, since it determines, to a large extent, the amount of processing and memory assigned to each station.
- The form of communication and the required layout of the network which impact the choice of synchronisation schemes and protocols used.
- The reduction of network traffic. A buffering scheme is introduced that does require extra memory but significantly reduces the amount of data transferred on the network.
- The correct distribution of power over the network to preserve a consistent model of light propagation. Care has to be taken when concurrently assessing the power levels on several stations.

- Correct termination. Because there can be arbitrary connections between stations, it is necessary to develop an algorithm that correctly recognises when all stations have terminated.

7.3.1. Achieving Initial Subdivision

The initial cell in FIAT contains all the objects in the scene. As was discussed in Chaper 5, light is propagated through the initial cell, and when a complicated part of the scene is reached by light, it is subdivided. Once more than one cells exist, it is possible to distribute them among stations. To allow better distribution of the load among stations, the system must achieve a state in which a large portion of the splitting process has already occurred. With more cells, an assignment algorithm has larger flexibility and can achieve better results.

At the beginning of the execution of FIAT, the external and internal light sources are shone onto the initial cell. As a result of this operation, the power flux from these sources is transformed into light beams on the sexels of the cell faces. As the algorithm encounters cells with more than one object, the cells are subdivided. It is often the case that empty cells result from subdivision, and light is immediately propagated through them. This whole process is effectively primary illumination of the scene and occurs at the beginning of the FIAT algorithm. Most cells that are reached by primary light are thus subdivided, and, since many contain several objects, they reach the maximum subdivision level. At this point, the octree has reached a comparatively stable state, and the only changes to its structure will occur as a result of secondary illumination later in the execution of the algorithm. This pattern is true for scenes in which primary illumination actually reaches most of the scene. It is, however, possible to construct a scene that is lit almost entirely by secondary illumination, using mirrors and directional lights. This relatively rare situation would be a worst case scenario for our distributed scheme.

Even assuming that most of the scene is lit by primary light, there is no simple way to determine when a 'stable' state has occurred. The fact that this state is not clearly defined makes this task even more formidable. It is therefore necessary to employ a heuristic that will indicate a point in the execution of FIAT at which the octree has reached a satisfactory level of subdivision. A simple way to deal with this would be to stop as soon as the first secondary light appears. This approach suffers from the drawback that in the case of high subdivision, cells with one object (or even a portion of one object) are very common and therefore will be selected for balancing before primary light has been spread to most of the scene. A more involved technique is required.

When the main FIAT loop begins, the cells with the highest imbalance ΔP in power are selected for balancing and shading. Initially, these ΔP correspond to the ΔE (emission) of cells containing light sources, or to the ΔI, the light incoming to the cell from the shining of external lights. In the first few iterations of the algorithm, lights are shone inside the cells where such light sources exist, and, as light reaches the cells containing more than one object, larger cells are split into smaller ones. When such a split occurs, the power is divided among the children, creating cells with smaller ΔP that are therefore not immediately chosen. Many cells that contain no objects are dealt with at this stage as light propagates further into the scene through these empty spaces. Many cells have at this stage either reached the maximum subdivision level, or contain one object. These cells have the higher ΔP, and therefore their selection for balancing and shading becomes more frequent.

To follow this process in the main FIAT loop (Figure 7.3) a record is kept of how many of the cells chosen by `ChooseCell()` are actually balanced and shaded by `ShootLight()`, indicating how many cells either contain one object or have reached the maximum subdivision level.

```
while ( not done ) {
    cell = ChooseCell()
    if ( CanDealWith (cell) ) {
        if( empty cell )
            PropagateThruEmpty( cell )
        else
            ShootLight( cell )
    }
    else
        SplitCell( cell )
}
```

Figure 7.3: Main FIAT Loop

When more than a certain proportion of the N previously selected cells are neither split nor empty, it is assumed that the system has entered a steady state†. Once the algorithm detects this occurrence, it signals that a desirable level of subdivision has been reached and allows the assignment algorithm to begin distributing cells to stations.

This method brings the octree to a relatively stable state, in which most of the cell splitting has already occurred. In Table 7.1, we show the results of our method for two scenes, Scene 1 containing an ashtray with 600 polygons and Scene 2 that consists of a room constructed from 12 polygons. We see that for the scene with many objects the subdivision occurs earlier, since cells often contain more than one object, while for the case with few polygons a significant part of the splitting occurs later.

Scene	Total Split Operations	% Split in Initial Phase
Scene 1	116	76.7%
Scene 2	88	28.4%

Table 7.1: Initial Splitting

As a result, the assignment algorithm has sufficient flexibility to choose the cells to be distributed to the various stations, since there are enough cells to distribute.

† One must note the fact that it is always possible to construct cases in which this steady state becomes unstable again.

7.3.2. Choosing Cell Regions

The algorithm presented in section 7.2 is essentially a static assignment approach. This is not an inherent limitation, and dynamic cell reallocation will be introduced later. In either case, with purely static assignment or dynamic cell migration, the initial assignment algorithm is of great importance. However intelligent a migration or dynamic load-balancing scheme, it always incurs a certain overhead. Therefore achieving satisfactory initial placement of cells on stations is a desirable goal.

7.3.2.1. Discussion of Assignment Strategies

The ideal goal for a segmentation of the octree of cells would be an assignment of cells to stations such that network traffic is minimal. In a high-speed network environment, this will allow good overall performance. In a more realistic, slower network environment, such an approach is more important, since communication overhead is a dominating factor.

In an ideal approach the faces through which the largest amount of power crosses, throughout the duration of the FIAT algorithm, should be identified. The algorithm should make an effort to keep the cells on either side of these faces on the same station, while sending cells that have common faces with low traffic to different stations. Predicting which faces will be 'busy' beforehand is very difficult. The traffic through faces is dependent on the position, orientation, and properties (directional, diffuse etc.) of light sources, and also on the position, orientation and surface properties of the objects in the scene.

A first-order approximation to this ideal goal is to use 'coherent regions' of cells. Such a segmentation of the octree attempts to keep neighbouring cells on the same station. By choosing cells with the highest power imbalance first and keeping their neighbours on the same station, it is most likely that faces through which large amounts of traffic will pass (at least initially) will not be separated.

7.3.2.2. Designing an Assignment Algorithm

Any assignment algorithm will need to know the stations on the network, together with information about their processing power and memory, and the octree of cells defined by the initial splitting. Stations can be rated by processing power, defined by the number and speed of processors and possibly by the amount of memory they have, both real and virtual. To assist in load balancing, each cell is assigned a 'potential load' indicator that approximates the amount of work that can result from distribution of light in this cell. Using the two ratings for stations and cells, the algorithm returns an assignment of cells to stations.

There are various ways to approximate 'potential load' of a cell. Before entering into more detail, it must be observed that this method is inherently limited by the fact that the indicator is assigned on a per cell basis, while the load that really affects the algorithm and its performance is the traffic through *faces*. A second limiting factor is that the properties used to determine the per cell load are static, existing at assignment time, while the traffic through faces is a *dynamic* phenomenon that changes as light travels through the environment.

The static properties are assessed after the initial subdivision phase, at which point a significant portion of the splitting of cells has already occurred. Properties that can be used in defining the load indicator are the power imbalance of the cell, the number of light beams and allocated sexels, the position and orientation of objects in a cell and their surface properties. None of these factors alone can accurately indicate which faces will exhibit higher 'traffic'. It is more likely that cells with a high power imbalance (i.e. with more sexels and beams) will have faces that will be busier than cells that have a smaller power imbalance.

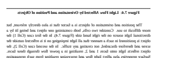


Figure 7.4: Light Flow Affected by Orientation and Position of Objects

The position and orientation of objects is a factor that is also directly relevant, but more difficult to use. Consider two cells, both containing one object and being lit by a directional light source on the right hand side (Figure 7.4). In the first case (Cell 1) the object is positioned in such a manner that all light impinging on it is reflected outside the scene and therefore discarded, not creating any traffic. In the second case (Cell 2), the object reflects light onto faces 1 and 2, resulting in a power flow through these faces. Surface properties also affect light flow and processing resulting from such propagation in an obvious manner. If the surface in cell 2 in the previous example is a highly absorbent, diffuse material, not much light will propagate to the cell faces, while if it is highly reflective, metallic material, the neighbouring cells will receive significant light power. Using the information of object placement and surface properties is a complex and difficult task. It is questionable if a practical *a priori* algorithm is possible. It seems that the assignment problem is more amenable to solutions that employ a feedback approach.

7.3.2.3. The Assignment Strategy Adopted

The assignment strategy adopted is power based, using the imbalance ΔP of each cell as a 'potential load' indicator. The total power level and the relative computational power of the stations determine the assignment.

The assignment algorithm rates each station by the number of processors it has. If there are a total of N processors of equal power† in all the stations, the power P_i that is to be assigned to each station i, with n_i processors is defined as the fraction:

$$P_i = \frac{n_i}{N} \times GPower$$

Recall from Chapter 5 that $GPower$ is the sum of the power that has not been absorbed or discarded still in the system. The implicit assumption is that more power results in more processing, therefore requiring more processors.

The assignment algorithm creates a list of cells to be migrated to a chosen station. First the cell with the highest ΔP is selected and then all its neighbours are added to the list, in an attempt to keep neighbouring cells on the same station. Once the neighbours are inserted in the list, their neighbours and so on recursively, are also added. This continues until P_i is reached or no more neighbours can be found that have not already been assigned to remote stations. The assignment algorithm can be summarised in the following pseudo-code (Figure 7.5):

```
CreateDistributionList( station i )
{
    do {
        add cell with highest ΔP  to list for station i
        AddNeighbours ( cell, station i )
    } while ( PowerAssigned_i < P_i )
}

AddNeighbours( cell, station i )
{
    if ( PowerAssigned_i + ΔP of cell > P_i )
        return;
    if ( cell not already remote ) {
        add cell to list for station i
        PowerAssigned_i += ΔP of cell
    }
    for ( each neighbour of cell )
        AddNeighbours ( neighbour, station i )
}
```

Figure 7.5: The Assignment Algorithm

† The homogeneity assumption can be removed by adjusting the fraction n_i/N by the relative power of each specific processor. This can be done by defining a 'base' processor speed (the speed of the slowest available processor), computing the speed ratio for each processor, and adjusting the computation of the fraction accordingly.

If the while loop in `CreateDistributionList()` executes more than one iteration, it is possible that cells that are not neighbours are sent to the same station. This is not particularly desirable, and, for reasonable scenes, it occurs rarely. When it does, it guarantees that each station will have an amount of work to do comparable to its processing power.

7.3.2.4. Possible Enhancements

There are a number of ways in which the assignment algorithm can be enhanced. One method includes the development of a set of predictors that attempt to determine the path of light through the environment, taking the orientation and properties of the light sources and the surfaces in the scene into account. Some fast approximation method can be used, such as a simple ray trace of a very small set of rays from the light sources. While the trace is being performed, statistics of the traffic on each face crossed by the rays are kept. After the trace is over, the cells associated with the faces through which the largest amount of light passed are kept on the same station, and faces with the lowest traffic are taken as starting points for remote distribution of the cells. This will not provide a perfect assignment strategy, but can help avoid the problem where two highly-reflective surfaces exist on either side of a boundary of two cells on different stations, creating a large amount of light propagation across the network (see Figure 7.6).

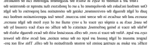


Figure 7.6: Avoiding High Traffic Assignments

A second improvement to the assignment strategy involves more sophisticated selection techniques that incorporate information on the possibly varying network connection speeds between stations. This would allow adaptation to multiple networks communicating through gateways. In combination with a suitable predictor strategy such as the one described previously, this could result in good usage of the available network. The cost of such precomputation for assignment would have to be carefully assessed so that the prediction actually does result in benefit, rather than simply adding extra overhead.

Similarly, by using information about the memory capabilities of the stations, a better assignment of work can be performed. Since busy faces mean that more beams will be assigned and thus more memory will be used, assigning more work to machines with larger real memories will result in better utilisation of the total available memory.

7.3.3. Network and Communication

The network communication channels required by the distributed solution to FIAT are arbitrary point-to-point connections between stations (Figure 7.1). The model adopted in effect creates small, fully-qualified FIAT environments in which the standard algorithm is run on each station. These environments need not necessarily be cubic, since they can result from arbitrary groupings of cells. If we make the simplifying assumption that the regions on each station always consist of connected cells, then we can view each such group as a *virtual cell* that communicates to other such virtuals cells over the network.

7.3.3.1. The Master/Worker Configuration

As we have already seen, one station on the network is selected as the *master*. The necessity for this has already been partially justified by the initialisation procedure that requires the spawning of the various *worker* processes on the other stations and the establishment of the communication channels between the master and the workers. The master is also responsible for the creation of communication channels between worker processes. Only connections between stations that have neighbouring cells are necessary, and therefore only these connections are established. The only station that must have a communication channel to all stations is the master.

The master is used as an arbitrator for synchronisation protocols, as we shall discuss later. An example of such a scheme is the termination algorithm that is made simpler due to the existence of the master (see section 7.3.5).

7.3.3.2. Reader/Writer Paradigm for Concurrency

In the distributed solution we have adopted, each worker process has a dual role: balancing and shading for the section of the octree it has been assigned, and reading the network data from the remote neighbouring cells. Although it would be possible to use a polling scheme, it is simpler to allow the operating system to deal with the problem of scheduling network reads. Specifically, two light-weight processes are created, one for each part of the work: a *writer* process that performs the balancing and shading and a *reader* process that is responsible for reading all the data from the network. The reader sleeps on the active communications channels to be awakened only when there are data to be read, while the writer executes the normal FIAT algorithm. On multiprocessor

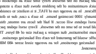


Figure 7.7: Remote Sexel Update

7.3.4. Reducing Network Traffic

A naive approach to implementing sexel updates across the network would be to send a sexel update packet on the network every time such an operation occurs for a remote cell. A quick examination of the problem shows that such a strategy would be exceptionally inefficient. In an average run of FIAT, it is realistic to consider that a cell will have approximately 1000 incoming beams. In such a case, due to distribution, on the order of 10^5 sexel updates will occur. If half the faces are remote, this means that 5×10^4 packets would have to be sent on the network. Every sexel update packet includes cell, sexel, and beam information, that require a packet size of 48 bytes. Therefore, 2.4 MB of network traffic would be generated for each cell balancing operation. With on the order of 200 cell balancing operations, for an average sized scene 480 MB of network traffic would be generated.

This approach is unacceptable. To alleviate the problem, a buffering technique is used. In this scheme, a buffer is created that has the size of the largest cell. It has preallocated sexels for each of the six faces. The memory is allocated after the assignment of the cells to the stations, so that the largest cell is already comparatively small. For a cell that that has an edge length of 8 (that is a normal size for a high subdivision case) the memory required for the buffer is 1.2 MB. This may sound high, but compared to the gains we achieve, it is a worthwhile tradeoff.

Every remote sexel update that occurs while balancing a cell is written to the buffer instead of directly generating a network packet. Updates to the same beam on the same sexel are coalesced in the buffer, in the same way they would be coalesced on the remote host. When the balancing for the specific cell is finished the faces of the buffer that received update information are sent to the corresponding hosts. As a result, the cell information is sent only once for every cell balancing operation and the sexel information only once for every sexel instead of transmitting both for every beam. The only information necessary is the beam index that is sent with the power information, to create a total packet size of 20 bytes per beam.

To understand the benefit from this approach consider its application to the example above. Because of the amount of coalescing that occurs (since there are many updates to the same beam) the number of beams actually affected for one cell balancing operation, would usually be not more than 10000. If again we take half of these to be remote we get 5000 beams to update. Using the buffering scheme this will generate only 100 Kbytes of traffic per cell balancing operation, instead of 2.4 MB, and for the total run, only 20 MB instead of 480 MB. The buffering scheme, therefore, gives over one order of magnitude reduction in the amount of network traffic. To confirm the validity of the above arguments in Table 7.2 we show the network traffic generated using the unbuffered and the buffered technique for Scene 1, containing an ashtray lit by a directional light source.

Traffic on the Network			
Unbuffered		Buffered	
Traffic (MB)	Time	Traffic (MB)	Time
145.9	3:15:32	13.3	20:16

Table 7.2: Benefits from Buffering

We see that the gain is just over an order of magnitude. The corresponding execution times are also reported, indicating the enormous benefit from using the buffering scheme.

7.3.5. Power and the Heap Across the Network

In the original algorithm, dealing with power was relatively simple. Initially the light sources generated some power that created a positive ΔP in some cells. Once a cell was balanced, its ΔP was set to zero, and the neighbours were affected by the light propagation resulting from the interaction of light with objects in that cell. The heap could then be reorganised, and the power balance for the whole system was once again consistent. The algorithm would then assess the total power still in the system to determine whether or not to continue.

In the distributed approach two main differences arise. The first is the fact that there are no longer specific points in time at which the system is guaranteed to have a known state, since sexel updates occur continuously from any remote neighbour. The second difference has two parts. First, the meaning of the sum of current global power† in the distributed case, and second the determination of threshold levels on each station.

† The current global power sum, GPower, introduced in Chapter 5, which is the amount of

The fact that the distributed configuration of FIAT is effectively a system in continuous state of flux, with power entering at arbitrary points in time, requires that the entire heap (containing an ordering of cell imbalances) be reorganised at predetermined intervals. The simplest way to do this is after every balance operation. Although this may become expensive when there are many cells, it does not add significant overhead, relative to cell balancing and shading.

Each station receives a fraction of the initial power existing on the master. As a result, the local $CPower$[1] variable signifies what is left of this initial fraction minus absorbed and discarded light. The threshold for the termination condition is broadcast from the master to the children, and is the same on every station.

The concept of the power heap in the distributed approach also raises some questions. Should the heap be distributed so that each station has knowledge of the power levels on other stations, or at least its neighbours? In the design adopted, each station maintains a private heap that includes only cells that are local to the station. The reason for this is that there is no apparent benefit from maintaining knowledge of power levels on other stations. The local heaps are used in exactly the same fashion as in the single-node algorithm, namely to maintain a fast method for accessing the cell with the highest imbalance for the local instantiation of FIAT.

7.3.6. Termination

Providing a correct termination algorithm in this environment is a complex problem. It may not be clear why this is so, since at a first glance it seems that simple schemes are possible. One such approach would be to maintain a list of state flags on the master, one for each worker station, and one for the master itself, since it is a worker also. When all these state flags indicate that the termination condition has been reached for all workers, the algorithm stops. This approach does not take the asynchronous nature of the communication channels into account.

Let us assume that there are three workers W_1, W_2, W_3, and a master M. Furthermore the master maintains four status flags, F_1, F_2, F_3 for the workers and F_m for itself. In this configuration, each station is connected to every other. At a given moment in time, suppose $F_1 = F_3 = F_m$ = *STOPPED*, indicating that workers W_1 and W_3 have arrived at the termination condition, as has the master, while F_2 = *RUNNING*. This state means that on W_1, W_3 and the master, the global power levels and the highest power imbalance in a cell have fallen below predefined thresholds. After a period of time worker W_2 reaches the termination condition, and notifies the master, which proceeds to initiate the termination procedure. In the meantime, W_3 has just received a final packet of sexel updates from W_2, and changes its state to *RUNNING*. If care is not taken, it will already be too late, and the master will have assumed that the algorithm has terminated, since, momentarily, $F_1 = F_2$

[1] power not yet discarded or absorbed.

$= F_1 = F_m$ = STOPPED. This is incorrect, since W_2 still has balancing and shading to perform.

Having described the problem, we introduce a solution that guarantees correct ordering of events. The general termination problem for asynchronous concurrent stations is complex. However, in our case, due to the existence of the master, things are somewhat simpler. If the master and all the workers but one, say W_1, have terminated, it must be guaranteed that the state transition from RUNNING to STOPPED for W_1 occurs after all other stations have consumed all the pending update sexels and have notified the master of any transitions from STOPPED to RUNNING. The solution described guarantees this correct ordering of events.

To achieve correct termination, each worker maintains a local state flag and an array of state flags for each of its neighbours. There are two possible states for the local flag, which we will call `WrState`: RUNNING or STOPPED. A worker is RUNNING when the writer is still computing, while it is STOPPED if the termination condition has been reached on the local station. There are two possible transitions, from RUNNING to STOPPED and vice versa. We will examine each separately.

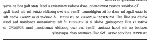


Figure 7.8: State Transitions

Running to Stopped. Suppose this change is performed by the writer on station W_1, once the power levels have fallen below the predefined thresholds. The state change is first sent to each neighbour, and the writer waits for an acknowledgement. We show this in Figure 7.8, state 2. The message is read by the reader of the neighbour, W_2, and therefore is processed after any pending sexel updates, originating from station W_i, have been dealt with. At this point there are four possibilities. Let us call *transaction* the time interval beginning when W_i decides to change state and ending when W_j reads the state change message from W_i.

- The neighbour W_j has been in a running state throughout the transaction. In this case there is no problem.
- The neighbour has remained in a stopped state throughout the transaction. Again there is no problem.
- The neighbour was running and during the transaction it stopped. The timing for such a transition is irrelevant, since we are only concerned that stations which resume computation are acknowledged in the correct order by the master.
- The neighbour was stopped, but the last updates cause it to resume again. As will be shown below, the `WrState` flag on station W_j will have the value *RUNNING*, and therefore it will not match the value of the local neighbour state flag maintained on W_i, that is still *STOPPED*. The mismatch is shown in state 3 of Figure 7.8, where W_i is looping, waiting for the `ChangeState` message to arrive. The writer on station W_i will wait until its local neighbour state flag is updated, and matches the `WrState` flag on the neighbour W_j. When this match occurs, it is guaranteed that the master has been informed of the change of state to *RUNNING* on station W_j (see below), and therefore station W_i can safely inform the master of its transition to *STOPPED*, without there being danger of the master assuming termination (state 4 in Figure 7.8).

Stopped to Running. When the writer is stopped, it waits on a semaphore for a signal from the reader. The reader, having read enough sexel updates to justify restarting the writer, first sets `WrState` to *RUNNING* and then awakens the writer. The writer sends the state change to the master reader, receives the acknowledgement, and then sends the change to the neighbouring readers, thus resolving any mismatch of the state flags between two neighbours. By notifying the master *first*, it is guaranteed that under no circumstances will the master incorrectly assume that all workers have terminated, thus resulting in a correct state at the end of the above transaction (state 5 in Figure 7.8).

Finally to initiate the LUB transmission, an `dState` variable is kept that is set to *TERMINATED* only when the master broadcasts a message to that effect to all workers. When this occurs, the writer will exit the main loop and send the LUB as soon as it awakes. In all other cases `RdState` is set to *RUNNING*, and whenever the writer is awakened it continues balancing and shading.

7.3.7. Summary of Loosely-Coupled Algorithm

All the problems discussed in the previous sections add a significant amount of complexity to the loosely-coupled scheme developed. To summarise the full solution, we present a high-level pseudo-code description of the full hierarchical solution. The code is naturally separated into Master Reader and Writer, and Worker Reader and Writer. This summary will show how steady state is achieved, how regions are selected, the techniques used for communication and synchronisation, the local heap manipulation and finally the details of the termination algorithm.

7.3.7.1. The Master Writer

Initially the master spawns the workers on the various stations and reads in the scene description, creating the object data-base and the initial cell (Figure 7.9). After the initialisation, the octree is brought to a steady state. This occurs when all ΔE (emission) are 0 and when over 50% of the last N cells chosen had one object, or had reached maximum subdivision. After this, the cell list to be sent to each station is created, as described previously.

```
MasterWriter()
{
    ReadHostFile()
    SpawnChildren()
    ReadSceneAndInitialise()
    /* perform initial phase */
    while( LightsNotShone and CellsDealtWith < 50% ) {
        cell = ChooseCell()
        if ( CanDealWith (cell) ) {
            if( empty cell )
                PropagateThruEmpty( cell )
            else {
                AdjustCellsDealtWith()
                ShootLight( cell )
            }
        }
        else {
            SplitCell( cell )
            BroadCastSplit( cell )
        }
        if( ΔE > 0 )
            ShineLights()
    }
    /* distribute cells to stations */
    DistributeOctree()
    SpawnReader()
    /* begin normal FIAT */
    while( !done ) {
        cell = ChooseCell()
        if( ΔP_cell < Threshold and GPower < GThreshold )
            if( MasterStopLocal() )
                done = TRUE
        NormalFIAT(cell)
    }
    /* output picture and cleanup */
    Terminate()
}
```

Figure 7.9: The Master Writer Main Loop

```
DistributeOctree()
{
    for( every station in StationList ){
        CreateDistributionList( station )
        cell_cnt = | DistributionList |
        Send( station, cell_cnt )
        for( every cell on DistributionList ) {
            SendCellToStation( station, cell )
            /* remove from local heap */
            HeapRemove( cell )
            cell.remoteFlag = TRUE
            cell.remoteStation = station
        }
    }
}

SendCellToStation( station, cell )
{
    Send( station, cell )
    Send( station, cell.beam_cnt )
    while( cell.beam_cnt-- > 0 ) {
        beam = NextBeam( cell )
        Send( station, beam )
    }
}
```

Figure 7.10: Distributing the Cells

The list of cells is then sent to each station, and the corresponding cells removed from the local heap. The cell is marked 'remote', and the station identifier is filled in so that remote sexel updates can be routed correctly (Figure 7.10). After these operations the master continues with the processing of the normal FIAT loop, balancing and shading for the remaining octree.

```
MasterStopLocal()
{
    WrState = STOPPED
    if ( all stations done ) {
        /* LUB collection has to be performed by the reader */
        SynchroniseWorkers()
        WaitForSignal()
        return TRUE
    }
    else {
        /* Sleep until a signal is received from reader */
        WaitForSignal()
        if( RdState = TERMINATED )
            return TRUE
        else
            WrState = RUNNING
            return FALSE
    }
}
```

Figure 7.11: Master Termination Code

Termination for the master has to deal with two possible situations. In the first case, all the worker stations have terminated before the master. The master then sends a synchronisation message to all workers that respond with a `TerminateSync` message (Figure 7.11). In the second case there will be some other station still working, in which case the master reader will handle termination, since the writer will be dormant. The `TerminateSync` message protocol allows LUB collection to be performed in a uniform way in both cases by the reader, since in both cases the `TerminateSync` message is used to determine termination (Figure 7.13).

7.3.7.2. The Master Reader

The reader consists of a main loop that sleeps on the select call waiting for I/O. There are other messages, but the only ones of interest are `SexelUpdate`, `StateChange`, and `TerminateSync` (Figure 7.12).

```
MasterReader()
{
    while ( true ) {
        channels_ready = select()
        while ( ! done ) {
            /* Try all channels in succession */
            channel = NextChannel( channels_ready )
            /* If the read would block try the next channel */
            if( (message = Read( channel )) = Would Block )
                continue
            switch( message ) {
                case SexelUpdate :
                    if( WrState = STOPPED )
                        DealWithChange()
                    UpdateSexel( message )
                case StateChange :
                    AcknowledgeChange()
                    if ( all stations stopped )
                        SynchroniseStations()
                case TerminateSync :
                    CheckTerminate()
                ...
            }
        }
    }
}
```

Figure 7.12: The Master Reader Main Loop

When the writer is asleep, the reader will check if the sexel update created enough work, and if necessary it will wake the writer. The master reader also has the responsibility for reading all the synchronisation messages and requesting the LUBs from the workers. This is performed by `CheckTerminate()` that is called only when the writer is stopped (Figure 7.13), since `TerminateSync` messages arrive only in this case.

```
DealWithChange()
{
    if( EnoughWork() ) {
        WrState = RUNNING
        SignalWriter()
    }
}
CheckTerminate()
{
    TerminateCnt ++
    if( TerminateCnt = NoOfWorkers ) {
        AskForLUBs()
        RdState = TERMINATED
        SignalWriter()
    }
}
```

Figure 7.13: Master Reader Auxiliary Functions

```
WorkerWriter()
{
    ReadSceneAndInitialize()
    InitialWait()
    SpawnReader()
    while( !done ) {
        cell = ChooseCell()
        if( ΔP_cell < Threshold and GPower < GThreshold )
            if( WorkerStopLocal() )
                done = TRUE
        NormalFIAT(cell)
    }
    SendLUB();
    /* and end ... */
}

InitialWait()
{
    master_channel = select()
    message = Read( master_channel )
    switch( message )
        case Split :
            CellSplit( message.cell )
        case InitialDone :
            cell_cnt = Read( master_channel )
            while( cell_cnt-- > 0 ) {
                cell = Read( master_channel )
                HeapInsert( cell )
                beam_cnt = Read( master_channel )
                while( beam_cnt-- > 0 ) {
                    ReadAndUpdateBeam( master_channel )
                }
                return
            }
        }
}
```

Figure 7.14: Worker Writer Main Loop

7.3.7.3. The Worker Writer

The worker writer's main loop consists of the initial waiting/reading loop and the normal FIAT loop (Figure 7.14). The function `InitialWait()` reads the split operations and when the master is ready to distribute cells, reads in the cell and beam information, inserting the new cells into the heap. The local stop function is exactly the same as the master's less the special termination code (Figure 7.15).

```
WorkerStopLocal()
{
    /* Sleep until a signal is received from reader */
    WaitForSignal()
    if( RdState == TERMINATED )
        return TRUE
    else {
        WrState = RUNNING
        return FALSE
    }
}
```

Figure 7.15: Worker Termination Code

7.3.7.4. The Worker Reader

The reader for the worker processes is very similar to the reader on the master. The worker, however, does not have to deal with termination conditions and signals the writer after receiving a TerminateSync, causing the writer to terminate after the LUB is sent.

```
WorkerReader()
{
    while ( true ) {
        channels_ready = select()
        while ( not done ) {
            /* Try all channels in succession */
            channel = NextChannel( channels_ready )
            /* If the read would block try the next channel */
            if( (message = Read( channel )) == Would Block )
                continue
            switch( message ) {
                case SexelUpdate :
                    if( WrState == STOPPED )
                        DealWithChange()
                    UpdateSexel( message )
                case StateChange :
                    AcknowledgeChange()
                case TerminateSync :
                    send TerminateSync to master
                case SendLUB :
                    RdState = TERMINATE
                    /* Writer Will Send the LUB */
                    SignalWriter()
                ...
            }
        }
    }
```

Figure 7.16: Worker Reader Main Loop

7.4. Dealing with Different Memory Capabilities

In an environment where the memory capabilities of the stations differ considerably, some stations will exhaust their memory faster than others. If no action is taken, these systems will fail from memory exhaustion, with undesirable ramifications. It is therefore necessary to transfer cells from stations reaching their limits to other stations with more available main store. For this purpose, a migration scheme was designed, in the form of a totally asynchronous protocol that does not interrupt the normal processing of the distributed algorithm, while at the same time guaranteeing correct migration of cells from one station to another.

7.4.1. A Migration Protocol

When a worker reaches the limits of its memory capacity, it signals the master. The master, on receipt of this message, queries the other stations and decides the best candidate based on the available memory on each station. The selected station is notified. The station that receives this message sends a message to the station wishing to migrate cells, and the migration of cells begins. There are several complications involved in this scheme since there are some potential concurrency problems that have to be dealt with. Before explaining some of these and how they are resolved, we present a time diagram that depicts the exchange of messages between the three participating stations in the migration protocol and the time dependencies involved (Figure 7.17).

Time	Master		Source		Destination	
	Reader	*Writer*	*Reader*	*Writer*	*Reader*	*Writer*
1				*Send Master Migration Request*		
2	*Rcv Request Enq Request*					
3		*Query Loads on all station*				
4		*Select Destination Send Notification*				
5					*Receive Notification Enqueue Notification*	
6						*Send Notification to Source*
7			*Receive Notification Enqueue Notification*			
8				*Send Cells to Destination*		
9					*Rcv Cells Update Structs*	
10						*Send Master Cell Relocation Info*
11	*Rcv Cell Relocation Info*					

Figure 7.17: The Migration Protocol

7.4.2. Synchronisation Problems

There are several important points that have to be made concerning the migration protocol. These are mainly timing issues that have to be considered so that the migration does not result in corruption of data, or loss of information, since during the transaction all three participants are continuing normal computation of the algorithm.

To be consistent with the reader/writer paradigm, it is necessary to use a queue to allow communication between reader and writer. In this way, for example, in time-slot 2 (Figure 7.17) the master reader receives the request for migration from the source station. The reader cannot initiate any action that requires possibly blocking write operations, since this could lead to a deadlock. Instead, the reader queues the request which is dealt with by the master when convenient. This queue is considered to be of high priority and is regularly flushed by the writer. The load query issued by the master in time-slot 3 (Figure 7.17), uses the 'synchronous' acknowledgement method discussed previously, and therefore does not require the intervention of the writer on the worker stations.

As cells migrate, there are periods of time during which their exact location is unclear, since the source writer is in the process of transmitting them to the destination reader. If care is not taken it is possible that updates to the cells being migrated may be lost. This could occur if an update is received for a cell that is still marked local, but has already been transferred to the new station. Before beginning to send the sexel information to the destination cell, the remote flag is set on the cell to be migrated. The master then has to explicitly synchronise with the reader to guarantee consistency of the data structures†. Any subsequent updates for this cell read by the reader are then treated as requests for a remote cell. Such requests are placed in a queue, in a manner similar to requests for the cell currently being balanced. The queue is flushed later by the writer, and the updates are sent to the new station that has now inserted the new cell in its data structures. In this sense, an 'automatic forwarder' exists that takes care of any sexel updates that may have been issued to the old location of a cell. This guarantees that information is not lost. The neighbours of a cell are notified immediately after the migration, but due to the asynchronous nature of the distributed solution, there is no guarantee when remaining sexel updates already on the network will reach a station.

During the migration process, care must be taken to ensure that the heaps on both the source and the destination are not corrupted. To this end, a lock is acquired before deleting cells from the heap on the source or inserting them on the destination station.

The method described above, allows a station that is reaching its memory limits to relieve some of the load to another station that still has available memory. The protocol is asynchronous so that the source station may have already run out of memory before the protocol can finish. In this case, the source station that has run out of memory sends a disaster message to the master, that simply collects all the LUB portions from the workers and terminates.

7.5. Results

We present the implementation results of the design described in this chapter. The numbers presented are from runs of the two test scenes used in Chapter 6, an ashtray lit by a directional light source which we will call Scene 1, and a room with diffuse walls lit

† If there is no synchronisation, the reader may change the cell that is being transmitted, fatally disrupting the transmission protocol.

by an extended diffuse light source, which we will refer to as Scene 2. We use higher subdivision parameters, thus increasing the memory requirements.

The presentation of these results is in the following format. The first two tables, Tables 7.3 and 7.4, present the execution times for the distributed scheme for each scene. To provide a meaningful measure of comparison, we compare the results of the hierarchical approach to the tightly-coupled approach using up to 4 processors.

The first four entries in each table are the results of the tightly-coupled scheme on a single multiprocessor utilising from one to four processors. All these tests where run on a 4-processor IRIS 4D/120 with 64 MB of main memory. Notice that the memory utilisation is larger than the real memory of the machine, thus incurring the overhead of paging, which now dominates the computation time. The speedup therefore is not very good.

The second half of these tables contain results for the full hierarchical solution and have entries for 5, 6 and 8 processors. The 5 processor configuration is one 4 processor host and one uniprocessor host, the 6 processor configuration is one 4 processor host and two uniprocessor hosts and finally the 8 processor configuration is one 4 processor, one 2 processor host and two 1 processor hosts. The 2 processor host has 24 MB of real memory, while the uniprocessors have 16 and 12 MB respectively.

Speedup (S) and processor utilisation (PU), are the same as defined in Chapter 3, ignoring the overhead of the network for the hierarchical coupled results. The memory used in the hierarchical case is the sum of the memory used on all the stations.

Timing Results for Scene 1				
No. of procs.	Time	S	PU	Memory
1	1:15:15.7	-	-	-
2	54:35.0	1.38	69.0%	83.9
3	42:12.0	1.78	59.3%	84.0
4	38:22.5	1.96	49.0%	84.1
5	26:19.6	2.86	57.2%	59.0
6	20:16.8	3.71	61.8%	54.1
8	12:21.0	6.09	76.1%	50.2

Table 7.3: Timing Results for Scene 1

Timing Results for Scene 2				
No. of procs.	Time	S	PU	Memory
1	1:32:46.7	-	-	-
2	1:03:41.9	1.46	73.0%	86.1
3	57:03.3	1.63	54.3%	86.2
4	59:37.5	1.56	39.0%	87.4
5	1:07:53.5	1.37	27.4%	81.8
6	57:04.2	1.63	27.2%	81.4
8	34:05.9	2.72	34.0%	76.2

Table 7.4: Timing Results for Scene 2

There are two things to notice in the above results:

Memory Savings. Less memory is used as the number of stations used is increased. There are two explanations for this. First, each station maintains the network buffer that substitutes a large part of the dynamically allocated memory in a non-fragmented manner, allowing for less memory waste. Second, with more stations, there are more balancing operations occurring concurrently, and therefore less memory is allocated for intermediate results.

Better Speedup for Specular/Directional Scene. The performance of the algorithm is much better for the case of the specular ashtray lit by a directional light source. In Tables 7.5 and 7.6 we see the network traffic for each of the scenes, and it is obvious that there is much more traffic in the case of the room scene, than there is with the ashtray, even though the memory utilisation for the single-station case is approximately the same. Additionally, due to the limited spread of light from the directional light-source and the specular surfaces, more coalescing occurs in the buffer, thus reducing the network traffic. This obviously results in less network overhead and better speedup.

Megabytes of Traffic	
# of Procs	Traffic
5	12.7
6	13.3
8	9.2

Table 7.5: Network Traffic for Scene 1

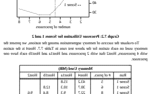
Megabytes of Traffic	
# of Procs	Traffic
5	21.2
6	33.8
8	32.5

Table 7.6: Network Traffic for Scene 2

We can see from Tables 7.5 and 7.6, that the amount of network traffic is significant, but much more so in the diffuse/extended light source case, since there are more beams, and, more importantly, the beams are spread among various directions. Additionally, the fact that in the diffuse case only 28.4% (see Table 7.1) of all cell splits occur in the initial phase, indicates that the assignment algorithm did not have sufficient information to perform a very satisfactory placement of cells. In the case of the directional\specular light source, beams are more concentrated in a fixed set of directions, creating less network traffic.

The following two graphs show speedup and processor utilisation for the two scenes (Graph 7.1 and 7.2).

Graph 7.1: Speedup for Scenes 1 and 2



Graph 7.2: Processor Utilisation for Scenes 1 and 2

To evaluate the success of memory segmentation among the stations, we present the memory used on each station for the above test runs in Table 7.7. Host4 is the station with 4 processors, Host2 that with 2 processors and Host1a and Host1b each have one processor.

Memory Used (MB)					
Run	# of procs.	Host4	Host1a	Host1b	Host2
Scene 1	5	43.2	15.8		
Scene 1	6	30.7	10.3	12.8	
Scene 1	8	22.7	9.3	8.4	9.6
Scene 2	5	57.0	24.8		
Scene 2	6	47.7	23.2	10.4	
Scene 2	8	33.2	6.3	10.7	25.9

Table 7.7: Memory Used on Each Host

We see from the table that the memory used is comparatively well distributed among stations, although not always optimally. Note that in all the test runs, the master was on Host4. Some of the memory used on Host4 was actually allocated before distribution.

The hierarchical approach has the additional benefit of producing pictures that require memory exceeding the capabilities of any single computer. The results of one such test run are shown in Table 7.8, that used a maximum subdivision level 6 and 4 sexels per edge of the smallest cell, requiring a total of 503 MB. The test was run on a 4 processor host (Host4) with 310 MB of swap space and 64 MB of real memory, and on a 2 processor host (Host2) with 170 MB of swap space and 64 MB of real memory. Execution time, memory usage and network traffic are shown.

Execution Time	Memory (MB) Host4	Host2	Total	Network Traffic (MB)
12:21:26	313.1	190.5	503.6	77.7

Table 7.8: Excessive Memory Utilisation Test

Overall, from the above results, it can be seen that the loosely-coupled solution is successful in both goals: memory segmentation is achieved, and there is significant speedup from the usage of more processors. The fact that in the hierarchical case the program does not exceed the real memory limitations of the hosts, allows faster speed when compared to the version run on one station that uses paging to satisfy the large memory requirements. The hierarchical version also saves memory, as a result of buffering and the elimination of the need for intermediate storage. Finally, pictures can now be computed that require memory exceeding the capacity of a single computer.

7.6. Possible Improvements

In this section we describe two possible improvements to the hierarchical approach described.

7.6.1. Using Migration for Load Balancing

The method described in section 7.4 can, with slight modifications, be adapted to be a general purpose migration technique for load balancing. The master can monitor the load in computation and memory on all the stations and reassign cells when necessary. To simplify this approach, the master can use the state change messages as cues. When a station changes to stopped state long before any of the others, the master can determine which of the other stations is more loaded from a point of view of memory and computation and initiate the migration protocol, moving cells to the less heavily loaded station.

7.6.2. Adaptive Virtual Faces

As was noted in the section on cell regions, an ideal algorithm would minimise the traffic through faces. This goal could be partially achieved by monitoring the transmission through faces connecting remote cells and relocating cells so that busy faces are local. If we adopt the paradigm of regions on stations being considered as virtual cells, this relocation mechanism can be viewed as a way of implementing virtual faces that adapt to the traffic flowing through them.

A similar idea to this was presented in [Dipp84], in which the corners of the generalised cubes used in the subdivision technique adapted according to the load in each region.

7.7. Summary and Discussion

The second level in the hierarchy of parallelism for FIAT is that of distributing the computation of cells among workstations connected on a network. Such a scheme was developed and implemented, indicating that hierarchical parallelism is possible for FIAT.

In summary, the solution adopted starts the FIAT algorithm on a single station, the master, which in turn spawns worker processes on the other stations on the network. The master proceeds to split the octree and distribute the cells among stations. Each station computes FIAT for the segment of the scene it is responsible for, communicating via sexel update requests with the other stations. When all stations have finished computing the image is combined.

There are a number of issues that had to be addressed in this design.

- A heuristic was developed to determine a point early in the execution of FIAT when the octree has reached a comparatively stable state of subdivision for distribution.
- An algorithm was developed to distribute groups of cells to stations. In this approach the load is divided as evenly as possible, and there is an attempt to minimise the amount of network traffic. The algorithm bases the selection on the power level of cells.
- A reader/writer paradigm was used that gives a simple solution to the problems of concurrency and network communications to avoid deadlocks and complex I/O scheduling.
- To reduce network traffic, a buffering scheme is used that allows sexel updates to be coalesced and that reduces the amount of information sent per beam. Approximately an order of magnitude of network traffic is saved using this scheme.
- A sophisticated termination algorithm was developed to avoid potential race conditions. The techniques used in the single-station approach to deal with power and the heap structure allowing fast access of cells with high power imbalance had to be adjusted for the loosely-coupled case.
- A migration scheme was developed to avoid memory exhaustion on stations with limited real memory. With this method, stations reaching their memory limits move cells to stations with more available memory.

The results of this scheme indicate success for both goals set out for the hierarchical scheme.

- Use of multiple parallel workstations connected over a LAN results in satisfactory speedup, with an efficiency that peaks at 76%. Considering the limited bandwidth of the Ethernet, this result is significant.
- The memory requirements are spread among the stations participating in the hierarchical scheme. Paging activity is significantly reduced for two reasons. First, overall more memory is available. Second, less memory is used as the number of stations

increase, because fewer intermediate results are stored, and by utilising the buffer, the memory wasted due to fragmentation is limited. The hierarchical approach also allows pictures to be computed at higher subdivision, requiring more memory, since it is now possible to combine the virtual address spaces of more than one station.

The results of this chapter demonstrate that FIAT is a good candidate for parallelism, and that the hierarchical solution is a viable and attractive approach.

Chapter 8

Conclusions

8.1. The Success of Hierarchical Parallelism

The research presented in this thesis is a design utilising *hierarchical parallelism* for FIAT, a space-subdivision global illumination algorithm. The two components of the hierarchy, a tightly-coupled solution for shared-memory multiprocessors and a loosely-coupled scheme for a network of multiprocessors and uniprocessors, were designed and implemented.

This thesis succeeds on two counts:

(1) It was proven possible to successfully parallelise FIAT. The benefits from the overall parallel algorithm were speedup of the computation and distribution of memory among workstations.

(2) It was shown that hierarchical parallelism can work and offer benefits, even if the network used in the loosely-coupled case is not especially fast. We believe that many algorithms can adapt nicely to hierarchical parallelism, and, therefore, it is a promising way of dealing with large and demanding computational tasks.

We believe that the success of the parallel design is an important step, both for computer graphics and for parallel computation:

For computer graphics, this success shows that otherwise prohibitively expensive problems can now be solved without having to resort to expensive supercomputer solutions. Research alternatives for global illumination techniques can now be investigated in more detail.

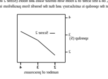
For parallel computation, it is demonstrated with a fully implemented design that hierarchical parallelism is an interesting paradigm for compute- and memory-intensive applications.

8.2. Recapitulation of the Results

The results of this thesis have two components. The first part consists of the results of the tightly-coupled parallel design for balancing and shading in a cell, while the second involves the full hierarchical solution that utilises a network of multiprocessors.

Tightly-Coupled Design. To decompose the problem of balancing, the direction space defined by the discretisation scheme used in FIAT is segmented, and a specific subset of the discrete directions is assigned to a given processor. The balancing calculations are reorganised so that when a beam intersects a surface, the intersection point is buffered. A second phase follows, in which each processor performs the sexel updates for the subset of directions it is assigned on all the buffered intersection points. For shading, each pixel is assigned to a processor and the shading calculations are performed independently.

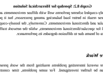
Because of its low contention, good load balancing, and minimal need for synchronisation, this design results in good speedup. Graph 8.1 shows the speedup for up to 4 processors, on a test scene of a room with diffuse walls and floors (Scene 2 in Chapter 6). We see that the speedup is satisfactory and that the benefit from parallelism is significant.



Graph 8.1: Speedup for Tightly-Coupled Solution

Hierarchical Solution. The method we developed takes the octree of cells produced by the FIAT algorithm and distributes groups of cells to the stations on the network, some of which are multiprocessors. The method developed involves a solution to the initial placement of cells regions to stations. To solve the problems of synchronisation, we introduced an asynchronous reader/writer protocol. To avoid the problems of high network traffic, a buffering scheme was employed that results in a reduction in the amount of data transferred on the network by an order of magnitude when compared to the naive approach. The hierarchical scheme also decreases the memory requirements of the algorithm, due to buffering and concurrent execution. To avoid the problem of memory exhaustion within a workstation, a dynamic memory load balancing algorithm was developed that migrates cells.

Overall, the hierarchical algorithm performs very well, giving significant speedup, especially considering the slow speed of the network used. In Graph 8.2 we see the speedup for configurations of 4 stations: one 4 processor station, one 2 processor station and two uniprocessor stations. The data for a total of 5 processors was run on one 4 processor and 1 uniprocessor station, with the addition of one more uniprocessor we get the results for 6 processors, and finally the results for 8 processors include all the processors on all stations operating together. These results are for the scene consisting of a specular ashtray lit by a directional light source (Scene 1 in Chapter 6).



Graph 8.2: Speedup for Hierarchical Solution

The tightly-coupled scheme performs well with diffuse environments and area light sources as these result in better load balancing among processors, but it performs less well for specular and directional environments. Conversely, the loosely-coupled solution performs well for specular/directional environments, while the network traffic generated by the plethora of beams due to diffusion degrades its performance.

8.3. Future Work

There are several interesting problems resulting from the work presented in this thesis that can be further investigated. For some problems, time constrains forbid further investigation, while for others, their solution seemed to involve work far beyond the scope of a M.Sc. thesis. We will attempt to classify these issues.

- As was mentioned in the previous section, the tightly-coupled and the loosely-coupled approach are complementary in the environments they handle best. Developing an approach that actually uses this property in the assignment of cell regions and in load balancing could result in an optimal solution.
- A load-balancing design for the tightly-coupled solution could be introduced. Such a scheme would allow the algorithm to deal better with the differences in the surface types and the light source properties. An attractive idea would be to introduce a scheme that uses the existing solution for diffuse surfaces, since it performs well, while applying an approach that dispatches work to processors when the scene has a high proportion of specular surfaces.
- For the loosely-coupled/hierarchical structure, the issue of initial assignment is of extreme importance. Although the existing approach is not totally arbitrary, an improved scheme could be developed that predicts the potential workload more accurately. Incorporating memory information and varying network speeds into the model would also be exciting.
- The concept of power levels and termination thresholds in the distributed case is effectively an open problem. A much deeper understanding of the properties of the cell regions assigned to stations is required so that correct thresholds can be computed.
- A fully dynamic load-balancing scheme, either using the 'virtual-face' or the simple migration scheme, or a combination of both, would be interesting. One necessary condition for such an approach is being able to determine the cost of the load-balancing attempt so that the overall effect is truly beneficial.
- Applying the algorithm to an architecture specifically built for hierarchical parallelism would be an interesting exercise. Such an environment would reduce the effort required to achieve the loosely-coupled distribution, since the actual architecture would provide most of the support, and eliminate the need for the elaborate buffering schemes. Hector [Stum89] would be a good platform, since in such an environment the network cost would become negligible, resulting in even better performance for the loosely-coupled solution and as a result for the hierarchical approach as a whole.

8.4. Retrospective

Looking back, one can identify many of the mistakes made, especially in the course of the implementation. Most of these were due to the problems of parallel programming, both for the tightly-coupled environment and the message-passing, loosely-coupled case. Programming in these environments is extremely difficult, since one has to cut away from habits of the past, and from a whole frame of mind that comes from years of serial programming.

One positive side effect of the effort was the construction a large working software system. The tightly-coupled solution is already being used for picture generation in the development process of FIAT, and once the loosely-coupled implementation stabilises it will also be used to compute images at higher subdivision.

We believe that hierarchical parallelism has a place in the future of computing and computer science. Moving to this model is not easy and will require significant effort. Once better tools supporting this paradigm become available to the programmer, hierarchical solutions will become more attractive.

The benefits of hierarchical solutions, both in computation time, and in distributed memory utilisation are just too large to be ignored. Hierarchical parallelism is no panacea; but it is here to stay.