Distributed Rendering of Particle Systems

Charles Smith
B.A.(Mod) Computer Science
Supervisor:Michael Manzke

April 22, 2003

Abstract

Scalable Rendering allows the creation of high performance rendering systems without specialist hardware by utilising clusters of rendering nodes with commercial graphics accelerators.

Particle systems describe a class of rendering techniques for displaying objects without a rigid structure, including fire, water, and atmospheric effects, which traditional Computer Graphics approaches find difficult. The term has also been extended to cover decentralised behavioural systems consisting of large numbers of entities which are independent but influence one another's motion.

Here we describe attempts to extend and improve on some existing computer graphics techniques by making use of parallel computation and rendering, to manage and display complex particle based systems in real-time on modest sized compute clusters.

Acknowledgements

I would like to thank my supervisor Michael Manzke for his support and advice on this project, and Dr. Carol O'Sullivan whose help has also been invaluable to me, the Computer Graphics group at Stanford whose work on Chromium gave me the basis for this project, and my friends, family and Anne for all their encouragement.

Charles

Produced with LaTeX

Contents

Chapter 1

Introduction

1.1 Distributed Rendering

Computer Graphics is one of the fastest growing fields of the Computer Science discipline, having progressed rapidly from the primitive vector graphics of the 1960's to complex and immersive 3-Dimensional environments with specialist supporting hardware acceleration. In recent years competition among the major manufacturers of 3D Graphics Accelerators has resulted in affordably priced cards with extremely fast dedicated graphics processors and growing amounts of video memory.

Accompanying these technological improvements has been continued increases in the demands which are placed on graphics subsystems, a demand which for the foreseeable future will always outstrip the capabilities of a given graphics accelerator. Research always continues toward more realistic, and hence compute intensive, rendering techniques, and both scientific and medical visualisation and the entertainment industry require more and more complex scenes and larger datasets to be displayed.

Traditionally, to boost rendering performance, specialist multiprocessor systems with a custom graphics processing pipeline have been constructed to meet the specifications required. This approach produces high speed and efficient rendering systems, but suffers from several disadvantages. Firstly it is not an option available to general consumers without the resources to construct such a system, and it is difficult to extend; usually this requires a redesign of the system.

Scalable rendering addresses these issues by providing an extensible high-performance rendering architecture that is constructed from readily available

general purpose components. Specifically, clusters of standard workstations with graphics acceleration hardware, combined with high performance interconnect. The arrangement of the system does not require any changes to the rendering pipeline within a graphics accelerator, allowing hardware upgrades without altering the configuration. Furthermore, nodes can to inserted into or removed from the cluster to attain the required level of performance.[11]

1.1.1 Applications

Applications for Distributed Rendering include tiled display walls, where an array of projectors or displays is used to create a very large composite image - each projector can be given its own machine in the rendering cluster, and CAVES, where multiple projectors are used to rear project stereo images onto the inside walls of a cube to create an immersive virtual reality simulation.[1]

1.2 The Chromium Project

Chromium[2] is an extension of the WireGL[3] project developed by the Computer Graphics Laboratory at Stanford[4]. It provides an scalable distributed rendering implementation of OpenGL, and adds functionality including custom OpenGL extensions, synchronisation primitives and Stream Processing capability.[12]

The basic structure of a Chromium cluster consists of 3 types of nodes:

- A single *Mothership* node — this node coordinates the cluster and provides configuration information
- *Server* nodes — these nodes accept rendering commands, and perform appropriate actions
- *Application* nodes — these nodes issue rendering commands to servers

Both server and application nodes can have an attached chain of Stream Processing Units (SPUs). SPUs implement the fundamental processes of

[1] http://cave.ncsa.uiuc.edu/
[2] http://chromium.sourceforge.net/
[3] http://graphics.stanford.edu/software/wiregl/
[4] http://graphics.stanford.edu/

distributed rendering and can also be used to produce special non-photo-realistic effects. An *SPU Chain* is a set of SPUs, where rendering commands enter at the head SPU of the chain and are passed down until the tail SPU which produces the final result. Any node can have a chain of SPUs, for application nodes the application's OpenGL commands are submitted to the head SPU, and the tail SPU will typically dispatch commands to servers. The head SPU of a server accepts incoming commands, and the tail SPU will either pass commands to another server or produce rendered output. This system allows SPUs which provide logging, performance measurement or extensions to OpenGL to be inserted without needing to modify applications.

To illustrate the use of SPUs and the basic operation of the system, consider a cluster with 3 nodes: one mothership node, one server node and one application node. The server node has one SPU, the *Render* SPU. This simply carries out the rendering commands that are issued to the server. The application node also has one SPU, the *Pack* SPU, which packs all OpenGL commands into buffers for transmission across the network to the server.

Upon initialisation, the sequence of operations is as follows:

1. The mothership is started, and waits for configuration requests from other nodes

2. The server node is started, it locates the mothership and requests configuration information regarding the arrangement of the cluster

3. The application node is started, and obtains configuration information in a similar fashion

4. The application begins issuing rendering commands. These are packed into network buffers by the Pack SPU, before being sent across the network to the server node, which unpacks them and passes them to the Render SPU which renders them to the display.

This is a somewhat trivial example which does not illustrate the power of distributed rendering. Chromium allows configurations with multiple servers and multiple clients, rendering and submitting commands in parallel, as well as chains of SPUs which perform complex manipulation of rendering primitives before they are finally displayed. These mechanisms will be explained in detail and discussed in section 5.

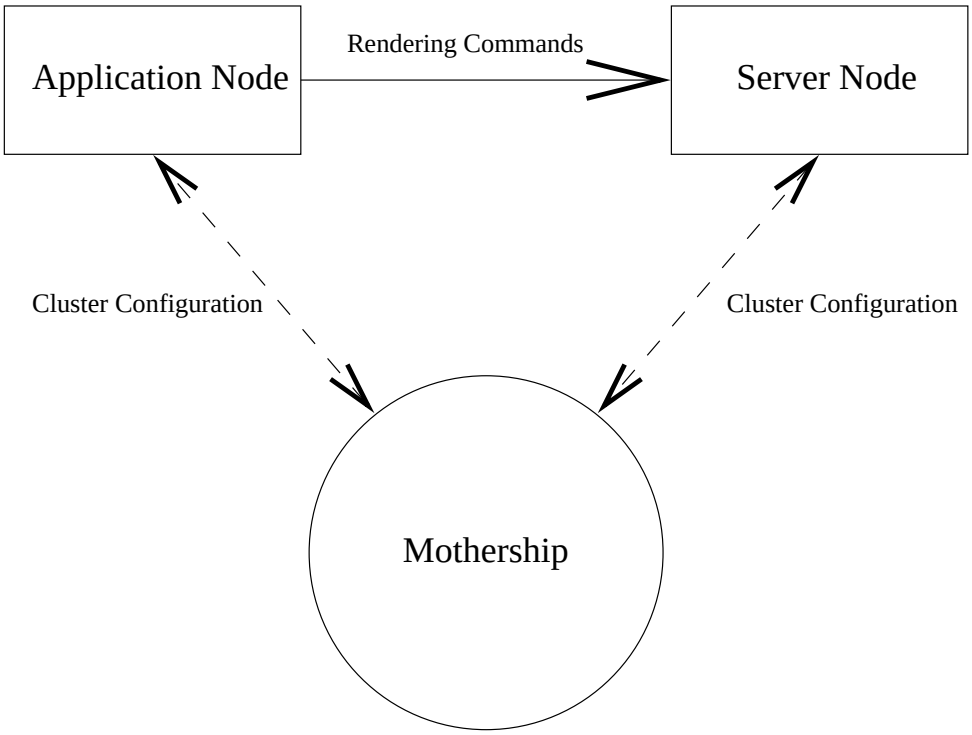


Figure 1.1: A minimal Chromium cluster contains a mothership, an application node and a server node

1.3 A History of Particle Systems

Particle Systems were first formally proposed as a rendering technique by William Reeves in 1983[15], although several of the concepts had been used previously. Reeves' paper was titled "Particle Systems — a Technique for Modelling a Class of Fuzzy Objects", and describes methods for modelling water, fire, grass and atmospheric effects, which pose difficulties for traditional polygonal representation and rendering approaches.

Objects and phenomena such as these are difficult to represent using polygons because they cannot be approximated satisfactorily with flat faces, they are dynamic and constantly change their form, and their motion is generally stochastic in nature. Particle Systems represent objects as clouds of points, each of which is capable of moving independently, the entire set of which defines the particular object. The motion of particles is given by certain rules, but with some degree of randomness. Generally the system is not static, and under certain conditions particles may die and new particles may be generated.

Particles can be thought of a points in 3-Dimensional space, with associated attributes which may change over time. These attributes generally include:

1. Position: the particle's location in 3-D space
2. Velocity: the direction and speed of the particle's motion
3. Size and shape: a description of the geometry of the particle
4. Colour and transparency: typically given as red, green, blue and alpha channel values
5. Lifetime: the amount of time before a particle dies out

Each particle may be represented as simply a coloured dot, as a single textured polygon, or as a complete polygonal model, depending on the application. Reeves proposed that the particles each be considered as a separate point light source, meaning that lighting and depth calculations can be ignored, giving a large saving in computation time.

The example of particles given by Reeves, and the first major instance of their use, was the Genesis Effect from the Lucasarts film Star Trek II: The

Wrath of Khan, where particles are used to model explosions and a wall of fire covering a planet. Physically Modellled particle systems are discussed in section 4.

1.4 Behavioural Systems

Since the publication of the first paper by Reeves, many extensions have been made to the initial particle system model. One of the most important was by Craig Reynolds, who published in 1987 "Flocks, herds and schools: A distributed behavioural model"[16], in which is described a system for modelling large scale decentralised behavioural systems, including flocks of birds, and schools of fish.

In this model, each particle represents, say, a bird, and the entire system of particles represents a flock of birds in flight. Reeves model is extended somewhat to allow the motion of particles to be influenced not only by their internal state, but also by the external state, that of the entire flock. This form of interaction between particles in the system is necessary to ensure realistic flocking behaviour.

Each bird in the virtual flock acts of various impulses determined by the world around it. These typically include obstacle avoidance, and maintaining the overall cohesion of the flock.

The interaction between particles in such a system necessarily makes it more compute intensive to simulate particle motion, however the underlying concepts remain much the same. These systems are discussed in section 3.

1.5 The Constraints of Realtime Simulation

The aim of this project is to produce particle system simulations at *interactive framerates*, making use of distributed rendering and computation to model more complex scenes than the constraints of realtime would normally allow on standard hardware. *Interactive framerates* typically means upwards of 25 frames per second, and the framerate must be approximately constant or have a worst case lower bound around this figure.

Issues to be considered include modelling particle systems in such a way that they can be distributed, the synchronisation issues involved in a parallel

solution, optimisation and load balancing considerations, and the choice of an efficient distributed rendering configuration.

In this report we will describe work on implementing effective parallel solutions for a behavioural flocking particle system, and multiple interacting physically based particle systems.

Chapter 2

Background

2.1 Chromium Parallel API and Parallel Computation

Highly demanding graphical applications generally suffer from one of a set of common bottlenecks on system resources[11]. Applications are *compute limited* if they cannot generate scene datasets fast enough to maintain the desired framerate. This tends to be the case where scenes are very large and computationally complex to update.

Applications are *graphics limited* if the graphics hardware cannot keep up with their requests, e.g. the primitives are very time consuming to render, or involve complex textures, blending, transparency etc.

A program which is limited by the rate at which it can issue commands to graphics hardware is said to be *interface limited*; this is often the case for large static scenes.

Other applications may be *display limited* in which case the display resolution is not sufficient to render the dataset at the desired level of detail.

Using multiple application nodes (i.e. compute parallelism) will benefit applications which are compute limited, whilst multiple server nodes (graphics parallelism) will be useful for graphics limited programs. Interface limitations will be lessened where there are multiple servers and application nodes, and using multiple tiled displays will solve display limitations.

The factors limiting the speed of a particle system will depend on the nature of the system; if the update cost per particle is low, then very large particle systems will tend to be interface limited. In cases where the computation of interactions between particles is expensive, then the system will normally be compute limited. A graphics limited situation can also arise

when the rendering cost associated with a single particle is high.

Chromium provides an API to allow applications to submit commands to rendering servers in parallel. The arrangement of the cluster is largely transparent to the application, and many applications can simply use the standard OpenGL API. Because rendering servers can accept simultaneous commands from several clients, they must operate in an asynchronous manner. To preserve the normal ordered semantics of OpenGL, additional synchronisation is needed, Chromium provides barriers and semaphores for this purpose.

Barriers are typically used before a clear of the frambuffer, or before swapping buffers in a double buffered configuration, to ensure that all applications have reached the same point before such an event occurs, since otherwise graphics data could be lost, or the system could become inconsistent.

2.2 "Embarrassingly Parallel"

If there is no interaction or interdependence between the particles in a system being modelled, then parallelising this system is simply a case of dividing the number of particles to be simulated evenly among the nodes of the cluster. Assuming that the compute and rendering costs are equal for all particles, this will give an optimum load balance, and a speed up approximately linear in the number of nodes in the cluster.

This falls into the set of problems known as "Embarrassingly Parallel", where no synchronisation or complex load balancing is required, and the problem can be implemented in parallel with only minimal modifications and with virtually no performance penalty in doing so.

Problems of this sort will not be discussed in this report, chiefly because there is little to no work remaining to be done in this area and because they do not represent a realistic evaluation of the potential of scalable rendering techniques. Therefore we will concentrate on situations where particles interact with one another, requiring synchronisation and careful load balancing.

2.3 Interacting Particle Systems

Interacting particle systems are those where the next state of a particle is a function not only of its current state, but of those of a number of neigh-

bouring particles, possibly the entire system. When the number of particles involved grows very large, the process of computing the next state for all particles can become very expensive. In such a situation, if an interactive or realtime solution is required (as in our case) then increased computational power (through parallelism) or decreased computational cost (though approximation and simplification) must be investigated. In practise a combination of both approaches is used for very large problems.

As discussed above, particle systems tend to be compute limited or interface limited, therefore a cluster configuration with an approximately equal number of application nodes and server nodes should provide the best performance gain.

2.4 The MPI Standard

As we have already determined that synchronisation among compute nodes will be necessary, a mechanism for this must be chosen. Here we use the Message Passing Interface Standard (MPI)[1] to exchange state information between nodes involved in the parallel computation in order to maintain synchronisation.

MPI allows for a wide variety of point to point and collective communication operations, although we will only make use of a small subset of these. The choice to use MPI greatly simplifies the task of developing a parallel solution as it is a well tested, widely understood standard, which has been implemented efficiently.

2.5 Hardware Setup

Our cluster consists of 3 machines, all with an identical hardware and software configuration. All have Intel Pentium II processors running at 450MHz, 256MB of RAM, and are connected using 100Mb Ethernet. All are running Red Hat Linux 7.3, and we use the MPICH[2] implementation of MPI.[9] The machines have S3 Virge video cards, which do not have hardware acceleration support for OpenGL, therefore in taking measurements of system

[1] http://www.mpi-forum.org/

[2] http://www-unix.mcs.anl.gov/mpi/mpich/

performance it is the relative performance gain from utilising larger numbers of nodes which must be taken into account, rather than an absolute measure such as framerate.

2.6 Parallel Architectures

It is important here to make clear the distinction between the usage of the term particle system in computer graphics and its use in mathematics, although there are many common elements. Within computer graphics, particle systems refers to a group of techniques for rendering loosely structured collections of simple objects, the emergent properties of whose motion gives rise to an apparent behaviour for the system as a whole.

When mathematicians speak of particle systems, they refer to a branch of their discipline which deals with computing the interactions (e.g. forces) among a system composed of independent bodies. It should be clear that there is a clear overlap between these two areas, since in order to render a particle system realistically it will be necessary to simulate in some fashion the motion of the particles, and therefore the forces acing upon them. In many cases however, we are simply interested in rendering a seemingly realistic scene, but computational limitations mean that major simplifications are employed. This is clearly different from the demands of mathematics which typically requires a simulation as close to reality as possible, and where it may be acceptable to perform calculations slower than realtime.

A very typical particle motion system in mathematics is that of the motion of planetary systems, which essentially involves computing gravitational forces among the bodies (planets) in the system. As these systems can grow very large, massively parallel supercomputers are sometimes employed to obtain a solution in reasonable time.[3]

In this report we are dealing with a very different situation, in terms of the aim of the simulation (convincing rendering) and the computational hardware available (a small Linux cluster). Therefore it is important to differentiate our approach from previous particle simulations in terms of several aspects of the architecture of the parallel system.

2.6.1 Tightly vs. Loosely Coupled

The supercomputers used for very large particle simulations (e.g. the Cray T3D), differ from a compute cluster in a very significant way: multiprocessor supercomputers are *tightly coupled*, meaning that they share memory space directly, whereas a multicomputer cluster is *loosely coupled*, each compute node has its own local memory space.

This has direct implications for the type of parallel solution which can be implemented. Essentially the lack of shared memory space[3] means that we must attempt to make the work done by each processor as close to independent as possible. The more interdependence that exists, the greater the amount of synchronisation that will be necessary. This synchronisation takes the form of message passing over Ethernet, and can quickly become very costly if not carefully managed and minimised. In a multiprocessor environment, all communication takes place over some manner of local bus or high speed interconnection network, which will be many times faster than communication over Ethernet.[4]

2.6.2 Scale and Efficiency Considerations

An additional important factor is the scale of the parallel system. Our aim is to produce parallel particle systems which are effective on modest sized clusters. This modest scale consideration means that the solutions must be quite efficient in terms of good load balancing and low time penalties from parallelisation, in order to produce a solution which clearly performs better than the existing serial algorithms.

2.7 Algorithmic Complexity — The n-Body Problem

As stated earlier we will only consider here particle systems in which the particles interact with one another. In such a system, a single particle can theoretically be influenced by any other particle, and in a system with n particles the algorithmic complexity will be $O(n^2)$. Even if a particle is only affected by (say) particles within a certain distance from its location, the task of identifying which, if any, particles lie within this region will still be $O(n^2)$.

[3] In section 6.3 we explore SCI based clusters (which support shared memory) for problems such as these

[4] Issues such as cache coherency on multiprocessor systems can slightly complicate this issue

This becomes very significant in a simulation with $100,000$ particles, for example, since $100,000^2 = 10,000,000,000$(ten billion) which will generally be an unfeasibly large number of possible interactions to consider; if the desired framerate is 25 frames per second then $250,000,000,000$ interactions would have to be computed per second.

This is a famous mathematical problem, known as the *n-Body Problem* (or the *Many Body Problem*), and many approaches have been proposed to overcome it, the key being to reduce the computational complexity from $O(n^2)$ to a more manageable level.

The standard way to achieve this is to subdivide the domain of the particles into smaller regions, and sort the particles according to the regions in which they lie. They speed gain arises since each particle need only be checked for interactions with those within its own region and those in neighbouring regions inside a certain distance.

Note that this type of approach does not necessarily need to be executed in parallel, and a performance gain can usually be achieved on serial systems. With a parallel architecture, the domain of the problem must be somehow decomposed further in order to balance the workload over the multiple processors, and a good load balance is the key to an effective solution.

Chapter 3

Boids — Behavioural Systems in Parallel

Boids is a name given by Craig Reynolds[16] to the particles in a flocking simulation. The particles were originally just birds, thus boid = bird-oid, and this is the term we will use henceforth. To begin evaluation of this type of system, it was necessary to first construct a serial version of a behavioural system. This implementation is closely based on the one described in Reynolds' paper.

In our implementation, a boid is an object, with the following attributes:

- Position [3D Vector Quantity]
- Velocity Vector [3D Vector Quantity]
- Acceleration Vector [3D Vector Quantity]
- Euler Orientation Angles [3 Scalar Quantities]

The number of boids in the system is static, no boids are destroyed, and none are created after the initialisation of the system. To begin with, the boids are randomly distributed within a sphere. The environment may contain static objects which act as obstacles which the boids attempt to avoid.

During each timestep of the system, each boid's behaviour is influenced by other boids nearby, and by the surrounding environment. The behaviour of the boids is given as a set of impulses (movement vectors), the weighted average of which given the new acceleration vector for the boid. To model the inertia of the boid, the new velocity vector is a linear combination of the previous velocity vector and the new acceleration vector.

The impulses acting on a boid are:

- Collision Avoidance: avoid collisions with obstacles and other boids
- Velocity Matching: attempt to match velocity with nearby boids
- Flock Centring: attempt to stay close to nearby boids

Several constants determine particular elements of boid behaviour:

- Sight Radius: the distance at which the boids can see other boids and obstacles
- Sight Angle: the field of view angle of the boids also determines what is visible to them
- Boid Avoidance Distance: the optimum distance which boids attempt to maintain from their neighbours
- Obstacle Avoidance Distance: the distance at which boids will attempt to steer away from an obstacle

The outline algorithm for determining the new states for all boids on a timestep is:
Compute the distances between all boids in the system, and for boids which are within the boid sight radius of one another, compute whether they can see one another according to the boid sight angle
FOR every boid DO

1. Compute the average velocity vector for all boids visible to this one
2. Compute the average position of all boids visible to this one, and compute a vector from the current boid to this point
3. For all neighbouring boids inside the boid avoidance distance, compute a *repulsion vector* as sum of vectors directly away from these boids, weighted by the inverse of the distance to the current boid
4. Compute a repulsion vector for static obstacles in a similar fashion
5. Compute the acceleration vector as a weighted average of these impulses

END

3.1 Previous Parallel Implementations

A simple parallel implementation of a flocking simulation[13] requires that all the processors in the parallel system be connected in a ring structure. Every processor is then assigned a particle, or group of particles. At every step of the computation, each processor receives information about a set of particles from its left neighbour, uses this to compute the next state for its own set of particles, and then sends this data to its right neighbour. If there are p processors, then $p-1$ communication steps are required, and the number of particles, n, will be divided such that every processor is assigned $\frac{n}{p}$ of the total.

The performance of this algorithm is very much dependent on the number of processors and the costs of the communication involved. If the processors are loosely coupled, then the costs of sending data on every particle around the entire network of processors may be very high. Additionally, a large number of processors will be required in any case to attain a reasonable speed gain. As we are looking to produce a solution that is effective on small clusters this approach is not suitable.

3.2 Binary Tree Domain Decomposition

As stated previously, a key element in reducing the time complexity of this computation is to divide the domain of the flock[2]. The Barnes-Hut Method[1] which was designed to simplify force calculations in simulating interactions of galactic bodies, uses a hierarchical octree representation of space, where each node contains a simplified representation the approximate total forces of all of its children.[1]

For our initial simulation, we are looking simply to subdivide the domain, and will not consider simplifications. Since we are dealing with small clusters, an octree representation is not ideal, as it quickly produces a very large number of regions[7]. Thus, it was decided to use a *binary space partition* representation for the space occupied by the flock.[2] The leaf nodes in the

[1]Each node contains the total mass and centre of mass of all the particles it contains, this so-called *monopole* approximation can be used to approximate the total force acting on distant particles

[2]A binary space partition is formed by cutting space by an n-dimensional hyperplane, then recursively partitioning each of the two resulting halfspaces. The result is a hierarchical division of space into convex regions. For more information see http://www.faqs.org/faqs/graphics/bsptree-faq/

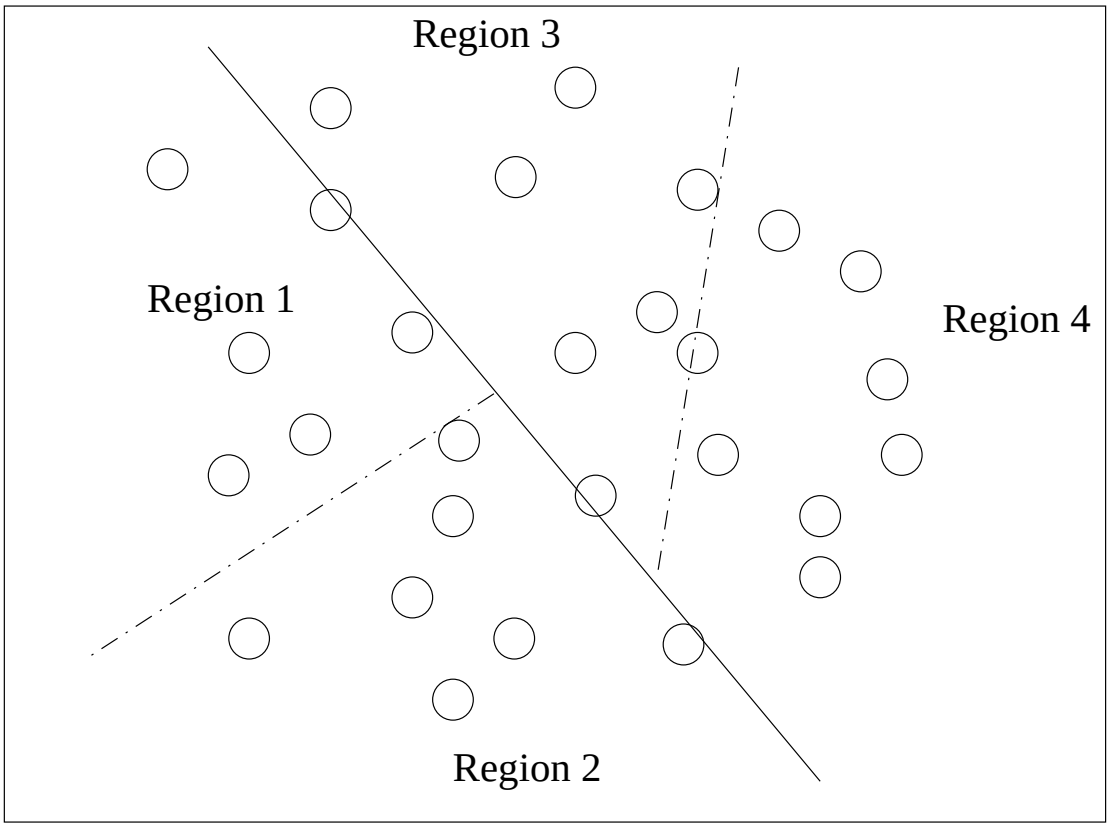


Figure 3.1: A 2D view of a binary space partition of a particle set into 4 disjoint regions. The solid line gives the split plane at the root node, the dashed lines are the two split planes of the nodes at the next level down

resulting tree will be the actual regions containing boids, therefore the depth of the tree will determine the total number of regions.

In a distributed environment, each machine[3] will be responsible for some set of nodes, and since it may be possible for boids to see other boids in regions managed by other machines; synchronisation mechanisms will be needed to ensure that all machines operate on up-to-date data. Synchronisation is also necessary to maintain the structure of the BSP as the flock moves. The overall BSP tree is distributed across the cluster, all machines will maintain a copy of nodes near the root of the tree, whilst nodes near the leaves may only be stored locally on individual machines. With this arrangement it is not necessary for every machine to have knowledge of the entire tree, only the *locally significant* nodes of the tree must be stored.

To simplify this representation, we add the constraint that if any machine is responsible for more than one region, then all the regions it manages must be children of the same node, and this node must have no children managed by other machines. We call this node the *local root* for this machine. Other machines in the cluster need only have knowledge of the local root of this machine, and perform any interactions with this machine via this node.[4]

In our implementation, there can be 3 types of nodes in a tree:

Branch Node a non-leaf node that may be local only or shared with all machines

Local Leaf Node a leaf node that is local to this machine

Remote Node a node that is the local root of some other machine

3.3 Region Splitting

The division process used to divide regions in the BSP tree must be carefully chosen to ensure an even load balance between both sides of the split. Additionally the process of choosing a split should be relatively fast since the region tree will need to be updated every frame of the simulation.[5]

[3]Note: to avoid confusion here, node refers only to a node in the BSP tree, and machine refers to a single machine in the cluster

[4]Conceptually this is because any node in a binary tree represents exactly the union of all the space occupied by its children

[5]A situation where the subdivision of the regions is static will quickly degenerate into a very unbalanced state as the boids move, since they tend to move together and would all cluster in a small subset of regions

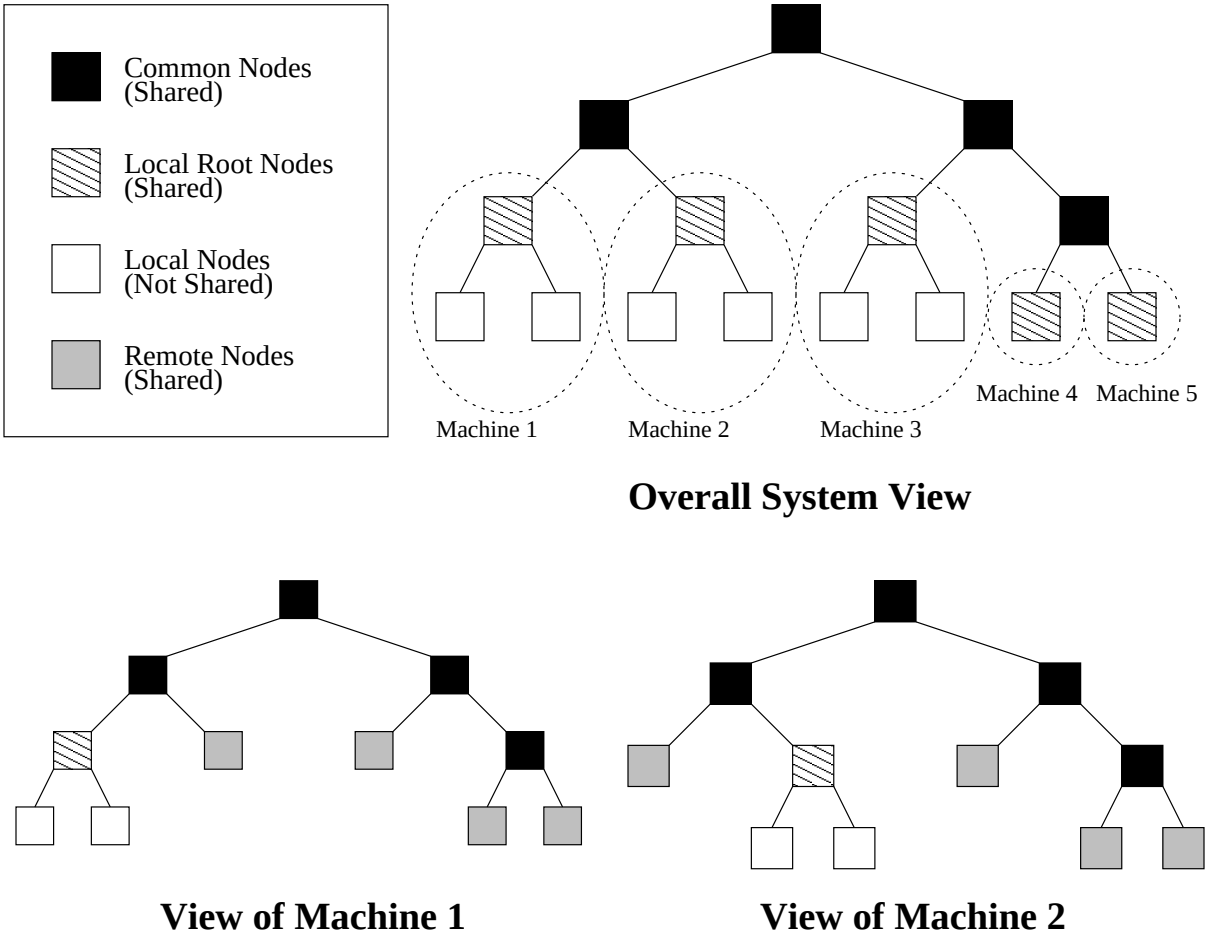


Figure 3.2: The topmost tree is the complete view of the system, below are shown the views of machine 1 and machine 2. Note that every machine interacts only with the local root nodes of the other machines

Here we will will treat the boids as 3-Dimensional vertices; there are many ways in which the domain of a problem such as this may be divided[17][6], however many are intended for graph structures where the vertices are connected by edges; an unconnected situation is considerably simpler.

3.3.1 Planar Bisection

The simplest method for dividing a set of vertices is known as recursive coordinate bisection(RCB). It is easy to implement and is relatively fast. The process is as follows:

1. Determine longest expansion of domain (x, y or z direction)
2. Sort vertices according to coordinate in selected direction
3. Assign half of the vertices to each subdomain
4. Repeat recursively (divide and conquer)

This approach is not ideal for our requirements, however, as it does not give us easy access to a hyperplane separating the two new subdomains. This is important for the flocking system as it will become necessary to test whether a boid has crossed such a hyperplane and entered a new region. Additionally, the need to sort the vertices means that the algorithm is $O(n \log n)$ could become very costly for large numbers of vertices.

An intuitive alternative is to take the mean of the coordinate values of all vertices, and construct a plane which passes though this point and is normal to the axis vector of a chosen dimension. This plane can then be used to partition the vertices. This does not guarantee to provide an even balance; consider the following example:

- let V be a set of vertices longest in the x dimension
- let X be the set of x coordinates of these vertices $= \{1, 2, 4, 21\}$
- $\bar{X} = (1 + 2 + 4 + 21) \div 4 = 7$
- Dividing the set at the point 7 will give two sets containing 3 and 1 elements respectively

Despite this shortcoming, we will use this form of evaluating splitting planes for regions, since it can be computed in $O(n)$ time. If the boids are distributed approximately evenly, then the amount of imbalance will be quite low. There is also a further property, the advantages of which will become clear later — all that is needed to compute a split plane is the sum of the coordinates of all boids in a region, and the number of boids that are present. If the determination of splitting planes is carried out bottom up, i.e. starting with the leaf nodes, then computing the split plane for a branch node is simply a matter of summing the information already computed for its children.

This is vital to the efficiency of a parallel solution, as it means that the only information which must be synchronised between machines in order to rebuild the BSP tree is the sum of the coordinates of all boids and the number of boids present, for every local root node. Since each machine has exactly one local root, the amount of message passing for this synchronisation step is linear in the number of nodes present, and independent of the number of boids in the system.

There is a further refinement to this process, which requires slightly more information, but has other desirable properties. To justify this, consider a situation where all the boids in the system are in a long line in one dimension.[6] If we split the domain along this dimension (see figure **3.3**), the boids will be divided evenly, however the distribution of boids will be "unstable" since small movements of boids will result in them crossing the splitting plane.[7]

Therefore an improved solution would be to split using a plane orthogonal to the longest dimension, as with RCB, which will eliminate the instability as there will be less boids in the immediate vicinity of the split plane. A further problem can arise, however, if the boids begin to move into an arrangement where the flock is oriented along a different dimension. At the point where the flock shape moves to being longer in the new dimension, the split plane will be reoriented orthogonal to this new dimension, and the resulting distribution of boids among the regions will likely be very different from the preceding arrangement. Again a large amount of migration between regions will result, which could cause irregular framerates when such a situation arises.

What is required is a method for choosing a splitting plane, which can be computed from the minimal amount of data, produces a reasonable and

[6]This is not an unrealistic situation, since birds often fly in such a formation

[7]This will be seen to be undesirable when we discuss boid migration

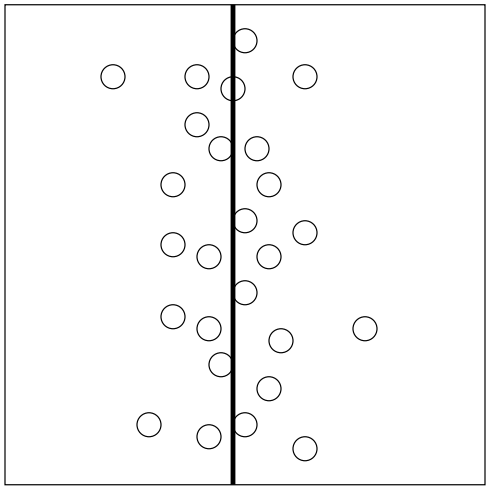


Figure 3.3: A set of boids has been split along the Y axis, leaving many in a position where slight changes in position would cause them to cross the split plane

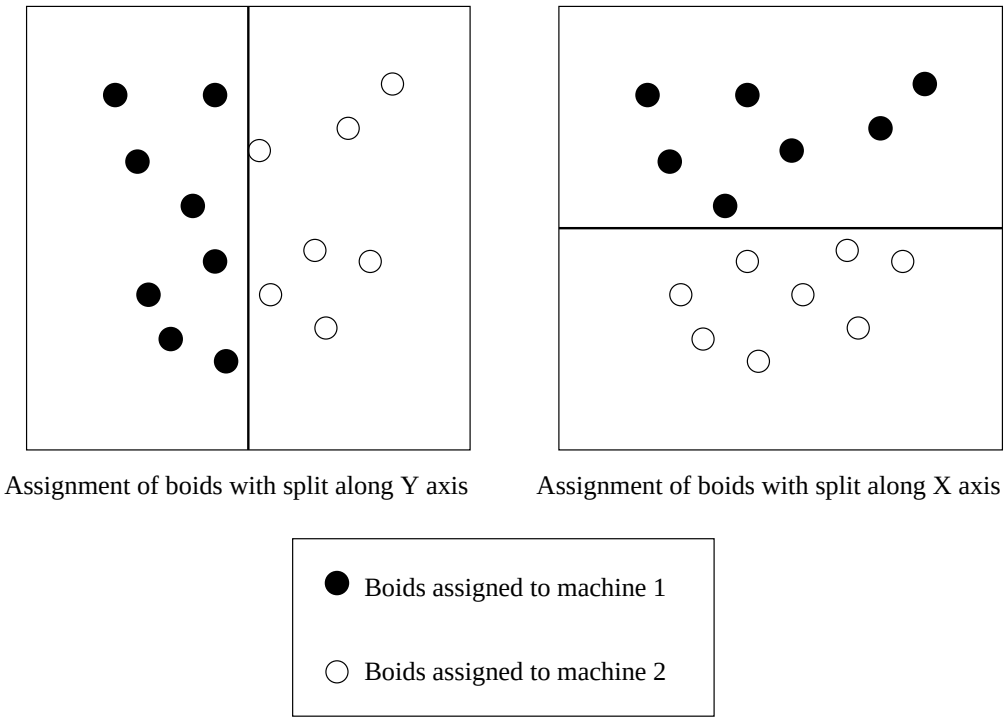


Figure 3.4: The distribution of boids between two machines shown over a period of two frames illustrates how changes to the axis chosen to divide the flock could cause many boids to migrate to another region

stable division of boids, and changes continuously as the flock moves. Here we will use a plane based on the coordinates of a bounding box surrounding a set of particles. The plane is defined as passing though the mean of the position of all the the boids, and orthogonal to a vector between two opposite corners of the bounding box. Choosing a different two corners of the bounding box to form plane normal will give an alternative splitting plane for the set of particles. Since we are splitting the domain recursively, the subdivision will be most effective if choice of which plane to use when splitting is cycled as depth in the tree increases, such that planes at adjacent levels in the tree meet with an angle of approximately 90 degrees. Since the coordinates of the bounding box of a set of boids can only move as fast as any given boid, this splitting plane will move continuously with the flock as we required.

3.4 Boid Sharing and Migration

The division of the domain gives rise to several issues which must be considered when boids interact with a neighbouring region. Firstly, boids may *migrate* to a neighbouring region if they cross one of the dividing split planes. These boids must be transmitted to their new region, and removed from that which they previously occupied.

A second issue is sharing of boids between regions, since a boid's sight radius may well extend into regions outside its own. If we do not employ any simplifications, then any boid which is within the boid sight radius of a split plane must be shared with the region on the other side of that plane. It may be the case that the region on the other side is itself composed of subregions, and the boid will then need to be shared with one or more of these.

In both cases, migration and sharing of boids, a boid may need to be transmitted to a remote machine. This will be discussed as part of the synchronisation mechanisms.

3.5 Load Balancing

Load balancing this system is relatively simple, if it is assumed that the approach to region subdivision employed is reasonably good. Since only the leaf nodes of a tree contain boids, the depth of a tree will be determined by

the number of machines in the cluster, such that there is at least one leaf node per machine.

A binary tree of depth d has 2^d leaf nodes. Therefore a cluster with m machines will need a tree of depth $d = \lceil \log_2 m \rceil$. The number of leaf nodes in such a tree will be 2^d which may be larger than m. In this case it will be necessary for some machines to manage more than one region. A problem would arise if it were necessary for a machine to manage, for example, 3 nodes, since we earlier imposed the constraint that all the nodes on a given machine must be the only children of a single parent node, thus the number of nodes on any machine must be a power of 2. We will show here that this will always be the case, and that furthermore it will never be necessary for a machine to manage more than 2 nodes.

Proof Let m = the number of machines in the cluster. Let d = the tree depth required = $\lceil \log_2 m \rceil$. Let n = the number of leaf nodes in the resulting tree = 2^d. Therefore $2^{d-1} < m \leq 2^d$. If one machine were to manage 3 nodes, then this would imply that all other machines were already managing 2 nodes each. In this case, the total number of nodes managed by the cluster would be $s = 3 + 2 \times (m - 1)$. Since $2^{d-1} < m$, we have $m - 1 \geq 2^{d-1}$, and $s \geq 3 + 2 \times 2^{d-1} = 3 + 2^d > n$. Therefore more nodes are being managed than exist in the tree, therefore there is no need for any machine to manage 3 nodes.

3.6 Synchronisation

The process of updating this system involves several steps, all performed in parallel across all machines.

Update boid positions the new acceleration, velocity and movement vectors for all boids are computed. This process is as described for the serial case

Compute local bounding regions the average position and the bounding box extents for every local region is computed bottom up

Synchronise bounding regions all machines pass the bounding region info for their local root nodes to all other machines. This is implemented as an MPI collective communication operation (MPI_Allgather), which

gathers data from each machine and distributes the complete set to every machine

Compute shared bounding regions the bounding region information is computed for all shared regions, again bottom up. Note that since all machines will have the same shared regions, this step will be identical on all machines

Compute split planes and boid migration and sharing this step is performed top down. The root node computes a new split plane based on its bounding region information, and tests the boids in all of its children against this plane. Once this is complete, the children of that node perform the same process recursively. If sharing or migration of boids is detected, the boids must be inserted into the new region. This region may of course be composed of subregions, so the process is again recursive, and boids are tested against split planes until the correct leaf node is identified.

The result then depends on the type of node the boids in question interact with. If the node is local to the current machine (i.e. the boids are moving from a local region to another local region), then the boids are transfered directly. If the node is a remote node (i.e. the boids are interacting with the local root of a different machine) then information about the boids must be buffered for transmission to the remote machine, and in the case of migration, the boids are marked as having migrated, but are not yet removed from their previous region

Synchronise migration and sharing in this step, the information on boids which have migrated, or which must be shared is transmitted between machines and cached. The details of this operation will be discussed under communication costs

Rendering all boids are rendered using their current state information

Apply migration and sharing the boids which have been received as having migrated or been shared from remote machines are inserted into the local trees on each machine

Remove migrated boids boids which have been marked as having migrated to a remote machine are removed from their previous machine

3.7 Reducing Communications Costs

3.7.1 Message Sizes and Efficiency

In a realtime situation such as this, latency is critical, therefore the costs of message passing should be reduced as far as possible. The first consideration is the size of the messages involved, and for our application this is quite simple to optimise. The performance measurements for the MPICH implementation of MPI[8] indicate that it is considerably more efficient to pass a small number of large messages rather than many smaller ones. For example, on the systems that were used for the performance study[8], sending a 200 byte message took approximately $1650\mu s$, while sending a 400 byte message took approximately $1750\mu s$, clearly better than the cost of two 200 byte messages.

This means that when transmitting information on migrated or shared boids between machines, it is preferable to pack all the information for a point to point transfer into a single buffer and send it as one message, as this will give a considerable speed gain over sending a single message for each such boid. Therefore, every node maintains an ingoing and an outgoing communication buffer for every other node, and during every frame of the simulation, when it is determined that a boid need be sent to a remote node the pertinent information (position,velocity) of the boid is packed into the appropriate buffer, to be sent during the synchronisation phase.

3.7.2 Point-to-point communication

This synchronisation phase (migration and sharing) requires every machine to send to every other machine a (possibly empty)[9] buffer. Collective communication operations would be cumbersome here, since machine A may want to send different data to machine B than it does to machine C, and the data is stored in separate buffers.

Clearly in this situation the total amount of message passing will be proportional to the square of the number of machines in the cluster. Ideally, if at all times during this phase, every machine were sending or receiving a message, then the total number of message passing steps required would be

[8]Two Sun SPARCStations connected by Ethernet, a situation comparable to ours

[9]Buffers must be sent even when empty, since communication is performed synchronously and otherwise the remote machine would stall on a blocking receive

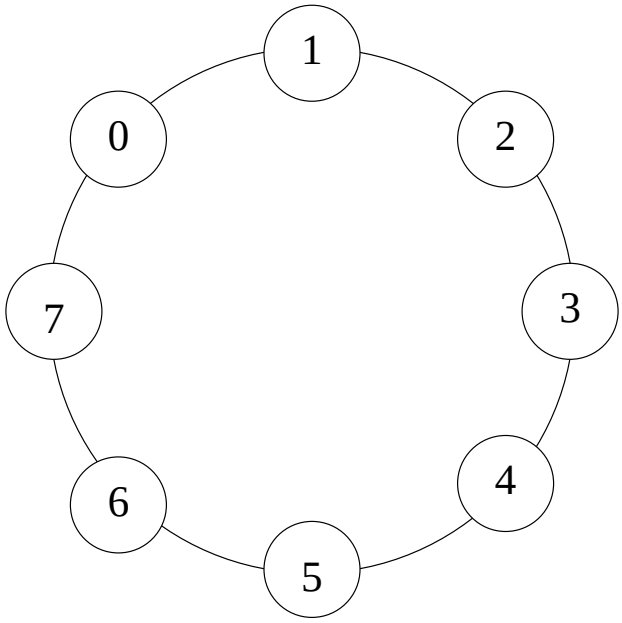


Figure 3.5: Machines of a cluster with their MPI IDs, viewed in a circular arrangement.

linear in the number of machines. A simple algorithm was devised to achieve this, the central concept is that of *distance* of message passing. To picture this, consider n machines $M_0 \dots M_{n-1}$ arranged consecutively according to their MPI ranks in a circle. The distance between machines M_a and M_{a+1} is 1, the distance between machines M_a and M_{a+2} is 2 etc. Note that M_{n-1} is considered adjacent to M_0.

```
FOR all machines DO
1. rank ← the rank of this machine
2. size ← the total number of machines
3. distance ← 1
4. maxDistance ← size ÷2
5. WHILE distance ≤ maxDistance DO
6.    nextNeighbour ← (rank + distance) % size
7.    previousNeighbour ← (rank - distance + size) % size
8.    IF (rank % (distance ×2)) < distance THEN
```

```
 9.      Send/Receive data with nextNeighbour
10.      IF nextNeighbour ≠ previousNeighbour THEN
11.        Send/Receive data with previousNeighbour
12.      ENDIF
13.    ELSE
14.      Send/Receive data with previousNeighbour
15.      IF nextNeighbour ≠ previousNeighbour THEN
16.        Send/Receive data with nextNeighbour
17.      ENDIF
18.    ENDIF
19.    distance ← distance + 1
20. ENDWHILE
END
```

The essence of this process is that all nodes at a distance of 1 from one another exchange data, followed by those at distance 2 etc. until all nodes have had a chance to exchange data. The data transfers are implemented with the MPI_Sendrecv operation, and are synchronous, therefore we ensure that the order in which nodes communicate with their neighbours is defined using an alternating pattern, the period of which grows with the distance of the transfers.

3.8 Time complexity

Before any performance evaluation is carried out, it is useful to estimate the time complexity of this algorithm when compared to the $O(n^2)$ serial solution. The basic step of computing boid positions is still $O(n^2)$, however this will involve now only the smaller number boids in each region, plus a small number of shared boids from neighbouring regions. If the total number of boids is n, and there are r regions then the time for this computation will

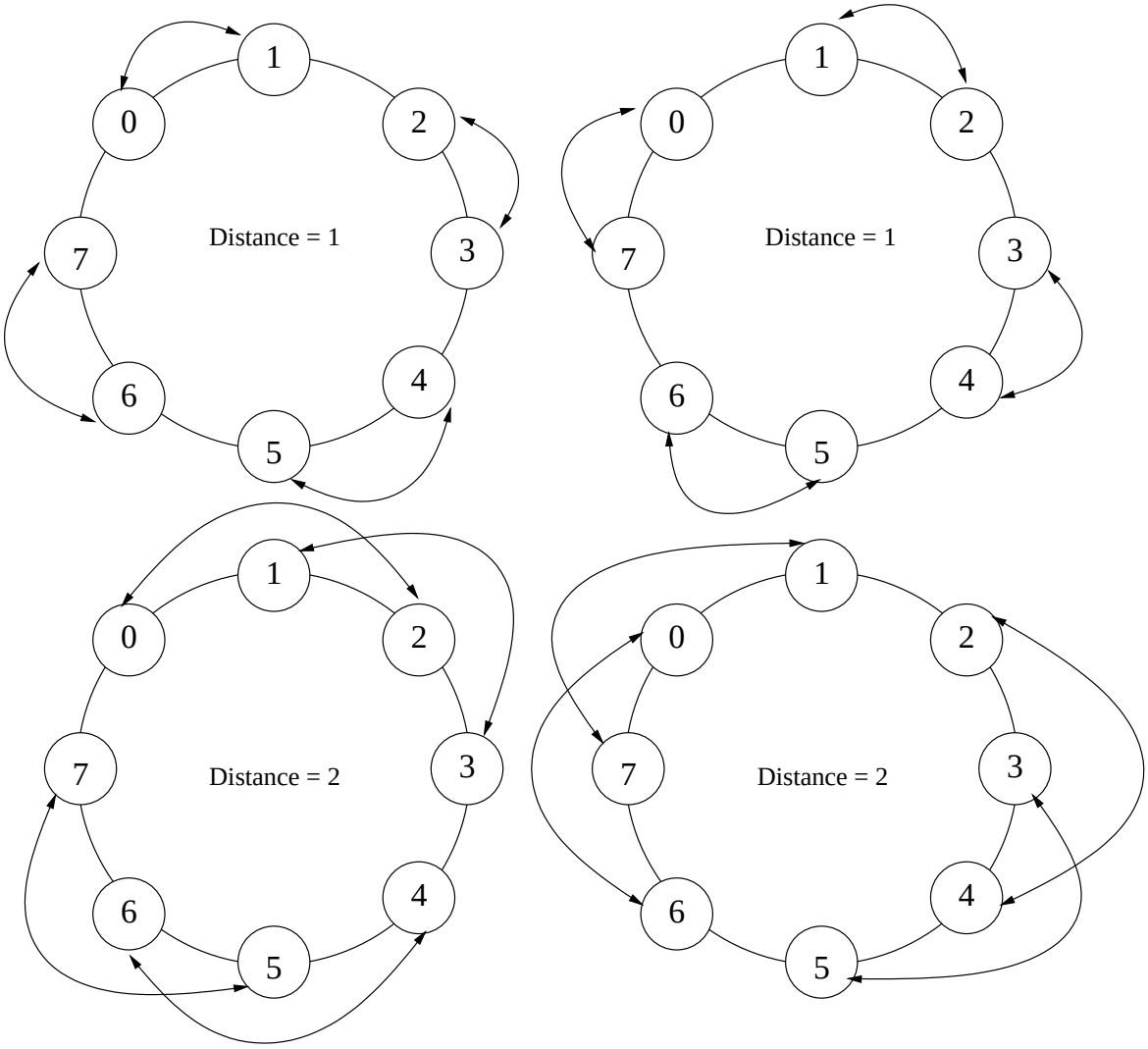


Figure 3.6: 2 distance steps in the point-to-point communication process. The topmost diagrams show the two stages for nodes at a distance 1 apart, the lower diagrams show the sequence for nodes at distance 2.

be $O((\frac{n}{r})^2)$. The time to update the bounding region information is $O(r)$ and the time to compute the interactions of boids with other regions is $O(n \log r)$. Since the number of regions, r, is proportional to the number of machines (m), the total complexity is $O((\frac{n}{m})^2 + m + n \log m)$.

This is not a realistic performance measure, since it ignores the costs of message passing. In section 6 we present performance measurements and an an evaluation of the performance of the parallel solution.

3.9 Simplification and Approximation

If instead of sharing information on individual boids between adjacent regions, we distribute only the averages of the properties of those boids in each region amongst machines, then we can obtain a considerable reduction in communication and computational costs. Much of this information is shared already to perform dynamic domain decomposition.

To use this approximation, every boid is influenced by those in its own region, and by the approximate representation of other nearby regions. The effect of the averages of other regions is scaled appropriately to reflect the number of boids in each region, to closely model the net effect of all boids in those regions. To increase the accuracy of this method we can subdivide regions further than the minimum amount given by the number of machines in the cluster. As shown in section 3.5, it is possible to produce a tree and distribution for any number of machines such that every machine manages 1 or 2 leaf node regions. Since any region can be divided into two children, the regions belonging to any machine can be subdivided into any power of 2 number of subregions.

There are some similarities between this approach and the Barnes-Hut algorithm mentioned earlier, specifically the use of a simplified representation for the particles in distant regions. The Barnes-Hut method, however, uses a *monopole acceptance criteria* to determine whether the approximation is satisfactory for a particular region, and if it is not, expands the region and examines its subregions, or individual particles. The monopole acceptance criteria compares the ratio of the dimensions of the region to the distance of the particle from the centre of mass of the region.

This approach is unsuitable for our situation since expanding a remote

region and examining the particles inside would require a request-response message passing transaction, and the response would typically be quite large, so the efficiency of the solution would be poor.[10]

[10]The Barnes-Hut method is effective on large multiprocessor systems, for example an nCUBE

Chapter 4

Physically Based Systems in Parallel

As an alternative to behavioural systems, we present here a discussion of particle systems based solely on physical laws. These particles do not interact except when they come into very close proximity of one another, where they may combine to produce particles of a different kind.[19] This means that the motion of a particle is unaffected by those around it, but that these neighbouring particles can potentially react with it, annihilating the original particles and producing particles of a different type.

To describe such a system we use the concept of several separate *particle sources* within a common *particle system*. Within each particle source, the particles share common appearance and behaviour, which is also subject to random variation. A set of rules is provided to the system, which govern which particle sources may interact, and what the result will be.

4.1 Stochastic Modelling of Interactions

A naive implementation of the arrangement just described would suffer from the same $O(n^2)$ performance problem as the behavioural systems, however our aim is to avoid this if possible. A possible alternative to doing distance calculations between every pair of particles is to model interactions stochastically — rather than considering interactions between individual particles, to operate on probabilities. Thus if there are A particles of type α and B particles of type β, and they can combine in a 1:1 ratio to produce particles of type γ, then assuming they are all within a region of space such that they may all interact, the expected number of particles of γ produced can be determined based on the probability of the above combination event occurring.

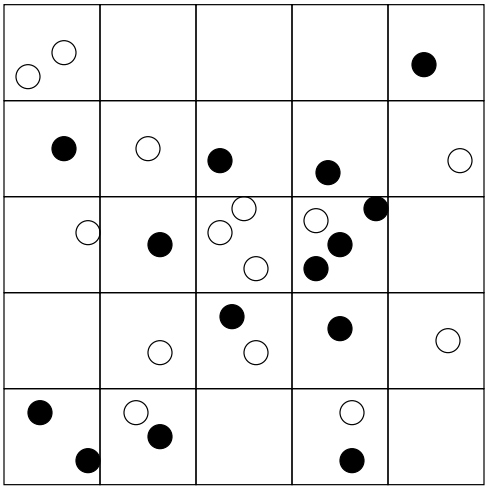


Figure 4.1: Two types of particles sorted into regular sized static buckets

4.2 Bucket-Sorted Regions

The advantage of this approach is that it once again allows us to subdivide space to reduce the scale of the problem. Here we will take a different approach to spatial subdivision, and divide space regularly into equal size *buckets*. Each bucket keeps track of the number of particles of every type that it contains in the current state of the system, and uses this information to derive the expected values for the next state.

Each particle in the system maintains a record of its current bucket, and when its position changes it moves into a new bucket, the relevant particle counts for these buckets must be modified to reflect the change, and similarly for the birth and death of particles. This is more efficient than recalculating the numbers of particles in each bucket at the end of every frame. After a new state is computed for the system, particles will be created or destroyed probabilistically. If, for example in bucket b there are currently 10 particles of type α and none of type β, and the next state counts dictate there should be 5 α particles and 5 β particles, the system will create 5 new β particles at random positions within the bucket, and will then randomly destroy α particles with probability 0.5.[1]

[1]Note that the probabilistic nature of this operation means that the actual number of α particles which die may not be 5 every time, but on average the expected number will be destroyed

2,0	0,0	0,0	0,0	0,1
0,1	1,0	0,1	0,1	1,0
1,0	0,1	3,0	1,3	0,0
0,0	1,0	1,1	0,1	1,0
0,2	1,1	0,0	1,1	0,0

Figure 4.2: The particle counts of the buckets in figure **4.1**, given as ordered pairs A,B where A = *Hollow Particles* and B = *Filled Particles*

4.2.1 Performance Tradeoff

The running time of this algorithm is strongly determined by the number of buckets in the system. If there are 128 buckets in every dimension, then there will be $128^3 = 2097152$ in total, and the time complexity is proportional to this figure. If this exceeds the total number of particles, the speed gain from modelling the system this way will be very poor. The other extreme, a small number of buckets, will be very efficient but the visual result will be highly unconvincing. Therefore there is a tradeoff between speed and the visual result, and the bucket count can be adjusted to attain a desired framerate.

4.3 Describing Interactions

We use a simple method to describe the particle sources and possible interactions within a system. Each source is given a name, and a type which will be taken from an (expandable) set of different sorts of particle behaviour and appearance. The rules used to describe interactions are restricted to being ternary; two particle sources interact to produce a third, in a defined ratio. An example of such a rule is **1[green] + 1[red] = 2[blue]**, the meaning is obvious — one particle from the source named **green** interacts with one from the source named **red** to produce 2 particles of the **blue** source. Complex behaviour within a system can be obtained by specifying multiple such rules.

4.3.1 Dealing with Non-Determinism

Non determinism could arise in this system if two rules affecting the same particle sources are both applicable to the state of a bucket at some timestep. To make this clearer, imagine three particle sources, **A**, **B** and **C**, and two interaction rules, $\mathbf{1[A]} + \mathbf{1[B]} = \mathbf{1[D]}$ and $\mathbf{1[A]} + \mathbf{1[C]} = \mathbf{1[E]}$. Any particle of type **A** could potentially react with one of type **B** or type **C**, so it is unclear what proportion of particles should participate in each reaction.

To address this issue, the approach of using Linear Programming[5] was examined, but quickly discounted, for the reason that an integer solution would be required, and the calculation of such a solution using Integer Programming is computationally expensive, and would detract from the effectiveness of this method of computing particle interactions.[2]

We use a simple approach to divide particle counts among different interaction rules, which will be seen to be advantageous when a parallel solution is considered. When updating then next state of a particular bucket, the set of rules which could possibly be applied (based on the presence of particles) is computed, and particle counts are divided evenly among those rules which affect them and can be applied at this timestep.

4.4 Implementation in Parallel

In the previous consideration of a parallel solution to a behavioural system, it was decided to divide the domain of the system among the various machines. Here we will use a different approach, and will divide the particle sources and rules among the machines, and give each machine a representation of the entire domain in terms of the particle counts in the buckets. The aim of this is to minimise the amount of message passing by exchanging only the summary information on total particle counts, and not the exact positions of the particles themselves. Every machine will then manage a set of particle sources and any rules which can *create* one of these sources. Since these rules may require information on particle sources on remote machines, this information will need to be exchanged between machines.

[2] Additionally, whilst the constraints in this system are simple to derive, it is not obvious which quantities should be minimised.

4.4.1 Synchronisation

A summary of the steps involved on each timestep of the system follows:

Synchronise Particle Counts Machines exchange information on particle counts per bucket

Update Bucket States Every machine applies the rules it manages to every bucket to compute the next state for the system

Remove Dead Particles Particles are destroyed probabilistically

Generate New Particles New particles are generated stochastically

Update Particle States The state of every particle source (postions and motion of particles) is updated by the machine managing it

4.5 Issue — Volume of Message Passing

In a similar way to the computation time for the system state, the volume of message passing in this system will increase rapidly with the number of buckets in every dimension, and with the number of types of particle sources being synchronised, for this reason they should be minimised as far as possible.

4.5.1 Optimising Distribution of Particle Sources

The number of particle sources whose counts are being exchanged among machines will have a large impact on communication time. If there are 4096 buckets, then each additional particle source will increase the number of data items to be transmitted by 4096.

It is not necessary that the particle counts for all sources be distributed, since some sources may not interact with any outside of their local machine, and it is conceivable that there may even be no such sources, making message passing unnecessary. This allows us to limit the number of sources which are "shared" and obtain more efficient communication.

Unfortunately, obtaining a distribution of particle sources and rules among the machines in a cluster that both gives a good load balance and minimises communication is not straightforward. This problem has many solutions if there are more than a trivial number of particle sources and machines, and

finding the optimum distribution might at worst require a search over the entire solution set.

For the purposes of this project, an algorithm was devised that chooses this distribution in a deterministic way, in polynomial time, although it may not produce an optimal solution. To begin with, a *dependency tree* is constructed, which describes the interactions between particle sources in terms of their dependencies on one another. If a rule $\mathbf{1[A]} + \mathbf{1[B]} = \mathbf{1[C]}$ exists, then **C** is dependent on **A** and **B**, which may be themselves dependent on other sources, leading to a unconnected directed graph structure of all dependencies. We impose the further constraint that each node must have out-degree ≤ 2 (i.e. each source may have only 2 direct dependencies), and that the graph be acyclic. If a system is required that would produce a graph violating one of these constraints, the graph can be made conformant by duplication of one or more particle sources to break the cycle or divide the number of direct dependencies of a particular node among two or more sources with identical behaviour.

The algorithm to divide this tree among the machines works by attempting to isolate subgraphs which have few dependencies on nodes in the rest of the graph, and assigning such a subgraph to a single machine. Load balancing is achieved by limiting the size of these subgraphs to ensure that all nodes get an approximately equal number of particle sources.[3] Each node in the tree contains a record of its total number of *dependencies* (nodes it depends on), and *outside dependants* (nodes which depend on it which are currently not allocated to the machine in question). The algorithm to allocate to a machine a subgraph of a particular size proceeds as follows:

1. Choose from the tree an unallocated node with the largest number of dependencies, and allocate it to the machine

2. Recursively decrease the number of outside dependent nodes count for all children of this node (since it is now allocated to this machine)

3. Add any unallocated immediate children of this node to a set of descendants for the root of this subgraph

4. If the descendant set is empty or the size of the subgraph equals the one specified, go to step 7

[3]This assumes that management of all particle sources involves the same computational cost

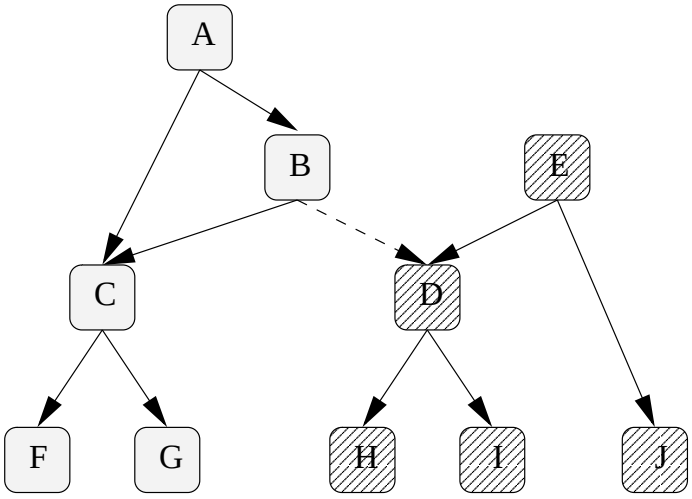


Figure 4.3: A dependency tree for 10 particle sources, divided in an optimal fashion between two machines; arrows show dependencies and the dotted arrow shows dependencies that will require passing of information between machines.

5. Choose from the descendant set an unallocated node with the least number of dependent nodes and allocate it to the machine
6. Go to step 2
7. If the size of the subgraph is not yet large enough go to step 1, otherwise the algorithm terminates

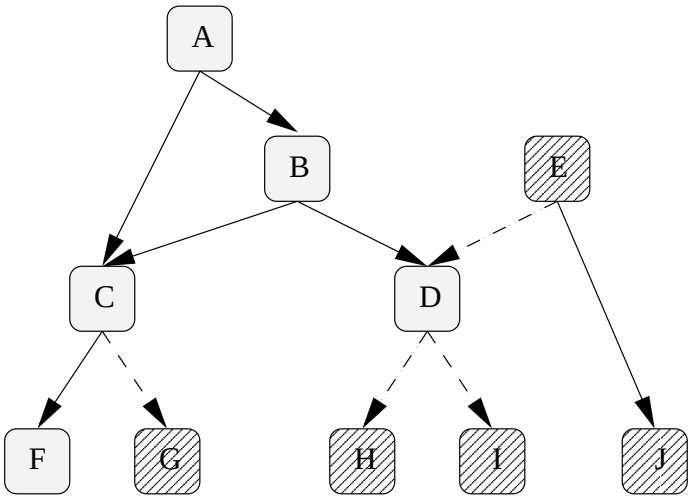


Figure 4.4: A non-optimal division of the tree in figure **4.3**. Note that it only differs in the assignment of two particle sources, but there are now 4 dependencies that will need synchronisation between machines.

Chapter 5

Distributed Rendering

Here we present a concise overview of distributed rendering techniques, with particular reference to how they are implemented in Chromium.

5.1 Conceptual Overview

A description of the arrangement of a distributed rendering cluster was given in section 1, which consisted of application nodes submitting rendering commands to server nodes which perform the rendering work. In a situation with multiple server nodes, a mechanism for distributing the rendering workload over the servers is required. Because the rendering servers use commercial graphics hardware, it is not possible to modify the graphics pipeline, therefore any steps to achieve parallelism must occur before or after the standard pipeline.[11] Methods which divide primitives before they enter the graphics pipeline are known as *sort-first* methods, while those which combine multiple scenes at the end of the pipeline are called *sort-last* methods.

5.2 Overview of Tilesort

Tilesort is a sort-first distributed rendering approach, where every server manages a rectangular portion of the viewport, rendering any primitives or parts thereof which project to this section of the screen. As applications submit primitives to the servers, the projected bounding boxes of these primitives are used to determine which tile they will occupy, and accordingly, to which rendering server they must be sent. This process is largely transparent to the application, being managed by SPUs on the application node.

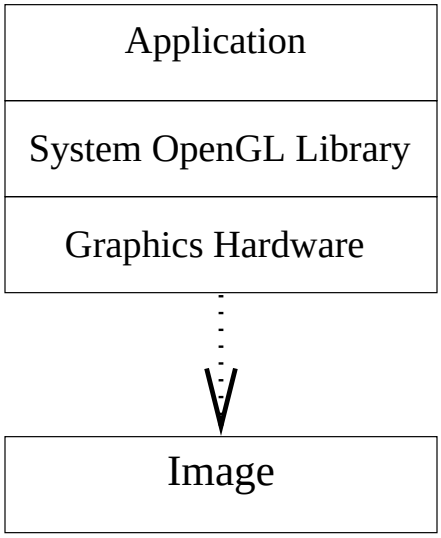


Figure 5.1: Standard Serial Rendering Process

The Chromium *Tilesort* SPU acts as a sort-first renderer by managing tile layouts, and performing the bounding box checks to determine to which tile primitives will project. These primitives are then sent to the servers as with the *Pack* SPU. The *Render* SPU on the server nodes also has knowledge of which tile it manages, which it uses to perform clipping and other transformations on the primitives it receives.

In a rendering configuration with multiple monitors, or multiple projectors (e.g. a CAVE), tilesort is a natural approach to distributing the rendering workload.

5.3 Overview of Z-Compositing

In order to perform sort-last rendering, the images produced by a number of renderers must be combined to create a composited image. Z-Compositing performs this operation using the Z-Buffer, which is an array containing depth information for every pixel in the viewport.[1] When multiple images are composited, the Z-Buffer information is used to select the pixel to appear in a location to be the one closest to the viewer.

To create a sort-last rendering configuration in Chromium, every render server contains the *Readback* SPU. This SPU renders the primitives it re-

[1] Alternatively, Alpha channel (translucency) values may be used to composite images

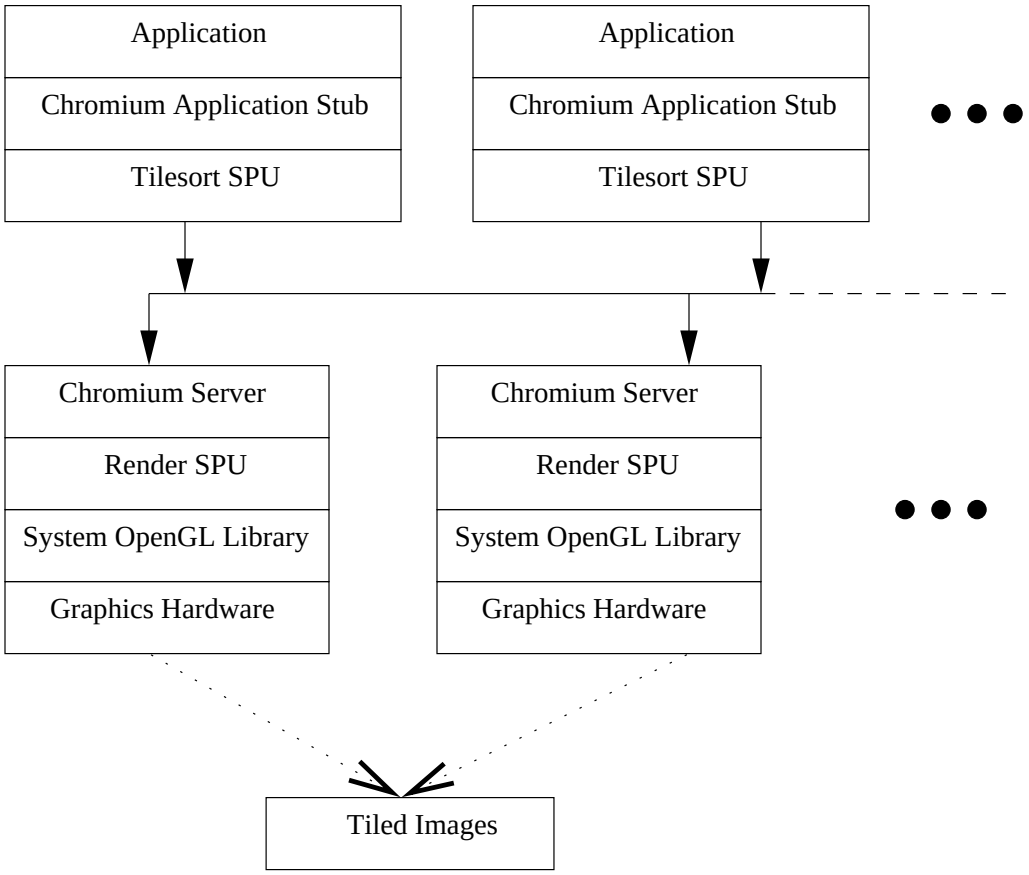


Figure 5.2: Sort-First (Tilesort) Rendering Process

ceives, and then passes colour and depth buffer information to the next SPU in the chain, which for a typical sort-last situation will be the pack SPU, to send the image to another render server, using the *Render* SPU to composite the images it receives.

5.4 Comparison

These approaches to distributed rendering differ considerably in terms of their performance and their suitability for different classes of rendering tasks. A major strength of Tilesort is that it can be used in almost any situation, including those with only a single application. Z-Compositing requires multiple input images, which suggests that there must be several application nodes running in parallel and submitting rendering commands to multiple rendering servers. Essentially, the process of dividing the viewport into tiles, and distributing primitives accordingly provides parallelism, while Z-Compositing depends on the application distributing the rendering work over several servers.

Z-Compositing does however allow the output of two different applications to be combined into a single output image, which may be more efficient than sorting the output of both into tiles for rendering. The relative performance of these techniques will be dependent on a variety of factors, mostly dependent on the nature of the application. If the majority of primitives rendered occupy several tiles, then time will be wasted on bounding box checks for primitives which will be submitted to most or all servers anyway. This will also result in a large communication overhead. The available hardware also needs to be taken into consideration, as it was discovered that in a situation with no hardware support for Z-Compositing (such as ours), the process of combining images in a sort-last setup is quite costly, and tilesort performs considerably better.

Since particle systems (typically) consist of a large number of very simple primitives, Tilesort is effective, since most can quickly be assigned a particular tile, and there are few primitives occupying multiple tiles, minimising the clipping required. In section 6 we present a performance evaluation of particle systems under various rendering configurations.

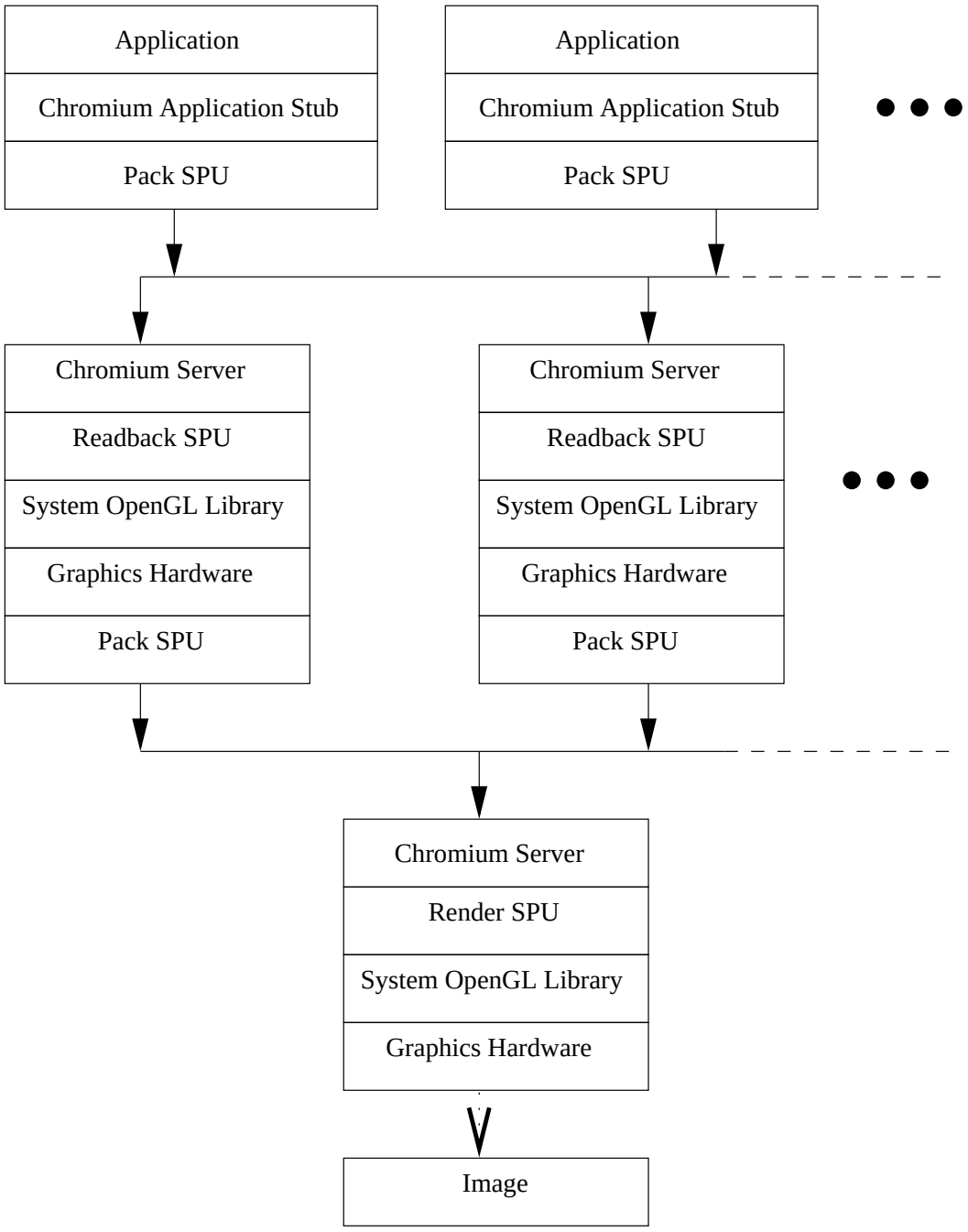


Figure 5.3: Sort-Last (Z-Compositing) rendering process

5.5 Dynamic Balancing of Tilesort

The efficiency of tilesort depends on the even distribution of graphics primitives between tiles. If this distribution is highly uneven, then the speed of the system will be limited by the tile with the most work and the cost of determining to which tile to send every primitive will make a tilesorted renderer perform poorly.

The distribution of the graphics workload between render servers is governed therefore by the screen-space projection of objects in the scene which may be difficult to predict and may change over time. To tackle this issue, Chromium allows the run-time reconfiguration of the tilesort SPU based on commands from and application, and provides feedback to the application regarding the load distribution.

A simple load balancing situation involves only 2 servers, each with one tile. An application requests information on the load balance, and this is provided in the form of the numbers of vertices submitted for rendering to each server. The application can examine this and resize tiles accordingly.

There is a computational cost associated with this dynamic rebalance, so it is preferable to perform it as seldom as possible. In situations where the scene is relatively stable, it may suffice to perform load balance for the first few frames, and then stop updating tile sizes once a stable situation has been reached. In the opposite case, where the scene moves often and randomly (e.g. an interactive simulation) it may not be effective to attempt this load balance, since it would need to be performed relatively often (otherwise scene changes could negate the benefits of applying it) and this would give a noticeable performance penalty.

Furthermore, Chromium can use a special optimised bucketing algorithm when the tiles are all the same size, so a static even arrangement may give the best results in certain cases. If the the cluster is being used to drive a CAVE or tiled display wall it may not be possible to alter the tile sizes so a form of regular grid arrangement will give optimal performance.

Chapter 6

Evaluation

6.1 Boids Performance Measurement

Figure **6.1** shows the performance of a system of boids executed on one, two or three machines in a cluster.[1] The cost to produce a frame in the one machine case can be seen to increase exponentially with the number of boids, illustrating the $O(n^2)$ nature of the basic algorithm. By contrast the graphs for two and three machines show the performance gain that a combination of domain subdivision, approximation and parallelisation produce, and in fact give a curve that is approximately linear for the range of values tested. The true time complexity is in fact $O((\frac{n}{m})^2 + m + n \log m)$ as given in section 3.9, so the linear appearance of the curve is due to the relatively small values of n and m. These performance measurements indicate however, that the costs incurred by message passing do not degrade the efficiency of the algorithm. Moreover, a visual examination of the status monitor for the Ethernet hub connecting the machines showed that the utilization of the transport medium rarely exceeded 20% during the computation, so further expansion of the cluster could be explored.

The performance of the two and three machine arrangements is very similar, this is a consequence of the relatively small numbers of boids present. The third machine presents additional processing power that is not really being utilized and is being counteracted by the additional communication costs of having more machines in an Ethernet cluster (collisions etc.). If the number of boids were increased further it is likely that a speed gain from using

[1] Note that every machine acts as both an application and a server, so essentially in the three machine setup for example, there are three application processes and three server processes, but only three processors

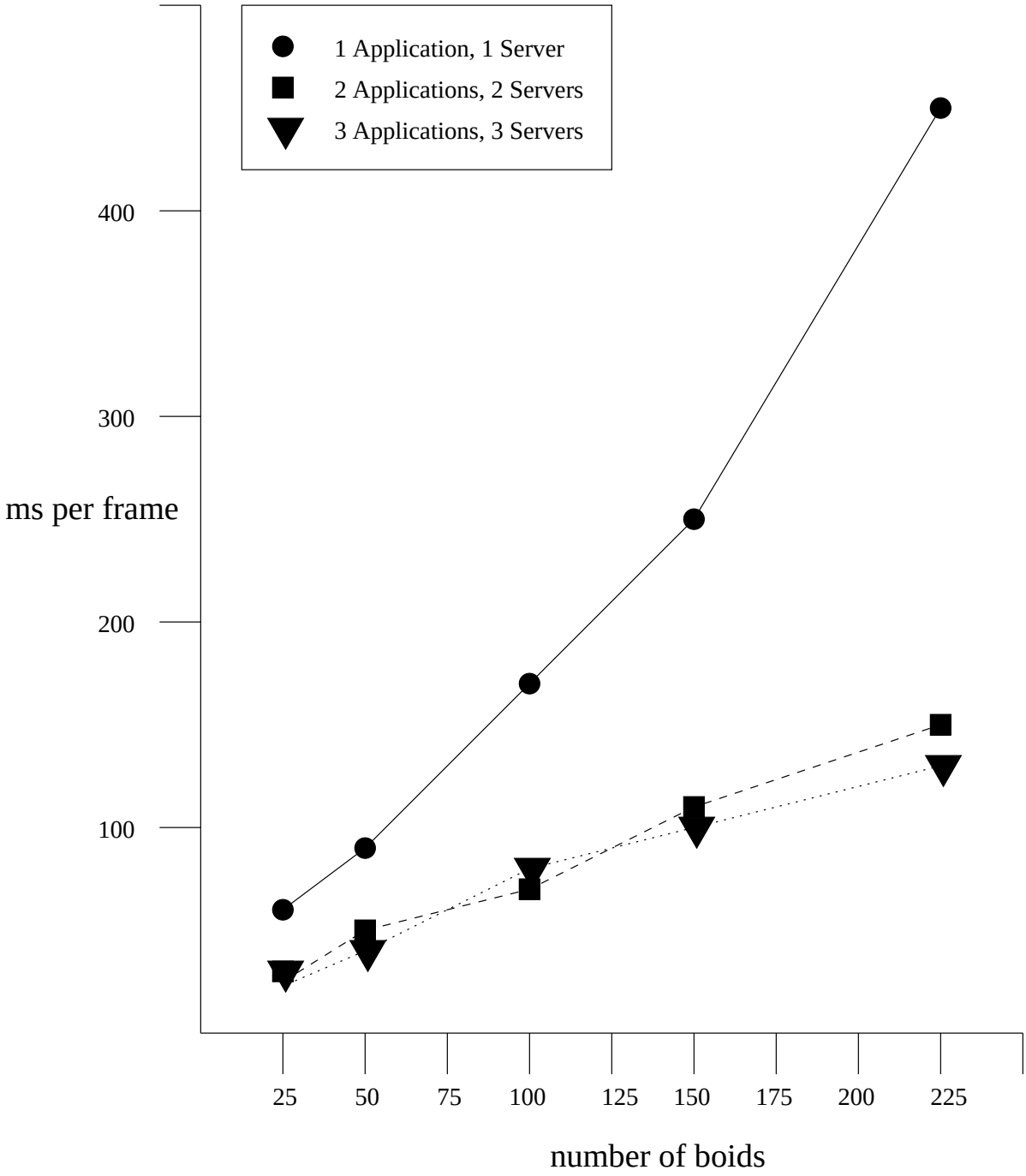


Figure 6.1: The performance of the boids implementation on various cluster sizes

more machines would be observed. The reason that this was not done is that 225 boids is a number approaching the limits of what the (non-accelerated) hardware can reasonably take. Using a greater number would make the process distictly graphics limited, and this would dominate all other factors is performance measurements. Therefore further analysis of this method would be desirable on larger clusters with more powerful graphics processing. The amount of inter-machine communication is quite small, and not greatly affected by the number of boids present, so the algorithm is expected to scale quite well to larger clusters of 8-16 nodes and to much larger flock sizes.

6.2 Interacting Systems Performance Measurement

The performance of an interacting set of particle sources is given in figure **6.2**, tested for one and two machine configurations. It is clear that the one machine case outperforms that with two, and this is due to the problem being extremely graphics limited. The graphics hardware becomes overloaded long before the computation time would become significant. A problem with 3000 particles without hardware acceleration places very large demands on the software rendering drivers and hence the CPU, however the computational cost to manage such a system is neglibible, since there are no per-particle interactions being performed. The parellel case has worse performance because the overheads of synchonisation are being imposed without giving any real performance gain, as the number of particles is so small. This area needs further work, again with superior graphics hardware to evalute whether the parallel situation is effective in practise.

We can see however that the graph is nearly linear, a consequence of the fact that the $O(n^2)$ nature of the problem has been eliminated, and the bucket sorting approach should be effective on serial graphics systems also.

6.3 Future Work — SCI Shared Memory Systems

The Scalable Coherent Interface (SCI) Standard[10] is a high-performance interconnect technology for compute clusters, and the potential for its use in a rendering cluster was examined as part of this project. SCI provides hardware shared memory across a cluster and manages issues such as error detection

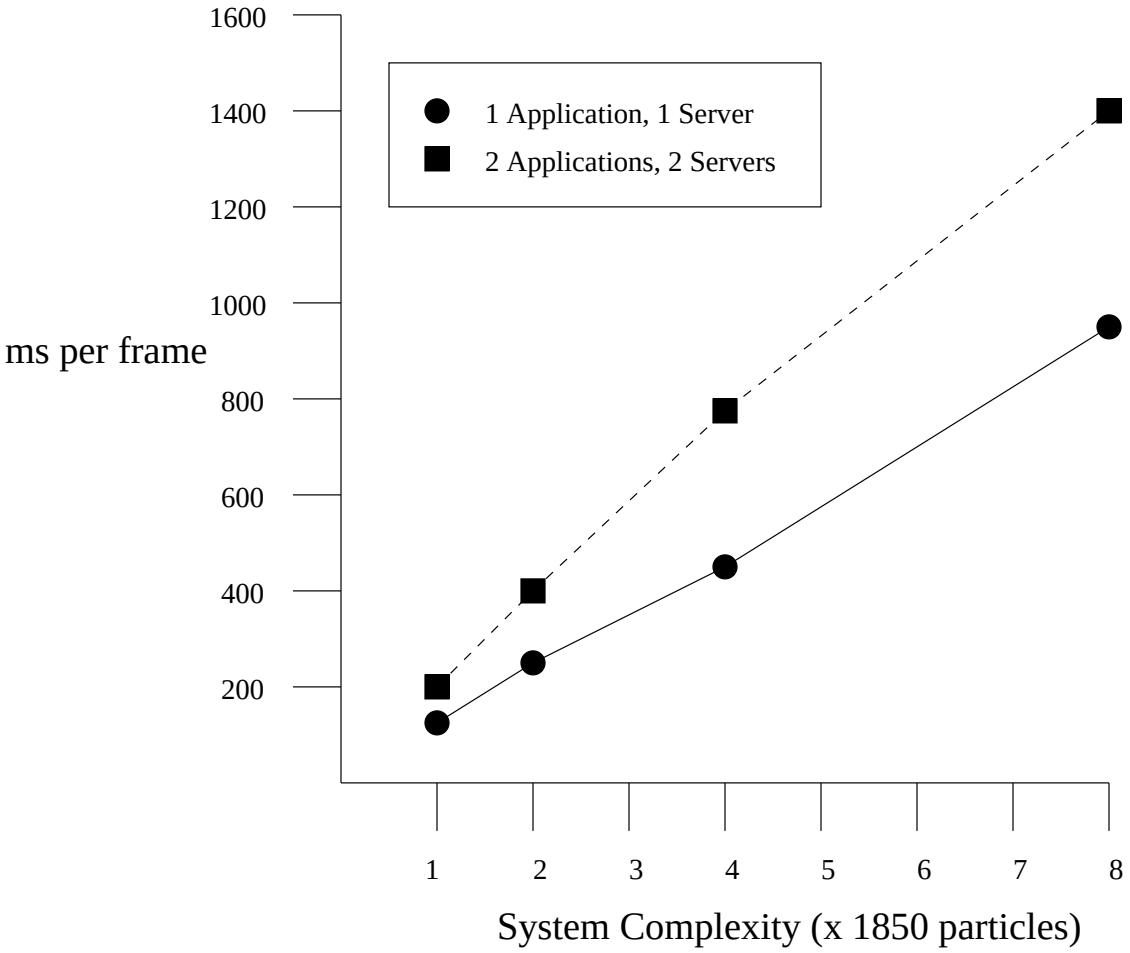


Figure 6.2: The performance of the implementation of stochastically modelled interacting systems

at the hardware level. For an overview of communication on an SCI cluster, refer to Appendix B. The SCI hardware discussed here is manufactured by Dolphin Interconnect Solutions Inc.[2]

6.3.1 SCI APIs and Abstractions

Several levels of driver support allow programmers to interact with SCI hardware in a manner appropriate to the needs of their application. The lowest level is the Interconnect Resource Manager (IRM), which provides basic access to SCI hardware functionality, including remote memory access, interrupts and Direct Memory Access (DMA) transfers. The Software Infrastructure for SCI (SISCI) API[18] provides higher level access to SCI functionality, including access to remote memory segments, DMA queues etc. through *SCI resource descriptors* which act as software handles to hardware functionality. The Shared Memory Interface (SMI) library[20], and the MP-MPICH library[21] provide further abstractions to the SCI interconnect. MP-MPICH is an implementation of the MPI standard, and SMI is a shared memory library with a similar interface. Both are designed to allow developers to design parallel applications using SCI without worrying about the details of SCI hardware. MP-MPICH operates as a wrapper to SMI.

6.3.2 Implementing the Chromium Networking Model on SCI

Chromium contains an abstract networking model which has currently been implemented on several types of hardware, including TCP/IP over Ethernet and GM Myrinet[14]. The networking abstraction requires that a network implementation provide basic common functionality, including:

- An initialisation routine
- A routine to create a connection
- A send routine
- A receive routine

This could be implemented using an MPI library, in our case MP-MPICH, as it provides all the necessary functionality for the send and receive semantics

[2] http://www.dolphinics.com/

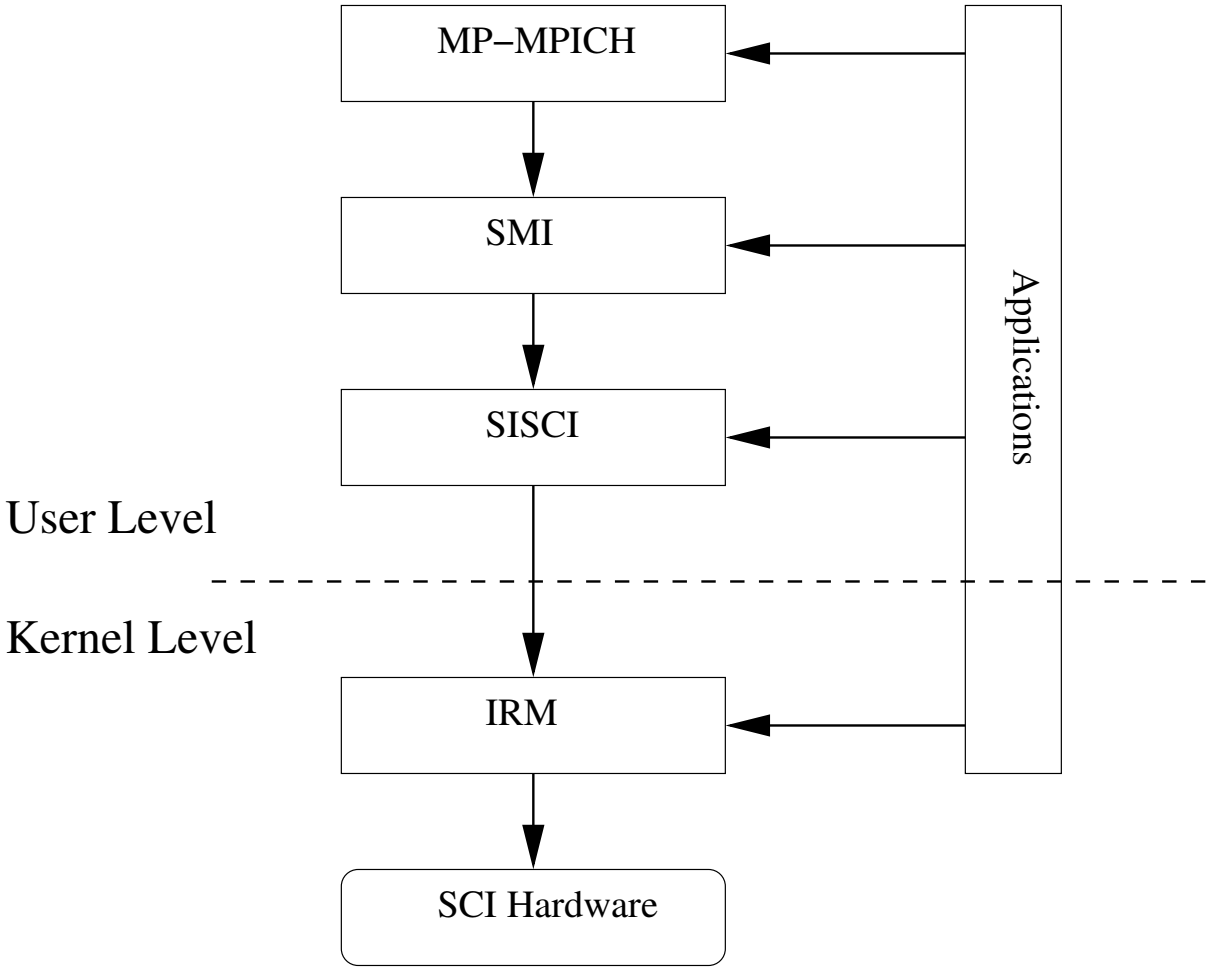


Figure 6.3: Different levels of SCI drivers and APIs

required by Chromium. The issues of initialisation and creating connections would need to be considered carefully, as MPI does not contain the concept of a connection as such, and requires special initialisation involving knowledge of the total number of nodes in the system.

Chromium requires that communication with the mothership occurs over TCP/IP, regardless of the network protocol used between application and server nodes. This would allow nodes using MPI to query the mothership regarding the rendering configuration (using TCP/IP) before initialising the MPI library. A "connection" under MPI will simply dispatch messages to the appropriate node, there is no need for any handshaking.

This does present a difficulty since the target of a connection is given as a URL, but in order to use MPI an MPI node ID is required as the destination of any message. This is complicated by the fact that MPI node IDs are assigned at initialisation, and may not be known in advance. This could possibly be solved by having all nodes exchange information regarding their URL and their MPI node ID in a collective communication operation as part of the initialisation process. This information could be cached on each node, and referred to whenever the node ID corresponding to a URL is required.

6.3.3 Distributed Textures

Chromium contains a *Distributed Texture* SPU, which allows textures to be placed as files on server nodes, so that they may be accessed locally. This avoids the cost of passing large texture data files across a cluster. The fast hardware supported shared memory provided by SCI could provide an alternative means of distributing textures.

Placing textures in shared memory segments available to both application and client nodes would support distribution of textures without the need to explicitly place them on servers. Typically texture data is placed in the local video memory of an accelerator card for fast access during rendering, however it remains in system memory also. This is because the amount of video memory is limited and textures must be swapped in and out as they are needed. AGP technology[4] greatly increases the bandwidth between graphics hardware and the CPU or main system memory, by providing a direct channel between the graphics card and the system bus. This allows textures to be read directly from main memory, with performance similar to if they were

stored locally on the video card, and the Graphics Address Remapping Table (GART) part of the AGP chipset fuctions as a virtual memory management unit for the video hardware, making the complete set of textures in local video memory and system memory appear as a single contiguous block, and allowing portions of textures to be paged into local video memory using DMA as they are requested.

The overall effect of this is similar to a multilevel cache, and extends naturally to allow the textures which appear to be in main memory to be stored in remote SCI memory segments. Caching of regularly accessed textures in local memory will greatly improve performance and will be especially effective since textures tend to be static i.e. don't change often during rendering. Portions of textures will be accessed gradually as they are needed, avoiding the need to copy the entire texture as a single chunk, and many servers may never need to refer to certain textures, saving the time which would otherwise be wasted in copying these to the server. If textures are stored as mip-maps (multiple levels of detail for different distances) then distant objects will only need the lowest level of detail, and only a small portion of the texture data will be accessed. As the objects come closer, the other levels of detail will be transferred as needed.

To implement this would require slight modifications to Chromium to allow server nodes to connect to the SCI memory segments containing texture data, and handle mapping these into local address space. If caching were implemented, then this would need to be implemented in software also. Chromium already provides OpenGL extensions to allow access to specific functionality, so this method would be preferable for adding an interface to distributed shared texture capability.

6.4 Summary

To conclude, the potential to enhace the scope and performance of particle systems through distributed rendering and computation has been examined. Even with a small cluster it is possible to obtain a noticeable speed gain, and although approximations to the computations were used to increase efficiency, the visual appearance of the simulation is not significantly disturbed.

Scalable Rendering is likely to become considerably more important in

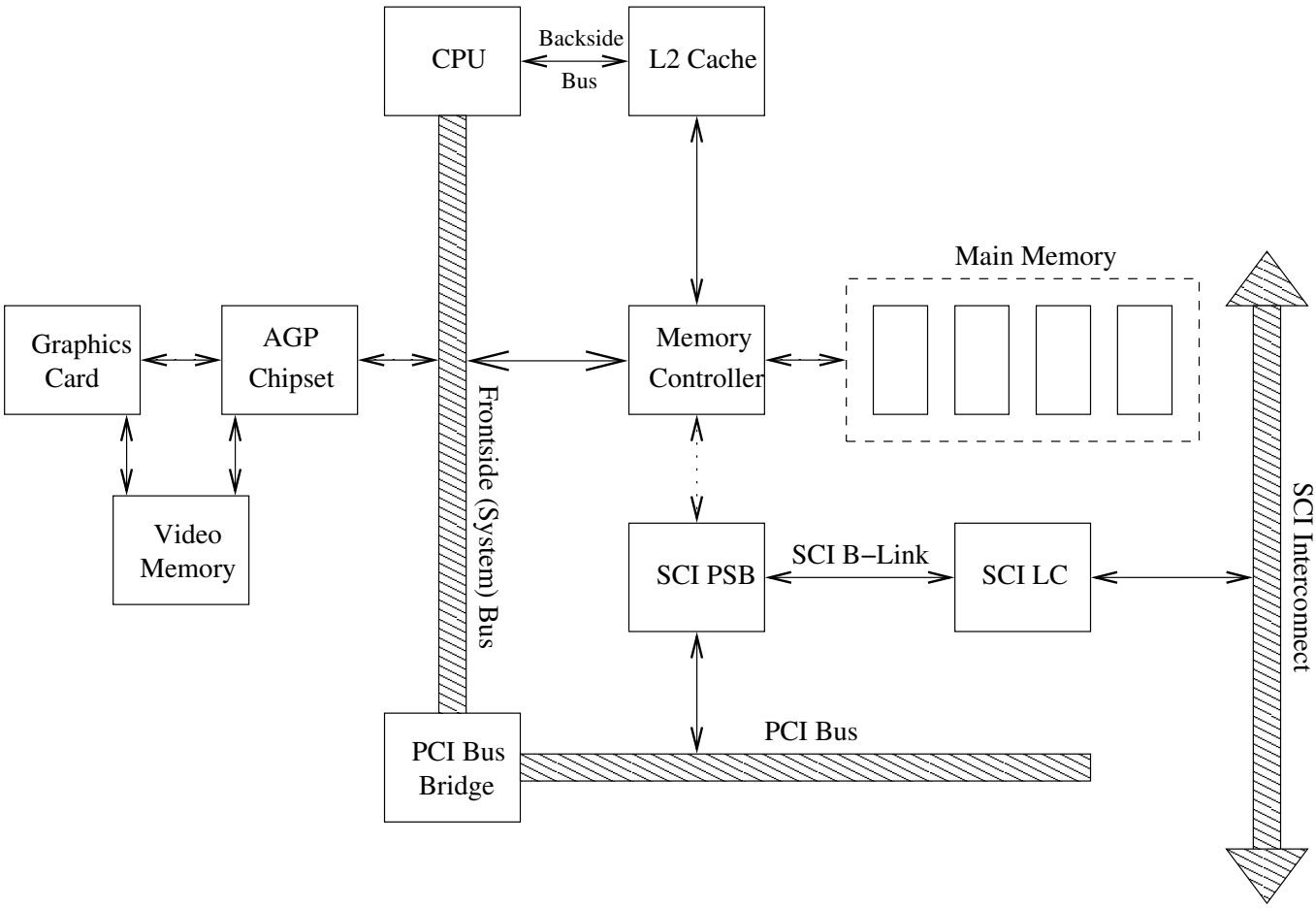


Figure 6.4: Overview of PCI bus, AGP port and SCI hardware

areas such as Virtual Reality, and its widespread accesibility to anyone with multiple machines and an Ethernet network will give it a major advantage. Particle Systems are generally only a part of an immersive rendering environment, which will typically also involve polygonal scenery. Rendering static polygonal environments is quite simple to distribute, and a major challenge instead is responding to user interaction. Future versions of Chromium will contain a distributed event handing model for such eventualities. Very high performance rendering will soon be available relatively cheaply to individuals for the first time, opening the way for many interesting new areas in Computer Graphics.

Bibliography

[1] J. Barnes and P. Hut. A Hierarchical $O(NlogN)$ Force Calculation Algorithm. *Nature*, 324:446–449, 1986.

[2] Guy Blelloch and Girija Narlikar. A practical comparison of n-body algorithms. In *Parallel Algorithms*, Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997.

[3] Jörg M. Colberg and Thomas J. MacFarland. Simulating the universe. In *Forschung und wissenschaftliches Rechnen*. 1998.

[4] Intel Corporation. Agp v3.0 interface specification. 2002.

[5] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[6] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works*. Morgan Kaufmann, 1994.

[7] Ananth Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulations of the Barnes–Hut method for n-body simulations. *Parallel Computing*, 24(5–6):797–822, 1998.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[9] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[10] Hermann Hellwagner and Alexander Reinefeld, editors. *SCI: Scalable Coherent Interface, Architecture and Software for High-Performance*

Compute Clusters, volume 1734 of *Lecture Notes in Computer Science*. Springer, 1999.

[11] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordan Stoll, Matthew Everett, and Pat Hanrahan. Wiregl: a scalable graphics system for clusters. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 129–140. ACM Press, 2001.

[12] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 693–702. ACM Press, 2002.

[13] Helmut Lorek and Matthew White. Parallel bird flocking simulation. In *Parallel Processing for Graphics and Scientific Visualization*, 1993.

[14] Myricom. *GM: A message-passing system for Myrinet networks*, 2002.

[15] William T. Reeves. Particle systems — a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics (TOG)*, 2(2):91–108, 1983.

[16] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34. ACM Press, 1987.

[17] Horst D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[18] Dolphin Interconnect Solutions. *SISCI API User Guide*, 2001.

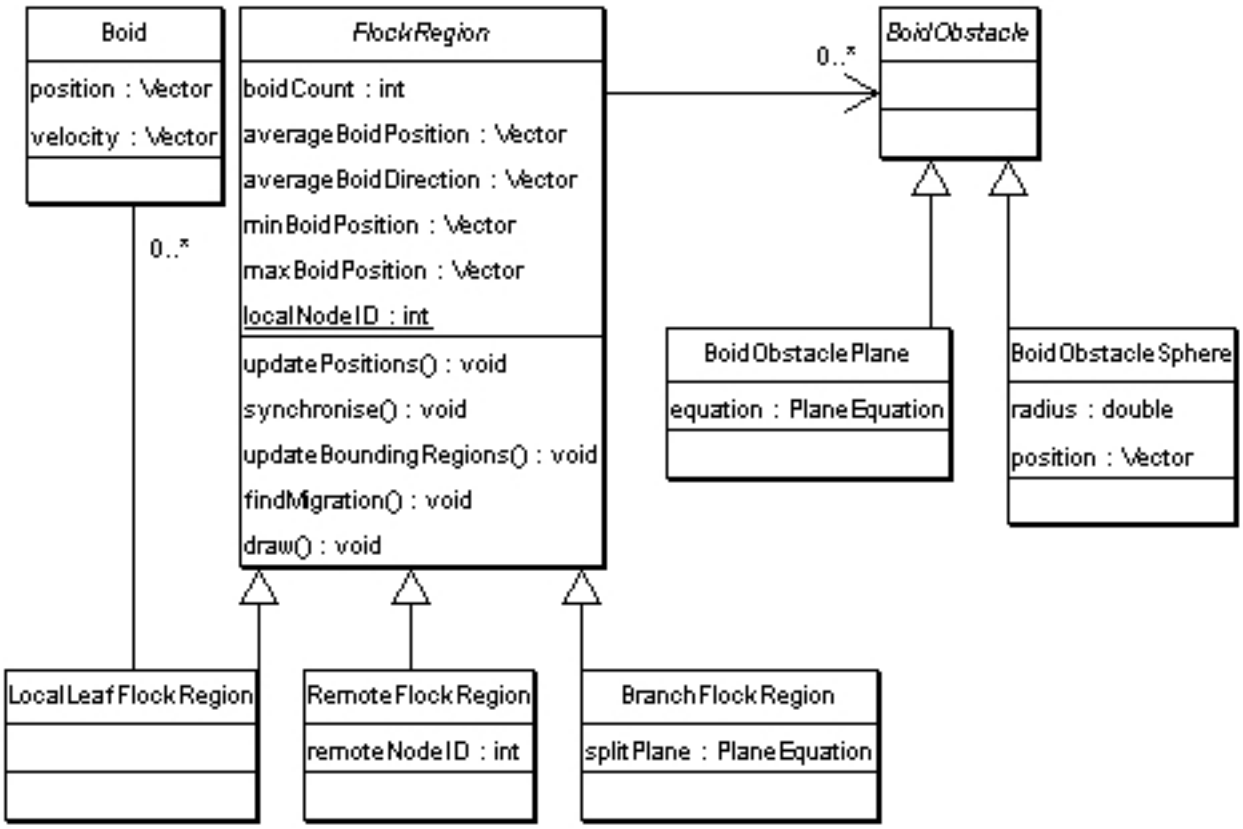
[19] Justin McCune (Cornell University). Interdependent particle systems. 1995.

[20] Joachim Worringen and Marcus Dormanns. *SMI - Shared Memory Interface: User & Reference Manual*. Lehrstuhl für Betriebssysteme RWTH Aachen, 2000.

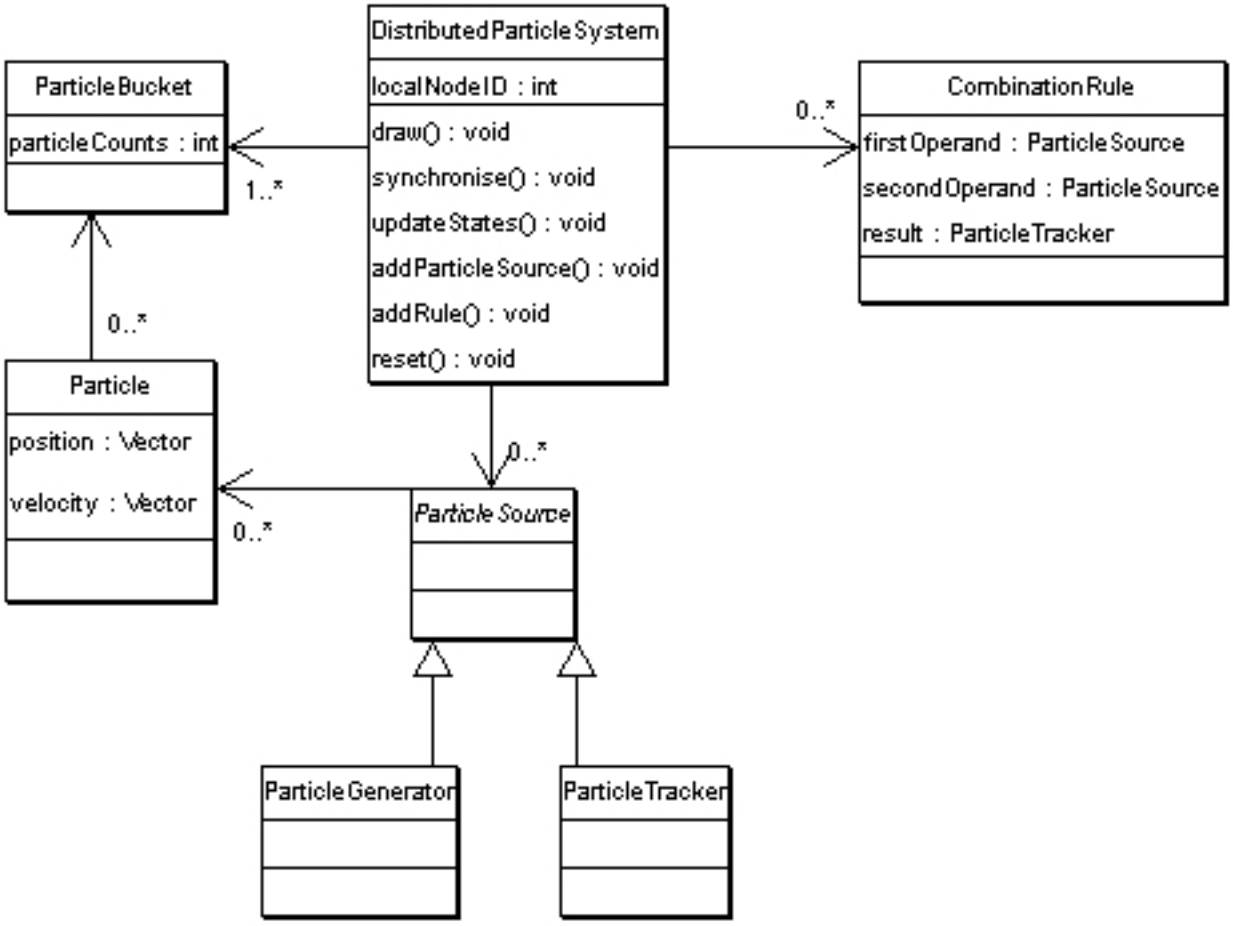
[21] Joachim Worringen and Karsten Scholtyssik. *MP-MPICH User Manual & Technical Notes*. Lehrstuhl für Betriebssysteme RWTH Aachen.

Appendix A — UML Diagrams

Distributed Behavioural System



Distributed Interdependent Physically Based System



Appendix B — SCI Transactions

The basic layout of an SCI cluster is a ring, or an n-dimensional torus. Nodes in a ring layout are connected to two neighbouring nodes, whilst those in a torus may be part of two or three rings depending on the dimensionality.

Communication occurs on an SCI network in the form of addressed packets. Every node in the cluster is assigned a *node ID*, which uniquely identifies it, and can be used to determine how to route packets. Every SCI transaction involves four packet transmissions: Request, Request-echo, Response, Response-echo. To initialise a transaction, node A sends a Request packet to node B. When the packet arrives at node B, this node immediately sends a Request-echo to node A. When the response is ready on node B, it is sent to node A, which acknowledges receipt of this with a Response-echo to node B. This form of handshake is used to guarantee reliable communication.

Upon receiving a packet, a node determines whether it is the destination, or whether to route it onward to another node, in which case it is placed in a bypass buffer, which has a higher priority for transmission than packets sent from this node.

The provision of shared memory by SCI means that nodes in a cluster can map segments of memory on remote nodes into their local memory space. A brief overview of the steps required to create, and connect to a shared memory segment is presented here.

1. The server node (the node exporting the segment) first allocates a portion of memory to be a shared segment
2. The server then maps this segment into the SCI address space
3. The segment is made available to remote nodes for connection
4. The client node (the node connecting to the segment) connects to the remote segment using a segment identifier

5. The client node maps the segment from SCI address space into its local addressable space

6. At this point the client node may access the segment as if the memory segment were local

List of Figures

Index