PHANTOM : Parallelization of Hierarchical Applications usiNg Treecodes On Multiprocessors

by
Sanjay Goil
M.Sc(Tech) Computer Science and M.Sc(Hons) Mathematics,
Birla Institute of Technology & Science, 1990
M.S Computer Science, Syracuse University, 1994

Final Report

August 1996

Advisors
Prof. Geoffrey Fox
Prof. Sanjay Ranka
Prof. Srinivas Aluru

Department of Electrical Engineering and Computer Science
and
Northeast Parallel Architectures Center
Syracuse University, Syracuse NY 13244

© Copyright 1996
Sanjay Goil

Contents

List of Tables vi

List of Figures vii

Abstract viii

1 Introduction 1

1.1 Distributed-Memory Machines . . . 2
1.2 Model of Parallel Computation . . . 3
1.3 Software System Requirements . . . 6
1.4 Organization of the report . . . 7

2 A Survey of Hierarchical Applications 9

2.1 N-body methods . . . 9
2.1.1 Parallelization on Shared memory machines . . . 12
2.1.2 Parallel Implementation on Distributed Memory Machines . . . 13
2.1.3 Data Partitioning . . . 15
2.1.4 Tree Construction . . . 17
2.1.5 Accessing Locally *Essential* Non Local Data . . . 17
2.1.6 Tree Traversal . . . 18
2.1.7 Summary . . . 19
2.2 Molecular Dynamics . . . 21
2.2.1 Parallelization on distributed memory machines . . . 22
2.2.2 Summary . . . 23
2.3 Volume Rendering . . . 24
2.3.1 Parallelization on Shared Memory Machines . . . 27
2.3.2 Parallelization on Distributed Memory Machines . . . 28
2.3.3 Data Partitioning and Coherence Issues . . . 29
2.3.4 Summary . . . 31
2.4 Adaptive Meshes . . . 33
2.4.1 Parallelization on distributed memory machines . . . 34
2.4.2 Summary . . . 36
2.5 Databases . . . 36
2.5.1 Spatial Indexing . . . 36
2.5.2 Parallelization . . . 37

2.5.3 Summary 38
2.6 Hierarchical Radiosity 39
2.6.1 Sequential Algorithm 40
2.6.2 Parallel approaches on a shared memory machine 41
2.6.3 Parallel approaches on a distributed memory machine 42
2.6.4 Summary 42
2.7 Image Compression 43

3 Parallel Selection Algorithms 45
3.1 Parallel Algorithms for Selection 45
3.1.1 Median of Medians Algorithm 45
3.1.2 Bucket-Based Algorithm 47
3.1.3 Randomized Selection Algorithm 47
3.1.4 Fast Randomized Selection Algorithm 50
3.2 Algorithms for load balancing 52
3.3 Implementation Results 52
3.4 Conclusions 55

4 Load Balancing Algorithms 56
4.1 Order Maintaining Load Balance 56
4.2 Dimension Exchange Method 58
4.3 Global Exchange 59

5 Constructing Multidimensional Binary Search Trees 61
5.1 Local Tree Construction 61
5.1.1 Sort-based Method 61
5.1.2 Median-based Method 62
5.1.3 Bucket-based Method 62
5.1.4 Experimental Results 63
5.2 Parallel Tree Construction for $\log p$ levels 65
5.2.1 Sort-based Method 65
5.2.2 Median-based Method 67
5.2.3 Bucket-based Method 69
5.2.4 Experimental Results 71
5.3 Global Tree Construction 73
5.4 Reducing the Data Movement 73
5.4.1 Experimental Results 75
5.5 Conclusions 76

6 Concatenated Parallelism 78
6.1 Introduction 78
6.2 Concatenated Parallelism 79
6.3 Deterministic Balanced Parallel Divide and Conquer 84
6.3.1 Applications - Multidimensional Binary Search trees 84
6.3.2 Experimental Results 85
6.4 Deterministic Unbalanced Parallel Divide and Conquer 86

6.4.1 Applications - Quadtrees and QuickHull 88
6.4.2 Experimental Results 89
6.5 Randomized Parallel Divide and Conquer 89
6.5.1 Applications - Quicksort 91
6.5.2 Experimental Results 91
6.6 Redistribution Strategies 92
6.7 Conclusions 93

7 Queries 96
7.1 Sample spatial queries 97
7.1.1 Point Location 97
7.1.2 Range Query 97
7.1.3 Circular region query 97
7.1.4 Minimum/Maximum in a range 98
7.1.5 Rectangle intersection 98
7.1.6 Rectangle containment 98
7.1.7 k-nearest neighbors 98
7.1.8 Parallelopiped region query 98
7.1.9 Polygonization 98
7.1.10 Spatial Joins 98

8 Primitives, Language and Run-time Support 99
8.1 Primitives required 99
8.1.1 Creating a tree 99
8.1.2 Data distribution 99
8.1.3 Fetch locally essential data 100
8.1.4 Incremental updates 100
8.1.5 Load Balancing 100
8.2 Usage of primitives for Volume Rendering 100
8.3 HPF and treecodes 101
8.3.1 Enhanced Mapping 101
8.3.2 Computation Control 102
8.3.3 Communication Optimization 102
8.4 Run time Support on distributed memory machines 102

9 Conclusions and Future Work 103
9.1 Conclusions 103
9.2 Future Work 104

Bibliography 105

List of Tables

1.1 Running times of various parallel primitives on cut-through routed hypercubes and square meshes with p processors. 5

2.1 Parallel approaches to N-body treecodes 14
2.2 Different approaches for parallel molecular dynamics 22
2.3 Different approaches on parallel volume rendering 29
2.4 Hierarchical data structures for spatial data used in practice 38

5.1 Computation time for local tree construction. 63
5.2 Time for tree construction up to $\log p$ levels on p processors for different strategies on a hypercube. 67

List of Figures

1.1 Software system for hierarchical applications 8

2.1 Creation of a quadtree as BH-tree 11
2.2 Creation of a k-d tree as BH-tree. 11
2.3 Barnes-Hut Algorithm 12
2.4 Parallel implementation of N-body algorithm using adaptive trees 14
2.5 Hashed Oct Tree (HOT) implementation (Warren and Salmon,1993) 16
2.6 A generic implementation of the N-body algorithm using incremental data structures 19
2.7 Molecular dynamics algorithm 22
2.8 Cell processor layout showing the extended volume for particle interactions 23
2.9 Overview of volume-rendering algorithm 26
2.10 Hierarchical enumeration of object space for N = 5 27
2.11 Ray tracing of hierarchical enumeration 28
2.12 Frame 1: Ray work profile for **brainsmall** and Frame 2 at a 5° rotation - Each pixel shows the computation work of a scanline. Scanlines for a slice progress from left to right on the horizontal axis. Slices of a frame are top to bottom on the vertical axis. 31
2.13 Ray intersections with volume (a) Viewing plane is aligned with the XY plane (b) Viewing plane is at an angle from the XY plane 32
2.14 (a) Grid refinement and (b) associated grid hierarchy 34
2.15 Radiosity algorithm 41
2.16 A subband decomposition scheme to be used with the Discrete Wavelet Transform. 44

3.1 Median on Medians selection Algorithm 46
3.2 Bucket-based selection algorithm 48
3.3 Randomized selection algorithm 49
3.4 Fast Randomized selection Algorithm 51
3.5 Performance of selection algorithms without load balancing (except for the median of medians algorithm for which load balancing is used) on random data sets. 53
3.6 Performance of fast randomized selection algorithm using random and sorted data sets. 53
3.7 Performance of the two randomized selection algorithms on sorted data sets using the best load balancing strategies for each algorithm 54

4.1 Modified order maintaining load balance 57
4.2 Dimension exchange method for load balancing 58
4.3 Global exchange method for load balancing 60

5.1 Local tree construction for random distribution of data of sizes 8K, 32K and 128K 64
5.2 Sort-based Method 66
5.3 Median-based Method 68
5.4 Bucket-based Method 70
5.5 Bucket-based method - Local tree construction 71
5.6 Tree construction to log p levels for p = 32, 64 and 128 using random distribution of data 71
5.7 Global tree construction for random distribution of data of size $\frac{N}{p}$ = 4K, 128K on p= 8, 32, 128. On the X-axis level i represents the tree is built to i levels (a tree having 2^i leaves) 74
5.8 Tree construction time using median-based method followed by median-based method (G3) on random data 75
5.9 Comparison of the two approaches for $\log p$ levels for different values of $\frac{N}{p}$. . 76

6.1 Illustration of Concatenated parallelism for $p = 8$. 80
6.2 Concatenated parallel algorithm 82
6.3 Comparison of Task Parallelism and Concatenated Parallelism for balanced divide and conquer in construction of k-d trees for different values of N/p. . 86
6.4 Time for tree construction up to $\log p$ levels divided into computation time for splitting the points plus the associated communication time and cost of redistribution for N = 2M and 4M on 8, 16, 32 and 64 processors. (TP: Task Parallel, CP: Concatenated Parallel) 87
6.5 Comparison of Task Parallelism and Concatenated Parallelism for unbalanced divide and conquer in quickhull for N = 128K, 512K and 2M random data. 90
6.6 Comparison of Task Parallelism allocating half the processors to a subproblem (A), Task Parallelism with processor allocation proportional to subproblem sizes (B) and Concatenated Parallelism for randomized divide and conquer in quicksort for N = 128K, 512K and 2M random data. 91
6.7 Illustration of load balancing during redistribution of tasks (a) Order preserving (b) Relative order combinations. Task allocation to processors is more balanced in (b). 93

7.1 (a) Range query (b) Circular region query (c) min(max) query in a range (d) intersection query (e) rectangle containment query (f) 4-nearest neighbors query (g) polygonization 97

PHANTOM : Parallelization of Hierarchical Applications usiNg Treecodes On Multiprocessors

by
Sanjay Goil

Abstract of Report
August 1996

In this report we address a class of problems consisting of highly structured computations on data sets that are described by hierarchical data structures. These are often represented as *tree* structures to optimize data storage requirements and perform efficient queries for data access. Specifically, applications that are dynamic and perform many iterations on data are of interest to us, since the requirements for data evolve over time and require modifying data structures incrementally. The computational relationship between the subdomains is known only at runtime, and may change between computation phases. Parallelization of such applications requires efficient distributed data management and has received some attention recently. We study the computational structure of a few irregular applications falling in this category. This helps us to evaluate data structures and the address issues of data partitioning, load balancing and communication requirements for these applications. Most recent efforts have been application specific and solutions are not portable across applications or parallel computing platforms. In this research we wish to characterize requirements for primitives and runtime software support needed to parallelize irregular applications.

A study of the computational structure of applications allows us to identify requirements for parallelization across a broad spectrum. Each application uses a tree data structure to organize data for efficient construction, manipulation and querying of related data. We enumerate algorithms that are used for solving these applications and discuss their parallelization. We present an outline of primitives that are needed to parallelize such applications. These can be used to provide language support by high performance languages to parallelize hierarchical applications.

Chapter 1

Introduction

Most previous research on coarse-grained MIMD machines has concentrated on parallelizing scientific High-level languages like Fortran-D [FHK+92], Vienna-Fortran [ZC92] which support parallel operations on uniform arrays. Efficient parallelizations of dynamic and irregular problems have received much attention only recently. Runtime support for irregular structures is available in PARTI [CFH+92]. These approaches take advantage of static data accesses and communication patterns. They provide little in terms of efficient solutions for dynamically changing data distribution and communication patterns.

We study a class of problems that consists of highly structured computations on sets of subdomains that are coupled hierarchically. The computational relationship between the subdomain is known only at runtime, and may change between computation phases. Parallelization of these applications on distributed memory machines require exploitation of the hierarchical nature both for data distribution, computational load balance as well as maintaining locality to reduce communication overheads. The following applications have been investigated: N-body simulation, Molecular Dynamics, Hierarchical Radiosity, Volume Rendering and Ray Tracing in Computer Graphics, Adaptive meshes, and Databases. These applications are usually efficiently represented and manipulated by using sparse data structures such as graphs, trees, and lists in sequential algorithms to reduce data storage sizes as well as to gain asymptotic performance for manipulation and retrieval of data.

The communication networks and software available on coarse-grained machines make local accesses at least an order of magnitude faster then nonlocal accesses. This is further accentuated by high latency costs of communication software on distributed-memory machines. Effective parallelization of these applications on coarse-grained MIMD machines requires careful attention for the following reasons:

- The amount of work done by the parallel algorithm should be within a small constant factor of the amount of work done by the sequential algorithm, since the number of processors used in practice is limited to from ten to a thousand. Parallel algorithms, which may be theoretically optimal, have limited use if the constants involved are large.

- The off-processor accesses generated by these applications are highly unstructured and may have many hot spots.

- For many applications, the data structures used have inherent locality of access and/or change incrementally. Exploitation of this information is necessary for efficient use of the various levels of memory hierarchy present in these architectures (register, caches, local accesses, nonlocal accesses, etc.). This requires fast methods for partitioning, repartitioning, replication, and migration of data.

- Static scheduling, dynamic scheduling, and load balancing are required to reduce processor idle time, and synchronization is required to achieve program correctness.

Our goal is to study the scalability of these applications on coarse-grained machines in a relatively architecture-independent fashion. We discuss the structure of the applications listed above in detail and present the data partitioning, communication and load balancing primitives that are required for their efficient parallelization.

The motivation for this work is from the recent efforts on parallelizing N-body applications on parallel machines, and extending their use to applications in other areas. Most implementations have been very specific to the problem and most often architecture dependent. The various phases in the implementation of the algorithm require immaculate control on the computation and communication functions. Our goal here is to develop an architecture independent infrastructure for parallelization of treecodes and apply it to challenging visualization applications such as ray-traced volume rendering [Lev90a], Ray Tracing and Hierarchical Radiosity [Aup93], Spatial databases, Molecular Dynamics and Adaptive Meshes. Singh [Sin93] discusses the parallelization needs of some graphics applications on a shared memory machine and elaborates the difficulties of parallelization on distributed memory machines.

1.1 Distributed-Memory Machines

A distributed-memory machine consists of a set of processors linked by interconnection networks. Each processor has its own memory that is directly accessible only by itself. Data exchange and global operations among processors are accomplished through message passing or appropriate hardware support [KGGK94, FJL+88].

The parallel-processing literature abounds with parallel algorithms designed for processors connected through special interconnection networks such as hypercubes, meshes, rings, or toroids. Available commercial architectures (IBM SP series, CM-5, nCUBE, and Intel Paragon) have subsets of the following properties:

1. *Distance.* With new routing techniques such as wormhole routing and randomized routing (as seen on the CM-5) [Lei85, KGGK94, DL87, NK93], the distance between communicating processors is less of a determining factor on the amount of time required for communication.

2. *Latency.* The start-up time for sending a message is an order of magnitude more than the cost of transmitting a few bytes of information. The latency of nonlocal access can be significantly reduced by using active messages [vECGS92, BK94, BR95]. Further, hardware support for accessing data from nonlocal memory (or moving pages into local memory) can provide a reduction in the effective latency [ea89, LLG+90, AA91].

3. *Node Contention.* A node can receive only one message (or a limited number of messages) at a time.

4. *Link Contention.* If two message paths have common links, the time required for their transmission may be affected. This effect is limited due to the use of virtual channels and because link bandwidth is much larger than node-interface bandwidths. Theoretical models typically assume only one virtual channel per link.

5. *Cross Section Bandwidth.* For machines which have an underlying mesh architecture (like Intel Paragon), the cross-section bandwidth may become a bottleneck.

Hardware support for cache coherence in machines (e.g., DASH, KSR) can significantly reduce programmers' efforts to maintain coherence of replicated data, and context switching can allow for multiple threads at low overheads [AA91]. We would concentrate on machine models without hardware support for shared memory and multi-threading. However, most of the techniques developed are of general applicability. One of our goals is to develop architecture-independent primitives that can be implemented on a wide variety of distributed-memory machines.

The specific interconnection network for which primitives would initially be developed for, assume that it can support arbitrary permutations (i.e., at a given time each node can send and receive one message from an arbitrary processor). This two-level memory model distinguishes elements needed for computation as being local or nonlocal to a processor. The logP [CKP+93] model and the postal model [BNK92] are theoretical models, based on the above philosophy, for coarse-grained machines. Due to larger link bandwidths, as compared to node interface bandwidths, many networks have behavior close to this model (e.g., the IBM SP Series and the CM-5). Later specific networks such as hypercubes and meshes would be studied. A large number of commercial and research architectures use these interconnection networks.

1.2 Model of Parallel Computation

Coarse Grained Machines (CGMs) consist of a set of processors (tens to a few thousand) connected through an interconnection network. The memory is physically distributed across the processors. Interaction between processors is either through message passing or through a shared address space. Popular interconnection topologies are buses (SGI Challenge), 2D meshes (Paragon, Delta), 3D meshes (Cray T3D), hypercubes (nCUBE), fat tree (CM5) and hierarchical networks (cedar, DASH).

CGMs have cut-through routed networks which will be used for modeling the communication cost of our algorithms. For a lightly loaded network, a message of size m traversing d hops of a cut-through (CT) routed network incurs a communication delay given by $T_{comm} = t_s + t_h d + t_w m$, where t_s represents the handshaking costs, t_h represents the signal propagation and switching delays and t_w represents the inverse bandwidth of the communication network. The startup time t_s is often large, and can be several hundred machine cycles or more. The per-word transfer time t_w is determined by the link bandwidth. t_w is often higher (an order to two orders of magnitude is typical) than t_c, the time to do a unit computation on data available in the cache. The per-hop component $t_h d$ can often be

subsumed into the startup time t_s without significant loss of accuracy. This is because the diameter of the network, which is the maximum of the distance between any pair of processors, is relatively small for most practical sized machines, and t_h also tends to be small. The above expressions adequately model communication time for lightly loaded networks. However, as the network becomes more congested, the finite network capacity becomes a bottleneck. Multiple messages attempting to traverse a particular link on the network are serialized. A good measure of the capacity of the network is its cross-section bandwidth (also referred to as the bisection width). For p processors, the bisection width is $\frac{p}{2}$, $2\sqrt{p}$, and 1 for a hypercube, wraparound mesh and for a shared bus respectively.

Our analysis will be done for the following interconnection networks: hypercubes and two dimensional meshes. The analysis for permutation networks and hypercubes is the same in most cases. These cover nearly all commercially available machines. A permutation network is one for which almost all of the permutations (each processor sending and receiving only one message of equal size) can be completed in nearly the same time (e.g. CM-5 and IBM SP Series).

Parallelization of applications requires distributing some or all of the data structures among the processors. Each processor needs to access all the non-local data required for its local computation. This generates aggregate or collective communication structures. Several algorithms have been described in the literature for these primitives and are part of standard textbooks [FJL+88, KGGK94]. The use of collective communication provides a level of architecture independence in the algorithm design. It also allows for precise analysis of an algorithm by replacing the cost of the primitive for the targeted architecture.

In the following, we describe some important parallel primitives that are repeatedly used in our algorithms and implementations. For commonly used primitives, we simply state the operation involved. The analysis of the running time is omitted and the interested reader is referred to [KGGK94]. For other primitives, a more detailed explanation is provided. Table 1 describes the collective communication routines used in the development of our algorithms and their time requirements on cut-through routed hypercubes and meshes. In what follows, p refers to the number of processors.

1. **Broadcast.** In a Broadcast operation, one processor has a message of size m to be broadcast to all other processors.

2. **Combine.** Given a vector of size m on each processor and a binary associative operation, the Combine operation computes a resultant vector of size m and stores it on every processor. The ith element of the resultant vector is the result of combining the ith element of the vectors stored on all the processors using the binary associative operation.

3. **Parallel Prefix.** Suppose that $x_0, x_1, \ldots, x_{p-1}$ are p data elements with processor P_i containing x_i. Let $\otimes$ be a binary associative operation. The Parallel Prefix operation stores the value of $x_0 \otimes x_1 \otimes \ldots \otimes x_i$ on processor P_i.

4. **Gather.** Given a vector of size m on each processor, the Gather operation collects all the data and stores the resulting vector of size mp in one of the processors.

Primitive	Running time on a p processor	
	Hypercube	Mesh
Broadcast	$O((t_s + t_w m)\log p)$	$O((t_s + t_w m)\log p + t_h\sqrt{p})$
Combine	$O((t_s + t_w m)\log p)$	$O((t_s + t_w m)\log p + t_h\sqrt{p})$
Parallel Prefix	$O((t_s + t_w)\log p)$	$O((t_s + t_w)\log p + t_h\sqrt{p})$
Gather	$O(t_s \log p + t_w mp)$	$O(t_s \log p + t_w mp + t_h\sqrt{p})$
Global Concatenate	$O(t_s \log p + t_w mp)$	$O(t_s \log p + t_w mp + t_h\sqrt{p})$
All-to-All Communication	$O((t_s + t_w m)p + t_h p \log p)$	$O((t_s + t_w mp)\sqrt{p})$
Transportation Primitive	$O(t_s p + t_w r + t_h p \log p)$	$O((t_s + t_w r)\sqrt{p})$
Order Maintaining Data Movement	$O(t_s p + t_w(s_{max} + r_{max}) + t_h p \log p$	$O((t_s + t_w(s_{max} + r_{max}))\sqrt{p} + t_h\sqrt{p})$
Non-order Maintaining Data Movement	$O(t_s p + t_w(s_{max} + r_{max}) + t_h p \log p)$	$O((t_s + t_w(s_{max} + r_{max}))\sqrt{p} + t_h\sqrt{p})$

Table 1.1: Running times of various parallel primitives on cut-through routed hypercubes and square meshes with p processors.

5. **Global Concatenate.** This is the same as Gather except that the collected data should be stored on all the processors.

6. **All-to-All Communication.** In this operation each processor sends a distinct message of size m to every processor.

7. **Transportation Primitive.** It performs many-to-many personalized communication with possibly high variance in message size. Let r be the maximum of outgoing or incoming traffic at any processor The transportation primitive breaks down the communication into two all-to-all communication phases where all the messages sent by any particular processor have uniform message sizes [RSA95]. If $r \geq p^2$, the running time of this operation is equal to two all-to-all communication operations with a maximum message size of $O(\frac{r}{p})$.

8. **Order Maintaining Data Movement.** Consider the following data movement problem, an abstraction of the data movement patterns that we encounter in subtask redistribution. Initially, processor P_i contains two integers s_i and r_i, and has s_i elements of data such that $\sum_{i=0}^{p-1} s_i = \sum_{i=0}^{p-1} r_i$. Let $s_{max} = max_{i=0}^{p-1} s_i$ and $r_{max} = max_{i=0}^{p-1} r_i$. The objective is to redistribute the data such that processor P_i contains r_i elements. Suppose that each processor has its set of elements stored in an array. We can view the $\sum_{i=0}^{p-1} s_i$ elements as if they are globally sorted based on processor and array indices. For any $i < j$, any element in processor P_i appears earlier in this sorted order than any element in processor P_j. In the order maintaining data movement problem, this global order should be preserved after the distribution of the data.

   The algorithm first performs a Parallel Prefix operation on the s_i's to find the position

of the elements each processor contains in the global order. Another parallel prefix operation on the r_i's determines the position in the global order of the elements needed by each processor. Using the results of the parallel prefix operations, each processor can figure out the processors to which it should send data and the amount of data to send to each processor. Similarly, each processor can figure out the amount of data it should receive, if any, from each processor. The communication is performed using the transportation primitive. The maximum number of elements sent out by any processor is s_{max}. The maximum number of elements received by any processor is r_{max}.

9. **Non-Order Maintaining Data Movement.** The order maintaining data movement algorithm may generate much more communication than necessary if preserving the global order of elements is not necessary. For example, consider the case where $r_i = s_i$ for $1 \leq i < p-1$ and $r_0 = s_0 + 1$ and $r_{p-1} = s_{p-1} - 1$. The optimal strategy is to transfer the one extra element from P_{p-1} to P_0. However, this algorithm transfers one element from P_i to P_{i-1} for every $1 \leq i < p-1$, generating $(p-1)$ messages.

   For data movements where preserving the order of data is not important, the following modification is done to the algorithm: Every processor retains $min\{s_i, r_i\}$ of its original elements. If $s_i > r_i$, the processor has $(s_i - r_i)$ elements in excess and is labeled a source. Otherwise, the processor needs $(r_i - s_i)$ elements and is labeled a sink. The excessive elements in the source processors and the number of elements needed by the sink processors are ranked separately using two Parallel Prefix operations. The data is transferred from sources to sinks using a strategy similar to the order maintaining data movement algorithm.

   The maximum number of outgoing elements at any processor is $max_{i=0}^{p-1}(s_i - r_i)$, which can be as high as s_{max}. The maximum number of incoming elements at any processor is $max_{i=0}^{p-1}(r_i - s_i)$, which can be as high as r_{max}. Therefore, the worst-case running time of this operation is identical to the order maintaining data movement operation. Nevertheless, the non-order maintaining data movement algorithm is expected to perform better in practice.

1.3 Software System Requirements

The software requirements for hierarchical applications can be divided into *Architecture dependent* and *Architecture independent* activities. Current distributed memory machines are available in different configurations with varying interconnection networks. Message passing routines are typically architecture dependent and provided by the vendor. Global communication operations built using the basic communication primitives *send* and *receive* are also implemented by the machine vendor.

Hierarchical applications deal with treecodes and require basic tree manipulation routines. Queries on the tree data structure need to be performed to access and modify data structures. A library of routines providing primitives for tree construction and providing operations on the distributed data structure is needed. Load balancing aspects for the tree should be addressed at different levels by Global tree and Local tree management routines.

Queries will extensively use data structure movement, which in most cases will be subtrees and will need encoding on the sender side for transmission. The receiver will decode the data to reconstruct the subtree. This *high level* data movement will use the low level data movement provided by the machine, using global communication whenever appropriate.

The main objective is to build an architecture independent software system for hierarchical applications and we would like to keep the architecture dependent part as small as possible. Communication libraries like PVM and MPI provide a layer of architecture independence and can be used.

Figure 1.1 highlights the software system that is needed for the applications discussed in this paper.

1.4 Organization of the report

Chapter 2 presents a survey of hierarchical applications and discusses issues in their parallelization on distributed memory parallel computers. This identifies the common structure of these applications, amenable to the *divide and conquer* paradigm. Quadtrees, k-d trees, R-trees and its variants are identified as spatial data structures that efficiently represent and manipulate data in these applications. For balanced construction of these structures, *median finding*, and more generally *selection*, is an important operation. Chapter 3 presents parallel algorithms for selection on distributed memory machines. Chapter 4 presents algorithms for parallel construction of k-d trees (multidimensional binary search trees). We have analyzed these algorithms and optimized communication for them by reducing the data movement at each stage. *Concatenated parallelism* describes this method and is presented in Chapter 5. Chapter 6 presents the common spatial queries that are required for applications surveyed in this report. Primitives for language and run-time support are presented in Chapter 7. Finally Chapter 8 presents our conclusions and looks into the future for this research.

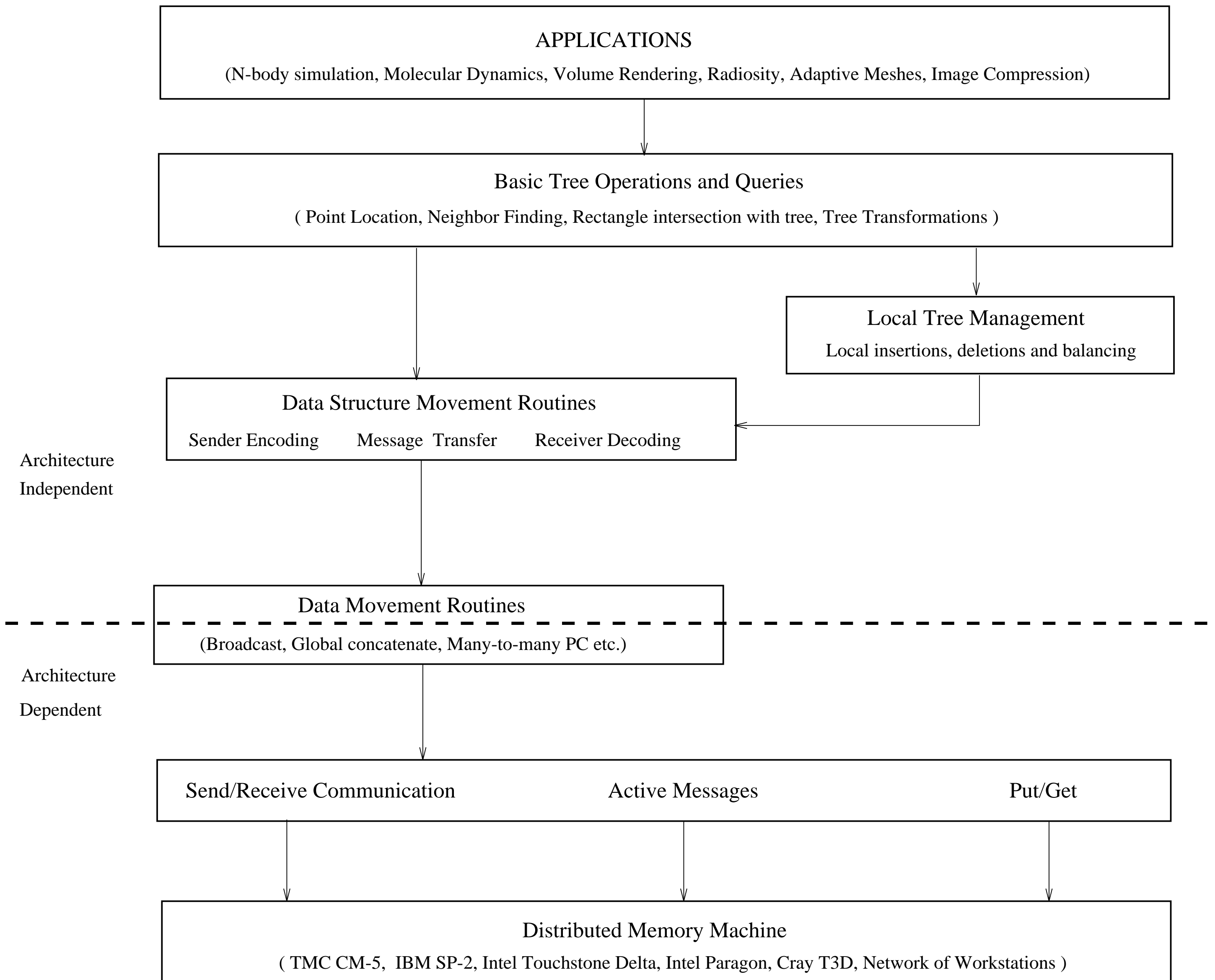


Figure 1.1: Software system for hierarchical applications

Chapter 2

A Survey of Hierarchical Applications

2.1 N-body methods

Computational methods to track the motions of bodies that interact with each other have been subjects of study for a long time in the areas of astrophysics, semiconductor device simulation, molecular dynamics and plasma physics. The N-body problem computes the state (position and velocity) of N bodies at a given time $t > 0$, given an initial state at $t = 0$. The most common approach is to iteratively calculate the solution by calculating all forces over a sequence of small time steps. Within each timestep the instantaneous acceleration is approximated by the instantaneous acceleration at the beginning of the time step, which is done by directly summing the force induced by each of the other $N-1$ bodies. This method is conceptually simple and vectorizes well but its $\Theta(N^2)$ arithmetic complexity rules it out for large-scale simulations involving millions of bodies.

Many physical systems exhibit a large range of scales in their information requirements, in both space and time. A point in physical domain requires progressively less information at a lesser frequency from parts of the domain that are farther away from it. Applying this fundamental insight Appel [App85] and Barnes and Hut [BH86] were the first to propose faster N-Body algorithms. Appel's method requires $O(N)$ steps but has a bigger constant attached to it, whereas the Barnes-Hut method is $O(N log N)$ and is used in practice. N-body simulations using adaptive tree data structures are referred to as *treecodes*. Parallel implementation of the Barnes-Hut method has been fairly recent and is reported in Salmon and Warren [WS92, WS93].

In an astrophysical N-body simulation, force interactions between celestial bodies are calculated according to the laws of Newtonian physics. Accelerations induce by forces due to the other $N-1$ bodies are calculated as

$$\frac{d^2\vec{x}_i}{dt^2} = \sum_{j\neq i} \vec{a}_{ij} = \sum_{j\neq i} -\frac{Gm_j\vec{d}_{ij}}{|d_{ij}|^3}, \qquad \vec{d}ij = \vec{x}_i - \vec{x}_j.$$

This formulation leads to a $O(N^2)$ algorithm. Several approximate methods have been used to reduce the overall time and allow larger simulations to be done. The approximation that reduces the interactions using treecodes is stated as [WS92]

$$\sum_j \frac{G m_j \vec{d}_{ij}}{|d_{ij}|^3} \approx \frac{G M \vec{d}_{i,cm}}{d^3{}_{i,cm}} + \ldots$$

where $\vec{d}_{i,cm} = \vec{x}_i - \vec{x}_{cm}$ is the vector from $\vec{x}_i$ to the center-of-mass of the particles that are summed in the left hand side in the above expression. Quadropole, octopole and further terms in the multipole expansion can be included for better approximations.

In the Fast Multipole Method (FMM) presented by Greengard and Rokhlin [GR87] particles are organized into cells and then cell-cell interactions are computed prior to the force calculation step. Once this has been determined, the force on a single particle can be obtained in a time independent of N, resulting in a *O(N)* scaling. The interactions here are more complex and the hidden constants in the notation are not very clear. Each cluster is characterized by a *multipole expansion* computed by traversing the tree in an upward phase. This is followed by a downward phase to combine multipole expansions and to propagate them to the leaves. At the end of the downward phase each leaf has data to compute the force induced by bodies in the *far field*, which is the area outside of this leaf and its neighbors.

The Barnes-Hut algorithm begins by constructing a tree, inserting the bodies into the cluster hierarchy one at a time. First an octree partition of the three-dimensional box (a region in space) is computed enclosing the set of bodies. The partition is computed recursively by dividing the original box into eight octants of equal volume until each undivided box contains exactly one body. Figure 2.1 is an example of recursive partition in two dimensions, the corresponding quadtree, which is called the Barnes-Hut (BH) tree. Alternatively, a k-d tree can be constructed to store the bodies. This is a binary tree in which elements are partitioned into two partitions, alternating the dimension to choose an element as a partitioner at each level. To minimize the number of interactions, each body computes interactions with the largest clusters for which the approximation can be applied. The bodies are added to the tree one at a time. The ith body is added into the BH-tree with i - 1 bodies, the newly inserted body descending down the tree until it reaches a box of which it is the sole occupant. If the body reaches a leaf, the leaf is subdivided until each of the two bodies is in its own box.

Each internal node of the BH-tree represents a cluster. Once the BH-tree has been built, the centers-of-mass of the internal nodes are computed by traversing the tree bottom-up. Once that is done, each cluster represents the bodies in its region for interaction. For computing accelerations each body traverses the tree in depth-first manner starting at the root. For any internal node sufficiently far away, the effect of the subtree on the body is approximated by a two-body interaction between the body and a point mass located at the center-of-mass of the tree node. The tree traversal continues, but the subtree is bypassed. When the traversal reaches a leaf, a direct two body interaction is computed. The set of nodes which contribute to the acceleration on a body are called the *essential nodes* for the body. Each body has a distinct set of essential nodes which changes with time. Two important criterion help in approximating the force fields. Firstly, for a body far away from the cluster, the effect of the cluster can be approximated by its center of mass rather than by each individual interaction with each body in the cluster. Secondly, the depth-first traversal ensures that each body interacts only with the *largest* clusters for which the approximation is valid. Once accelerations on each body are known, the new positions and velocities are computed. The entire process is repeated for the desired number of time steps.

The Barnes-Hut algorithm is shown in Figure 2.3.

For any particle p the force can be approximated by starting at the root cell of the tree. Let l is the length of the cell currently being processed and D the distance of p from the cell's center-of-mass. If $l \ / \ D < \theta$, where θ is a fixed accuracy parameter ~ 1, then the interaction between this cell and p is included in the total being accumulated. Otherwise, the cell is resolved into eight subcells for an octree, and two for a k-d tree, each one being recursively examined.

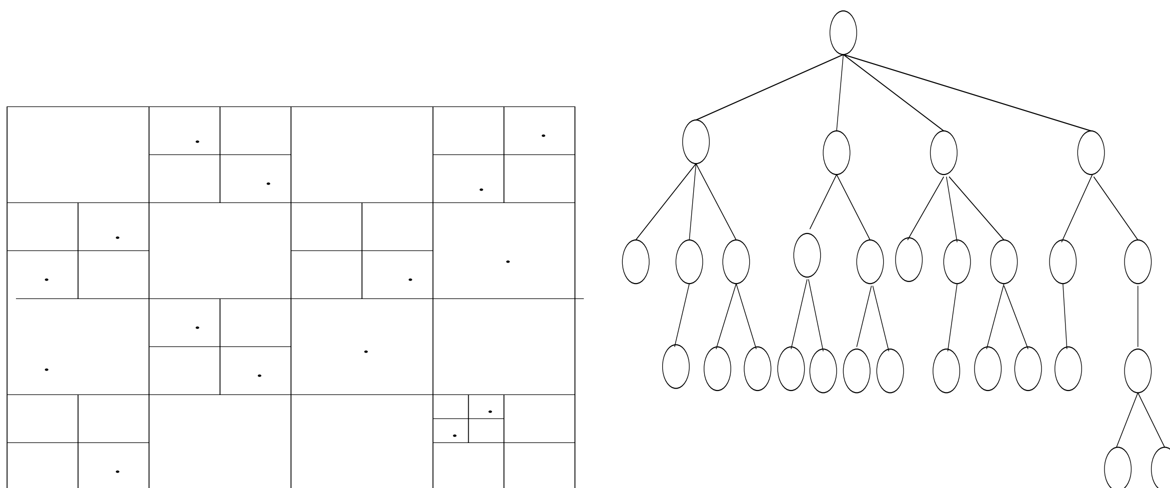


Figure 2.1: Creation of a quadtree as BH-tree

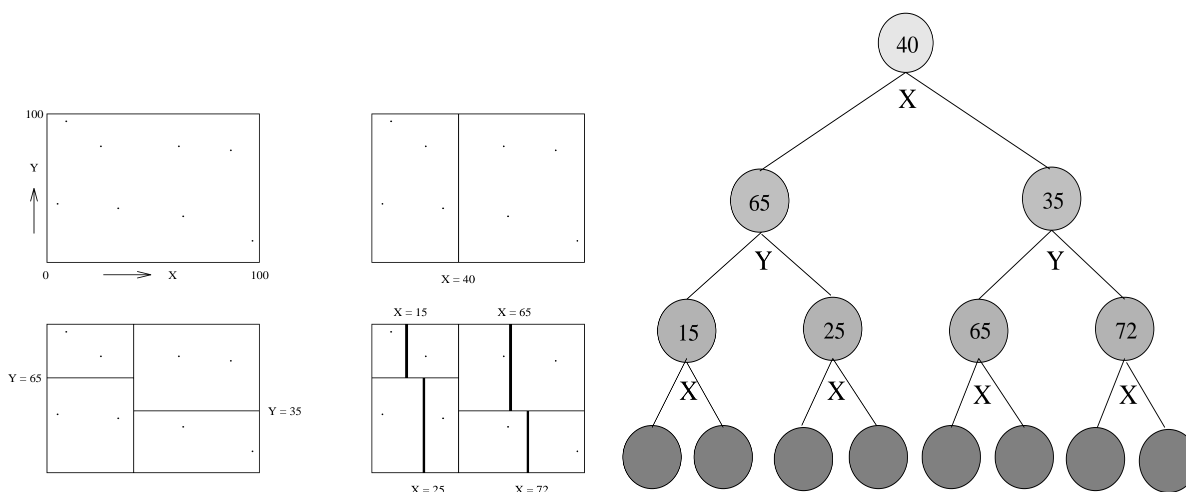


Figure 2.2: Creation of a k-d tree as BH-tree.

In this paper, we limit ourselves to the Barnes-Hut algorithm due to its popularity as well as its hierarchical nature. We provide a brief summary of the different issues required for its parallelization on distributed memory machines with and without hardware support for shared memory.

```
For each time step:
    Step 1. Build the BH-tree.
    Step 2. Compute centers-of-mass bottom-up.
    For each body
        Step 3. Start a depth-first traversal of the tree, truncating the
                search at a internal node where the approximation is applicable.
        Step 4. Update the contribution of the node to the acceleration of the body
    Step 5. Update the velocity and position of each body.
```

Figure 2.3: Barnes-Hut Algorithm

2.1.1 Parallelization on Shared memory machines

A parallel implementation of the Barnes-Hut algorithm on the DASH [LLG+90], a cache coherent shared memory multiprocessor shared memory machine, has been reported in [Sin93] . The data partitioning is done using *costzones* [Sin93] or by using Orthogonal Recursive Bisection (ORB). The data is globally shared among the processors in shared memory. The tree construction is parallelized by letting processors insert their particles into the shared tree concurrently. Whenever a processor has to modify the tree, either by putting a particle in a cell or by subdividing a cell, it must first obtain a lock on that cell to ensure that no other processor tries to modify it at the same time. As the number of processors increase, the overhead of executing these locks and unlocks becomes substantial. The ORB partition allocates every processor contiguous partition of space. This effectively divides the tree into distinct sections and assign a processor to each section, which means that there is little contention for locks after the first few levels of the tree are built, since processors will construct their own disjoint sections without much interference. Much of the contention is due to many processors simultaneously trying to update the upper levels of the tree.

In another approach outlined in [Sin93], every processor builds its own version of the tree using only its own particles. These individual trees are then merged together into a single tree used in the rest of the computation. The number of times a processor has to obtain a lock on the global tree goes down as entire subtrees are usually merged into the global tree. There is a tradeoff when merging entire subtrees, since not much concurrency is available in the merging phase. However, results in [Sin93] show that this method outperforms the previous one. This is essentially the approach one would take on a distributed memory machine, by building local trees and then getting a global representation by combining them appropriately to store global information. The tree building and center-of-mass computation phases require both interprocessor communication and synchronization. The force calculation for a particle requires the communication of position and mass information from other particles and cells, but this is not modified during the force calculation phase. Forces on different particles can be computed in parallel without synchronization. Each processor will read the data it needs from the global tree, a single copy of which is shared among all processors. The work for a particle in the update phase is entirely local to that particle and requires neither communication nor synchronization.

2.1.2 Parallel Implementation on Distributed Memory Machines

On a distributed memory machine the data is distributed across the memory of processors. A data distribution that gives more or less equal work to all processors in desirable to obtain higher efficiencies. The work metric used in the N-body simulation is the calculation of forces for each body. Hence, just the count of the number of bodies on a processor is not enough, their distribution in space also needs to be taken into account. This determines the force calculations that will be performed for each body and ideally we would like that to be perfectly load balanced. During the computation phase if data for force calculation is not available locally it must be fetched from the processor that has it. Data should be assigned to processors such that most data for computation should be available locally. Typically communication for fetching off-processor data is much more expensive than a local read. Data partitioning must preserve data locality to reduce communication requirements. An inappropriate data mapping can increase communication costs and degrade overall performance by adding to the overhead. A good partitioning would take both load balancing and data locality into account.

A distinguishing feature of the BH-tree is that it evolves continuously due to ongoing computation. It is a dynamic data structure and data must dynamically be updated to adapt to the evolving system. A static data mapping may not distribute data evenly after a period of time. The data mapping can either be done again at each time step or adjusted incrementally to reflect the changes in the system. Also, as bodies move and the distribution of bodies in space changes, the work associated with calculating forces can also change leading to differential load on processors. The mapping of bodies to processors must be adjusted to ensure load balance. Thus the BH-tree is adaptive to dynamic and irregular distribution of bodies. Also, the movement of bodies requires dynamic data mapping and distributed data management.

Since the BH-tree is distributed there is a need for off-processor data during computation. The data mapping can change from one step to another, the communication pattern for such data will also change, making it irregular and dynamic. A static data pattern cannot be calculated for optimizing communication for that pattern. It is unpredictable at compile time and hard to optimize. So finding a good communication schedule at the first iteration to be reused at later times does not work. A high-performance code can be developed by addressing the following issues

- Mapping of the BH-tree to processors must change adaptively as the simulation proceeds.
- Efficient fetching and update of the *locally-essential* tree for each body.
- Provision for load balancing to take into account the work each body performs rather than the number of bodies on each processor.

In the following subsections, we describe the important approaches for data partitioning, tree construction, accessing non local data and tree traversals studied in the literature. See Table 2.1 for a summary.

For each time step:

Step 1. Partition the bodies to processors using a k-d tree

On each processor : Step 2. Obtain the *locally essential tree* for this processor

For each body on this processor

Step 3. Start a depth-first traversal of the tree at the root, truncating the search at a internal node where the approximation is applicable.

Step 4. Update the contribution of the node to the acceleration of the body

5. Update the velocity and position of each body belonging to this processor.

Figure 2.4: Parallel implementation of N-body algorithm using adaptive trees

	Phase	Implementation
1.	Tree Construction	Distributed Adaptive Trees (Warren and Salmon 1992)
		Hashed Octree (HOT) (Warren and Salmon 1993)
		Octree in shared memory (Singh, Hennessey and Gupta 1992)
2.	Data Partitioning	ORB tree for bodies. (Warren and Salmon 1992)
		Spatial coordinates to keys (Warren and Salmon 1993)
		Costzones (Singh, Hennessey and Gupta 1992)
3.	Tree Traversal	Latency hiding tree traversal (Warren and Salmon 1992)
4.	Locally essential data	
	(Receiver-oriented)	Gather essential data for force computation (Warren and Salmon 1992)
	(Sender-oriented)	Send essential data to processor needing it (Liu P. 1994)
5.	Incremental Updates	
	(Sender-oriented)	Incremental Tree Updates (Liu P. 1994)
		Incremental updates of locally essential data (Liu P. 1994)

Table 2.1: Parallel approaches to N-body treecodes

2.1.3 Data Partitioning

A distributed octree or a k-d tree representation is maintained on a set of processors. Each processor owns portions of data, the amount of which is guided by the workload associated with it. Every body, on a processor, can only see a fraction of the complete tree in a distributed memory scenario. The distant parts are seen only at a coarse level of detail, while the nearby sections are seen all the way down to the leaves, observing that nearby bodies see similar trees. This means that the upper levels of the tree, have nodes that do not contain data but contain indirection pointers to processors where the data can be found. Data partitioning can be done using orthogonal recursive bisection (ORB), where space is recursively divided in two, and half the processors are assigned to each domain until there is one processor associated with each rectangular domain. A multidimensional binary k-d tree, call it the ORB tree, can be used by alternating the dimensions of the split. A later chapter presents algorithms for the construction of k-d trees in parallel. A copy of the ORB tree is stored on every processor. Each internal node of the ORB tree represents a bisector plane and the domain it bisects, and each leaf node is a processor domain. The ORB decomposition of space among processors allocates points in space to processors. This is useful for *sender-directed* communication of essential data used in relocating bodies which cross processor boundaries and for building the global BH tree. A owner of data can calculate which processors, if any, require its data by looking at the partitioning boundaries. This is in contrast with *receiver-directed* communication, where a data request is sent out and the appropriate processor(s) will fulfill it. ORB preserves data locality well and the cost of incremental load-balancing is negligible [Liu94].

The load distribution changes only slowly across iterations and the ORB can be adjusted with minimum changes to balance the load again. The incremental update begins with each processor computing the total number of interactions used to update the state of the local bodies. A tree reduction yields the number of operations for the subset of processors corresponding to each internal node. A node is overloaded if its weight exceeds the average weight of the nodes at that level by some fixed quantity. A top-down search on ORB tree marks those internal nodes which are not overloaded but one of their children is overloaded. This node is called the *initiator*. Only the processors within the corresponding subtree participate in balancing the load for the region of space associated with the initiator. Since the subtrees for different initiators are disjoint, the non-overlapping regions can be balanced in parallel. The bodies of the overloaded child have to be moved to the non-overloaded child at each step. A new bisector plane is computed so that the right amount of workload can be shifted to the under-loaded child. This is done by determining the weight within the old plane and any given plane to find the correct bisecting plane by a binary search. The workload within the parallelepiped is computed by traversing the local BH-tree [Liu94].

Another approach of partitioning data is by mapping the bodies by converting the locality information in terms of a one dimensional key. Each possible cell is identified with a key and by performing simple bit arithmetic on a key, keys for daughter or parent cells are determined. The translation of keys into memory locations where cell data is stored is achieved via hash table lookup. This scheme provides a uniform addressing mechanism to retrieve data which is in another processor. This representation of data is called the *Hashed Octree* method [WS93]. The key is defined as a result of a map of d floating point

---

For each processor DO in parallel

Step 1. Construct one dimensional keys for each body interleaving its spatial coordinates

Step 2. Sort the keys in an increasing order

Step 3. Divide the list onto processors weighted by the amount of work corresponding to each body

On each processor

Step 4. Construct the tree using the bodies local to the processor.

Step 5. Make copies of branches on every processor to complete the full representation of the tree

Step 6. Gather locally essential data needed for the bodies on this processor

For each body on the processor

Step 7. Traverse the tree representation in the local memory

Figure 2.5: Hashed Oct Tree (HOT) implementation (Warren and Salmon,1993)

numbers, body coordinates in d-dimensional space, into a single set of bits. The floating point numbers are converted into integers and then bits of the d integers are interleaved into a single key. This is identical to Morton ordering (also called Z or N ordering). This function maps each body in the system to a unique key. A hash table is used to map the key to the memory location holding this data. A hashing function maps the k-bit key to the h-bit long address. Collisions in the hash table can be resolved by chaining. During tree traversals, daughter nodes are found by shifting the parent key left by d bits and the result is *OR*'ed to daughter numbers 0 to $2^h - 1$. The key provides immediate *O(1)* access to any object of the tree. Access to data can be generalized to a global accessing scheme implementable on a message passing architecture. By taking advantage of the properties of mapping spatial coordinates to keys, a sorted list of one-dimensional body key coordinates are divided into equal pieces, weighted by the amount of work corresponding to each body. The work for each body is readily approximated by counting the number of interactions the body was involved in on the previous timestep. Using Morton ordered decomposition a processor domain can span one of the spatial discontinuity. *Peano-Hilbert* ordering can be used for domain decomposition, which does not contain any spatial discontinuity, but is harder to describe by body coordinates.

The data distribution described above, leads to a mapping with irregular boundaries. Incremental modification of the tree structures, the HOT tree and the locally essential tree, becomes increasingly difficult in such a case. Since boundaries are no longer uniform, the movement of bodies from one partition to another cannot be ascertained purely by looking at spatial coordinates. Hence, a sender initiated protocol is not suitable for such a scheme.

2.1.4 Tree Construction

Once the bodies are allocated to a processor using the ORB data partitioning, the adaptive BH-tree is constructed by building local trees on every processor using the local bodies only. A local tree is built with respect to the entire spatial domain and not its own processor domain. It represents the local view of the distribution of bodies within a processor domain. The local trees might not be structurally consistent with respect to each other. Local trees are made structurally consistent by adjusting the levels of all leaf nodes which are split by ORB bisector planes. For a tree with N bodies, each leaf in the BH-tree can contain up to L bodies, $L << N$, to adapt to the fast multipole expansion. This accelerates force calculations by reducing the number of tree traversals, but makes level adjustment more tricky.

Updating the BH-tree incrementally by adjusting the levels after each insertion/deletion within the localtree is described in [Liu94]. Consider a body λ in level ℓ of a local tree consisting of n levels ($n >= l$). λ may actually be in a deeper level $\ell' > \ell$ in the global tree. The level of the cell needs to be adjusted down to ℓ' for the owners on the path from ℓ' to ℓ to receive contributions from λ. Only bodies that are not in the correct levels need to be adjusted. If the processor domain covers the entire leaf, consisting of atmost L bodies, the bodies within the leaf do not require adjustment. Otherwise the leaf's domain spans multiple processors. For each such leaf, ρ, define *covering processors* $\mathcal{CP}(\rho)$ as those processors whose domains overlap with ρ. The level of a body then is determined only by its covering processors. Initially the level information of a body λ contains its split leaf and the local bodies in it. The correct level information is found by refining this information further. Each body λ receives level information from each member of $\mathcal{CP}(\rho)$. Then each processor chooses the deeper leaf as the new level.

Once level adjustment is done, each processor computes the center-of-mass and multipole moments of its local tree. Next, each processor sends its contribution to an internal node to the owner of the node. The transmitted nodes are combined by the receiving processors completing the construction of the global BH-tree. At this stage each processor contains a tree with nodes containing bodies it owns and some empty nodes at upper levels to identify all the bodies on other processors. The owner of a tree node is the processor that contains its geometric center. An owner of a node keeps track of the forces and the acceleration of the bodies in it. This mapping function can be easily constructed by different processors consistently using the ORB tree. A local tree node has correct node information if it is completely covered by one processor domain.

During a time step a body may move into another processor domain and has to be moved from one local tree to another. Also, the body may remain in the same processor but move into a new tree node. If there is no body in the leaf then the leaf has to be deleted. If the body joins a leaf with already L bodies inside, it must be divided until no more than L bodies are in any new leaf.

2.1.5 Accessing Locally *Essential* Non Local Data

The ORB has a recursive structure, consisting of $log_2 P$ levels and $P-1$ spatial bisectors to divide data into P partitions. For each processor a top-down BH-tree traversal collects the data that is needed for calculating accelerations for the bodies it owns. This is referred

to as *locally essential* data, some of which might be owned by other processors. This data can be acquired in two ways. In the first method, a processors sends a request for data to the owning processor. The owners of data then fulfill the request by sending the requested data. This can either be demand driven, that is, essential data is requested on a need basis during force computation, or a communication phase before the start of force calculations can accumulate all the essential data. This scheme has the advantage that data partitioning schemes that preserve locality, but result in uneven boundaries, can be used. The disadvantages are that the overheads of fine-grained communications are high, which have been addressed by using multi-threaded tree traversals at the expense of considerable complexity [WS93].

The second method uses a one way message transfer by doing away with requests for data. Each processor exploits the ORB partitioning information, to calculate the processors which require its local data as essential data. A *sender-directed* communication mechanism is used in which every processor identifies its own exportable data, and then exchanges that data with a processor in the complimentary partition on the other side of the bisector. After $log_2 P$ exchanges every processor is in possession of its locally essential tree.

2.1.6 Tree Traversal

Tree traversals are needed in the force calculation phase. By calculating forces on bodies in groups the cost of tree traversal can be amortized over several bodies. The key point here being, nearby bodies on a node will see similar tree structures and will traverse similar paths [Bar90]. A multipole acceptability criteria (MAC) is used to approximate the far-field forces. This determines the depth of the tree traversals for each body or a group of bodies, if that is the case. An additional MAC is the evaluation of the force at any point in a processor's domain. This can be controlled by the distance from the edge of the cell to the boundary of the rectangular domain. The positions of the bodies can be updated by traversing the locally essential trees. Nodes which do not have global information can safely be skipped since that cannot be essential data. There is no communication required in this phase. By calculating accelerations on groups of bodies the vector units in machines can be utilized effectively. There is a reduction in time spent in traversing the tree although the number of calculations increase [Bar90]. A further reduction in tree traversal can be obtained by caching essential nodes. The key observation is that the set of essential nodes for two distinct groups close together in space are likely to have many elements in common, so the cache can be utilized effectively.

A *walk-list* of cell nodes is maintained, which on the first pass contains only the root cell. Each daughter cell of the input walk list nodes is tested against the MAC. If it passes then the corresponding cell data is placed on the *interaction list*. If a daughter cell fails the MAC, it is placed on the output walk list. After the entire input list is processed the output walk list is copied to the walk list and the process iterates. The process continues till there are nodes in the walk list. After this the calculations in the interaction list are processed.

The advantage of this new method is offset by losing the advantage of sender-directed communication. It is difficult for the BH node to compute the set of processors where its data is essential. This is due to the processor domains having complicated shapes because

For each processor DO in parallel:

- Step 1. Build local BH-trees.
- For every time step do:
  - Step 2. Construct the global BH-tree representation
    - Step 2a. Adjust node levels
    - Step 2b. Compute partial node values on local trees
    - Step 2c. Combine partial node values at owning processors
  - Step 3. Owners send essential data to other processors
  - Step 4. Calculate accelerations on owned bodies
  - Step 5. Update velocities and positions of bodies
  - Step 6. Update local BH-trees incrementally
  - If workload is not balanced
    - Step 7. update the ORB incrementally

Figure 2.6: A generic implementation of the N-body algorithm using incremental data structures

partitioning is now done according to the bodies position in the BH-tree. Also, the force computation stage is slowed down by fine-grained communication. The startup cost for a large number of small sized messages is high on current architectures. This is overcome by using multiple threads to pipeline tree traversals and to update accelerations. If a body does not obtain an essential node in its local tree it initiates the communication to get the data and continues with the tree traversal on some other part of the BH-tree. The communication throughput is increased by packing requests/data to/from the same processor into longer messages, at the expense of added complexity in code.

The MAC used above is difficult to calculate because the parallel algorithm requires identification of locally essential data before the tree traversal begins. With a data-dependent MAC it is difficult to determine before hand which non-local cells are required before the traversal begins. A different approach that does not require building the locally essential trees was proposed by Salmon [WS93]. It provides a mechanism to retrieve non-local data as it is needed during the tree traversal. Calculating accelerations is the most time consuming step. Building locally essential trees lets each processor obtain the data it requires for calculating accelerations on the bodies owned by the processor.

2.1.7 Summary

We summarize the phases of computation and communication for N-body simulations and global operations and collective communication requirements that are needed on a distributed memory MIMD machine.

- *Data partitioning:* Data partitioning can be done using k-d trees or using orthogonal recursive bisection. This ensures regular spatial boundaries which can be used for incremental tree updates. However, ORB is not so good in preserving spatial locality. The k-d tree results in some spatial locality. A space filling approach, like the Morton ordering or the Peano-Hilbert ordering provides good data locality properties. They lead to uneven processor boundaries and incremental aspects become hard to implement.

- *Tree construction:* A l-d tree or an octree is employed to apply the multipole approximations for force calculations. A globally consistent tree is maintained by each processor for the bodies it owns, and pointers to locate bodies that are owned by other processors. A *many-to-many communication* phase is needed to make this tree structurally consistent.

- *Incremental tree updates:* As the simulation proceeds and the new positions of bodies are evaluated, the partitioning and the BH-tree need to be updated to reflect the changes. This may involve movement of bodies across processor boundaries. In one case a tree needs *delete* a body and in the other an *insert* operation is needed. The BH-tree needs to be kept balanced for efficient tree traversals. The ORB tree partitions also need to be adjusted incrementally, otherwise the ORB tree would need to be constructed from scratch at every iteration.

- *Gathering essential data:* As the BH-tree is traversed top-down, bodies in some nodes are not available in the local memory. If such a case arises, the body data needs to be fetched from the appropriate processor, information of which is available at the node. This can either be done during the force calculation phase or in a pre-fetching phase, where all data is gathered beforehand and plugged appropriately in the local BH-tree. However, if the MAC is data-dependent, as in the case of [WS93], then essential data cannot be ascertained before force calculations begin. This phase requires several *many-to-many* communication steps.

- *Tree walking:* A node is opened if it fails the MAC criteria. Then each of the daughter cells is traversed recursively. This process starts from the root of the tree. It is interrupted when a node is reached which does not have data in local memory, but has indirection indices to data in other processors. Multiple threads have been used to overlap tree walking with communication delays [WS93].

- *Load balancing:* This issue can be addressed by adjusting the partitioners incrementally. Otherwise, the data partition tree needs to be constructed from scratch at every iteration. Some amount of load-imbalance can be tolerated depending on the cost of such an operation. Space filling curves can be remapped to maintain load balance.

- *Queries:* Region queries asking for points in a region specified by low and high coordinates in all dimensions ($[x_1, x_2] \times [y_1, y_2] \times [z_1, z_2]$ in 3D) are needed to find particles for interaction. *Locally essential data* can be gathered on a processor with an appropriate query.

2.2 Molecular Dynamics

Molecular Dynamics is a widely used technique for the studying liquids, solids, complex molecular systems in Chemistry, Biology, Statistical Physics and Materials Science. Molecular Dynamics simulates the local and global motion of atoms, molecules or some larger unit, by integrating Newtonian equations for a system of N particles. It is a computationally intensive problem and parallel computers have been used to simulate larger and more realistic systems.

Let us consider a system with N particles represented by a collection of positions and velocities $\vec{x_1}, \vec{x_2}, ..., \vec{x_N}$ and $\vec{v_1}, \vec{v_2}, ..., \vec{v_N}$ respectively. Let r_{ij} be the distance between particle i and particle j. The total energy of the system is given by $E = \sum_i \sum_j U(r_{ij})$, where $U(r_{ij})$ denotes the inter-particle potential between particles i and j. Lennard-Jones potential is the most widely used potential in modeling liquid behavior, and is given by the equation

$U(r_{ij}) = 4\epsilon[(\frac{\sigma}{r_{ij}})^{12} - (\frac{\sigma}{r_{ij}})^6]$,

where σ is the length at which the potential crosses zero and ϵ defines the energy scale. A cutoff distance r_c can be defined beyond which the potential is very small and can be taken to be effectively zero. The force is given by the gradient of the potential, $\vec{F_{ij}} = -\nabla U(r_{ij})$ and $F_{ij} = -F_{ji}$. From the forces the acceleration is calculated by using the equation $\vec{a_{ij}} = \frac{F_{ij}}{m}\vec{r_{ij}}$. By integrating the equations of motion , the new set of coordinates and velocities can be found for time $t + \delta t$ from the values at time t.

The computational tasks in molecular dynamics are to find the interacting neighbors of a particle (particles at a distance $\leq r_c$), compute the forces and integrate the equations of motion and this cycle continues for the period of the simulation.

In a system of N particles, each particle interacts with $\frac{4}{3}\pi {r_c}^3\rho$ particles, where r_c is the cutoff distance and ρ is the average particle density. For each particle, a search for neighbors within a distance of r_c is done which takes $O(N^2)$ time. A widely used technique is to divide the space into cells or rectangular domains and search for interacting particles in the neighboring cells. this reduces the complexity of neighbor searching to $O(N)$, which can further be reduced by constructing Verlet neighbor tables and using the neighborhood information over several iterations. The use of cell method also reduces the complexity of constructing the Verlet neighbor table from $O(N^2)$ to $O(N)$. For a rapidly decaying potential, like the Lennard-Jones, the interactions greater than a distance r_c do not contribute and can be ignored. This reduces the search such that for a particle i the search would be done in the neighborhood $\aleph_i = j \mid |r_i - r_j| < r_c$. The physical domain is divided into rectangular domains, called cells, of size $\sim r_c$ so that the search can be restricted to the neighboring cells. Figure 2.7 describes the molecular dynamics algorithm.

N-body methods have been applied to molecular dynamics by using a hierarchical spatial octree to represent data. A list of interacting neighbors is maintained for each particle. This method is called the *Verlet neighbor table method* and a list of interacting pairs for which the interaction forces have to be computed is used to save computation. The overhead of constructing the table is significant and should be amortized by making use of the table foe several time steps. The table can be reused if no pair of particles originally apart at a distance r_s comes closer than r_c, where $r_s = r_c + \delta_s$, δ_s being a safety distance. The table can be constructed by using a "cell" method in $O(N)$ time for N particles.

---

n_update := Counter for updating verlet tables.

Step 1. Initialize particles and geometry.

For each processor DO in parallel:

for $iter = 1$ to max_iter

Step 2. If ($iter$ % $n_update == 0$) Build Verlet tables.

Step 3. (*Communication*) Send and receive neighboring particles.

Step 4. (*Force Computation*) Compute forces between particles.

Step 5. (*Integration*) Integrate equations of motions to obtain new positions and velocities of particles.

Figure 2.7: Molecular dynamics algorithm

2.2.1 Parallelization on distributed memory machines

CHARMM [BH92] is a program which calculates empirical energy functions to model macromolecular systems. Data decomposition and force decomposition methods are employed for parallelization of CHARMM, a molecular dynamics software, on MIMD machines in [HDS92]. Table 2.2 summarizes the different parallelization approaches to molecular dynamics.

Approach	Target Architecture	Description
SPasM (1994)(Los Alamos)	TMC CM-5/Cray T3D	Parallel Cell Method
Parallel Verlet Table (1994) (Boston Univ. and TMC)	TMC CM-5	Builds and uses Verlet neighbor tables for particles in the neighborhood.
Hwang, Das and Saltz (1993) (Univ. of Maryland)	MIMD DMPCs	Data parallel implementation of CHARMM using a library of irregular communication runtime primitives using both data decomposition and force decomposition.
Plimpton (1995) (Sandia Labs)	MIMD DMPCs	Atom decomposition, force decomposition and spatial decomposition for short range molecular dynamics.

Table 2.2: Different approaches for parallel molecular dynamics

A regular force decomposition algorithm partitions atoms evenly over processors. However, the non-bonded interaction list is distributed unevenly and results in severe load imbalance. The nature of force interactions is irregular and hence any parallelization needs to take that in account. A hierarchical nature is embedded in the way force computations are performed. Bonded interactions occur only between atoms in close proximity to each other and non-bonded interactions are excluded beyond a certain cutoff range. To preserve locality and reduce communication it is reasonable to assign atoms that are close to each other on the same processor. A k-d tree can be constructed to distribute the atoms to processors to maintain spatial locality. The amount of computation associated with an atom

depends on the number of atoms with which it interacts.

A parallel cell method and a parallel verlet neighbor table method is described in [ea95]. In these implementations, processors must synchronize after the completion of the distributed force calculation before any processor can begin the update of the particle coordinates. This requiresthe loads to be uniformly balanced to reduce processor idle-time. Force calculations can proceed without synchronization on a MIMD platform. For calculating forces, each pair of interacting particles must have their mutual force calculated by bringing together the particles on a single CPU. This is similar to building the locally-essential trees in the N-body methods. Each processor is mapped a rectangular volume of space using either a k-d tree or an ORB tree. Particles in the cells of this region are allocated to processors. Each particle is owned by some processor, which is responsible for its force calculations. The particles at the boundary region of the processor may be needed by the neighboring processors. An extended space can be defined for each processor (as shown in Figure 2.8). Once the neighbor table is constructed for an iteration, a time step consists of a communication step, a force computation step and an integration of the equations of motion. The communication step is for the particles at the node boundaries to be sent to the neighboring processors. Another communication step is needed for updating the neighbor table to reflect the movement of particles.

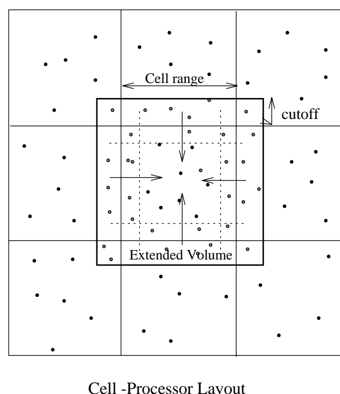


Figure 2.8: Cell processor layout showing the extended volume for particle interactions

2.2.2 Summary

The issues in parallelization of this application for our framework of hierarchical applications are summarized in this section.

- *Data Partitioning:* Equal work should be allocated to all processors. The measure of work is the number of interactions by all the particles on a processor. Particles or cells can be mapped to processors. The amount of work per cell is distribution dependent and hence data partitioning is an important issue to maintain load balance. A balanced k-d tree constructed on the atoms leads to a balanced partitioning. This structure needs to be maintained in parallel as atoms change positions due to forces acting on them.

- *Tree construction:* A k-d tree can be constructed by using orthogonal recursive bisection, which can be traversed to calculate forces on the atoms allocated to a processor. Particles are spatially distributed and we need to create the neighbor interaction list, which can be done by tree traversals. Since the interactions are limited to neighboring cells, the tree traversals will be localized in a region, to access parent and sibling nodes.

- *Incremental Updates:* The neighbor table needs to be updated as particles change position as the simulation progresses. Particles are deleted from and inserted into the appropriate partitions. Doing this incrementally is more efficient than constructing the neighbor list from scratch.

- *Queries:* The queries are of the type where a neighbor list for all particles at a distance r_c need to be identified. These queries are spherical and can be approximated by using a rectangular range query, leading to some duplicated effort. Communication is generated to acquire the appropriate data, which might have a irregular pattern. Collective communication can be utilized to optimize data access requests.

2.3 Volume Rendering

Volume rendering is a technique to visualize three dimensional volume data by projecting it onto a two dimensional plane. It is used in diverse fields like medical imaging, modeling physical phenomenon and molecular structures. Most of these require generating multiple views of the volumes at different orientations from the viewer. Due to requirements for it to perform in real-time, parallel algorithms have been proposed and implemented to accelerate volume rendering.

Volume is visualized by sampled scalar or vector fields of three spatial dimensions without fitting geometric primitives to the data. Images are generated by computing 2-D projections of volume, where the color and opacity at each point (called a volume element or *voxel*) are derived from data using some local operators. Since all voxels participate in the generation of each image, rendering time grows linearly with the size of the data set. The principal advantages of these techniques over others are their superior image quality and the ability to generate images without explicitly defining surface geometry. This gives the images a degree of visual realism.

Rays are cast from every pixel in the image plane into the volume and the data is resampled at regular intervals along each ray. At each sample point, the eight data values are trilinearly interpolated to provide a value and gradient that corresponds to the sample's location. Interpolation is necessary since the volume slice samplings may not coincide with the point the ray is driven through. This value is then classified to give equivalent color and opacity values. The color is shaded by calculating the dot product of the local gradient with each light source which is composited to the ray. Data volumes with contiguous subregions of voxels classified as having zero opacity values do not contribute to the final image and their resampling is unnecessary.

The algorithm described in [Lev88] assumes a scalar-valued array forming a cube that is N voxels on a side. Voxels are indexed by a vector $\mathbf{i} = (i, j, k)$ where $i, j, k = 1, ..., N$,

and the value of voxel $\mathbf{i}$ is denoted by $f(\mathbf{i})$. Using local operators, a scalar or vector color $C(\mathbf{i})$ and an opacity $\alpha(\mathbf{i})$ are derived for each voxel. Parallel rays are then traced into the data from an observer position. It is assumed that the image is a square measuring P pixels on a side and that one ray is cast per pixel. Pixels, and hence, rays are indexed by a vector $\mathbf{u} = (u, v)$ where $u, v = 1, ..., P$. For each ray, a vector of colors and opacities is computed by resampling the data at W evenly spaced locations along the ray and by trilinearly interpolating from the colors and opacities in the eight voxels surrounding each sample location. Samples are indexed by a vector $\mathbf{U} = (u, v, w)$ where (u, v) identifies the ray and $w = 1, ..., W$ corresponds to the distance along the ray with $w = 1$ being closest to the eye. The color and opacity of sample $\mathbf{U}$ are denoted $C(\mathbf{U})$ and $\alpha(\mathbf{U})$, respectively. Finally, a fully opaque background is draped behind the dataset, and the resampled colors and opacities are composited with each other and with the background to yield a color for the ray. This color is denoted by $C(\mathbf{u})$. Working front to back color and opacity are composited at each sample location *under* the ray. Specifically, the color $C_{out}(\mathbf{u}; \mathbf{U})$ and opacity $\alpha_{out}(\mathbf{u}; \mathbf{U})$ of ray $\mathbf{u}$ after processing sample $\mathbf{U}$ are related to the color $C_z(\mathbf{u}; \mathbf{U})$ and opacity $\alpha_{in}(\mathbf{u}; \mathbf{U})$ of the ray before processing the sample and color $C(\mathbf{U})$ and opacity $\alpha(\mathbf{U})$ of the sample by the transparency formula

$$C_{out}(\mathbf{u}; \mathbf{U})\alpha_{out}(\mathbf{u}; \mathbf{U}) = C_{in}(\mathbf{u}; \mathbf{U})\alpha_{in}(\mathbf{u}; \mathbf{U}) + C(\mathbf{U})\alpha(\mathbf{U})(1 - \alpha_{in}(\mathbf{u}; \mathbf{U}))$$

and

$$\alpha_{out}(\mathbf{u}; \mathbf{U}) = \alpha_{in}(\mathbf{u}; \mathbf{U}) + \alpha(\mathbf{U})(1 - \alpha_{in}(\mathbf{u}; \mathbf{U}))$$

Many datasets contain coherent regions of empty voxels. A voxel is defined as empty if its opacity is zero. These do not change the opacity of the ray and need not be rendered. An optimization to improve performance is to ignore empty voxels while rendering. Methods for encoding coherence in volume data include octree hierarchical spatial enumeration, polygonal representation of bounding surfaces and octree representation of bounding surfaces. The second optimization is based on the observation that, once a ray has struck an opaque object or has progressed a sufficient distance through a semitransparent object, opacity accumulates to a level where the color of the ray stabilizes and ray tracing can be terminated. Adaptive termination is implemented by stopping each ray when its opacity reaches a user-selected threshold level.

In this section we will concentrate on methods that use hierarchical spatial enumeration. An octree is used to represent voxel data which helps in skipping empty regions of the data set by appropriate tree traversals[Lev90a]. For a dataset measuring N voxels on a side where $N = 2^M + 1$ for some integer M, the hierarchical spatial enumeration can be represented by a pyramid of $M + 1$ binary volumes. Volumes in this pyramid are indexed by a level number m where $m = 0, ..., M$, and the volume at level m is denoted by V_m. Volume V_0 measures $N - 1$ cells to a side, volume V_1 measures $(N - 1)/2$ cells on a side, and so on upto volume V_m which is a single cell. Cells are indexed by a level number m and a vector $\mathbf{i}$ = *(i,j,k)* where $i, j, k = 1, ..., N - 1$, and the value contained in cell $\mathbf{i}$ on level m is denoted $V_m(\mathbf{i})$. The ray-tracing, resampling and compositing steps now use this pyramidal data structure.

For each ray, the point where the ray enters the single cell at the top level is calculated. The pyramid is then traversed in the following manner: After entering a cell, its value is tested. If it contains a zero, we advance along the ray to the next cell on the same level. If the parent of the new cell differs from the parent of the old cell, we move up to the parent of the new cell. This is done since if the parent of the new cell is unoccupied we can

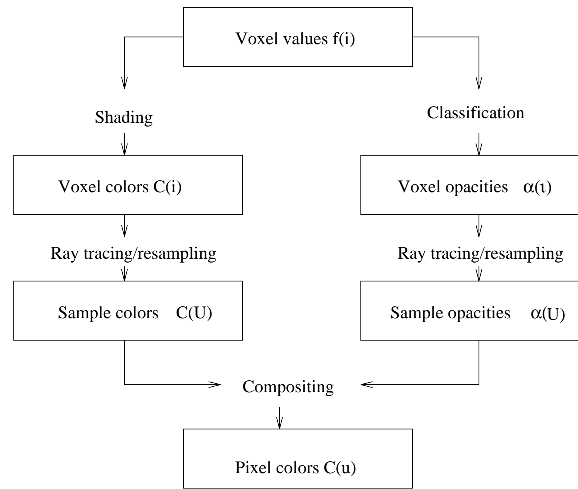


Figure 2.9: Overview of volume-rendering algorithm

advance the ray further on the next iteration than if we had remained at the lower level. If, however, the cell being tested contains a one, we move down one level, entering whichever cell encloses our current location. At the lowest level, samples are drawn at evenly spaced locations along that portion of the ray falling within the cell, resample the data at these sample locations, and composite the resulting color and opacity into the color an opacity of the ray.

Adaptive termination of ray tracing is done by quickly identifying the last sample location along a ray that significantly changes the color of the ray i.e if $C_{out}(\mathbf{u};\mathbf{U}) - C_{in}(\mathbf{u};\mathbf{U}) > \epsilon$, for some small $\epsilon > 0$. Since $\alpha_{in}(\mathbf{u};\mathbf{U})$ increases monotonically along the ray, no significant color change occurs beyond the point where $\alpha_{out}(\mathbf{u};\mathbf{U})$ first exceeds $1 - \epsilon$. Higher value of ϵ reduce rendering time, while lower values reduce image artifacts.

An algorithm in which image quality is adaptively refined over time is presented in [Lev90b]. An initial image is generated by casting a uniform but sparse grid of rays into the volume data, less than one ray per pixel, and interpolating between resulting colors and resampling at the display resolution. Subsequent images are generated by alternately casting more rays and interpolating. Rays are distributed according to measures of local image complexity. Recursive subdivision based on color differences is used to concentrate these rays in regions of high image complexity, and recursive bi-linear interpolation is used to form images from the resulting non-uniform array of colors.

Figure 2.11 shows in two dimensions how a typical ray might traverse a hierarchical enumeration. The level-zero cell corresponding to each nonempty voxel is denoted by a shaded box. The largest empty cell enclosing each empty voxel is denoted by an unshaded box. The sequence of points where the ray enters the next cell at the same level is denoted by circular dots. In regions containing many nonempty level-zero cells, the spacing between these dots is close to the spacing between voxels. These points are not evenly spaced on the ray. If the data is resampled at nonuniformly spaced points, a noise component may be added to the resulting image. To avoid this, a set of evenly spaced sample locations is superimposed, shown as dividing lines in Figure 2.11, and limit to resampling the data at

these locations.

Recently a shear-warped algorithm has been reported in [LL94]. It is currently acknowledged to be the fastest sequential volume rendering algorithm. It is a modification of the object space technique discussed above. Considering a $N \times N$ pixel viewing plane and an $N \times N \times N$ voxel dataset, we observe that $\theta(N^2)$ rays are driven through the N slices of the volume (the volume is N slices of $N \times N$ voxels). A total of N^3 interpolations and $k \times N^3$ resampling weights are computed for each iteration, where k is the number of weights that need to be computed for each iteration. Shear-warp method reduces the resampling calculations to N, by shearing the volume such that each ray can be assumed perpendicular to the slices. Each slice can then be translated and resampled using weights which are invariant across the slices. However, this generates an intermediate image which then needs to be warped to produce the final image. For each pixel in the final image, the four nearest neighbors in the intermediate image are located and the final value of the pixel is interpolated from the color value of these neighbors. This requires $\theta(N^2)$ computation. Using early ray termination, skipping runs of transparent voxels by using run-length encodings, this technique has been improved further.

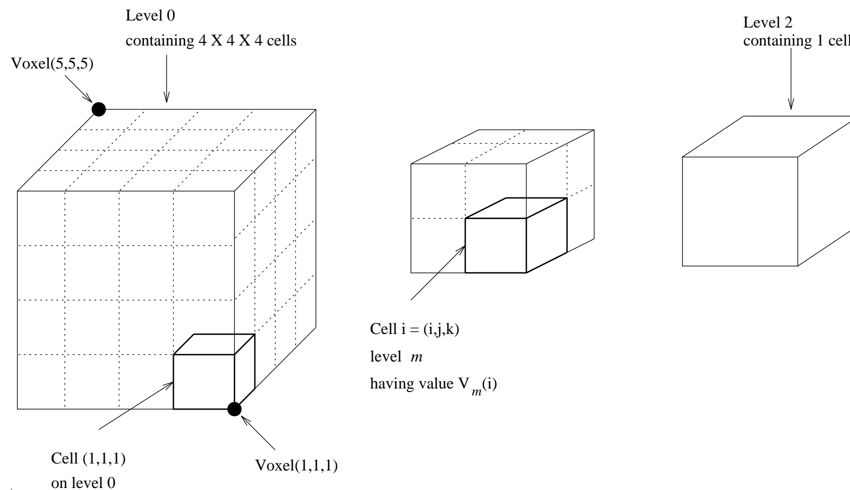


Figure 2.10: Hierarchical enumeration of object space for N = 5

2.3.1 Parallelization on Shared Memory Machines

An implementation of ray traced parallel volume rendering using the single address space distributed memory has been done on the Stanford DASH [NL92]. The data volume is partitioned and distributed to the local memories of the multiprocessor nodes. The image space is statically divided and allocated in equal areas to the processors. Rays are cast by each processor from the subimages that they are responsible for. Voxels required at the resampling points along each ray are accessed directly if available in local memory, or are fetched from the remote nodes where they reside. Dynamic load balancing is realized by allowing idle processors to grab images from others that still have work to do. The algorithm uses an octree representation, adaptive image sampling and early ray termination

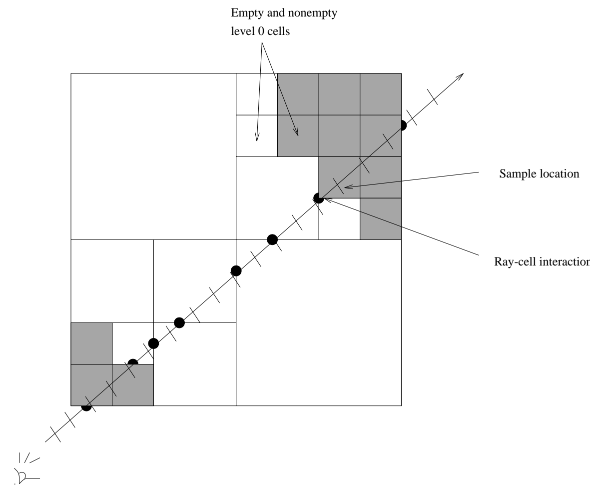


Figure 2.11: Ray tracing of hierarchical enumeration

as optimizations. The performance of the algorithm depends on data coherence. Adjacent rays traversing through the same volume will tend to access the same voxel. Communication overheads will diminish at each processor if this fact is utilized in the algorithm. Though good cache performance is realized in [NL92], no analysis is presented for expecting such behavior. We do not discuss other efforts on shared memory machines, since our primary focus is on distributed memory machines.

2.3.2 Parallelization on Distributed Memory Machines

Parallel data distributed volume rendering was first done on an NCUBE by Montani *et al.*, dividing the volume in slices along one dimension of the volume and using a static load balancing scheme to redistribute the data among processors [MPS92]. Processing nodes are organized as clusters and the image space is partitioned as to assign a subset of pixels to each cluster. A hybrid strategy to ray tracing parallelization is applied, using ray-dataflow within an image partitioning approach.

A ray tracing volume renderer that samples data volumes decomposed and distributed over the local memories of the parallel nodes is implemented by Karia [Kar89]. Every node renders and creates a partial image of the projection of each of the subvolumes it holds. Subsequently, all nodes communicate and composite their partial images in a divide and conquer way to generate the final image. The rendering stage, which requires the bulk of the computation, does not require any communication. Communication is only required in the final compositing stage. Prior to rendering every processor classifies the data stored locally by mapping each voxel's value to its respective color and opacity. After data distribution, certain processors may hold subvolumes that are empty, and avoid rendering them altogether. This creates an imbalance among the nodes in the amount of useful rendering being done for the volume.

Green and Paddon [GP90] have discussed methods of exploiting coherence of references

to entries in the object space which use a combination of dynamic and static caching techniques. A frequency distribution of object usage is determined by measuring the frequency of reference of objects while processing different rays. They have advocated the use of a software cache on a processor to exploit the *locality of reference* that arises from consecutive memory references to similar addresses in main memory.

The colors and opacities computed at each sampling point along a ray are composited using the **over** operator. For any two sample points S_i and S_j, whose colors and opacities are respectively $[c_i, \alpha_i]$ and $[c_j, \alpha_j]$, their composition using the over operator is defined as $S_i \quad \textbf{over} \quad S_j = S_i + (1 - \alpha_i)S_j$

The *over* operator is associative [MPHK93]. Hence, its application to any sequence of samples $S_i ... S_n$ may be grouped arbitrarily as follows

$$\begin{array}{l} (S_1 \quad \textbf{over} \quad S_2 ... S_i) \\ \textbf{over}\ (S_{i+1} \quad \textbf{over} \quad S_{i+2} ... S_j) \\ \textbf{over}\ ... \\ \textbf{over}\ (S_{k+1} \quad \textbf{over} \quad S_{k+2} ... S_n) \end{array}$$

Table 2.3 summarizes the various approaches that have been reported in the literature for volume rendering on distributed memory machines.

Approach	Target Architecture	Description
Montani et al. (1992)	nCUBE	Hybrid image partitioning - ray dataflow approach. Processing nodes organized as a set of *clusters*. Image space is partitioned, Volume data is replicated on each cluster. Static load balancing is used for distributing data.
Nieh (1992)	Stanford DASH	Data interleaved among processor memories. Image partitioned into contiguous blocks for assignment to processors. Task Stealing is used for dynamic load balancing.
Schröder & Stoll (1992)	CM-2	Data parallel SIMD implementation, rays proceed in lock step
Vezina et al. (1992)	MP-1	Also SIMD with volume transposition to localize data access
Ma, Painter et al. (1993)	CM-5	Static input data partitioning into subvolumes using a k-D tree Processing nodes perform local raytracing of their subvolume concurrently.
Karia (1994)	Fujitsu AP1000	Data is decomposed into subvolumes and rendered locally on each processor Scattered decomposition is used for load balancing.

Table 2.3: Different approaches on parallel volume rendering

2.3.3 Data Partitioning and Coherence Issues

Parallelization strategies for volume rendering have two goals. Each processor needs to be assigned equal load and any mapping of data to processors needs to maintain locality. The former helps to reduce processor idle time and the latter helps in keeping overheads of communication low. These are referred to as *load balancing* and maintaining *data locality*, two often conflicting goals. We discuss each of these to motivate the approach we have taken to analyze requirements for each of the two goals.

Strategies used for data partitioning are classified as follows.

Image Space Partitioning The pixels of an image are distributed across processors. Each processor traces rays for the pixels assigned to it. The volume data is replicated on each processor. Portions of the image from each processor are then combined to yield the final images. This method achieves near linear speedup but is not feasible if the object data set is larger than the available memory on each node.

Object Space Partitioning The volume data is partitioned and distributed among processors. Each processor traces each ray in the local partition only. Each non-resolved ray is transmitted to the next processor for further tracing. Once each ray has finished, the final composited values are collected to form the final image.

Object Dataflow A partition of the image is assigned to each processor, which locally traces and resolves each assigned ray. Volume data is partitioned among nodes too. Non-resolved rays will be sent to appropriate processors for tracing and the "owner" of the ray will get the finished result back.

Image/Object Partitioning The volume data is partitioned among processors. The image data is also partitioned among processors. Each processor is responsible to trace rays from pixels assigned to it. Pixels may be traced in the local volume data that is in the processors memory or it might fetch data that it needs from other processors.

Communication costs are typically higher than computation costs on most real machines. To keep communication costs down, various forms of data locality need to be exploited. There are three kinds of coherence in images which we discuss next.

Image Coherence Image coherence is the property that adjacent pixels of an image are illuminated in a similar way. Portions of the image are similar in nearby areas, and this fact can be exploited when allocating pixels to processors. Nearby pixels go to same processors. This coherence exists in two dimensions. Exploiting this property for load balancing is not as straight forward. In an irregular image, where some portions are bright and some are dark, the work done in compositing a ray can vary a lot. Any load distribution strategy should take that into account.

Object Coherence Adjacent rays will travel similar portions of the object. This is the essential idea of object coherence. In hierarchical volume representations, ray intersections with the octree can be optimized for adjacent rays (explained in a later section). This helps in reducing communication costs for essential volume data on processors for ray tracing by allowing reuse of offprocessor data. The data can be incrementally modified as the previous data can usually be reused. Another kind of coherence is available as the rays traverse the slices of the volume. The volume data gradually changes as the ray traverses through each slice. From one slice to another *slice coherence* is present as seen in Figure 2.12.

Frame Coherence Consecutive frames in a multiframe sequence are quite similar. Figure 2.12 shows frame coherence for the *brain* dataset. This fact is used in the dynamic load balancing strategy that we have proposed in [GR95]. Workload information from previous frame can be used to predict the load characteristics of the current frame.

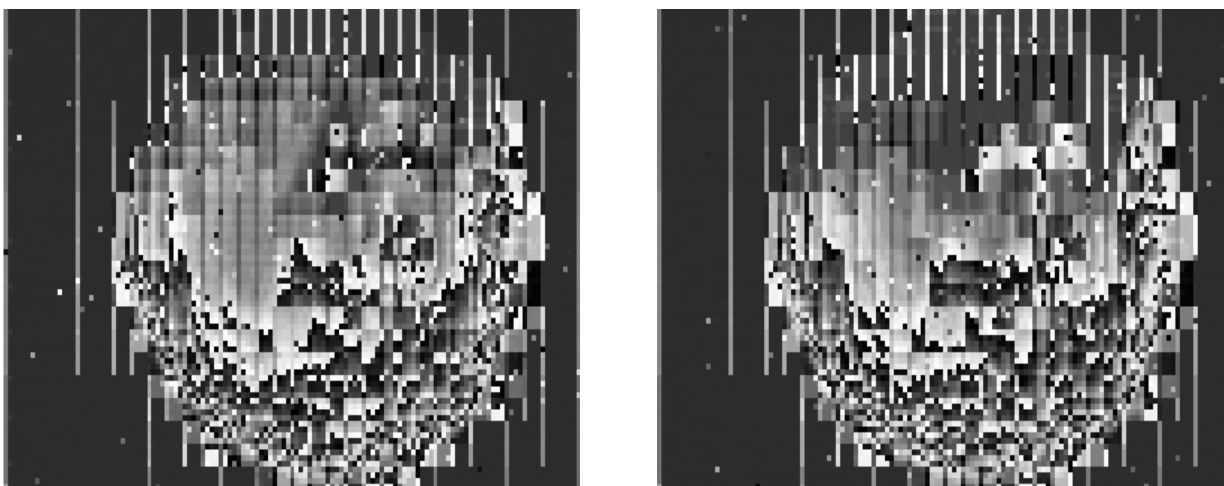


Figure 2.12: Frame 1: Ray work profile for **brainsmall** and Frame 2 at a 5° rotation - Each pixel shows the computation work of a scanline. Scanlines for a slice progress from left to right on the horizontal axis. Slices of a frame are top to bottom on the vertical axis.

Data partitioning for the image space needs to provide image coherence. By proceeding scanline by scanline within a slice, scanline coherence is exploited. To preserve this, image partitioning is done by dividing the scanlines to processors, keeping the load balanced. A record of the load on each processor is kept which is used to partition scanlines in the next frame. The first frame is load-balanced by looking at the load of scanlines of each slice on the processor. This strategy works well with a replicated object. For distributed volume, data partitioning the image and the volume are complimentary. Allocation of rays to processors needs to be guided by the volume data that is assigned to a processor. Two dimensional locality needs to be exploited for maximum reuse of data on processors to keep the communication costs low.

Each ray intersects the tree starting at the root. Tree traversal is done to find the smallest cell with a uniform color (white or black) that the ray intersects in a slice. The size of the cell determines the number of rays that are covered by this traversal. There is no need for additional tree traversals for these rays for the slice. This is done for each scanline in a slice. Rays in the scanline then pass through each slice accumulating opacities as they traverse the volume.

2.3.4 Summary

In this section we present the requirements for maintaining hierarchical data structures for volume rendering of multiple scenes of a 3D volume.

- *Data Partitioning:* Pixels from the resulting 2D view frame are distributed to processors, such that the work on each processor is balanced. This cannot be ensured by allocating equal number of pixels, since pixels have different work associated with

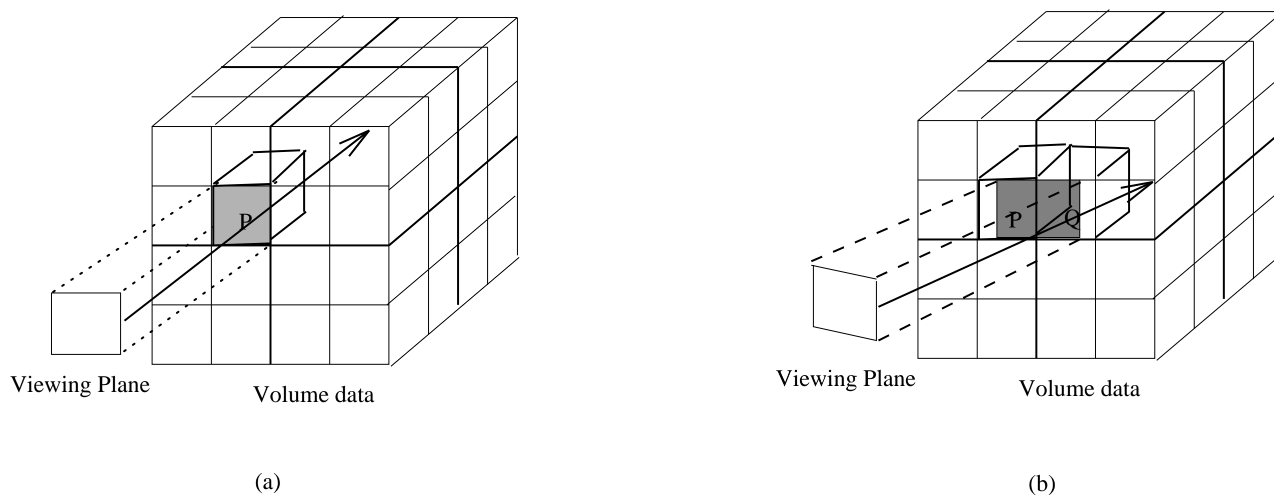


Figure 2.13: Ray intersections with volume (a) Viewing plane is aligned with the XY plane (b) Viewing plane is at an angle from the XY plane

them. A measure of work of rays being traced through pixels is maintained to estimate the workload on a processor. Work is represented by resampling, translation, tree traversal and compositing for the ray fired through a pixel. Scanline coherence can be preserved by allocating contiguous rows of scanlines to processors. The number of scanlines per processor would be data dependent and can either be adjusted between slices or between frames. See [GR95] for details.

- *Tree Construction:* The object data is represented by an octree. In parallel implementations, a distributed tree needs to be maintained to distribute data across the available processors, considering that each processor has memory only addressable by it. A k-d tree can be maintained to organize spatial volume data into partitions. A local tree can be built from the data a processor owns.

- *Locally essential data:* Each processor fires rays into the volume through the pixels it owns. Since the volume data is distributed, rays might need to fetch offprocessor volume data for compositing calculations. It can be determined by each processor independently which processor requires the data it owns, depending on the orientation of the view plane. A *sender-oriented* protocol can be used to supply each processor with the data it needs for intersection and compositing calculations.

- *Incremental updates:* For rendering multiple sequences, with a change in the viewing angle, the rays fired from the pixels of each processor will intersect different parts of the volume data. Owing to object coherence and small angle changes in contiguous frames, most of the offprocessor data can be reused. The tree can be incrementally modified for each frame, instead of fetching all the data from scratch. Figure 2.13 shows the incremental data region when the view angle is changed by a small amount.

- *Tree traversals:* Intersection of rays with voxel data at the appropriate level by skipping the empty regions is achieved by a traversing the tree from the top and finding

the cell node with a homogeneous color. Intersection calculations are done at the highest level so that unncessary calculations can be saved.

- *Queries:* Region queries need to be supported to gather off-processor voxels for composition of a portion of the volume through which the rays pass. Given a region in the image space, a volume is returned, whose voxels are required for image composition.

2.4 Adaptive Meshes

Many large physical systems can be modeled and represented by partial differential equations. Multigrid methods have been used to find solutions to these equations. A continuous domain can be discretized by overlaying a grid on top of it. The new discretized domain is now defined by the grid points. The grid points define a mesh. The characteristics of the solution guide the structure of the starting grid. In many problems, portions of the domain where high resolution is desired are localized. These allow the use of adaptive mesh refinement techniques, which allow a finer solution in more interesting areas of the problem domain and coarser solutions in areas of lesser interest. This can be considered as a multilevel method, where new levels are created by subdividing the mesh at the existing level. Subdivision is adaptive and is controlled by an error estimate. The lower the error tolerance for a specific subdomain, the finer is the mesh at a deeper level of the grid hierarchy. Adaptive mesh refinement can be considered as a hierarchical tree representation of grids with each level, except the root, representing a uniform subdivision of the parent grid. The root represents the starting mesh with the initial grid points.

The Berger-Oliger adaptive mesh refinement scheme [BO84] is popularly accepted for the formulation of adaptive finite difference (AFD) methods. An adaptive grid hierarchy can be represented as a directed acyclic graph (DAG), where each node of the graph represents a component grid. The root corresponds to the base grid and the levels of the DAG correspond to the refinement in the grid hierarchy, nodes at the same level comprising of the component grids at the same level of refinement. A node can be denoted as G_n^l, $0 \leq l < L$, L being the lowest level in the hierarchy, and $0 \leq n < 2^l$. The adaptive grid hierarchy is shown in Figure 2.14. The grid spacing at a level l is an integral multiple of the grid spacing at level $l+1$. Also, component grids at the same level must be locally uniform with space and time resolutions. The AFD integration algorithm defines the order of operations on the grid hierarchies and is composed of the *time integration*, *error estimation and regridding* and *inter-grid operations* components. All component grids at a level $l+1$ must be integrated to the current time T before integration begins for level l at time $T + \delta t$. Regions needing refinement are flagged based on the error estimate and a refined grid is generated wherever the error estimate is greater than required. Inter-grid operations are needed for the following:

- Initialization of the refined component grids by using the interior values of an intersecting component grid at same level or by prolongating values from the underlying coarser component grid.

- Updating underlying coarse component grids using values on a nested finer grid integrated to the same time T. This can be done by using the same values (injection) or

an appropriate interpolation.

- Averaging any overlapping component grids at any level to update the coarser component grids underlying the overlap region.

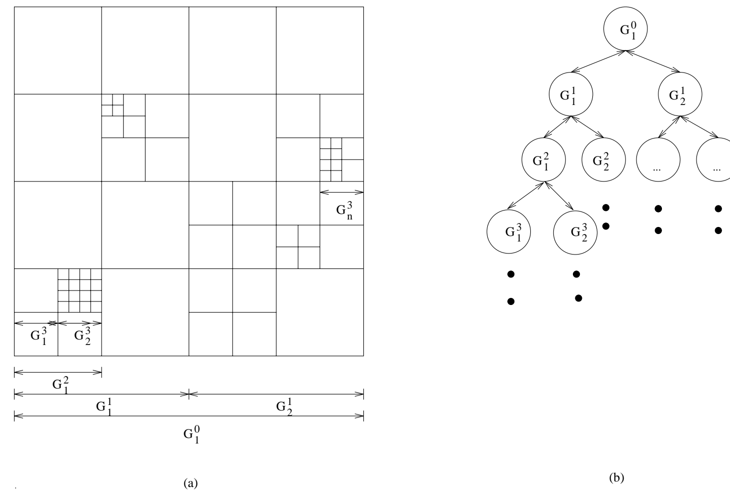


Figure 2.14: (a) Grid refinement and (b) associated grid hierarchy

The hierarchical structure of adaptive mesh refinement technique can be modeled as the generic tree-structured computation we are addressing in this paper. Inter-grid operations can be viewed as operations on tree nodes.

2.4.1 Parallelization on distributed memory machines

A parallel implementation of the adaptive mesh refinement method will require operations on the hierarchy of grids which include, creation of the grid, grid partitioning among processors, communication among grids at one level and communication between grids at different levels. This allows us to primarily exploit *data-parallelism* by partitioning the grids across processors and *task-parallelism* expressed as independent updates to component grids and integration across levels in the grid hierarchy. The refinement at a particular level is not known a priori and hence can differ across the processors. This can lead to imbalance in the amount of work allocated to each processor. Hence dynamic load balancing is needed at runtime to distribute the work evenly among the processors.

A distributed DAG of grids has to be maintained across the processors. This is updated when grids are refined at each level. Communication is generated for inter level grid updates since every processor needs to keep a global view of the grid hierarchy. Prolongation defined from a coarser grid to a nested finer grid, and restrictions defined from a finer grid to it's parent grid, generate irregular communication of *scatter/gather* nature. Generally, a finer grid is distributed over a larger number of processors than a coarser grid, since the former has more grid points and hence more work associated with it. Intra-grid interactions may result in near-neighbor communication. This can be overlapped with computation if the interior-to-boundary ratio is large. Communication patterns are random when component

grids are clustered or during grid redistribution. The grid decomposition scheme is critical to the effective parallelization of AFD methods. A composite distribution scheme can limit the cost of irregular communication for inter-grid updates by mapping the overlapping grids at different levels to the same processor. Locality maintaining mappings like the space filling curves define a mapping from a d-dimensional space to a linear ordering. Especially, *Morton ordering* and *Peano Hilbert ordering* are useful in partitioning space to preserve locality. Data partitioning can be performed by splitting the one-dimensional list appropriately. Redistribution can also be done by readjusting the points of the split. Grid partitioning among processors can be done in one of the following ways [PB95]:

- *Individual grid partitioning:* Each grid at every level is partitioned across processors. This generates communication pattern in which many processors may communicate with one processor generating a bottleneck. This happens when a fine grid needs to update its underlying coarse grid. Moreover, inter-level parallelism is not expolited since the grids at different levels are allocated to the same processor and their integration is sequential.

- *Comprehensive partitioning:* The work total across all the grids at all levels is distributed across processors. This distribution does not exploit the parallelism available in the problem since it will leave some processors idle while performing update calculations.

- *Independent level partitioning:* Each level of the grid hierarchy is partitioned independently. This scheme expolits the parallelism across grids at the same level, but generates irregular communication that if not scheduled appropriately may cause a bottleneck.

The fast adaptive composite grid method (FAC) [MQ89] is a multilevel scheme that nominally uses global and local uniform grids for adaptive solution of partial differential equations. It provides parallelism by producing several independent refinement regions. Asynchronous version of FAC (AFAC) allows for processing of the refinement levels in a parallel mode. Using multigrid as the individual grid solver, FAC has been applied to a variety of fluid flow problems, including incompressible Navier-Stokes equations, in both two and three dimensions. The local grids add computational load irregularly to processors. Load balancing is thus required to implement AFAC efficiently on distributed memory machines. The independence of the various refinement levels in the AFAC process allows the assignments of computational tasks to processors to be made level by level. This simplifies the load balancing, since the levels can be ordered in a list and the partition of such a list is inherently one-dimensional. The relationship between the levels of the composite grid is expressed in a tree of arbitrary branching factor at each vertex. Each grid exists as a data structure at one node of this tree. The composite grid tree is replicated in each node. In the case of a change to the composite grid (adding, deleting or moving a grid), the change is communicated globally to all processors so that the representation of the composite grid in each processor is consistent. Storage of the matrix values uses a one-dimensional array of pointers to row values. These rows are allocated and deallocated dynamically, allowing the partitioned matrices to be repartitioned without recopying the entire matrix.

Edelsohn [Ede93] has discussed the motivation of using the hierarchical approach for adaptive subdivision of meshes.

2.4.2 Summary

- *Data Partitioning:* Space filling curves can be used to partition the grid hierarchy across the processors. By maintaining appropriate data structures, repartitioning during load balancing can also be achieved.
- *Tree Construction:* Grid hierarchy represented as a distributed DAG is a hierarchical representation with an arbitrary degree at each node. A distributed tree structure with the appropriate fields is constructed as the base grid.
- *Incremental updates:* The grid hierarchy is dynamic and hence needs to be updated at each level. Indexing by code of the space filling curve can be used to move data appropriately.
- *Tree traversals:* Inter-grid and intra-grid updates need to go up or down in the grid hierarchy. This can be modeled as tree traversals on the nodes which basically are coarser or finer grids at different levels of the tree.

2.5 Databases

Applications that rely on spatial databases can be found in CAD, VLSI, robotics, Geographical information systems, Online Analytical Processing (OLAP) and a host of other areas. Lately, the challenges have been posed because these databases have grown in size and spatial representations to represent such data for compact storage and efficient querying have become issues. The use of hierarchical data structures in spatial databases enables focus of computational resources to regions of interest in the data set. They are attractive due to their compact representation and facilitate search and update operations. Spatial data usually made of different data types such as regions, points, lines, polygons and volumes can be represented by hierarchical data structures. Some queries on these data structures are described in a later chapter.

2.5.1 Spatial Indexing

Spatial occupancy methods, known as *bucketing methods*, decompose the space from which data is drawn into regions called *buckets*. There are several methods to achieve bucketing. One approach uses the minimum bounding rectangle to group objects into hierarchies and then store them in another structure such as the B-tree. This is the R-tree method [Gut84]. The R-tree and its variants are designed to organize a collection of arbitrary spatial objects (e.g. 2D rectangles) by representing them as d-dimensional rectangles. The objects are represented by the smallest aligned rectangle representing them. Each node in the tree corresponds to the smallest d-dimensional rectangle that encloses its son nodes. Leaf nodes contain pointers to the actual objects in the database. A R-tree or a R^+-tree of order (m, M) has the property that each node in the tree, except the root, contains between

$m \leq \lceil \frac{M}{2} \rceil$ and M entries. The root node has atleast two entries unless it is a leaf node. Bounding rectangles corresponding to different nodes may overlap. Also, an object may be spatially contained in several nodes, yet it is associated with only one node. A spatial query can visit several nodes to find the existence of an object.

Often the nodes correspond to disk pages and the parameters defining the tree can be chosen to minimize the number of nodes visited during the spatial query.

The decomposition of space into disjoint cells can be achieved by constructing a R^+-tree. To determine the area covered by a particular object, all cells that it occupies need to be retrieved. Deletion is also expensive, but it is not a very frequent operation. A related drawback is that a query to determine all objects in a given region, retrieves many of the objects more than once. Method such as the R^+-tree and the cell tree have data-dependent decompositions which is their drawback. In contrast, the uniform grid and the quad-tree method are data-independent and also lead to disjoint decompositions.

The uniform grid is ideal for uniformly distributed data, while quad-tree based approaches are suited for arbitrarily distributed data. Both these structures are amenable to set-theoretic operations, like composition, on data they represent. Quad-tree methods, however, are sensitive to the placement of data with respect to the decomposition lines of the space, which affects the storage costs and the amount of decomposition that takes place.

Region data can either be represented by its boundary or its interior. Using the interior representation as an image array, there is an interest in reducing the image array by aggregating similar pixels. A *region quadtree* is based on successive subdivision of the image array into four equal-sized quadrants. If the given array does not contain entirely of 1s or 0s, it is subdivided into quadrants until homogeneous blocks are obtained. The root node corresponds to the entire array. Each son represents a quadrant. Octrees can be used to represent volume data similarly.

Rectangles are often used to approximate other objects in an image for which they serve as the minimum rectilinear enclosing object. Bounding rectangles can be used in cartographic applications to approximate objects like lakes, forests etc. The approximation gives an idea of the object, and the exact object boundary is retrieved as a refinement step. Rectangles are also used in VLSI design rule checking as a model of chip components for analysis of their proper placement. The size of the collection is quite large here and the sizes of the rectangles are several orders of magnitude smaller than the space they are drawn from.

There are several data structures that are used to represent spatial data. Their choice depends on the nature of the data, whether it is point, line, polygon or region data, and the queries that can be efficiently answered on them. Table 2.4 summarizes the data structures proposed for different data types.

2.5.2 Parallelization

Performance of spatial operations in a data parallel environment using hierarchical data structures has been described in [HS94]. The spatial operations require a significant amount of computation to make the use of parallelism attractive. Data structure creation, polygonization and spatial joins are some of these. The algorithms presented are main memory resident.

	Data	Spatial data structure
1.	Point	PR quadtree, k-d tree
2.	Line	R-tree, R^+-tree, R^*-tree, PMR quadtree
3.	Region	Region quadtree
4.	Rectangle	R-tree, R^+-tree, MX-CIF tree

Table 2.4: Hierarchical data structures for spatial data used in practice

Polygonization is the process of determining all closed polygons formed by a collection of planar line segments. Spatial joins are quite common in databases. Two elements in space are joined when the elements cover identical spaces. Data parallel algorithms using PMR quadtree, R-tree and R^+ tree are discussed in [HS94]. A data parallel PMR quadtree for n line segments can be built in $O(\log n)$ time. A data parallel R-tree construction operation takes $O(\log^2)$ time, where each of the $O(\log n)$ subdivision stages requires $O(\log n)$ computations, which include a constant number of scans along with two bounding box sorts.

Map intersection and a spatial range query find all lines in a map that are within a distance d of any line in a second map. Polygonization is the process of determining all closed polygons formed by a collection of planar line segments. A description of the data parallel algorithms for the above is given in [HS94]. They report better performance of these operations using PMR quadtrees when compared to R-trees or R^+-trees. This is due to the property of regular decomposition of space in the PMR quadtree which is not the case for the other two. Spatial joins are efficiently implemented without using a spatial decomposition such as employed by the R-tree and the R^+-tree. This implies that the use of irregular and non-disjoint decompositions invoke a higher overhead when compared to regular and disjoint decompositions.

Parallelization of data structures representing point data sets can be achieved by using parallel algorithms for k-d tree construction and quadtree construction. Parallel algorithms for k-d tree construction are presented in Chapter 4. Incremental aspects are not discussed and will form part of our future work in this area.

2.5.3 Summary

The following attributes lead to our categorization of spatial databases in the class of applications that need hierarchical data structures. These will benefit from the run-time support provided for hierarchical algorithms on parallel machines.

- *Data Partitioning:* The data to be represented is distributed among the processors either using spatial proximity or in a round robin manner to maintain load balance. The issues are the same as in other applications and are also guided by the queries posed on these structures. If these issues are attached with large data sets and I/O using multiple disks, then for small queries, a small number of disks should be activated, and for large queries a large number of disks should participate for good performance.

- *Tree Construction:* Efficient algorithms for construction of trees for point, line, rectangle and region data are required on distributed memory parallel machines. These data structures need to be maintained to make the implementation of queries efficient. In particular, k-d trees, quadtrees, PMR quadtrees, R-trees and its variants are some of the applicable data structures.

- *Queries:* The *raison d'etre* for any database is to store data and answer queries posed on it. The range of queries is wide and varied. Queries involving multiple parameters are posed as queries on multidimensional data, where each parameter is seen as a dimension. Temporal queries can be seen as queries with time as an extra dimension. Range queries, nearest-neighbor queries, OnLine Analytical Processing (OLAP) uses multidimensional data processing and has enjoyed spectacular growth in recent times. Data organization on disk can be guided by spatial data structures and spatial indices can be maintained for data access.

2.6 Hierarchical Radiosity

A more complex problem using hierarchical algorithms is that of calculating radiosity of a scene in Computer Graphics. The *radiosity* of a surface is defined as the light energy leaving the surface per unit area. Given a description of a scene the idea is to calculate the radiosities of all surfaces resulting in the calculation of illumination of the scene. A *scene* is a collection of large polygonal patches. These polygons are subdivided into small enough *elements* that the radiosity of an element can be assumed to be uniform over its surface. Any larger piece is termed as a *patch*, formed by combining elements or other patches including the original polygon. The radiosity of an element i can be expressed as a linear combination of all other elements j. The coefficients in the linear combination are the **form factors** between the elements. Form factor between element j and i (F_{ji}) is the fraction of light energy leaving element j arriving at i. This leads to a linear system of equations which can be solved for the element radiosities once all the form factors are known. Enforcing a energy balance at every element yields a system of equations of the form :

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} B_j$$

where B_i is the radiosity, E_i is the emissivity, ρ_i is the diffuse reflectance, F_{ij} is the form factor and n is the number of elements in the scene. This system of equations can be efficiently solved using iterative techniques like the Gauss-Siedel method. Its physical implication is equivalent to each patch successively *gathering* light. The other alternative is of *shooting* light from patches in order of their brightness. The most expensive part of the calculation is computing form factors, given for two infinitesimal elements by

$$F_{ij} = \frac{\cos\theta_i \cos\theta_j}{\pi {r_{ij}}^2} dA_j$$

The angle $\theta_i(\theta_j)$ relates to the normal vector of element i(or j) to the vector joining the two elements.

To take into account occlusion, differential form factors are accumulated only if the two infinitesimal elements are mutually visible. The form factor matrix is $n \times n$, where n is the number of elements. The order of form factor calculation is thus $O(n^2)$. Applying the insights of the N-body problem that

1. Numerical calculations are subject to error, and therefore, the force acting on a particle need only be calculated to within a given precision.

2. The force due to a cluster of particles at some distant point can be approximated within a given precision, with a single term, reducing the total number of interactions.

the complexity is reduced to $O(n + k^2)$, where k is the number of polygons.

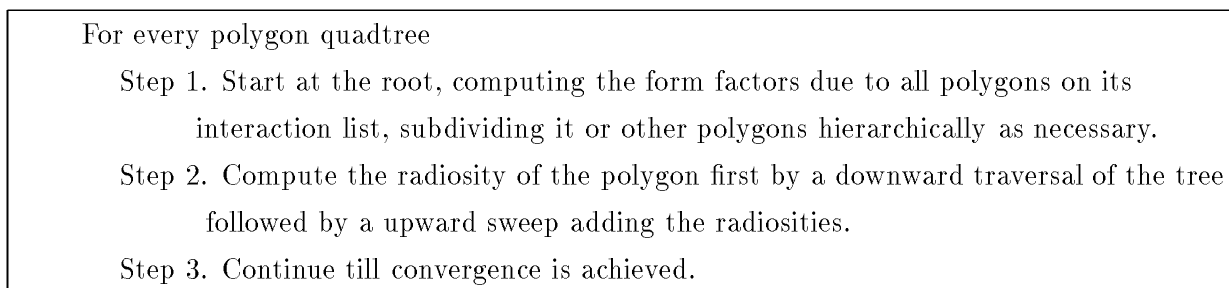
The radiosity problem shares many similarities with the N-body problem. First there are $n(n-1)/2$ pairs of interactions in both. The magnitude of the form factor falls off as $1/r^2$, same as the gravitational force.

The major difference between the two problems is the way the hierarchical data structures are formed. The N-body algorithm begins with n particles and clusters them in larger and larger groups the hierarchical radiosity algorithm begins with a few large polygons and subdivides them into smaller and smaller patches. Subdividing is based on the error of a potential interaction which gives an automatic method for discretization of the scene. The principle of superposition, that the potential due to cluster of particles is the sum of the potentials of the individual particles, cannot be directly adopted because of occlusion. Intervening opaque surfaces can block the transport of light between two other surfaces which makes the system non-linear. Lastly, the N-body problem is based on a differential equation , whereas the radiosity problem is based on an integral equation.

2.6.1 Sequential Algorithm

The input to the algorithm is a set of polygons depicting the scene. These are inserted into a Binary Space Partitioning (BSP) tree [FAG83] to facilitate efficient visibility computation between pairs of patches. Every input polygon is initially given a list of other input polygons that are potentially visible from it to enable it to compute interactions. The polygon radiosities are computed by iterating over the steps shown in Figure 2.15.

The tree data structure used here is not a single tree but a forest of quadtrees representing individual polygons. Each polygon has its own quadtree, with the roots being leaves of the BSP tree used for visibility testing. At every quadtree node visited in this traversal, interactions of the patch at that node are computed with all other patches in its interaction list. The interaction between two patches involves computing both the *visibility* and the *unoccluded form factor* between them and multiplying the two to obtain the actual form factor. Both of these quantities are computed approximately, introducing an error in the computed form factor. An estimate of the error is also computed. If this is larger than a user defined tolerance the patch with the larger area is subdivided to compute a more accurate interaction. Children are created for the subdivided patch in its quadtree if they do not already exist. If the patch being visited (say i) is subdivided, patch j is removed from its interaction list and added to each of its children's interaction lists. If patch j is subdivided, it is replaced by its children on patch i's interaction list. Patch i's interaction



For every polygon quadtree

Step 1. Start at the root, computing the form factors due to all polygons on its interaction list, subdividing it or other polygons hierarchically as necessary.

Step 2. Compute the radiosity of the polygon first by a downward traversal of the tree followed by a upward sweep adding the radiosities.

Step 3. Continue till convergence is achieved.

Figure 2.15: Radiosity algorithm

list is completely processed in this manner before visiting its children in the tree traversal. During this downward traversal, the radiosities gathered at the ancestors of patch i and at patch i itself from other patches are accumulated and passed on to i's children. After the traversal of a quadtree is completed, an upward pass is made through the tree to accumulate the area-weighted radiosities of a patch's descendants into its own radiosity. Thus the radiosity of a patch is the sum of three quantities

1. Area-weighted radiosities of its descendants
2. The radiosity a patch gathers in its own interactions
3. The radiosity gathered by its ancestors

Parallelism is available at three stages. First, the polygons are independent of each other and can be processed simultaneously. Second the visibility computation can proceed in parallel. Third, the interactions computed for patches can be done in parallel. Singh [Sin93] has discussed the parallel approaches on shared and distributed memory machines.

2.6.2 Parallel approaches on a shared memory machine

To exploit the parallelism across polygon-polygon interactions every processor needs to be assigned an equal number of them. This can either be done statically or processes can obtain interactions dynamically until none are left in the queue. It has been shown in [Sin93] that the dynamic scheme works better than the static scheme. Apart from the distance between two patches, the angle and visibility between them are also important factors for interactions. Usual schemes that rely only on spatial distribution thus do not suffice. There are three primary forms of locality in the application.

1. A form of object locality can be obtained by having a processor work mostly on interactions involving the same input polygon or its subpatches in every iteration.
2. Locality can be exploited across each patch by processing sibling patches consecutively, using a breadth first traversal of the quadtree.
3. During visibility testing, using a depth first search, locality is exploited by ensuring that the same subset of BSP-tree nodes is traversed by a processor in successive visibility calculations

Load balancing can be provided by allowing on the fly task stealing. Each processor has a queue of polygons on which the interaction have to be calculated. A processor can either steal polygons from other processors and process them or steal patch-patch interactions. The granularity of tasks is a patch and all its interactions in the first case and a single patch-patch interaction in the second. If an interaction subdivides one of the patches and thus spawns a new interaction it is placed at the end of the creating processor's queue.

2.6.3 Parallel approaches on a distributed memory machine

The local quadtrees approach lets each processor maintain local copies of all the quadtree data that they need, modify the data locally as needed in an iteration and only communicate the modifications to other interested processors at iteration boundaries. In the absence of task stealing, the approach consists of phases of local computation punctuated by phases of communication. Also, there is no complete logical version of the forest of quadtrees on any one processor. Polygons are statically assigned to processors. The growing interactions on processors might lead to load imbalance in the system. Allowing idle processors to steal tasks from other processors can lead to load balancing.

In the global quadtrees approach a single copy of the forest of quadtrees is maintained in distributed form over the processing nodes. Every processor holds the BSP tree and the quadtrees assigned to it in its local memory and knows the locations of the rest of the quadtrees through a global naming scheme. A processor can store non-local data it references in a local cache. To maintain coherence caches can be flushed at iteration boundaries. Modifications to data are always communicated to the master copy. The main disadvantages of this approach are that it requires fine-grained communication that is not phase-structured. It requires every reference to a quadtree datum to check whether the datum is local, nonlocal but cached locally, or remote.

2.6.4 Summary

The complications to hierarchical radiosity arise from the dynamically changing quadtrees of patches, since they are built as computation proceeds. These data structures are not read-only but are actively read and written by different processors in the same computational phase during the calculation of the form factors. The following issues have to be addressed for any viable implementation

1. Naming of patches on different processors in a globally consistent manner to ease access to data during form factoe calculations.

2. Local quadtree versus Global quadtree approach needs to be investigated for its communication overhead. A level-by-level approach is proposed in this paper. Polygons are distributed among the processors. Each processor processes upto k levels, for a constant k, at which time the load at each processor is checked and if it falls below a threshold, patch-patch interactions are transferred to it from processors having more load.

3. Task stealing has been advocated for load balancing which has not been implemented efficiently yet on a distributed memory machine. Coherency is complicated by movement of patches between processors. We will investigate issues in handling messages for data, control, coherence, synchronization and load balancing while performing computation.

2.7 Image Compression

Applications using wavelet theory can use quadtree based decomposition for thier solution. Binary data compression and image compression are areas where *wavelet theory* has been widely applied. An approximation to the original image can be formed by using a *wavelet transform* A discrete sequence $x(n)$ of length N where $n, N \in Z$ is used to derive two subsampled signals $y_h(n)$ and $y_g(n)$ corresponding to the low and high pass versions of the original sequence $x(n)$ respectively. Each of the signals is of length $N/2$. The signal $y_h(n)$ is obtained by convolving the sequence $x(n)$ with a low pass filter $h(n)$ and dropping every other sample. Similarly $y_g(n)$ is obtained by using a high pass filter $g(n)$. The process of decomposing the sequence $x(n)$ into two subsequences at half resolution can be iterated on either or both sequences. To achieve better frequency resolution at lower frequencies, the scheme is commonly iterated on the lower band.

In the first stage of wavelet decomposition this scheme is applied to 2-D images by applying the above scheme along the rows and then along the columns. The second stage applies the same procedure for the low-pass band. To obtain further stages of wavelet decomposition, the procedure can be applied to the low-pass band of the previous stage. This generates a pyramidal representation of the input image. Figure 2.16 shows a subband decomposition scheme that can be represented as a tree for computation.

Each wavelet picks up information about the image essentially at a given location and at a given scale. For portions of an image that has more interesting features, more coefficients can be used and where the image is nice and smooth a fewer coefficients can result in a good quality approximation. Hence there is an adaptive nature to the subdivision of the image form which makes image compression fall in the category of applications discussed so far. The details of this method can be found in [JS94]. The Discrete Wavelet Transform (DWT) was developed in filter bank representations for subband coding for images and speech signals. The work on Quadrature Mirror Filters (QMF) has beeen used to decompose and reconstruct a signal.

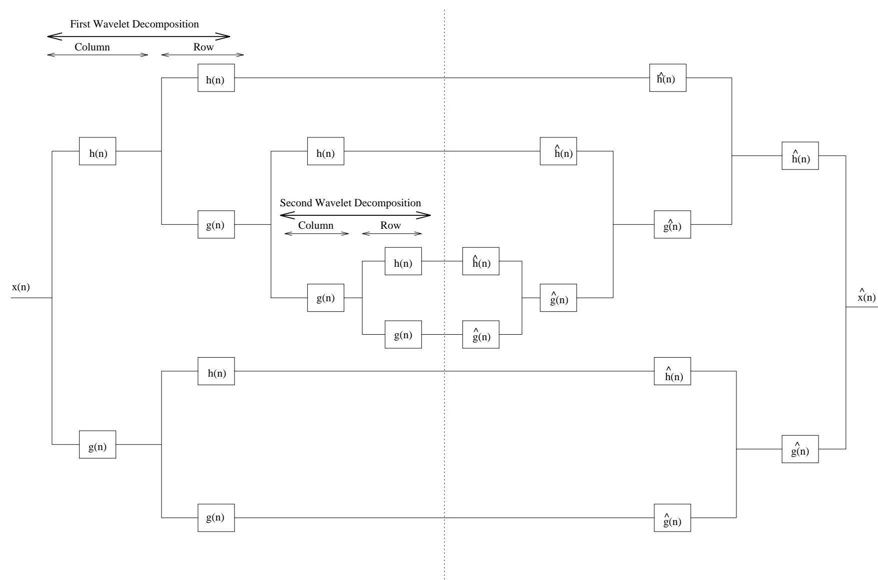


Figure 2.16: A subband decomposition scheme to be used with the Discrete Wavelet Transform.

Chapter 3

Parallel Selection Algorithms

Given a set of N elements, a total order defined on the elements, and a number k, the selection problem is to find the kth smallest element in the given set of elements. The problem has several applications in computer science and statistics. A special case of the problem, often found useful, is to find the median of the given data. The median of N elements is defined to be the element with rank $\lceil \frac{N}{2} \rceil$.

Parallel selection algorithms are useful in such practical applications as dynamic distribution of multidimensional data sets, parallel graph partitioning and parallel construction of multidimensional binary search trees. Many parallel algorithms for selection have been designed for the PRAM model and for various network models including trees, meshes, hypercubes and reconfigurable architectures [BFMP93]. More recently, Bader et.al. [BJ95] implement a parallel deterministic selection algorithm on several distributed memory machines. In this chapter, we consider and evaluate parallel selection algorithms for coarse-grained distributed memory parallel computers.

3.1 Parallel Algorithms for Selection

Parallel algorithms for selection are iterative and work by reducing the number of elements to be considered from iteration to iteration. The elements are distributed across processors and each iteration is performed in parallel by all the processors. Let n be the number of elements and p be the number of processors. To begin with, each processor is given $\lceil \frac{n}{p} \rceil$ or $\lfloor \frac{n}{p} \rfloor$ elements. Let $n_i^{(j)}$ be the number of elements in processor P_i at the beginning of iteration j. let $n^{(j)} = \sum_{i=0}^{p-1} n_i^{(j)}$. Let $k^{(j)}$ be the rank of the element we need to identify among these $n^{(j)}$ elements.

3.1.1 Median of Medians Algorithm

The median of medians algorithm is a straightforward parallelization of the deterministic sequential algorithm [BFP+72] and has recently been suggested and implemented by Bader et. al. [BJ95] (Figure 3.1). This algorithm requires load balancing at the beginning of each iteration.

Algorithm 1 *Median of Medians selection algorithm*

n - Total number of elements

p - Total number of processors labeled from 0 to $p-1$

L_i - List of elements on processor P_i, where $|L_i| = \frac{n}{p}$

$rank$ - desired rank among the total elements

$l = 0$; $r = \frac{n}{p} - 1$

On each processor P_i

while $n > p^2$

Step 1. Use **sequential selection** to find median m_i of list $L_i[l, r]$

Step 2. $M =$ **Gather**(m_i)

Step 3. On P_0

Find median of M, say MoM, and broadcast it to all processors.

Step 4. Partition L_i into $\leq MoM$ and $> MoM$ to give $index_i$, the split index

Step 5. $count =$ **Combine**$(index_i$, add) calculates the number of elements $< MoM$

Step 6. If $(rank \leq count)$

$n = count$; $r = index_i$; $rank = count$

else

$n = n - count$; $l = index_i$; $rank = rank - count$

Step 7. **LoadBalance**(L_i, n, p)

Step 8. $L =$ **Gather**$(L_i[l, r])$

Step 9. On P_0

Perform sequential selection to find element q of $rank$ in L

$result =$ **Broadcast**(q)

Figure 3.1: Median on Medians selection Algorithm

At the beginning of iteration j, each processor finds the median of its $n_i^{(j)} = \lceil \frac{n^{(j)}}{p} \rceil$ or $\lfloor \frac{n^{(j)}}{p} \rfloor$ elements using the sequential deterministic algorithm. All such medians are gathered on one processor, which then finds the median of these medians. The median of medians is then estimated to be the median of all the $n^{(j)}$ elements. The estimated median is broadcast to all the processors. Each processor scans through its set of points and splits them into two subsets containing elements less than or equal to and greater than the estimated median, respectively. A Combine operation and a comparison with $k^{(j)}$ determines which of these two subsets to be discarded and the value of $k^{(j+1)}$ needed for the next iteration.

Selecting the median of medians as the estimated median ensures that the estimated median will have at least a guaranteed fraction of the number of elements below it and above it. This ensures that the worst case number of iterations required is $O(\log n)$. Let $n_{max}^{(j)} = max_{i=0}^{p-1} n_i^{(j)}$. Thus, finding the local median and splitting the set of points into two subsets based on the estimated median each requires $O(n_{max}^{(j)})$ time in the jth iteration. The remaining work is one Gather, one Broadcast and one Combine operation. Therefore, the worst-case running time of this algorithm is $\sum_{j=0}^{\log n - 1} O(n_{max}^{(j)} + \tau \log p + \mu p)$. Since $n_{max}^{(j)} = O(\frac{n}{p})$, the running time is $O(\frac{n}{p} \log n + \tau \log p \log n + \mu p \log n)$. This is the running time of the Median of Medians algorithm assuming load balancing but ignoring the cost of it.

3.1.2 Bucket-Based Algorithm

The bucket-based algorithm [RCY94] attempts to reduce the worst-case running time of the above algorithm without requiring load balance. The algorithm is shown in Figure 3.2. As before, local medians are computed on each processor. However, the estimated median is taken to be the weighted median of the local medians, with each median weighted by the number of elements on the corresponding processor. This will again guarantee that a fixed fraction of the elements is dropped from consideration every iteration. The number of iterations of the algorithm remains $O(\log n)$.

The dominant computational work in the median of medians algorithm is the computation of the local median and scanning through the local elements to split them into two sets based on the estimated median. In order to reduce this work which is repeated every iteration, the bucket-based approach preprocesses the local data into $\log p$ buckets such that for any $0 \leq i < j < \log p$, every element in bucket i is smaller than any element in bucket j. This requires $O(\frac{n}{p} \log \log p)$ time. The cost of finding the local median reduces from $O(\frac{n}{p})$ to $O(\log \log p + \frac{n}{p \log p})$. To split the local data into two sets based on the estimated median, first identify the bucket that should contain the estimated median. Only the elements in this bucket need to be split. Thus, this operation also requires only $O(\log \log p + \frac{n}{p \log p})$ time.

After preprocessing, the worst-case run time for selection is $O(\log \log p \log n + \frac{n}{p \log p} \log n + \tau \log p \log n + \mu p \log n) = O(\frac{n}{p \log p} \log n + \tau \log p \log n + \mu p \log n)$ for $n > p^2 \log \log p$. Therefore, the worst-case run time of the bucket-based approach is $O(\frac{n}{p}(\log \log p + \frac{\log n}{\log p}) + \tau \log p \log n + \mu p \log n)$ without any load balancing.

Algorithm 2 *Bucket-based selection algorithm*

n - Total number of elements

p - Total number of processors labeled from 0 to $p-1$

L_i - List of elements on processor P_i, where $|L_i| = \frac{n}{p}$

C - is a constant

$rank$ - desired rank among the total elements

$l = 0$; $r = \frac{n}{p} - 1$

On each processor P_i

Step 0. Partition L_i on P_i into $\log p$ buckets of equal size such that if $r \in bucket_j$, and $s \in bucket_k$, then $r < s$ if $j < k$

while$(n > p^2)$

> **Step** 1. Find the bucket bkt_k containing the median element using a binary search on the remaining buckets. This is followed by finding the appropriate rank in bkt_k to find the median m_i. Let N_i be the number of remaining keys on P_i.
>
> **Step** 2. $M = \textbf{Gather}(m_i)$; $Q = \textbf{Gather}(N_i)$
>
> **Step** 3. On P_0
>
> Find the weighted median of M, say WM and broadcast it.
>
> **Step** 4. Partition L_i into $\leq WM$ and $> WM$ using the *buckets* to give $index_i$, the split index
>
> **Step** 5. $count = \textbf{Combine}(index_i, \text{add})$ calculates the number of elements less than WM
>
> **Step** 6. If $(rank \leq count)$
>
> $n = count$; $r = index_i$; $rank = count$
>
> else
>
> $n = n - count$; $l = index_i$; $rank = rank - count$

Step 7. $L = \textbf{Gather}(L_i)$

Step 8. On P_0

> Perform sequential selection to find element q of $rank$ in L
>
> $result = \textbf{Broadcast}(q)$

Figure 3.2: Bucket-based selection algorithm

3.1.3 Randomized Selection Algorithm

Algorithm 3 *Randomized selection algorithm*

n - Total number of elements
p - Total number of processors labeled from 0 to $p-1$
L_i - List of elements on processor P_i, where $|L_i| = \frac{n}{p}$
$rank$ - desired rank among the total elements
$l = 0$; $r = \frac{n}{p} - 1$
On each processor P_i
$while(n > p^2)$

Step 0. $n_i = r - l + 1$
Step 1. $s = \mathbf{PrefixSum}(n_i, p)$
Step 2. Generate a random number n_r (same on all processors) between 0 and $n-1$
Step 3. On P_k where $(n_r \in [s - n_i, s])$

$m_{guess} = \mathbf{Broadcast}(L_i[n_r - (s - n_i)])$

Step 4. Partition L_i into $\leq m_{guess}$ and $> m_{guess}$ to give $index_i$, the split index
Step 5. $count = \mathbf{Combine}(index_i,$ add) calculates the number of elements less than m_{guess}
Step 6. If $(rank \leq count)$

$n = count$; $r = index_i$; $rank = count$
else
$n = n - count$; $l = index_i$; $rank = rank - count$

Step 7. $L = \mathbf{Gather}(L_i[l, r])$
Step 8. On P_0

Perform sequential selection to find element q of $rank$ in L
$result = \mathbf{Broadcast}(q)$

Figure 3.3: Randomized selection algorithm

Sequentially, the randomized selection algorithm works as follows. A random element is selected to be the estimated median. The set is split into two subsets S_1 and S_2 of elements smaller than or equal to and greater than the estimated median. If $|S_1| >= k$, recursively find the element with rank k in S_1. If not, recursively find the element with rank $(k - |S_1|)$ in S_2. This can be parallelized easily (Figure 3.3). All processors use the same random number generator with the same seed so that they can produce identical random numbers. Consider the behavior of the algorithm in iteration j. First, a parallel prefix operation is performed on the $n_i^{(j)}$'s. All processors generate a random number between 1 and $n^{(j)}$ to pick an element at random, which is taken to be the estimate median. From the parallel prefix operation, each processor can determine if it has the estimated median and if so broadcasts it. Each processor scans through its set of points and splits them into two subsets containing elements less than or equal to and greater than the estimated median,

respectively. A Combine operation and a comparison with $k^{(j)}$ determines which of these two subsets to be discarded and the value of $k^{(j+1)}$ needed for the next iteration.

It can be shown that the number of iterations is $O(\log n)$ with high probability. Let $n_{max}^{(j)} = max_{i=0}^{p-1} n_i^{(j)}$. Thus, splitting the set of points into two subsets based on the median requires $O(n_{max}^{(j)})$ time in the jth iteration. The remaining work is one Parallel Prefix, one Broadcast and one Combine operation. Therefore, the total expected running time of the algorithm is $\sum_{j=0}^{\log n - 1} O(n_{max}^{(j)} + (\tau + \mu) \log p)$.

With load balancing and ignoring the cost of it, the running time of the algorithm reduces to $O(\frac{n}{p} + (\tau + \mu) \log p \log n)$. Even without this load balancing, assuming that the initial data is randomly distributed, the running time is expected to be $O(\frac{n}{p} + (\tau + \mu) \log p \log n)$.

3.1.4 Fast Randomized Selection Algorithm

The expected number of iterations required for the randomized median finding algorithm is $O(\log n)$. In this section we discuss an approach due to Rajasekharan et. al. [RCY94] that requires only $O(\log \log n)$ iterations for convergence with high probability (Figure 3.4).

Suppose we want to find the kth smallest element among a given set of n elements. Sample a set S of $o(n)$ keys at random and sort S. The element with rank $m = \lceil \frac{k|S|}{n} \rceil$ in S will have an expected rank of k in the set of all points. Identify two keys l_1 and l_2 in S with ranks $m - \delta$ and $m + \delta$ where δ is a small integer such that with high probability the rank of l_1 is $< k$ and the rank of l_2 is $> k$ in the given set of points. With this, all the elements that are either $< l_1$ or $> l_2$ can be eliminated. Recursively find the element with rank $k - rank(l_1)$ in the remaining $(rank(l_2) - rank(l_1) - 1)$ elements. If the number of elements is sufficiently small, they can be directly sorted to find the required element.

In iteration j, Processor $P_i^{(j)}$ randomly selects $n_i^{(j)} \frac{n^\epsilon}{n^{(j)}}$ of its $n_i^{(j)}$ elements. The selected elements are sorted using a parallel sorting algorithm. Once sorted, the processors containing the elements $l_1^{(j)}$ and $l_2^{(j)}$ broadcast them. Each processor finds the number of elements less than $l_1^{(j)}$ and greater than $l_2^{(j)}$ contained by it. Using Combine operations, the ranks of $l_1^{(j)}$ and $l_2^{(j)}$ are computed and the appropriate action of discarding elements is undertaken by each processor. A large value of ϵ increases the overhead due to sorting. A small value of ϵ increases the probability that both the selected elements ($l_1^{(j)}$ and $l_2^{(j)}$) lie on one side of the element with rank $k^{(j)}$, thus causing an unsuccessful iteration. By experimentation, we found a value of 0.6 to be appropriate.

Rajasekharan et. al. show that the expected number of iterations of this median finding algorithm is $O(\log \log n)$ and that the expected number of points decreases geometrically after each iteration. If $n^{(j)}$ is the number of points at the start of the jth iteration, only a sample of $o(n^{(j)})$ keys is sorted. Thus, the cost of sorting, $o(n^{(j)} \log n^{(j)})$ is dominated by the $O(n^{(j)})$ work involved in scanning the points.

As in the randomized median finding algorithm, one iteration of the median finding algorithm takes $O(n_{max}^{(j)} + (\tau + \mu) \log p)$ time. Assuming load balancing and ignoring the cost of load balancing, the running time of median finding is $O(\frac{n}{p} + (\tau + \mu) \log p \log \log n)$. Even without this load balancing, the running time is expected to be $O(\frac{n}{p} + (\tau + \mu) \log p \log \log n)$ for random distribution of data.

Algorithm 4 *Fast randomized selection algorithm*

n - Total number of elements

p - Total number of processors labeled from 0 to $p-1$

L_i - List of elements on processor P_i, where $|L_i| = \frac{n}{p}$

$rank$ - desired rank among the total elements

C - a constant

$l = 0$; $r = \frac{n}{p} - 1$

On each processor P_i

$while(n > p^2)$

Step 0. $n_i = r - l + 1$

Step 1. Collect a sample S_i from $L_i[l, r]$ by picking $n_i \frac{n^\epsilon}{n}$ elements at random on P_i between l and r.

Step 2. $S =$ **ParallelSort**(S_i, p)

On P_0

Step 3. Pick k_1, k_2 from S with ranks $\lceil \frac{i|S|}{n} - \sqrt{|S|logn} \rceil$ and $\lceil \frac{i|S|}{n} + \sqrt{|S|logn} \rceil$

Step 4. Broadcast k_1 and k_2.The $rank$ to be found will be in $[k_1, k_2]$ with high probability.

Step 5. Partition L_i between l and r into $< k_1$, $[k_1, k_2]$ and $> k_2$ to give counts $less$, $middle$ and $high$ and splitters s_1 and s_2.

Step 6. $c_{mid} =$ **Combine**$(middle$, add)

Step 7. $c_{less} =$ **Combine**$(less$, add)

Step 8. If $(rank \in (c_{less}, c_{mid}])$

$n = c_{mid}$; $l = s_1$; $r = s_2$; $rank = rank - c_{less}$

else

if($rank < c_{less}$)

$r = s_1$; $n = c_{less}$

else

$n = n - (c_{less} + c_{mid})$; $l = s_2$; $rank = rank - (c_{less} + c_{mid})$

Step 9. $L =$ **Gather**$(L_i[l.r])$

Step 10. On P_0

Perform sequential selection to find element q of $rank$ in L

$result =$ **Broadcast**(q)

Figure 3.4: Fast Randomized selection Algorithm

3.2 Algorithms for load balancing

In order to ensure that the computational load on each processor is approximately the same during every iteration of a selection algorithm, we need to dynamically redistribute the data such that every processor has nearly equal number of elements. We have developed and compared several algorithms for load balancing for the median finding problems. In the following, we describe only one of them for brevity. The rest of them are presented in Chapter 4.

Initially, processor P_i has two integers s_i and r_i and s_i elements of data such that $\sum_{i=0}^{p-1} s_i = \sum_{i=0}^{p-1} r_i$. Let $s_{max} = max_{i=0}^{p-1} s_i$ and $r_{max} = max_{i=0}^{p-1} r_i$. The objective is to redistribute the data such that processor P_i contains r_i elements. Every processor retains $min\{s_i, r_i\}$ of its original elements. If $s_i > r_i$, the processor has $(s_i - r_i)$ elements in excess and is labeled a source. Otherwise, the processor needs $(r_i - s_i)$ elements and is labeled a sink. The excessive elements in the source processors and the number of elements needed by the sink processors are transferred using the transportation primitive. The maximum number of messages sent out by a processor is $O(p)$ and the maximum number of elements sent is $(n_{max} - n_{avg})$, where $n_{avg} = \lfloor \frac{n}{p} \rfloor$.

The maximum number of elements received by a processor is n_{avg}. The worst-case running time is $O(\tau p + \mu(n_{max} - n_{avg}))$.

We call this algorithm non-order maintaining load balance algorithm as it does not maintain the ordering of the data. We also considered two other load balancing algorithms: the dimension exchange method and the global exchange method [AfAGR96b].

3.3 Implementation Results

We have implemented all the selection algorithms and the load balancing techniques on the CM-5. To experimentally evaluate the algorithms, we have chosen the problem of finding the median of a given set of numbers. We ran each selection algorithm with and without any load balancing (except for the bucket-based approach which does not use load balancing). In the following we briefly summarize a subset of the results. For details, the reader is referred to [AfAGR96b].

The algorithms are run until the total number of elements falls below p^2, at which point the elements are gathered on one processor and the problem is solved by sequential selection. This was used to provide a consistent comparison between the different schemes. For each value of the total number of elements, we have run each of the algorithms on two types of inputs - random and sorted. We ran each experiment on five different random sets of data and used the average running time. Random data sets constitute close to the best case input for the selection algorithms. The sorted input is a close to the worst-case input for the selection algorithms.

The execution times of the four different selection algorithms without using load balancing for random data (except for median of medians algorithm requiring load balancing) with 2M numbers is shown in Figure 3.5. An immediate observation is that the randomized algorithms are superior to the deterministic algorithms by an order of magnitude. The factor of improvement in randomized parallel selection algorithms over deterministic parallel

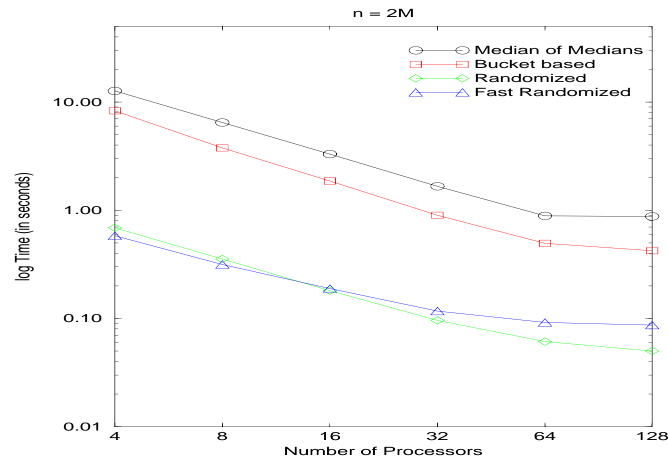


Figure 3.5: Performance of selection algorithms without load balancing (except for the median of medians algorithm for which load balancing is used) on random data sets.

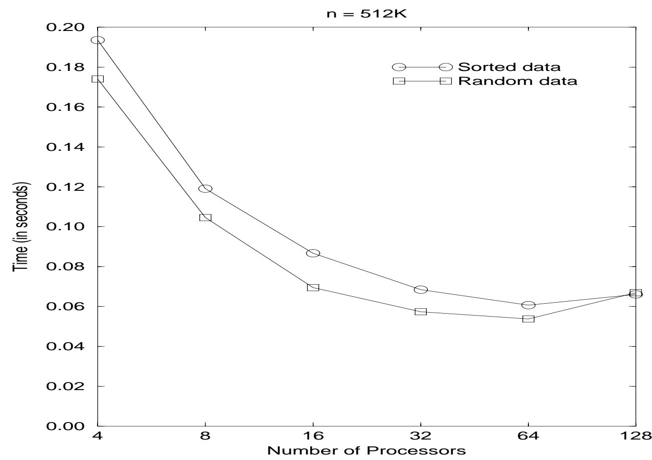


Figure 3.6: Performance of fast randomized selection algorithm using random and sorted data sets.

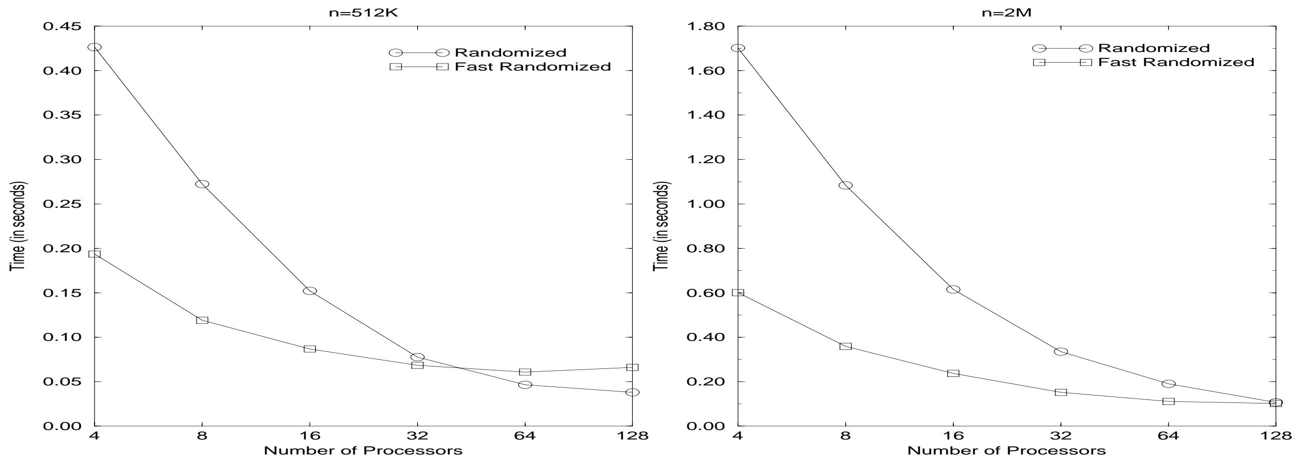


Figure 3.7: Performance of the two randomized selection algorithms on sorted data sets using the best load balancing strategies for each algorithm

selection is due to improvements in both the sequential and parallel parts. Among the deterministic algorithms, the bucket-based approach consistently performed better than the median of medians approach by about a factor of two for random data. For sorted data, the bucket-based approach which does not use any load balancing ran only about 25% slower than median of medians approach with load balancing [AfAGR96b].

The effect of the various load balancing techniques on the fast randomized algorithm is given in Figure 3.6. The execution times are consistently better for sorted data without using any load balancing. For the randomized selection the cost of load balancing offset the improvements resulting in approximately the same overall cost. Load balancing for random data almost always had a negative effect on the total execution time for both the randomized methods [AfAGR96b]. Consider the variance in the running times between random and sorted data for both the randomized algorithms. The randomized selection algorithm ran 2 to 2.5 times faster for random data than for sorted data [AfAGR96b]. The fast randomized selection with load balancing performs equally well on both best and worst-case data.

In Figure 3.7, we see a comparison of the two randomized algorithms for sorted data with the best load balancing strategies for each algorithm – no load balancing for randomized selection and non-order maintaining load balancing for fast randomized algorithm (which performed slightly better than other strategies). We see that, for large n, fast randomized selection is superior. For small data sets the cost of sorting the sample offsets the benefits of reduction in number of iterations.

3.4 Conclusions

We conclude that randomized algorithms are faster by an order of magnitude. If determinism is desired, the bucket-based approach is superior to the median of medians algorithm. Of the two randomized algorithms, fast randomized selection with load balancing delivers good performance for all types of input distributions with very little variation in the running time. The overhead of using load balancing with well-behaved data is insignificant.

Any of the load balancing techniques described can be used without significant variation in the running time. Randomized selection performs well for well-behaved data. There is a large variation in the running time between best and worst-case data. Load balancing does not improve the performance of randomized selection irrespective of the input data distribution.

Chapter 4

Load Balancing Algorithms

In order to ensure that the computational load on each processor is approximately the same during every iteration of a selection algorithm, we need to dynamically redistribute the data such that every processor has nearly equal number of elements. We present three algorithms for performing such a load balancing. The algorithms can also be used in other problems that require dynamic redistribution of data and where there is no restriction on the assignment of data to processors.

We use the following notation to describe the algorithms for load balancing: Initially, processor P_i contains n_i elements. n is the total number of elements on all the processors, i.e. $n = \sum_{i=0}^{p-1} n_i$. Let $n_{max} = max_{i=0}^{p-1} n_i$. Let $n_{avg} = \lfloor \frac{n}{p} \rfloor$.

4.1 Order Maintaining Load Balance

Suppose that each processor has its set of elements stored in an array. We can view the n elements as if they were globally sorted based on processor and array indices. For any $i < j$, any element in processor P_i appears earlier in this sorted order than any element in processor P_j. The order maintaining load balance algorithm is a parallel prefix based algorithm that preserves this global order of data after load balancing.

The algorithm first performs a Parallel Prefix operation to find the position of the elements it contains in the global order. The objective is to redistribute data such that processor P_i contains the elements with positions $n_{avg} i \ldots n_{avg}(i+1) - 1$ in the global order. Using the parallel prefix operation, each processor can figure out the processors to which it should send data and the amount of data to send to each processor. Similarly, each processor can figure out the amount of data it should receive, if any, from each processor. Communication is generated according to this and the data is redistributed.

In our model of computation, the running time of this algorithm only depends on the maximum communication generated/received by a processor. The maximum number of messages sent out by a processor is $\lceil \frac{n_{max}}{n_{avg}} \rceil + 1$ and the maximum number of elements sent is n_{max}. The maximum number of elements received by a processor is n_{avg}. Therefore, the running time is $O(\tau \frac{n_{max}}{n_{avg}} + \mu n_{max})$ [RSA95].

The order maintaining load balance algorithm may generate much more communication than necessary. For example, consider the case where all processors have n_{avg} elements

Algorithm 5 *Modified order maintaining load balance*

n - Number of total elements

p - Total number of processors labeled from 0 to $p-1$

L_i - List of elements on processor P_i of size n_i

On each processor P_i

Step 0. $n_{avg} = \lceil \frac{n}{p} \rceil$; if $p < n \bmod p$, increment n_{avg}

Step 1. M = **Global_Concat**(n_i)

for $j \leftarrow 0$ to $p-1$

Step 2. $diff[j] = M[j]$ - n_{avg}

Step 3. If $diff[j]$ is positive P_j is labeled as a source. If $diff[j]$ is negative P_j is labeled as a sink.

Step 4. If P_i is a source calculate the prefix sum of the positive $diff[*]$ in an array p_src, else calculate the prefix sums for sinks using negative $diff[*]$ in p_snk.

if(source[P_i])

Step 5. $l_i = |p_src[i]| - diff[i]$

Step 6. $r_i = |p_src[i]| - 1$

Step 7. Calculate the range of destination processors $[P_l, P_r]$ using a binary search on p_snk.

Step 8. while($l \leq r$)

Send $[min(r_i, p_snk[P_l]) - l_i]$ elements to P_l and increment l

if(sink[P_i])

Step 5. $l_i = p_snk[i] - diff[i]$

Step 6. $r_i = p_snk[i] - 1$

Step 7. Calculate the range of source processors $[P_l, P_r]$ using a binary search on p_src.

Step 8. while($l \leq r$)

Receive $[min(r_i, p_src[P_l]) - l_i]$ elements from P_l and increment l

Figure 4.1: Modified order maintaining load balance

except that P_0 has one element less and P_{p-1} has one element more than n_{avg}. The optimal strategy is to transfer the one extra element from P_p to P_0. However, this algorithm transfers one element from P_i to P_{i-1} for every $1 \leq i < p-1$, generating $(p-1)$ messages.

Since preserving the order of data is not important for the selection algorithm, the following modification is done to the algorithm: Every processor retains $min\{n_i, n_{avg}\}$ of its original elements. If $n_i > n_{avg}$, the processor has $(n_i - n_{avg})$ elements in excess and is labeled a source. Otherwise, the processor needs $(n_{avg} - n_i)$ elements and is labeled a sink. The excessive elements in the source processors and the number of elements needed by the sink processors are ranked separately using two Parallel Prefix operations. The data is transferred from sources to sinks using a strategy similar to the order maintaining load balance algorithm. This algorithm (Figure 4.1, which we call modified order maintaining load balance algorithm (modified OMLB), is implemented in [BJ95].

The maximum number of messages sent out by a processor in modified OMLB is $O(p)$ and the maximum number of elements sent is $(n_{max} - n_{avg})$. The maximum number of elements received by a processor is n_{avg}. The worst-case running time is $O(\tau p + \mu(n_{max} - n_{avg}))$.

4.2 Dimension Exchange Method

Algorithm 6 *Dimension exchange method*

n - Number of total elements

p - Total number of processors labeled from 0 to $p-1$

L_i - List of elements on processor P_i of size n_i

On each processor P_i

for $j \leftarrow 0$ *to* $\log p - 1$

Step 1. $P_l = P_i$ XOR 2^j

Step 2. Exchange the count of elements between $P_i(n_i)$ and $P_l(n_l)$

Step 3. $n_{avg} = \lceil \frac{n_i + n_l}{2} \rceil$

if $(n_i > n_{avg})$

Step 4. **Send** $n_i - n_{avg}$ elements from $L_i[n_{avg}]$ to processor P_l

Step 5. $n_i = n_{avg}$

else

if ($n_l > n_{avg}$)

Step 4. **Receive** $n_l - n_{avg}$ elements from processor P_l at $L_i[n_i]$

Step 5. Increment n_i by $n_l - n_{avg}$

Figure 4.2: Dimension exchange method for load balancing

The dimension exchange method (Figure 4.2) is a load balancing technique originally proposed for hypercubes [Cyb89][RWS88]. In each iteration of this method, processors are

paired to balance the load locally among themselves which eventually leads to global load balance. The algorithm runs in $\log p$ iterations. In iteration i ($0 \leq i < \log p$), processors that differ in the ith least significant bit position of their id's exchange and balance the load. After iteration i, for any $0 \leq j < \lfloor \frac{p}{2^i} \rfloor$, processors $P_{j2^i} \ldots P_{j2^{i+1}-1}$ have the same number of elements.

In each iteration, $\frac{p}{2}$ pairs of processors communicate in parallel. No processor communicates more than $\frac{n_{max}}{2}$ elements in an iteration. Therefore, the running time is $O(\tau \log p + \mu n_{max} \log p)$. However, since 2^j processors hold the maximum number of elements in iteration j, it is likely that either n_{max} is small or far fewer elements than $\frac{n_{max}}{2}$ are communicated. Therefore, the running time in practice is expected to be much better than what is predicated by the worst-case.

4.3 Global Exchange

This algorithm is similar to the modified order maintaining load balance algorithm except that processors with large amounts of data are directly paired with processor with small amounts of data to minimize the number of messages (Figure 4.3).

As in the modified order maintaining load balance algorithm, every processor retains $min\{n_i, n_{avg}\}$ of its original elements. If $n_i > n_{avg}$, the processor has $(n_i - n_{avg})$ elements in excess and is labeled a source. Otherwise, the processor needs $(n_{avg} - n_i)$ elements and is labeled a sink. All the source processors are sorted in non-increasing order of the number of excess elements each processor holds. Similarly, all the sink processors are sorted in non-increasing order of the number of elements each processor needs. The information on the number of excessive elements in each source processor is collected using a Global Concatenate operation. Each processor locally ranks the excessive elements using a prefix operation according to the order of the processors obtained by the sorting. Another Global Concatenate operation collects the number of elements needed by each sink processor. These elements are then ranked locally by each processor using a prefix operation performed using the ordering of the sink processors obtained by sorting.

Using the results of the prefix operation, each source processor can find the sink processors to which its excessive elements should be sent and the number of element that should be sent to each such processor. The sink processors can similarly compute information on the number of elements to be received from each source processor. The data is transferred from sources to sinks. Since the sources containing large number of excessive elements send data to sinks containing large number of excessive elements, this may reduce the total number of messages sent.

In the worst-case, there may be only one processor containing all the excessive elements and thus the total number of messages sent out by the algorithm is $O(p)$. No processor will send more than $(n_{max} - n_{avg})$ elements of data and the maximum number of elements received by any processor is n_{avg}. The worst-case run time is $O(\tau p + \mu(n_{max} - n_{avg}))$.

Algorithm 7 *Global Exchange load balance*

n - Number of total elements

p - Total number of processors labeled from 0 to $p-1$

L_i - List of elements on processor P_i of size n_i

On each processor P_i

Step 0. $n_{avg} = \lceil \frac{n}{p} \rceil$; if $p < n \bmod p$, increment n_{avg}

Step 1. M = **Global_Concat**(n_i)

for $j \leftarrow 0$ *to* $p-1$

- **Step** 2. $diff[j] = M[j]$ - n_{avg}
- **Step** 3. If $diff[j]$ is positive P_j is labeled as a source. If $diff[j]$ is negative P_j is labeled as a sink.

Step 4. For $k \in [0, p-1]$ sort $diff[k]$ for sources in descending order maintaining appropriate processor indices. Also sort $diff[k]$ for sinks in ascending order.

Step 5. If P_i is a source calculate the prefix sum of the positive $diff[*]$ in an array p_src, else calculate the prefix sums for sinks using negative $diff[*]$ in p_snk.

Step 6. If P_i is a source calculate the prefix sum of the positive $diff[*]$ in an array p_src, else calculate the prefix sums for sinks using negative $diff[*]$ in p_snk.

if(source[P_i])

- **Step** 7. $l_i = |p_src[i]| - diff[i]$
- **Step** 8. $r_i = |p_src[i]| - 1$
- **Step** 9. Calculate the range of destination processors $[P_l, P_r]$ using a binary search on p_snk.
- **Step** 10. while($l \leq r$)
  - **Send** $[min(r_i, p_snk[P_l]) - l_i]$ elements to P_l and increment l

if(sink[P_i])

- **Step** 7. $l_i = p_snk[i] - diff[i]$
- **Step** 8. $r_i = p_snk[i] - 1$
- **Step** 9. Calculate the range of source processors $[P_l, P_r]$ using a binary search on p_src.
- **Step** 10. while($l \leq r$)
  - **Receive** $[min(r_i, p_src[P_l]) - l_i]$ elements from P_l and increment l

---

Figure 4.3: Global exchange method for load balancing

Chapter 5

Constructing Multidimensional Binary Search Trees

Consider a set of n points in k dimensional space. Let $d_1, d_2, \ldots, d_k$ denote the k dimensions. A k-d tree [Ben75] on the points is constructed as follows: The root of the tree corresponds to the set of all points. Choose a dimension d_l, and find the median coordinate of all the points along dimension d_l. We can partition the points into two approximately equal sized sets - one set containing all the points whose coordinates along dimension d_l are less than or equal to this median and a second set containing all the remaining points. The two subpartitions are represented by the children of the root node. The tree is built recursively until each leaf corresponds to one point. In homogeneous trees, internal nodes are used to store the median points. Non-homogeneous trees store points only at the leaves.

Several applications require partial construction of k-d trees. In parallel graph partitioning, we are interested in creating p partitions to distribute the graph to p processors, requiring the construction of only the first $\log p$ levels of the k-d tree. In hierarchical applications like the n-body simulation, clustering of physically proximate objects is essential and the k-d tree offers such a clustering scheme. In databases, records can be treated as points in an appropriate space by mapping each key to a coordinate and the resulting point set can be organized using a k-d tree. In constructing the tree, a node is partitioned only if all its records do not fit in one disk sector.

In this chapter, we focus on efficient parallel construction of balanced, non-homogeneous k-d trees on coarse-grained distributed memory parallel computers. For all nodes at level i of the tree (defining the root to be at level 0), we use dimension $d_{(i \ mod \ k)+1}$. Other variations can be easily implemented with minor changes in our algorithms without significantly affecting their running time.

5.1 Local Tree Construction

5.1.1 Sort-based Method

In this method, we maintain k arrays $A_1, A_2, \ldots, A_k$. Initially, A_l contains all the points sorted according to dimension d_l. Any node in the k-d tree corresponding to a partition that is yet to be split has two pointers i and j ($i < j$) associated with it such that the subarray

$A_l[i..j]$ contains the points of the partition sorted along d_l. If the partition is split based on d_1, splitting the array $A_1[i..j]$ can be done in constant time as it requires just computing the pointers of the sub-partitions. Consider splitting $A_l[i..j]$ for any other dimension d_l. If we simply scan $A_l[i..j]$ from either end and swap points when necessary (as can be done in a median finding method), the sorted order will be destroyed. Therefore, it is required to go over the points twice: Once to count the number of points in the two subpartitions and a second time to actually move the data. Counting is necessary because the number of points less than or equal to the median need not be exactly equal to half the points.

Also, the arrays contain pointers to records and not actual records. With this, copying a point requires only $O(1)$ time. This method requires a preprocessing step of sorting the n points along each of the k dimensions, taking $O(kn \log n)$ time. Constructing each level of the k-d tree involves scanning each of the k arrays and takes $O(kn)$ time. After preprocessing, $\log m$ levels of the tree can be built in $O(kn \log m)$ time. In some cases, such as when the sort-based method is used to construct the first $\log p$ levels of the tree, the data may already be present in sorted order.

It is possible to reduce the $O(kn)$ time per level to $O(n)$ by maintaining a single copy of the data and operating on a list of pointers for each of the k dimensions. However, this method is not expected to perform better for small values of k such as 2 and 3, which cover many practical applications such as graph partitioning and hierarchical methods.

5.1.2 Median-based Method

In this approach, a partition is represented by an unordered set of points and the median is explicitly computed in order to split the partition (see Figure 5.3) We have tested various median finding algorithms and found that the randomized algorithm of Floyd et. al. [FR75] results in the best performance. The algorithm works by picking a random element of the set, partitioning the set based on this element, throwing away the partition not containing the desired element and repeatedly performing the same procedure on the partition containing the desired element. The worst-case run time is $O(n^2)$ but the expected run time is only $O(n)$. The expected number of iterations is $O(\log n)$. A different approach can be used to reduce the expected number of iterations to $O(\log \log n)$ [RCY94]. We found that the parallel versions of these two methods have comparable running times, but Floyd's algorithm fares better sequentially [AfAGR96c].

Note that the very process of computing the median of a partition splits the partition into two subpartitions. In constructing level i of the local tree, we have to compute 2^i medians, each on a partition containing $\frac{n}{2^i}$ points. Since the total number of points in all partitions at any level of the local tree is n, building each level takes $O(n)$ time. The required $\log m$ levels can be built in $O(n \log m)$ time.

5.1.3 Bucket-based Method

The bucket-based method is a hybrid approach combining the median-based and sort-based methods. It starts with inducing partial order on the data which is refined further only as it is needed.

Sample a set of n^ϵ points ($0 < \epsilon < 1$) and sort them according to dimension d_1. This take $O(n)$ time. Using the sorted sample, divide the range containing the points into b ranges,

Method	Tree construction up to $\log m$ levels		
	Preprocessing (X)	Median Finding(c)	Data Movement (s)
Median-based	-	$O(n \log m)$	$O(n \log m)$
Sort-based	$O(kn \log n)$	$O(m)$	$O(kn \log m)$
Bucket-based	$O(kn \log b)$	$O(\frac{n}{b} \frac{m^{1/k}-1}{2^{1/k}-1})$	$O(kn \log m)$

Table 5.1: Computation time for local tree construction.

called buckets. After defining the buckets, find the bucket that should contain each of the n points. This can be accomplished using binary search in $O(\log b)$ time. The n points are now distributed among the b buckets and the expected number of points in a bucket is $O(\frac{n}{b})$ with high probability (assuming b is $o(\frac{n}{\log n})$). The same procedure is repeated for all dimensions, for a total preprocessing cost of $O(kn \log b)$.

At any stage of the algorithm, we have a partition and k arrays storing the points of the partition partially sorted into buckets using their coordinates along each dimension. Without loss of generality, assume that the partition should be split along d_1. The bucket containing the median is easily identified in time logarithmic in the number of buckets. Finding the median translates to finding the element with the appropriate rank in the bucket containing the median. To split the partition, we need to compute the partially sorted arrays corresponding to the subpartitions. This is accomplished along d_1 by merely splitting the bucket containing the median into two buckets. To create the arrays along any other dimension d_l, each bucket in the partially sorted array along d_l is split into two buckets. All the buckets with points having a smaller coordinate along d_1 than the median are grouped into one subpartition and the rest of buckets are grouped into the second subpartition.

When a partition is split into two subpartitions, the number of points in the partition is split into half. The number of buckets remains approximately the same (except that one bucket may be split into two) along the dimension which is used to split the partition. Along all other dimensions, the number of buckets increases by a factor of two. At a stage when level i of the tree is to be built, there are 2^i partitions and $\Theta(b2^{i(k-1)/k})$ buckets. Thus, building level i of the tree requires solving 2^i median finding problems each working on a bucket of expected size $O(\frac{n}{b2^{i(k-1)/k}})$. This time is dominated by the $O(kn)$ time to split the buckets along $k-1$ dimensions. Since the constant associated with median finding is high, this method has the advantage that it performs median finding on smaller sized data. The asymptotic complexity for building $\log m$ levels is $O(kn \log m)$.

5.1.4 Experimental Results

The computation time required for local tree construction, as summarized in Table 5.1, can be decomposed into different parts: the time required for preprocessing (X), the cost of finding the median at every level (c), the data movement time to decompose arrays into subarrays based on the median (s). The sorting and bucket-based methods are required

to have **stable order property**. This means that the relative ordering of data has to be preserved during the data movement. resulting in a higher value for s. Data has to be moved for $k-1$ lists in the sort-based and bucket-based strategies as compared to a single list in the median-based strategy. There is an overhead r attached with maintenance and processing of sublists, which double at every level. This overhead is the smallest for sorting, slightly larger for median-method and highest for bucketing since the cost for bucketing grows exponentially with the increase in number of levels.

An important practical aspect of median finding is that the data movement step and median finding step can be combined resulting in a small overall constant (one of the reasons quicksort has been shown to work well in practice). We limit our experimental results to the two-dimensional case because the results we obtained are such that conclusions about higher dimensional point sets can be drawn.

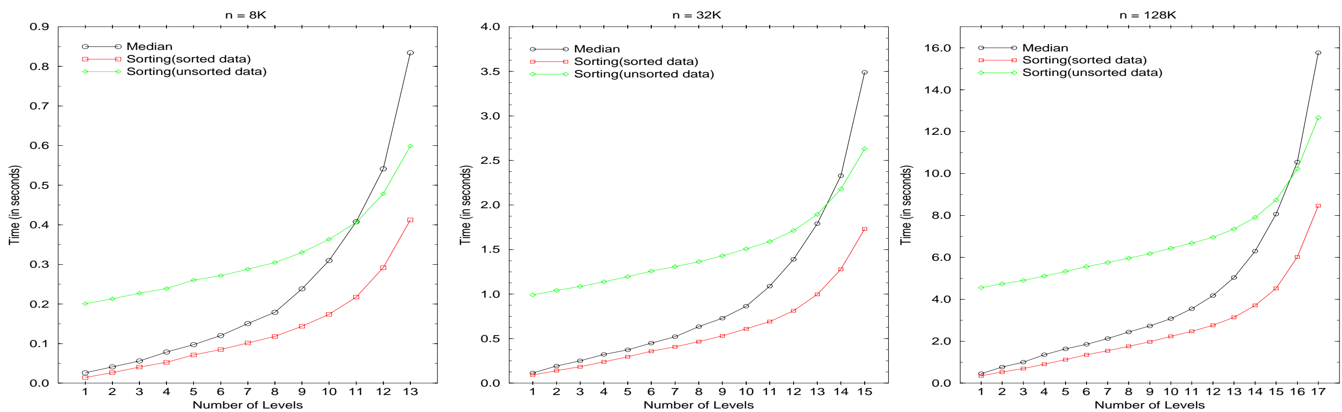


Figure 5.1: Local tree construction for random distribution of data of sizes 8K, 32K and 128K

A comparison of sort-based method and the bucket-based methods shows that bucket-based strategy has a lower value of X, similar value of s, much larger values of r and c. We experimented with different bucket sizes for the bucketing strategy. There is a tradeoff between r and c. The former is directly proportional to the number of buckets while the latter is inversely proportional to the number of buckets. The effect of the overhead r is per list and increases exponentially with increase in the number of levels. We found that the values of r and c are sufficiently large that sort-based method is better than the bucket-based method except when the number of levels is very small (less than 4). However, for these cases, the median-based approach works better. In fact, we found that the median-based strategy is much better than the bucket-based strategy for small levels even ignoring the time required for bucketing.

A comparison of the median and sort-based methods show that the median-based strategy has zero value of X, smaller value of s, a higher value of r, and much larger value of c as seen in Table 5.1 and verified by our experimental results. One would expect that the median-based method to be better for small number of levels and sort-based strategy to be better for larger number of levels, expecting the preprocessing cost to be amortized over several levels. A comparison of these methods for different data sets (of size 8K, 32K and

128K) is provided in Figure 5.1. These results show that median-based method is better than the sort-based method except when the number of levels is close to $\log N$. For larger levels the increase due to higher c becomes significant for the median-based method. These results also show that when data is already sorted, the sort-based method has a better time than the median-based method.

Median-based strategy is the best unless the number of levels is close to $\log N$ or if the data is already available sorted, for which the sort-based strategy is the best. For larger values of k ($k \geq 3$) using a median-based strategy would be comparable or better than the other strategies unless preprocessing information used for sorting method is already available.

5.2 Parallel Tree Construction for $\log p$ levels

5.2.1 Sort-based Method

This algorithm is shown in Table 5.2. We preprocess the data by sorting it along each dimension using parallel sample sort [SS92] to create k sorted arrays. The total time required for sorting is $O(\frac{N}{p}\log N + p\log^2 p + t_s p + t_w(\frac{N}{p} + p^2) + t_h p \log p)$ on a hypercube and $O(\frac{N}{p}\log N + p\log^2 p + t_s\sqrt{p} + t_w(\frac{N}{\sqrt{p}} + p^2) + t_h\sqrt{p})$ on a mesh. For large values of N ($N \geq O(t_s p^2 + t_w p^3)$), the running time is $O(\frac{N}{p}\log N + t_w\frac{N}{P})$ on a hypercube and $O(\frac{N}{p}\log N + t_w(\frac{N}{\sqrt{p}}))$ on a mesh.

Without loss of generality, assume that the initial partition should be split along dimension d_1. The processor containing the median coordinate along d_1 broadcasts it to all the processors. We want to split the partition and assign the resulting subpartitions to the appropriate processors. Assigning a subpartition amounts to computing the sorted arrays for the subpartition. This is already true along d_1. For any other dimension d_l, each processor scans through its part of the array sorted by d_l and splits it into two subarrays depending upon the coordinate along d_1. The subpartitions are moved using order maintaining data movement.

Consider the time required for building the first $\log p$ levels of the tree: At level i of the tree, we are dealing with 2^i partitions containing $\frac{N}{2^i}$ points each. A partition is represented by k sorted arrays distributed evenly on $\frac{p}{2^i}$ processors. Splitting the local arrays and preparing the data for communication requires $O((k-1)k\frac{N}{p})$ time. The required data movement must be done using Order maintaining data movement because the sorted order of the data must be preserved.

Given sorted data, the time required to build the first $\log p$ levels of the tree on a hypercube is $\sum_{i=0}^{\log p - 1} O(k(k-1)\frac{N}{p} + t_s\frac{p}{2^i} + t_w k(k-1)\frac{N}{p} + t_h\frac{p}{2^i}\log\frac{p}{2^i}) = O(k(k-1)\frac{N}{p}\log p + t_s p + t_w k(k-1)\frac{N}{p}\log p + t_h p\log p)$. The corresponding time on a mesh is $\sum_{i=0}^{\log p-1} O(k(k-1)\frac{N}{p} + t_s\sqrt{\frac{p}{2^i}} + t_w k(k-1)\frac{N}{p}\sqrt{\frac{p}{2^i}} + t_h\sqrt{\frac{p}{2^i}}) = O(k(k-1)\frac{N}{p}\log p + t_s\sqrt{p} + t_w k(k-1)\frac{N}{\sqrt{p}} + t_h\sqrt{p})$. For large values of N ($N \geq O(t_s p^2 + t_w p^3)$) the running time of the algorithm is $O(k(k-1)t_w\frac{N}{p}\log N)$ on a hypercube and $O(k(k-1)t_w(\frac{N}{\sqrt{p}}))$ on a mesh.

As in the sequential sort-based method, it is possible to reduce the computational cost at every level from $O(kn)$ to $O(n)$. However, the method requires dereferencing pointers to

---

Algorithm 8 *Tree Construction using sorting*

n - Total number of elements

p - Total number of processors labeled from 0 to $p-1$

L_x, L_y - Two lists of points (x, y) on P_i

m - Number of levels of the k-d tree to be constructed ($m \geq \log p$)

$d_j = x;\ d_k = y;\ n_i = n;$

Initialize the processor pool to contain p processors.

Step 0. Use sample sort to sort L_x in x dimension and L_y in y dimension across all the p processors

for $l \leftarrow 1$ to $\log p$ levels

- **Step** 1. The middle processor in the processor pool picks the last element from L_{d_j} as the median q, in dimension d_j. Broadcast q to all processors in the processor pool.
- **Step** 2. Consider the processors in the processor pool as two halves.
- **Step** 3. Move all points of $L_{d_k} \leq q$ in dimension d_j to the lower half processors and all points $> q$ in dimension d_j to the upper half processors using order maintaining data movement.
- **Step** 4. Switch dimension d_j with d_k and set the processor pool to the appropriate half of the processors to which this processor belongs. The roles of the lists L_{d_j} and L_{d_k} is reversed in the next iteration.
- **Step** 5. Set n_i to the number of points in the current processor pool.

Step 6. Construct the tree locally using **LocalTreeConstruction** on L_x and L_y, each list is of size n_i.

LocalTreeConstruction(L_{d_j}, L_{d_k})

if$(|L_{d_j}| > \frac{n}{2^m})$

- **Step** 1. Pick the middle element of L_{d_j} in dimension d_j as the median q.
- **Step** 2. Perform an order maintaining data movement on L_{d_k} to partition it into two halves, one $\leq q$ and another $> q$ in dimension d_j.
- **Step** 3. Switch dimension d_j with d_k.
- **Step** 4. Recursively find left subtree on $L_{d_j}[0, \frac{n_i}{2} - 1]$ using **LocalTreeConstruction**.
- **Step** 5. Recursively find right subtree on $L_{d_j}[\frac{n_i}{2}, n_i - 1]$ using **LocalTreeConstruction**.

Figure 5.2: Sort-based Method

points on other processors. The resulting communication makes this method impractical even for large values of k.

5.2.2 Median-based Method

Method	Parallel tree construction up to $\log p$ levels			
	Preprocessing (X)	Median finding (c)	Local processing (s)	Communication due to data movement(T)
Median-based	-	$O(\frac{N}{p}\log p + (t_s+t_w)\log^2 p \log N)$	$O(k\frac{N}{p}\log p)$	$O(t_s p + t_w(k\frac{N}{p}\log p))$
Sort-based	$O(\frac{N}{p}\log N + p^2\log p + t_s p + t_w(\frac{N}{p}+p^2))$	$O(\log p)$	$O(k(k-1)\frac{N}{p}\log p)$	$O(t_s p + t_w(k(k-1)\frac{N}{p}\log p))$
Bucket-based	$O(\frac{N}{p}(k+\log p) + t_s p + t_w(k\frac{N}{p}))$	$O(\frac{N}{p})$	$O(k(k-1)\frac{N}{p}\log p)$	$O(t_s p + t_w(k(k-1)\frac{N}{p}\log p))$

Table 5.2: Time for tree construction up to $\log p$ levels on p processors for different strategies on a hypercube.

We compared different deterministic and randomized parallel selection algorithms for coarse grained machines [AfAGR96c] and found that a straightforward parallelization of Floyd's sequential algorithm results in the best performance for random data. Non-random data can be randomized by allocating each point to a random processor and using a transportation primitive to move points to appropriate processors. The cost of this randomization is insignificant compared to the cost of construction of the k-d tree. This algorithm is shown in Figure 5.3.

Let $N_i^{(j)}$ be the number of elements in processor P_i at the beginning of iteration j of the median finding algorithm. Let $N^{(j)} = \sum_{i=0}^{p-1} N_i^{(j)}$. Let $k^{(j)}$ be the rank of the desired element. All processors use the same random number generator with the same seed to produce identical random numbers. Consider the behavior of the algorithm in iteration j. First, a parallel prefix operation is performed on the $N_i^{(j)}$'s. A random number between 1 and $N^{(j)}$ is used to pick the estimated median. From the parallel prefix operation, each processor can determine if it has the estimated median and if so broadcasts it. Each processor scans through its set of points and splits them into two subsets based on the estimated median. A Combine operation and a comparison with $k^{(j)}$ determines which of these two subsets is to be discarded and the value of $k^{(j+1)}$ needed for the next iteration.

Let $N_{max}^{(j)} = max_{i=0}^{p-1} N_i^{(j)}$. Thus, splitting the set of points into two subsets based on the median requires $O(N_{max}^{(j)})$ time in the jth iteration. For random data, it can be shown that the number of remaining points left after each iteration are mapped equally among all the processors with high probability (i.e. the maximum is close to mean) unless the number of remaining points is very small. Thus, the total expected time spent in computation is $O(\frac{N}{p})$. Another option is to ensure that a load balancing is done after every iteration. However, such a load balancing always resulted in an increase in the running time [AfAGR96c].

Algorithm 9 *Tree Construction using median finding*

n - Total number of elements

p - Total number of processors labeled from 0 to $p-1$

L_i - List of points (x, y) on P_i

m - Number of levels of the k-d tree to be constructed $(m \geq \log p)$

$d_j = x;\ d_k = y;\ n_i = n;$

Initialize the processor pool to contain p processors.

for $l \leftarrow 1$ to $\log p$ levels

Step 1. Find median q of n_i points in dimension d_j using fast randomized median method. This also splits L_i into two portions, one $\leq q$ and another $> q$ in dimension d_j

Step 2. Consider the processors in the processor pool as two halves.

Step 3. Move all points $\leq q$ in dimension d_j to the lower half processors and all points $> q$ in dimension d_j to the upper half processors.

Step 4. Switch dimension d_j with d_k and set the processor pool to the appropriate half of the processors to which this processor belongs.

Step 5. Set n_i to the number of points in the current processor pool.

Step 6. Construct the tree locally using **LocalTreeConstruction** on the list $L_i[0, n_i - 1]$

LocalTreeConstruction(L_i)

if$(|L_i| > \frac{n}{2^m})$

Step 1. Find median of L_i in dimension d_j. This also splits the list into two halves, one lesser than the median and another greater than the median.

Step 2. Switch dimension d_j with d_k

Step 3. Recursively find left subtree on $L_i[0, \frac{n_i}{2} - 1]$ using **LocalTreeConstruction**

Step 4. Recursively find right subtree on $L_i[\frac{n_i}{2}, n_i - 1]$ using **LocalTreeConstruction**

Figure 5.3: Median-based Method

The number of iterations required for N points is $O(\log N)$ with high probability. Therefore, the expected running time of parallel median finding, is $O(\frac{N}{p} + (t_s + t_w)\log p \log N)$ on the hypercube and $O(\frac{N}{p} + (t_s + t_w)\log p \log N + t_h\sqrt{p}\log N)$ on the mesh.

In building the first $\log p$ levels of the tree, the task at level i of the tree is to solve 2^i median finding problems in parallel with each median finding involving $\frac{N}{2^i}$ points and $\frac{p}{2^i}$ processors. After finding the median of $\frac{N}{2^i}$ points on $\frac{p}{2^i}$ processors, all the elements less than or equal to the median are moved to the first $\frac{p}{2^{i+1}}$ processors while the other elements are move to the next $\frac{p}{2^{i+1}}$ processors. The maximum number of elements sent out or received by any processor is $\frac{N}{p}$. We assume that this data movement results in a random distribution of the two lists to the two subsets of processors. This can be ensured by randomly permuting the data without increasing the asymptotic complexity. Even if pointers are used for local computation instead of records, the records have to be communicated when data is moved across processors.

Building the first $\log p$ levels of the tree on the hypercube requires $\sum_{i=0}^{\log p-1} O(\frac{N}{2^i}/\frac{p}{2^i} + (t_s+t_w)\log\frac{p}{2^i}\log\frac{N}{2^i} + k\frac{N}{p} + t_s\frac{p}{2^i} + t_w\frac{kN}{p} + t_h\frac{p}{2^i}\log\frac{p}{2^i}) = O(\frac{kN}{p}\log p + t_s(p+\log^2 p\log N) + t_w(k\frac{N}{p}\log p + \log^2 p\log N) + t_h p\log p)$ time. The time required on the mesh is $\sum_{i=0}^{\log p-1} O(\frac{N}{2^i}/\frac{p}{2^i} + (t_s+t_w)\log\frac{p}{2^i}\log\frac{N}{2^i} + t_h\sqrt{\frac{p}{2^i}}\log\frac{N}{2^i} + k\frac{N}{p} + (t_s + t_w\frac{kN}{p})\sqrt{\frac{p}{2^i}} + t_h\sqrt{\frac{p}{2^i}}) = O(\frac{kN}{p}\log p + t_s(\sqrt{p} + \log^2 p\log N) + t_w(\frac{kN}{\sqrt{p}} + \log^2 p\log N) + t_h\sqrt{p}\log N)$.

For large values of N ($N \geq O(t_s p^2 + t_w p^3)$), the total time required by the algorithm is $O(kt_w\frac{N}{p}\log N)$ for the hypercube and $O(kt_w(\frac{N}{\sqrt{p}}))$ on the mesh. Thus, the data movement time dominates.

5.2.3 Bucket-based Method

This algorithm is shown in Figure 5.4 and Figure 5.5. We first create the required bucketing using p processors and construct the first $\log p$ levels of the tree in parallel as before. Bucketing along a dimension, say d_1, can be computed as follows: Select a total of n^ϵ points ($0 < \epsilon < 1$) and sort them along d_1. Using the sorted sample, divide the range containing the points into p intervals called buckets. Using a global concatenate operation, the p intervals are stored on each processor. Each processor scans through its $\frac{N}{p}$ points and for each point determines the bucket it belongs to in $O(\frac{N}{p}\log p)$ time. The points are thus split into p lists, one for each bucket. All the lists belonging to bucket i are moved to processor P_i using the transportation primitive. The expected number of points per bucket is $O(\frac{N}{p})$ with high probability. Apart from the time used in sorting, the time for bucketing is $O(\frac{N}{p}(k+\log p) + t_s p + t_w\frac{kN}{p} + t_h p\log p)$ on a hypercube and $O(\frac{N}{p}(k+\log p) + t_s\sqrt{p} + t_w\frac{kN}{\sqrt{p}} + t_h\sqrt{p})$ on a mesh. Clearly, this dominates the time for sorting as only a sample of size n^ϵ is sorted.

Consider building the first $\log p$ levels of the tree. Suppose that the first split is along dimension d_1. By a parallel prefix operation, the bucket containing the median is easily identified. The median is found by finding the element with the appropriate rank in this bucket using the sequential selection algorithm. The median is then broadcast to all the processors which split their buckets along all other dimensions based on the median. Using Order maintaining data movement, the buckets are routed to the appropriate processors.

Algorithm 10 *Tree Construction using bucketing*

n - Total number of elements

p - Total number of processors labeled from 0 to $p-1$.

L_x, L_y - Two lists of points (x, y) on P_i

B_x, B_y - List of buckets for L_x and L_y on P_i.

m - Number of levels of the k-d tree to be constructed ($m \geq \log p$).

$d_j = x$; $d_k = y$; $n_i = n$;

Initialize the processor pool to contain p processors.

Step 0. Divide L_x into p buckets across all p processors, one bucket per processor such that if $r \in P_j$ and $s \in P_k$ in dimension x then $r < s$ if $j < k$. Similarly divide L_y into p buckets across the processors using the y dimension.

for $l \leftarrow 1$ to $\log p$ levels

Step 1. Perform a global concatenate on the bucket boundaries of B_{d_j} in the processor pool and perform a local prefix sum to calculate the bucket boundaries.

Step 2. Each processors identifies the median containing bucket bkt_m among the buckets on the processor pool and the processor it is on. This can be achieved by a binary search.

Step 3. The processor containing the median bucket performs a selection of the appropriate rank in bkt_m, using randomized selection to calculate median q in dimension d_j. q is broadcast to each processor in the processor pool.

Step 4. Each processor divides the buckets in L_{d_k} into $\leq q$ (lower buckets) and $> q$ (higher buckets) in dimension d_j. The number of buckets on each processor doubles in this phase.

Step 5. Consider the processors in the processor pool as two halves.

Step 6. Move all points of L_{d_k} in lower buckets to the lower half processors and all points in the higher buckets to the upper half processors using order maintaining data movement and maintaining the bucket ordering.

Step 7. Switch dimension d_j with d_k and set the processor pool to the appropriate half of the processors to which this processor belongs. The roles of the lists L_{d_j} and L_{d_k} is reversed in the next iteration.

Step 8. Set n_i to the number of points in the current processor pool.

Step 9. Construct the tree locally using **LocalTreeConstruction** on L_x with buckets B_x and L_y with buckets B_y. Each list is of size n_i.

Figure 5.4: Bucket-based Method

LocalTreeConstruction$(L_{d_j}, B_{d_j}, L_{d_k}, B_{d_k})$

if$(|L_{d_j}| > \frac{n}{2^m})$

Step 1. Use L_{d_j} and find the bucket containing the median, bkt_m, in B_{d_j}, by a binary search on the prefix sum of the bucket indices.

Step 2. Find the median q using randomized selection on bkt_m.

Step 3. Partition the buckets in L_{d_k} into $\leq q$ and $> q$ in dimension d_j.

Step 4. Perform an order maintaining data movement on L_{d_k} to move the lower buckets to the lower half of the list and the upper buckets to the upper half. Update appropriate bucket boundaries.

Step 5. Switch dimension d_j with d_k.

Step 6. Recursively find left subtree on $L_{d_j}[0, \frac{n_i}{2} - 1]$ using **LocalTreeConstruction**.

Step 7. Recursively find right subtree on $L_{d_j}[\frac{n_i}{2}, n_i - 1]$ using **LocalTreeConstruction**.

Figure 5.5: Bucket-based method - Local tree construction

Since the bucket size is smaller than the number of elements in a processor, the time for locating the median in the bucket is dominated by the time for splitting the buckets along each dimension. The first $\log p$ levels of the tree can be built in $\sum_{i=0}^{\log p-1} O(k(k-1)\frac{N}{p} + t_s\frac{p}{2^i} + t_w k(k-1)\frac{N}{p} + t_h\frac{p}{2^i}\log\frac{p}{2^i}) = O(k(k-1)\frac{N}{p}\log p + t_s p + t_w k(k-1)\frac{N}{p}\log p + t_h p\log p)$ time on a hypercube and $\sum_{i=0}^{\log p-1} O(k(k-1)\frac{N}{p} + t_s\sqrt{\frac{p}{2^i}} + t_w k(k-1)\frac{N}{\sqrt{p}} + t_h\sqrt{\frac{p}{2^i}}) = O(k(k-1)\frac{N}{p}\log p + t_s\sqrt{p} + t_w k(k-1)\frac{N}{\sqrt{p}}\log p + t_h\sqrt{p})$ time on a mesh.

5.2.4 Experimental Results

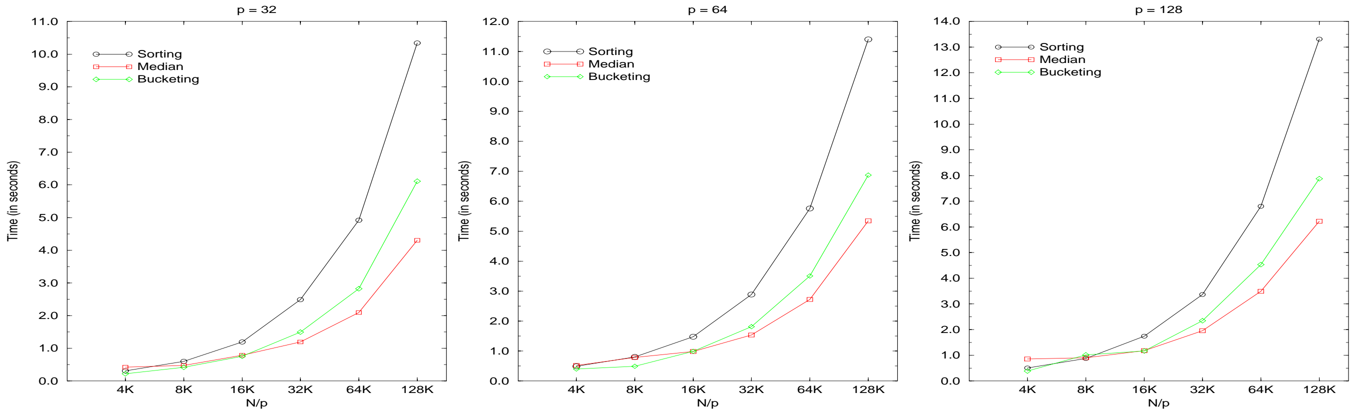


Figure 5.6: Tree construction to log p levels for p = 32, 64 and 128 using random distribution of data

In this section, we compare the three algorithms experimentally using implementations on the CM-5 for which most of the analysis presented for the hypercube is applicable. The

execution time is decomposed into several parts as summarized in Table 5.2 (the t_h term is insignificant in the overall communication time and is ignored): the time required for preprocessing (X), the cost of finding the median at every level (c), the local processing time to rearrange data based on the median (s) and the communication due to data movement (T). The data movement cost for $k-1$ arrays is expected to be higher for the bucket and sort-based methods as they require preserving the order of the data as opposed to a non-order maintaining data movement for only one array in the median-based method. Overhead cost r is associated with maintaining and processing sublists at every level on a per list basis and grows exponentially with the increase in number of levels.

Once again, we limit our experimental results to the two-dimensional case as it proved to be sufficient to draw conclusions for higher dimensional problems. A comparison of sort-based and median-based approaches show that the former has higher values of X, s and T, and a smaller value of r. For small values of $\frac{N}{p}$, median-based method is not expected to parallelize well as the communication cost in c dominates, which increases with increase in p. However, for small p, the median-based method performs well when it offsets the large value of X in the sort-based method. For large values of $\frac{N}{p}$, one would expect median finding to parallelize reasonably well. The communication cost in median finding is significantly lower than T, required for both strategies. The value of T is smaller for the median-based approach as compared to the sort-based approach since it does non-order maintaining data movement. One would expect median-based method to perform better, except for very large values of p.

A comparison of bucket-based and median finding approaches show that the former has larger values of X, s, r and T. For small values of $\frac{N}{p}$, we would expect median-based method to perform worse than the bucket-based approach which has small communication overhead for median finding unless p is small. A small p would keep c low for median-based method but X will still be large for the bucket-based method. For large values of $\frac{N}{p}$, c in median finding should not dominate the overall cost and should be significantly lower than T, required for both strategies at every level. Bucket-based method requires order maintaining data movement and hence has a higher T than the median-based method. Hence, one would expect the median-based method to be better than the bucket-based method, except for a very large p.

Bucket-based method has lower values for X and c when compared to the sort-based method. Hence, the bucket-based method would work better when the difference in preprocessing time is larger than the total time for median finding. The cost of the latter decreases at every level (as the bucket sizes decrease with increase in number of levels).

Figure 5.6 presents experimental results for k-d tree construction up to $\log p$ levels for different values of $\frac{N}{p}$. These show that the bucket-based method is always better than the sort-based method. The median-based method is the best approach for large values of $\frac{N}{p}$ (greater than 8K per processor) while the bucket-based approach is the best for small values of $\frac{N}{p}$ (less than 4K). The improvements of each of these methods over the other are substantial for these ranges. For larger values of k it is expected that the time requirements of the bucket-based strategy would grow faster than the median-based strategy due to overheads in the lists and bucket management.

We emphasize that the above conclusions are drawn for the randomized median-based method. The constants involved in a deterministic algorithm for median finding may make

sort-based and bucket-based methods better for small values of k and for a wide range of $\frac{N}{p}$.

5.3 Global Tree Construction

The parallel tree construction can be decomposed into two parts: constructing the tree till $\log p$ levels followed by local tree construction. Potentially a different strategy can be used for the two parts, resulting in nine possible combinations. Based on the discussion in the previous two sections, the following are the only viable options: sort-based approach followed by sort-based approach (G1), median-based approach followed by sort-based (G2), median-based approach followed by median-based approach (G3), bucket-based approach followed by median-based approach (G4), and bucket-based approach followed by sorting each of the buckets, followed by sort-based approach (G5).

These five approaches are compared for different number of levels ($\log p$ to $\log N$) for different number of processors (8, 32, 128) for different values of $\frac{N}{p}$ (4K and 128K) (see Figure 5.7). These results show that for large values of $\frac{N}{p}$, strategy G3 is the best unless the number of levels are close to $\log N$ for which G2 may be preferable. For small values of $\frac{N}{p}$, the strategy G4 is preferable. If the number of levels are close to $\log N$, G1 and G5 are the best.

For larger k, we would expect that one of the median or bucket-based strategies should be used to construct the tree till $\log p$ levels. This would depend on the value of $\frac{N}{p}$ and the target architecture. The bucket-based strategy would be better for small values of $\frac{N}{p}$ and k. The local tree construction should use median-based strategy. Using the above approach results in software which will be close to the best for nearly all values of the parameters.

5.4 Reducing the Data Movement

The algorithms described for parallel construction of the first $\log p$ levels of the tree require massive data movement using transportation primitive at every level. This can be significantly reduced by using a different approach.

Consider the median-based method. Initially, all the N points belong to one partition and are distributed uniformly on all p processors. After finding the median, the local data is divided into two subarrays (typically of unequal size), each belonging to one of the subpartitions. Instead of moving the data such that each subpartition is assigned to a different subset of processors, one can assume that these subpartitions are divided among all the processors. A median can be found for each of the subpartitions in parallel by combining the communication and computation for both. Because each processor always has a total of $\frac{N}{p}$ points, computational load is perfectly balanced among all the processors. This approach is repeatedly applied until the number of subpartitions is equal to p. At this stage, the data is distributed among the processors such that each processor has all the points of one of the p subpartitions and local trees are constructed.

Suppose that i levels of the tree are already constructed. At this stage, there are 2^i subpartitions, each divided among all the processors. It is desired to find the medians

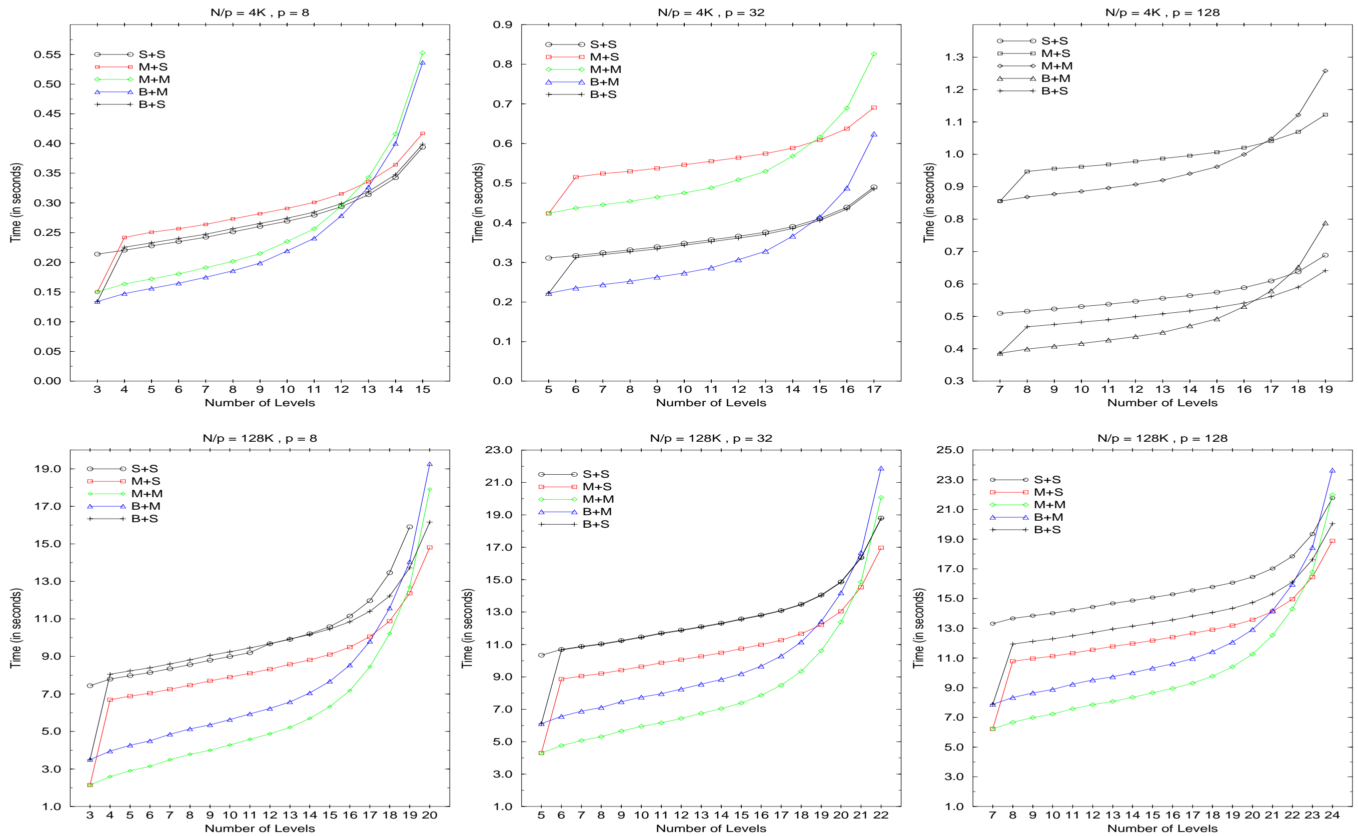


Figure 5.7: Global tree construction for random distribution of data of size $\frac{N}{p}$ = 4K, 128K on p= 8, 32, 128. On the X-axis level i represents the tree is built to i levels (a tree having 2^i leaves)

for all the subpartitions together. A parallel prefix operation is performed for each of the subpartitions to number the points in each processor belonging to a subpartition. All the 2^i parallel prefix operations can be combined together. By generating appropriate random numbers, each processor determines if it has the estimated median of each subpartition. A processor updates the corresponding entry in an array of size 2^i if it has guessed the median for the ith subpartition, otherwise it stores a 0. By a combine operation on this array using the '+' operation, the required medians are stored on each processor. Using the 2^i estimated medians, all processors together reduce the size of the subpartitions under consideration. The iterations are repeated until the total size of all the subpartitions falls below a constant. At this stage, all the subpartitions can be gathered in one processor and the required medians can be found.

The cost of this algorithm in constructing level $i+1$ of the tree from level i is $O(\frac{N}{p} + t_s \log p \log N + t_w 2^i \log p \log N)$ on a hypercube and $O(\frac{N}{p} + t_s \log p \log N + t_w 2^i \log p \log N + t_h \sqrt{p} \log N)$ on a mesh. Constructing the first $\log p$ levels of the tree on a hypercube requires $\sum_{i=0}^{\log p - 1} O(\frac{N}{p} + t_s \log p \log N + t_w 2^i \log p \log N)$ time plus $O(\frac{kN}{p} + t_s p + t_w \frac{kN}{p} + t_h p \log p)$ time for the final data movement. Thus, the run time is $O(\frac{N}{p}(\log p + k) + t_s(p + \log^2 p \log N) + t_w(\frac{kN}{p} + p \log p \log N) + t_h p \log p)$. The corresponding time on the mesh is $\sum_{i=0}^{\log p - 1} O(\frac{N}{p} + t_s \log p \log N + t_w 2^i \log p \log N + t_h \sqrt{p} \log N) + O(\frac{kN}{p} + t_s \sqrt{p} + t_w \frac{kN}{\sqrt{p}}) = O(\frac{N}{p}(\log p + k) + t_s(\sqrt{p} + \log^2 p \log N) + t_w(\frac{kN}{\sqrt{p}} + p \log p \log N) + t_h \sqrt{p} \log p \log N)$.

For both hypercubes and meshes, the computational cost reduces from $\frac{kN}{p} \log p$ to $\frac{N}{p}(k + \log p)$. This is the cost involved in local computations plus the cost in copying the records to arrays for data movement. For hypercubes (or permutation networks), the amount of data transferred reduces by a factor of $\log p$ from $\frac{kN}{p} \log p$ to $\frac{kN}{p}$. However, the data transferred improves only by a small constant factor on the mesh. A similar strategy can be used to reduce the data movement for sort-based and bucket-based methods.

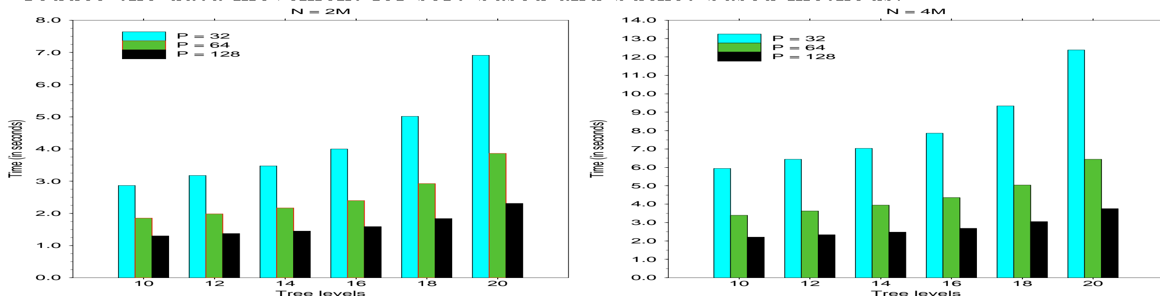


Figure 5.8: Tree construction time using median-based method followed by median-based method (G3) on random data

5.4.1 Experimental Results

We limited ourselves to applying this strategy to the median-based approach only. Figure 5.9 gives a comparison of the two approaches (with and without data movement) for different values of $\frac{N}{p}$ for $\log p$ levels of parallel tree construction. These results show that using the new strategy gives significant improvements due to lower data movement.

The median-based method with reduced data movement potentially reduces the data movement cost by a factor of $\log P$. Since the data movement cost is proportional to the size of the records, the effect of extra overhead due to communication in the new method reduces significantly for higher dimensional data. Thus, the new strategy should give improved performance in higher dimensions.

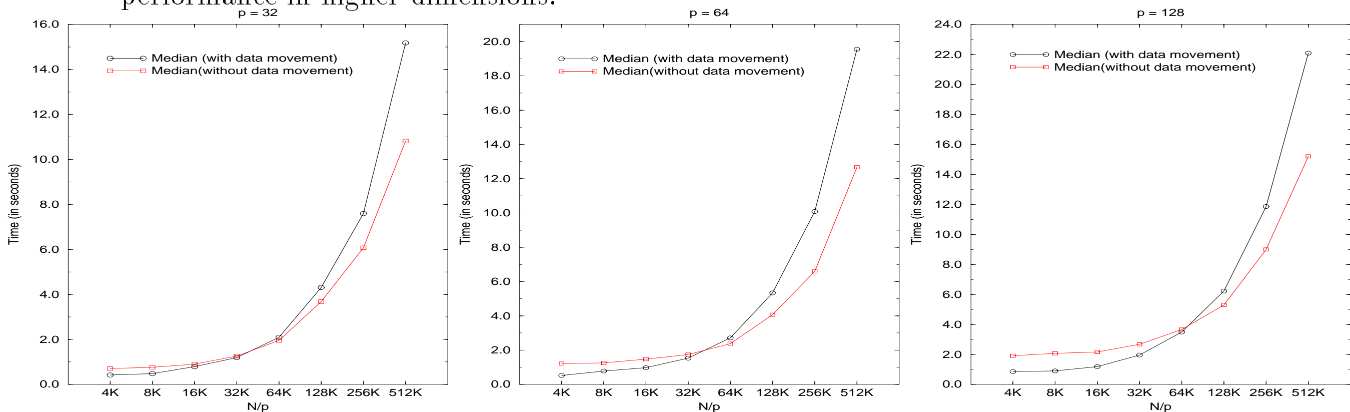


Figure 5.9: Comparison of the two approaches for $\log p$ levels for different values of $\frac{N}{p}$.

5.5 Conclusions

In this paper, we have looked at various strategies for parallel construction of multidimensional binary search trees. Traditionally, a sort-based strategy is advocated for sequential construction [PS85] or parallel construction [Ble90] of complete k-d trees. For two dimensional point sets, we found that the the median-based strategy is the fastest for large values of $\frac{N}{p}$, unless the number of levels is very close to $\log N$. In such a case, using the median-based approach up to $\log p$ levels followed by using a sort-based approach locally performed better. For small values of $\frac{N}{p}$, using the bucket-based approach up to $\log p$ levels followed by using a median-based approach locally is the best except when the tree is built almost completely. In such a case, using the bucket-based method followed by using sort-based method locally outperformed a complete sort-based strategy except when the number of processors is very small.

It is interesting to note that a complete sort-based approach did not perform better even if the tree is built completely. The median-based approach also exhibits good scaling as can be seen by the run-time analysis and experimental results. This is true mainly because of the performance of randomized median finding on randomly distributed data sets. For arbitrary data sets, a randomization step can be performed without much additional cost. We found that deterministic median finding algorithms are slower by an order of magnitude and their use would lead to entirely different conclusions. Bucket-based strategy is useful for applications such as graph partitioning when the number of points per processor is small.

For data in k dimensions, the preprocessing cost and the cost of tree construction per level of sort-based and bucket-based methods increases proportional to k. The median-based method remains unaffected in this regard. While the data movement time in median-based methods increases proportional to k, it increases proportional to $k(k-1)$ in other methods.

Thus, based on the dismal performance of sort-based and bucket-based methods for $k = 2$, we conclude that the median-based method is superior for $k \geq 3$.

Our experiments with comparing median finding with data movement at every stage and median finding with reduced data movement associate well with the idea of task versus data parallelism. For this, we showed that utilizing task parallelism leads to worse results as compared to "concatenated" data parallelism for large granularities. Often, problems amenable to the divide and conquer paradigm are solved in parallel by mapping the corresponding divide and conquer tree using task parallelism. Our technique can be effectively utilized to solve such problems efficiently, especially when the task sizes are non-uniform. We are currently exploring this strategy.

Random distribution of data has potential advantages in a number of applications. We feel that data-parallel languages such as High Performance Fortran should provide constructs for random distribution to facilitate coding such applications.

A generalized version of the problem we have considered in this paper is the construction of weighted multidimensional binary search trees. In this case, each point has a weight associated with it and a partition is split based on the weighted median. Therefore, the number of points in the subpartitions at level i of the tree can be very non-uniform. In such a case, our method with reduced data movement is clearly superior since it keeps the number of points on each processor balanced throughout the construction of the tree.

If a perfectly balanced k-d tree is not required, a random sampling based approach can be used to reduce the overall cost significantly. Choose a small random sample of all the points. Construct the tree for this random sample using one of the methods described in the paper up to a fixed number of levels. Assign the remaining points to one of the leafs based on the "sample" tree. Each leaf node can be constructed recursively using the same strategy.

Chapter 6

Concatenated Parallelism

6.1 Introduction

Scheduling a number of tasks on a parallel machine to minimize the running time for the completion of all the tasks is a well-studied problem in parallel computing. In the most general case, the tasks can be of varying sizes and each task itself can be solved in parallel. Two basic types of parallelism can be exploited: Scheduling of independent tasks to different groups of processors such that the tasks can be solved simultaneously in parallel is called *Task Parallelism*. Solving each individual task in parallel using all the processors and solving the tasks one after the other is called *Data Parallelism*. Of course, it is possible to use a combination of these strategies for optimal scheduling, and such a strategy is referred to as *Mixed Parallelism*. Several researchers have worked on exploiting mixed parallelism, both in theory [BB90, FST92, LT94, TWY92] and in practice [CDY95, Cha91, RSB94, SSOG93].

In a number of problems, all the tasks may not be known in advance but may be generated dynamically as existing tasks are processed. This is the case with problems whose efficient solutions use the divide and conquer strategy. The execution of an instance of such a problem can be represented by a divide and conquer tree. Each internal node of the tree corresponds to a task. After performing some computation, the task is split (divide step) into several subtasks which are represented by the children of the node. The subtasks are solved recursively and the solutions may need to be combined to find the solution for the task (merge step).

Several important issues arise in parallelizing such applications using task and data parallelism. Suppose that a task is currently distributed on a group of processors. After performing the work required to divide the task into subtasks in parallel, several options exist for the solution of the subtasks. In the task parallelism approach, the processors are divided into subgroups, perhaps according to the size of the subtasks, and the subtasks are moved to their respective subgroups of processors and solved independently. This requires movement of data to the appropriate processor subgroup. In the data parallelism approach, the subtasks are solved one after another using all the processors. Unfortunately, each subtask may not be uniformly spread across the processors even though the parent task is. Hence, data parallelism may lead to severe load imbalance. Also, the practical efficiency of a parallel algorithm often decreases with increase in the number of processors.

If the time required to divide the subtasks is significantly higher than the cost of redistribution, communication time due to allocation of the subtasks can be ignored. Such an assumption is often made in the literature [CDY95]. Unfortunately, it is not valid for several important problems, which include quicksort, quickhull, construction of quad/octrees and multidimensional binary search trees.

In this chapter, we propose a new strategy called *Concatenated Parallelism* for efficient solution of problems resulting in divide and conquer trees. The basis idea is to solve all the subtasks together using all the processors. Even though the distribution of each subtask is non-uniform, this scheme does not lead to load imbalance (as in data parallelism) because the sum of the sizes of the subtasks allocated to each processor is uniform. This scheme also eliminates the communication due to data movement in the intermediate steps, the drawback of task parallelism. The strategy is particularly useful when the sizes of the subtasks may be non-uniform. The only disadvantage of concatenated parallelism is that communication in solving the subtasks involves all the processors as opposed to increasingly smaller subsets as in task parallelism. However, we can often considerably reduce this expense by spooling the communication required for all the subtasks. Such a strategy significantly reduces the communication cost because set up times for communication are typically higher than transmission costs by two orders of magnitude. We use the concatenated parallelism strategy until enough subtasks are generated to map them uniformly to individual processors. At this stage, one redistribution is performed followed by sequentially solving the subtasks.

Our focus in this chapter is in designing strategies that have practical efficiency on parallel computers. Therefore, we use coarse-grained distributed memory parallel computers as our models of parallel computation as most existing parallel computers belong to this category. A coarse-grained parallel computer consists of several relatively powerful processors connected by an interconnection network. Instead of making specific assumptions about the network connecting processors, we describe our algorithms in terms of some basic communication primitives. The running time of our algorithms on a specific interconnection network can be easily derived by substituting the running times of the communication primitives. We provide such an analysis for hypercubes and meshes.

6.2 Concatenated Parallelism

A generic divide and conquer algorithm divides a given task into a number of subtasks which are solved recursively until the size of the subtasks is small enough to be solved directly. Consider a task of size N on p processors, initially distributed such that each processor has $\frac{N}{p}$ elements. Without loss of generality, assume that both N and p are powers of 2. For convenience of presentation, assume that a task of size S is divided into two subtasks S_1 and S_2. We are considering problems with $|S_1| + |S_2| \leq |S|$, where $|S|$ denotes the size of S. At stage i ($0 \leq i < \log N$) of the subdivision, 2^i subtasks are created. Our technique can be extended to problems in which the number of subtasks is more than two or the sum of the sizes of the subtasks is larger than the size of the parent task (e.g. binary space partitions).

There are two conventional approaches to solving a given number of tasks in parallel: *Task parallelism* and *Data parallelism*. We describe each of them below and provide a comparison with *Concatenated parallelism* that we propose in this chapter. The benefit

of concatenated parallelism, as will be clear from the discussion below, comes from eliminating repeated redistributions of subtasks and providing load balancing. Concatenated Parallelism is decidedly superior to Task Parallelism only when the time required to divide a task is linear in the size of the task or close to linear. However, this restriction is valid for a large variety of practical and useful problems.

In task parallel execution, a different group of processors is allocated for each task and all the tasks are executed in parallel. A task parallel divide and conquer algorithm divides the initial task S, using all the p processors. After the subdivision of S into two subtasks S_1 and S_2, each of these will be allocated to a different subgroup of processors. Two allocation schemes are possible: either the processors are divided into two equal sized subgroups, or processor subdivision is proportional to the sizes of S_1 and S_2. The choice depends on topological considerations for a given architecture. This process is repeated recursively until there are p subtasks, one on each processor. A sequential algorithm is now used to solve the subtasks.

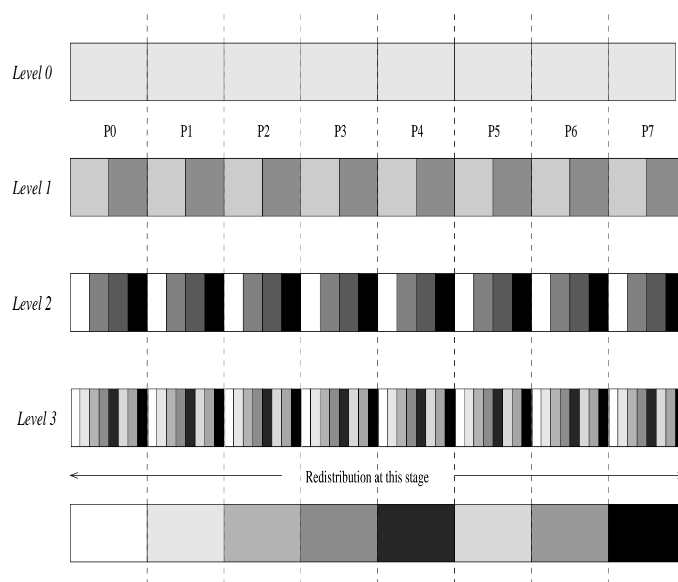


Figure 6.1: Illustration of Concatenated parallelism for $p = 8$.

When a task S of size N is divided into two subtasks S_1 and S_2, each processor will have some data of both the subtasks. Moving the subtasks to different processor groups causes redistribution of data, which can be performed using the Transportation Primitive outlined in Section 2. Because the total amount of data on each processor is $O(\frac{N}{p})$, such a redistribution cost is proportional to N on both hypercubes and meshes. If the sequential cost of dividing the task S is $O(N^{\alpha})$ for some $\alpha > 1$, this cost dominates the cost of redistribution. In this case, strategies to reduce redistribution cost will not be effective. If $\alpha = 1$, the cost of redistribution can no longer be ignored. Also, the constant involved in the redistribution cost (which requires communication) is typically larger than the constant in the subdivision cost (in which the dominant term is usually due to computation) by an order of magnitude. Due to this reason, the communication cost cannot be ignored in

practice for some small values of α greater than 1. Tasks where the subdivision cost is like $O(N \log N)$ can potentially benefit by reducing the redistribution cost for practical values of N.

A data parallel algorithm solves one task after another in parallel using all p processors. A task S is subdivided into two and kept locally. Given tasks $S_1, S_2, \ldots S_{2^i}$ distributed on p processors, after i subdivisions, the tasks are further subdivided one after another. Each processor has a portion of each of the subtasks. There are two drawbacks to this approach: The distribution of each subtask across processors may not be uniform. This creates problems due to load imbalance. Also, since the sizes of the subtasks keep decreasing, the "grain-size" of the computation (size of the problem on one processor) keeps reducing. Because unit communication is more expensive than unit computation by two to three orders of magnitude, there is a threshold grain-size below which the communication overhead severely limits any benefits due to parallelization.

In concatenated parallelism, all problems are solved together using all the p processors. A task is divided into subtasks on each processor and subtasks are kept locally. A processor contains portions of each of the 2^i subtasks at the ith level, as shown in Figure 6.1. All k $(1 \leq k \leq 2^i)$ subtasks are solved together in parallel using all the p processors. This process of dividing a task into subtasks is repeated until a stage is reached where enough subtasks have been created to be distributed to individual processors and solved sequentially. The first such stage is reached at the $\log p^{th}$ level. At this stage there are p subtasks, distributed across all the processors. A redistribution step can gather a subtask on each processor. This works well when subdivision of tasks leads to balanced subtasks. However, when dealing with unbalanced tasks, this can lead to grave load imbalance. This can be rectified by using concatenated parallelism further, continuing to divide the tasks beyond $\log p$ levels. This helps to achieve a better load balance as relatively fine-grained tasks are being gathered for allocation to processors. Apart from load balancing considerations, the level at which concatenated parallelism should be stopped and redistribution done depends on the comparative cost of task subdivision versus task redistribution. A later section describes redistribution strategies and criteria for applying redistribution. The framework for Concatenated Parallelism is illustrated in Figure 6.2.

At every stage, a processor works with $\frac{N}{p}$ elements even though they might belong to different subtasks. This is an important feature of the algorithm because it guarantees load balance at every stage even though individual subtasks are not balanced. Another view of the Concatenated Parallelism is that the grain-size is always $\frac{N}{p}$ and never decreases irrespective of how small the individual subtasks are. Divide and conquer algorithms resulting in unbalanced or randomized subdivisions especially benefit from this algorithm.

We distinguish between three different types of divide and conquer trees: deterministic balanced trees, deterministic unbalanced trees and randomized trees. A generic divide and conquer algorithm for Task Parallelism is modeled by the recurrence relation which can be written as $T(N,p) = \max\{T(x, \lfloor \alpha p \rfloor), T(N-x, \lceil (1-\alpha)p \rceil)\} + f(N,p) + g(N,p)$, where α is a factor governing processor allocation, $f(N,p)$ is the computation cost and $g(N,p)$ is the communication cost in dividing a task of size N on p processors. A generic recurrence relation for Data Parallelism is $T(N,p) = T(x,p) + T(N-x,p) + f(N,p) + g(N,p)$. In Concatenated parallelism all tasks at each stage are solved together spooling the communication, unlike the case of Data Parallelism, where tasks are solved one after

---

Algorithm 11 *Concatenated parallel algorithm*

N : Total size of the task.

p : Total number of processors labeled from 0 to $p-1$.

$f(s,p)$: Computation work for a task of size sp distributed on p processors.

$g(s,p)$: Communication required along with performing $f(s,p)$.

$h(s,p)$: Redistribution cost for tasks with total size s on p processors.

Q_j : size of the task j, $1 \le j \le 2^k$, at stage k. Initially k=0 and $|Q_1| = N$.

x : $1 \le x < N$, a task of size N splits into subproblems of size x and $N-x$.

K : A value for $|Q_j|$ below which partitioning will stop.

done = **false**

while(! done)

  while ($\sum_j g(Q_j, p) < h(\sum_j Q_j, p)$ and $\max_j |Q_j| > K$)

    Step 1. Split each of the j tasks into two subtasks by appropriate subdivision of Q_j on all the processors. Increment k by 1.

  /* *Redistribution phase* */

  If $\max_j |Q_j| < K$ then

    Step 2. Redistribute the task Q_j, $1 \le j \le 2^k$ to individual processors.

    Step 3. done = **true**

  else

    Step 4. Redistribute the problems Q_j, $1 \le j \le 2^k$ to processor pools of size $\frac{P}{2^m}$ for a maximum m, $1 \le m \le \log P$ such that $\sum_j g(Q_j, \frac{p}{2^m}) < h(\sum_j Q_j, \frac{p}{2^m})$.

    Step 5. Set p to $\frac{p}{2^m}$.

    Step 6. *If* ($m == \log P$) done = **true**.

Figure 6.2: Concatenated parallel algorithm

another. As discussed previously, we only consider problems for which $f(N,p) = O(\frac{N}{p})$. $g(N,p)$ is chosen to be $O((t_s + t_w)\log p)$ in our analysis on both hypercubes and meshes, because the communication required for subdividing a task typically requires a combination of Parallel Prefix, Combine and Broadcast operations for many applications. Of course, given a particular problem, analysis specific to that problem can always be performed.

In a deterministic balanced divide and conquer algorithm, problems are split into two halves, and therefore x is $\frac{N}{2}$. In the other two categories, x has no guaranteed size. In deterministic unbalanced trees, x can be any integer from 1 to N. However, the value of x is completely determined by the input and two runs on the same input will give the same value of x. In randomized trees, x can take any value from 1 to N with some probability associated with each possible value. Two runs on the same input may lead to two different values of x. In studying Task Parallelism, we either allocate half the processors to each subproblem in which case α will be set to $\frac{1}{2}$, or allocate processors proportional to the subtask sizes in which case α will be set to $\frac{x}{N}$.

At a stage i, in Concatenated Parallelism, each of the 2^i subtasks need to be divided into two subtasks. This requires communication between processors which can be spooled together. Since a processor contains portions of all tasks, it might need to broadcast data for a subtask. Different processors might end up broadcasting elements for different subtasks. To avoid 2^i broadcasts, we adopt the following strategy for spooling communication: Each processor has an array of size 2^i corresponding to the 2^i subtasks, with all elements initialized to zero. If a processor has the element to broadcast for a subtask, it fills the corresponding element of the array with that element. By doing a Combine operation on this array using the '+' operation, the required elements for all subtasks are stored on each processor.

Our goal is to analyze Concatenated Parallelism and compare it with both Task Parallelism and Data Parallelism. One can easily show that Data Parallelism always performs worse than Concatenated Parallelism irrespective of the nature of the tree, size of the problem or number of processors. Consider a case when the divide and conquer tree is evaluated up to i levels and let the 2^i subtasks be $S_1, S_2, \ldots S_{2^i}$. Let ${S_j}^k$ refer to the portion of the jth subtask on the kth processor, $0 \leq k < p$ and $1 \leq i \leq 2^i$. Let T_{dp} denote the computation time to divide all these subtasks into further subtasks. Since tasks are performed in sequence and each subsequent task is solved on all p processors, the computation time for subtask j is $\max_{k=0}^{p-1}|{S_j}^k|$. The time for all the 2^i tasks is $\sum_{j=1}^{2^i}\max_{k=0}^{p-1}|{S_j}^k|$. It is easily seen that $T_{dp} \geq O(\frac{N}{p})$. The equality is obtained when subtasks are uniformly distributed among the processors. In Concatenated Parallelism, the computation time is $O(\frac{N}{p})$. Also, in Concatenated Parallelism, the time spent in necessary communication for subdividing the subtasks is always smaller than communication time spent in Data Parallelism. This is because communication for all the subtasks is spooled together and this saves expensive set-up costs in individual communications. One can easily see that even when the sequential cost of subdividing a task of size N is any general function of N (not necessarily linear), Concatenated Parallelism always provides at least as good a load balance as Data Parallelism ($\max_{k=0}^{p-1}\sum_{j=1}^{2^i}(|{S_j}^k|)^\alpha \leq \sum_{j=1}^{2^i}\max_{k=0}^{p-1}(|{S_j}^k|)^\alpha$). Therefore, we limit our theoretical and experimental comparisons to comparing Concatenated Parallelism with Task Parallelism only.

The following sections present each of the three categories of divide and conquer algorithms. For each category, we analyze Concatenated Parallelism and Task Parallelism with the assumption that $f(N,p) = O(\frac{N}{p})$ and $g(N,p) = O((t_s + t_w)\log p)$. A different $g(N,p)$ can be analyzed for a problem if the need arises. This is done in a later section for construction of multidimensional binary search trees, where this function is $O((t_s + t_w)\log p \log N)$. Example applications follow each category and performance results on the CM-5 are provided to supplement the analysis.

6.3 Deterministic Balanced Parallel Divide and Conquer

Deterministic balanced divide and conquer algorithms result in the subdivision of a task of size N into two subtasks of size $\frac{N}{2}$, at each stage. Thus after i iterations, each of the 2^i subtasks will have size $\frac{N}{2^i}$. Every processor contains portions of each subtask. A balanced divide and conquer algorithm results in p perfectly balanced tasks after $\log p$ iterations. An appropriate criteria is used for redistributing the p subtasks to individual processors. At a stage i, $\sum_{k=0}^{2^i} f(\frac{N}{2^i}, p)$ is the computation cost to achieve the subdivision of all tasks. Communication is combined for all the tasks. After $\log p$ steps of the subdivision, a final redistribution cost $h(N,p)$ is incurred.

The running time to create p subtasks, one for each processor, using spooling of communication for deterministic balanced divide and conquer is $\sum_{i=0}^{\log p-1} O(2^i\frac{N}{2^i p} + t_s \log p + t_w 2^i \log p)$ on both a hypercube and a mesh. Adding $h(N,p) = O(t_s p + t_w \frac{N}{p})$ on a hypercube and $O(t_s\sqrt{p} + t_w\frac{N}{\sqrt{p}})$ on a mesh, we obtain $O(\frac{N}{p}\log p + t_s(p + \log^2 p) + t_w\frac{N}{p})$ as the running time on a hypercube and $O(\frac{N}{p}\log p + t_s(\sqrt{p} + \log^2 p) + t_w\frac{N}{\sqrt{p}})$ on a mesh, for a deterministic balanced divide and conquer algorithm.

In a task parallel approach which divides a processor subgroup equally at each level, there are 2^i subtasks, each being distributed on processor subgroups of size $\frac{p}{2^i}$ at some level i. Each subtask has size $\frac{N}{2^i}$ which needs to be solved in parallel on the processor subgroup. We obtain the running time for this method using the recurrence relation for Task Parallelism as $\sum_{i=0}^{\log p-1} O(\frac{N}{2^i}/\frac{p}{2^i} + t_s(\frac{p}{2^i} + \log\frac{p}{2^i}) + t_w(\log\frac{p}{2^i} + \frac{N}{p}))$ on a hypercube. The redistribution cost is a part of the recurrence relation in this case. This gives the running time as $O(\frac{N}{p}\log p + t_s(p + \log^2 p) + t_w\frac{N}{p}\log p)$. The corresponding running time on a mesh is $\sum_{i=0}^{\log p-1} O(\frac{N}{2^i}/\frac{p}{2^i} + t_s(\sqrt{\frac{p}{2^i}} + \log\frac{p}{2^i}) + t_w(\log\frac{p}{2^i} + \frac{N}{p}\sqrt{\frac{p}{2^i}})) = O(\frac{N}{p}\log p + t_s(\sqrt{p} + \log^2 p) + t_w\frac{N}{\sqrt{p}})$.

Note that this category yields the best case for task parallelism. However, concatenated parallelism does better in this case since it reduces the redistribution costs from $\frac{N}{p}\log p$ to $\frac{N}{p}$ on a hypercube.

6.3.1 Applications - Multidimensional Binary Search trees

Construction of multidimensional binary search trees (abbreviated k-d trees) [AfAGR96a], in which dividing a task is based on the median element of the corresponding data set, uses a balanced divide and conquer algorithm. The root of the k-d tree corresponds to the set of all points. Assume k dimensional data with $d_1, d_2, \ldots, d_k$ as the dimensions. Choose a dimension d_l and partition points into two sets, one containing points with coordinates

less than or equal to this median along dimension d_l and another containing points with coordinates greater than the median. The two partitions are represented by the children of the root node. The tree is built recursively until each node corresponds to a prespecified number of points.

Consider the construction of a k-d tree of N points on p processors with $\frac{N}{p}$ points on each processor initially. We use a randomized median finding algorithm [AfAGR96c] to calculate the median for points along dimension d_1. Points are divided into two subsets, S_1 and S_2, using the median. S_1 contains points with their d_1 coordinate values less than or equal to the median and S_2 contains points having their d_1 coordinate values greater than the median. Considering the processors partitioned into two halves, task parallelism would gather S_1 on the lower half subgroup of processors and S_2 on the upper half subgroup of processors. This process is repeated recursively, changing dimensions in a cyclic manner to find the median, until each processor has a subset and each processor subgroup contains a single processor. A sequential algorithm is then applied to solve the problem locally.

Thus, for k-d tree construction, $f(N,p)$ is the cost of parallel median finding of N elements on p processors plus the cost of partitioning local data into two portions, one containing elements less than or equal to the median and another containing elements greater than the median. $g(N,p)$ is the associated communication cost. For this method, $f(N,p)$ is $O(\frac{N}{p})$ and $g(N,p)$ is $O((t_s+t_w)\log p \log N)$. Using Task Parallelism, building the first $\log p$ levels of the tree on a hypercube requires $\sum_{i=0}^{\log p-1} O(\frac{N}{2^i}/\frac{p}{2^i} + (t_s+t_w)\log\frac{p}{2^i}\log\frac{N}{2^i} + k\frac{N}{p} + t_s\frac{p}{2^i} + t_w\frac{kN}{p}) = O(\frac{kN}{p}\log p + t_s(p+\log^2 p\log N) + t_w(k\frac{N}{p}\log p + \log^2 p \log N))$ time. The time required on a mesh is $O(\frac{kN}{p}\log p + t_s(\sqrt{p}+\log^2 p\log N) + t_w(\frac{kN}{\sqrt{p}} + \log^2 p\log N))$.

A Concatenated Parallel algorithm divides the tasks across each processor. Starting with a single task, subdivision is applied recursively to each subset resulting in 2^i subsets after i partitions. After redistribution, a sequential algorithm is used on each processor to construct the k-d tree locally. In this case, communication for the 2^i subtasks is combined together using the technique described in an earlier section. The cost of median finding at level i is $O(\frac{N}{p} + t_s\log p\log N + t_w 2^i \log p \log N)$ for a hypercube and $O(\frac{N}{p} + t_s\log p\log N + t_w 2^i\log p\log N)$ for a mesh. Median finding is done for each partition and the data in a partition is automatically subpartitioned into two subsets in the process. Cost of the redistribution is $O(\frac{kN}{p} + t_s p + t_w\frac{kN}{p})$ on a hypercube and $O(\frac{kN}{p} + t_s\sqrt{p} + t_w\frac{kN}{\sqrt{p}})$ on a mesh. Combining these costs, the running time for a deterministic balanced divide and conquer algorithm using concatenated parallelism is $O(\frac{N}{p}(\log p + k) + t_s(p+\log^2 p\log N) + t_w(\frac{kN}{p} + p\log p\log N))$ on a hypercube. The corresponding time on a mesh is $O(\frac{N}{p}(\log p + k) + t_s(\sqrt{p} + \log^2 p\log N) + t_w(\frac{kN}{\sqrt{p}} + p\log p\log N))$.

The data movement due to redistribution reduces by a factor of $\log p$ on hypercubes by using Concatenated Parallelism. The cost of redistribution does not reduce on the mesh. Computation cost reduces from $O(\frac{N}{p}k\log p)$ to $O(\frac{N}{p}(k+\log p))$ when Concatenated Parallelism is used. Therefore, higher dimensional data results in even better performance by using Concatenated parallelism.

6.3.2 Experimental Results

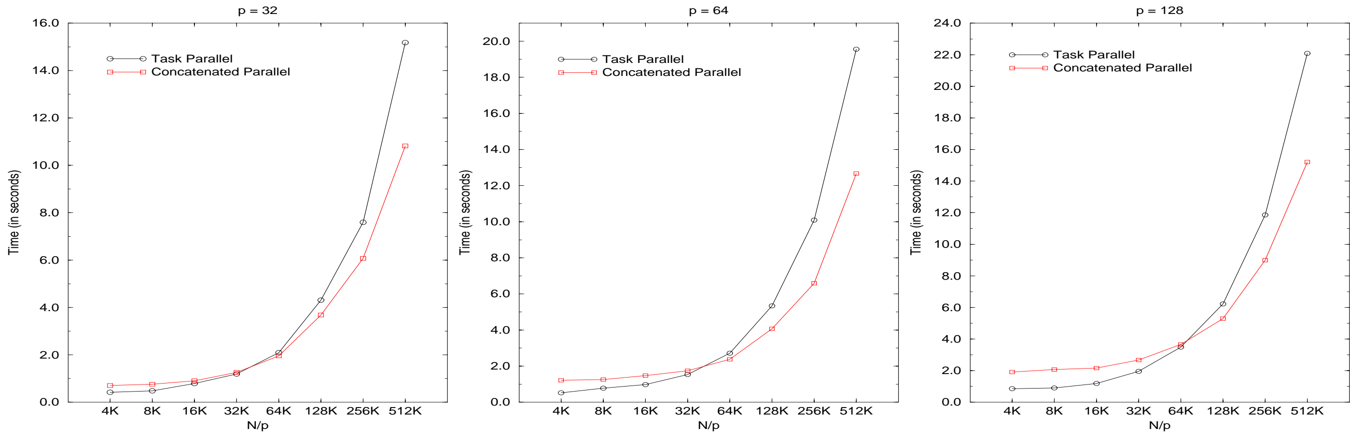


Figure 6.3: Comparison of Task Parallelism and Concatenated Parallelism for balanced divide and conquer in construction of k-d trees for different values of N/p.

We have implemented the construction of two dimensional binary search trees on the CM-5. Two algorithms are compared: Task parallelism with processor groups being divided equally into two subgroups at each stage and Concatenated parallel. Figure 6.3 presents a comparison of the two approaches for various values of $\frac{N}{p}$ for tree construction up to $\log p$ levels. The results show that concatenated parallelism works better than task parallelism for balanced divide and conquer only when there are a large number of points on a processor. The gains from reducing redistribution cost can be obtained only when the data is large in the balanced case. In the unbalanced case, as we will observe in a later section, come both from reducing redistribution cost and providing good load balance. For higher dimensional data, the cost of redistribution is more significant since it involves a higher volume of data to be redistributed and hence Concatenated Parallelism is expected to perform even better.

Figure 6.4 shows the component times for k-d tree construction up to $\log p$ levels with N = 2M and 4M. The total time for tree construction is divided into computation time and communication time for median finding at all levels and data redistribution time. We observe that redistribution time for Concatenated Parallelism is significantly lower than for Task Parallelism. The computation time for Concatenated Parallelism is slightly higher due to added list management. Concatenated Parallelism involves all p processors at all stages for communication whereas for Task Parallelism communication occurs in processor subgroups. Certain parallel machines (e.g CM-5) use a special network when all the processors are participating in the communication. The cost of global communication is lower in such a case.

6.4 Deterministic Unbalanced Parallel Divide and Conquer

Subdivision of tasks in certain divide and conquer algorithms result in subtasks of different sizes. The subdivision at each level remains unchanged for a given input and hence the algorithms are deterministic in nature. Using Concatenated parallelism, the work associated with the portions of subtasks on a processor may vary as a result of such subdivisions, but

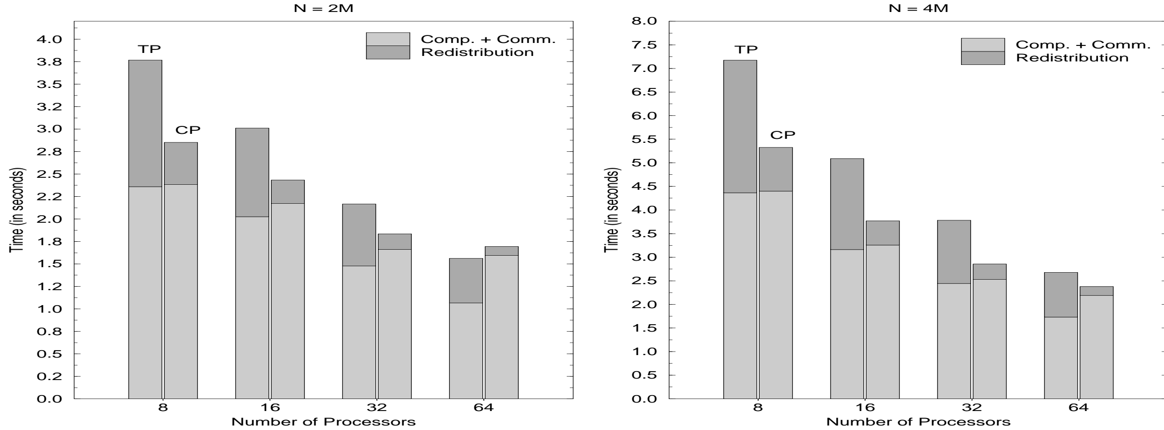


Figure 6.4: Time for tree construction up to $\log p$ levels divided into computation time for splitting the points plus the associated communication time and cost of redistribution for N = 2M and 4M on 8, 16, 32 and 64 processors. (TP: Task Parallel, CP: Concatenated Parallel)

each processor still works on $\frac{N}{p}$ elements. This method provides good load balancing for unbalanced parallel divide and conquer.

The number of points belonging to a subset may be highly imbalanced. Consider a problem of size N at some stage of the subdivision process. Suppose the problem is divided into two subproblems of sizes c and $N - c$, where c is a constant. Note that this definition will not subdivide a problem of size c further. This will lead to the worst-case analysis as the problem size is reduced by a constant. There is only one task to be solved at each stage. Concatenated Parallelism works the same as Data Parallelism in this worst case. A task is subdivided into two tasks at each stage. Let K be the size of a task after which it would not be subdivided. Subdivision of tasks is continued till a stage where all tasks have a size at least K. The value of K is set appropriately so that there are at least p subtasks and these can be allocated to processors to provide adequate load balance. A choice of $K = \frac{N}{p}$ is good because it guarantees that no processor will have more than $\frac{2N}{p}$ elements. At a level i, all the p processors are working on a problem of size $N - ic$. Choosing values of $f(N,p)$ as $O(\frac{N}{p})$ and $g(N,p)$ as $O((t_s + t_w)\log p)$ the running time of a Concatenated Parallel unbalanced divide and conquer is $O(\frac{N^2}{p^2}) + \sum_{i=0}^{\lceil \frac{N-K}{c} \rceil - 1} \frac{N-ic}{p} + (t_s + t_w)\log p$. Redistribution cost is $O(t_s p + t_w \frac{N}{p})$ on a hypercube and $O(t_s p + t_w \frac{N}{\sqrt{p}})$ on a mesh. The time for unbalanced divide and conquer using Concatenated Parallelism is $O(\frac{N^2}{p} + t_s(p + \frac{N-K}{c}\log p) + t_w(\frac{N-K}{c}\log p + \frac{N}{p}))$ on a hypercube and $O(\frac{N^2}{p} + t_s(p + \frac{N-K}{c}\log p) + t_w(\frac{N-K}{c}\log p + \frac{N}{\sqrt{p}}))$ on a mesh.

Task Parallelism can either partition the processors into two equal subgroups and allocate each to a subtask, or processor allocation to subgroups can be done proportional to subtask sizes. Unbalanced subdivisions lead to idling of processors as some processors have more work to do than others in the case where half of the processors are allocated to a subgroup. The earlier the imbalance occurs while subdividing, the

worse the performance will be because larger subgroups of processors are allocated during the earlier levels. Assume that a problem of size N is subdivided into subproblems of sizes c and $N-c$, represented by the following recurrence relation $T(N,p) = T(N-c, \frac{p}{2}) + f(N,p) + g(N,p) + h(N,p)$. The redistribution cost at each level can be bounded by an all-to-all communication using the transportation primitive. Substituting the generic values for $f(N,p)$ and $g(N,p)$ defined earlier, we obtain a running time to create p subtasks as $\sum_{i=0}^{\log p-1}(N-ic)/\frac{p}{2^i} + (t_s+t_w)\log\frac{p}{2^i} + t_s\frac{p}{2^i} + t_w(N-ic)/\frac{p}{2^i}$ on a hypercube and $\sum_{i=0}^{\log p-1}(N-ic)/\frac{p}{2^i} + (t_s+t_w)\log\frac{p}{2^i} + t_s\sqrt{\frac{p}{2^i}} + t_w(N-ic)/\sqrt{\frac{p}{2^i}}$ on a mesh. After $\log p$ iterations, one processor contains a task of size $O(N - c\log p)$. The running time for deterministic unbalanced divide and conquer algorithms using Task Parallelism is then $O((N - c\log p)^2 + t_s(p + \log^2 p) + t_w N)$ on a hypercube and $O((N - c\log p)^2 + t_s(\sqrt{p} + \log^2 p) + t_w N)$ on a mesh.

The computation cost using Task Parallelism is $O((N - c\log p)^2)$. This is reduced to $O(\frac{N^2}{p})$ using Concatenated Parallelism. Processor partitioning proportional to subproblem sizes for Task Parallelism would lead to allocating a processor to a subproblem of size c. This would result in $O((N - cp)^2)$ computation, not much better than the other case of Task Parallelism. Concatenated Parallelism provides better load balance than both kinds of Task Parallelism. Redistribution cost is reduced from $O(N)$ to $O(\frac{N}{p})$ on a hypercube, and to $O(\frac{N}{\sqrt{p}})$ on a mesh by using Concatenated Parallelism.

Algorithms for building quadtrees and finding the convex hull using the quickhull technique follow the unbalanced divide and conquer paradigm. We describe both these algorithms below.

6.4.1 Applications - Quadtrees and QuickHull

Consider building a quadtree for a set of points S in a $R \times R$ planar space [Sam84]. Four partitions each of size $\frac{R}{2} \times \frac{R}{2}$ are created using the median coordinates of the space. This process is repeated recursively until each point in the set belongs to a separate partition. At each stage of the subdivision the number of points belonging to each partition depends on the input data. It can potentially lead to unbalanced partitions and hence an unbalanced tree.

Given N points and p processors with $\frac{N}{p}$ points on each processor initially, each processor partitions its points into four subsets, each of the subsets corresponding to points lying in a quadrant of the given sample space. Each subset corresponds to a child node in the tree rooted with a node representing a subspace with dimensions $R \times R$ for some $m > 0$. Each of these subsets is partitioned recursively giving rise to 4^i subsets after the ith stage of the subdivision. An appropriate stage is chosen to redistribute subsets to individual processors to be solved sequentially.

In computational geometry, algorithms for finding the convex hull of points in space [PS85], like QuickHull, follow the divide and conquer approach of quicksort. In this case the subpartitions at any stage of partitioning are not guaranteed to be balanced as they depend on the input data. It follows that, given n points, the sequential algorithm for quickhull takes $O(n^2)$ worst case time, the average case run time being $O(n\log n)$.

The convex hull of a set S containing N points is the smallest convex set containing

S. It is represented by a polygonal chain [1] which contains the points on the convex hull. The Quickhull algorithm partitions S into two subsets, each of which computes a polygonal chain whose concatenation gives the convex hull polygon. The initial partition is determined by the line passing through l and r, the two points with the minimum and the maximum x-coordinates. Let $S^{(1)}$ be the subset of the points on or above the line lr and let $S^{(2)}$ be the subset of points below lr. A subset $S^{(k)}$ is processed in the following manner: A point $h \in S^{(k)}$ is determined such that the triangle (hlr) has the maximum area among all the triangles (plr), $p \in S^{(k)}$. h is a point on the convex hull. The next step is to construct two lines, one directed from l to h, $(\overline{lh})$, and another directed from h to r, $(\overline{hr})$. Points belonging to $S^{(k)}$ are tested with respect to these two lines. Clearly, points in the triangle (lrh) are interior points and have to be discarded. Points not to the left of $\overline{hr}$ but lying on or to the left of $\overline{lh}$ form a set $S^{(k,1)}$. $S^{(k,2)}$ is similarly formed by the points not to the left of $\overline{lh}$ but on or to the left of $\overline{hr}$. This process is repeated recursively on the newly formed subsets. A parallel algorithm for quickhull will start by finding the points with minimum and maximum x coordinates on all processors and performing a Combine operation to find the global minimum, min_x,and the global maximum max_x. These two points are on the hull as they are at extremities of the set. Each processor finds the point h, that gives the maximum area triangle with the line passing through min_x and max_x. Another Combine operation finds the maximum, h_{max}, among all the h points. h_{max} is a point on the hull. Points on the right of $\overline{h_{max}max_x}$ and to the left of $\overline{min_x h_{max}}$ form a subpartition. Another subpartition is defined for the points on the right of $\overline{min_x h_{max}}$ but lying to the left of $\overline{h_{max}max_x}$. Each of these subsets is solved in parallel recursively on all p processors, creating 2^i subsets after i steps of the algorithm. An appropriate stage is chosen for redistribution of subsets to individual processors, which apply the sequential algorithm to each subset separately.

6.4.2 Experimental Results

We present results for the quickhull algorithm on the CM-5. A concatenated parallel quickhull is compared with a task parallel quickhull implementation in Figure 6.5. The convex hull is constructed for 128K, 512K and 2M random points on 4, 8, 16, 32, 64 and 128 processors. Concatenated parallel algorithm clearly performs much better than a task parallel quickhull. We observe from the results that the gains of Concatenated Parallelism over Task Parallelism are diminishing with the increase in p. A smaller value of $\frac{N}{p}$ results in lesser load imbalance for Task Parallelism, a factor where Concatenated Parallelism gains the most. Redistribution costs are also lower and the gains from lowering this cost are also lower. These observations are endorsed by the analysis of both the methods in the previous section.

Similar results on experiments with building of quadtrees lead us to the conclusion that Concatenated Parallelism parallelizes well for deterministic unbalanced divide and conquer methods.

[1] A *chain* is a planar straight-line graph with vertex set $u_1, u_2, \ldots, u_p$ and edge set (u_i, u_{i+1}): $(1 \leq i \leq p-1)$

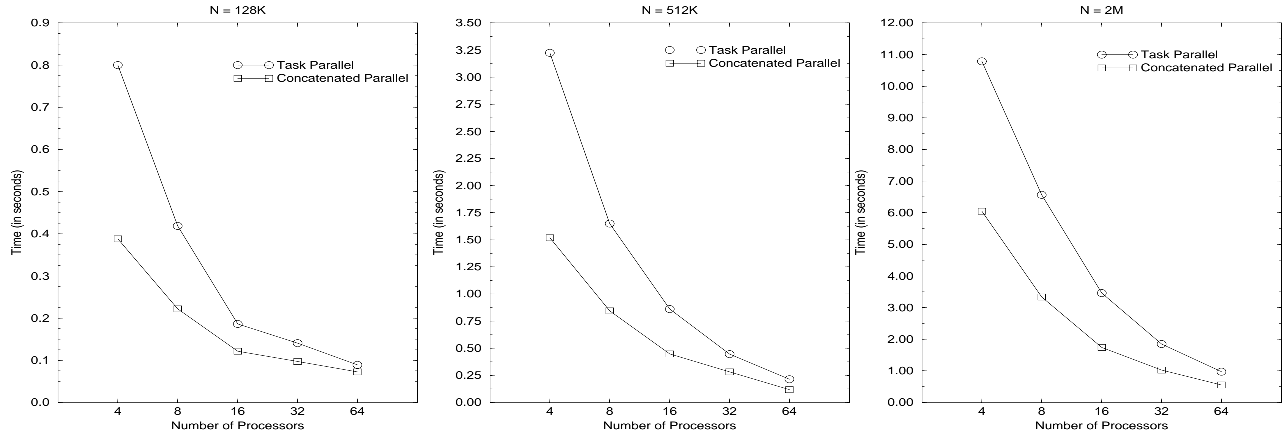


Figure 6.5: Comparison of Task Parallelism and Concatenated Parallelism for unbalanced divide and conquer in quickhull for N = 128K, 512K and 2M random data.

6.5 Randomized Parallel Divide and Conquer

The sizes of tasks after subdivision into two subtasks can be considered as random variables in some divide and conquer algorithms. Randomized quicksort is such an application. The subdivision process does not guarantee balanced partitions. However, something can be said about the expected value of the subtasks and hence about the number of iterations required to reduce the problem size to a specified level so that redistribution can be done. In this case the subtask sizes are a random variables. For the same input we could get different subtask sizes, because the criteria for subdivision is random and each element in the set is equally likely to be picked. In cases of unbalanced partitioning, Concatenated Parallelism will perform better as seen in the previous section.

In a randomized divide and conquer tree, the sizes of the subtasks of a given task depend on random choices made in the algorithm. We again limit our attention to the case where each task divides into two subtasks and sum of the sizes of the subtasks is the same as the size of the parent task. Suppose a task of size N is split into two subtasks. The sizes of the subtasks are given by x and $N - x$, where x is a random variable that can take any value between 1 and N. There is a probability associated with x assuming each of the allowable values. The probability distribution is often uniform. For example in quicksort, a random element from a given array is picked as a pivot and used to partition the array into two subarrays.

For randomized divide and conquer trees with uniform distributions and for which the cost of dividing a set is linear in the size of the set, a sequential analysis similar to quicksort shows that the expected running time is $O(N \log N)$, even though the worst-case run time is $O(N^2)$. However, there is a severe drawback to using Task Parallelism with equal subdivision of processors for randomized trees. Suppose that the subtask sizes when the root of the tree is subdivided are extremely unbalanced. This does not affect the expected running time of the subtasks in the sequential algorithm. However, half the processors are committed to a small subtask in Task Parallelism and the effect of this allocation will have a significant effect on the running times of all the subtasks in the subtree of the larger subtask.

For uniform distributions, the expected size of a subtask at level i of the tree is $\frac{N}{2^i}$. However, the variance in the sizes of the subtasks at level i of the tree decreases with increase in i. When a task of size N is split into two subtasks, the variance in the sizes of the subtasks can be shown to be $O(N)$, of the same order as the expected size. Concatenated Parallelism exploits this by advancing on the tree, level by level in parallel using all the processors until a balanced distribution of the subtasks to individual processors is possible. Since the variance decreases with levels, one can expect the performance of Concatenated Parallelism on randomized trees with uniform distribution to be similar to its performance on deterministic balanced trees.

6.5.1 Applications - Quicksort

Consider sorting a set S of N numbers in ascending order. Pick an element x from S and treat it as a pivot to partition S into two subsets S_1. containing elements smaller than or equal to x, and S_2, containing the remaining elements. This process is recursively applied to S_1 and S_2 to get the sorted order for S.

Assume a set S containing N elements divided equally among p processors. To parallelize quicksort on p processors assume that each processor contains $\frac{N}{p}$ elements to begin with. Pick a pivot at random from any of the p processors and broadcast it to all others. Each processor subdivides their elements using this pivot. This process is repeated recursively for each of the two subsets of elements. After i such subdivisions there are 2^i subsets to work with. An appropriate stage is chosen to redistribute these subsets to processors which apply the sequential quicksort algorithm to the sets allocated on them.

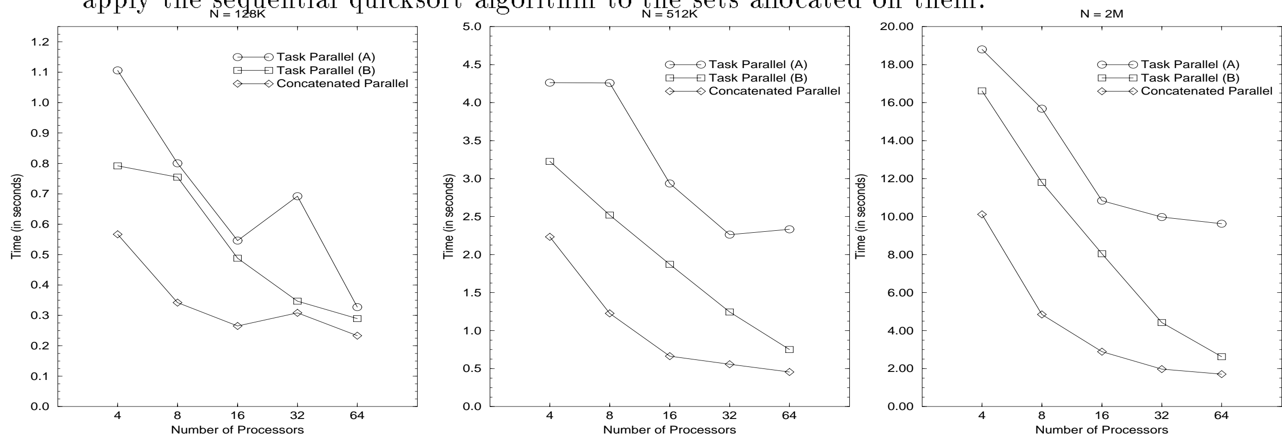


Figure 6.6: Comparison of Task Parallelism allocating half the processors to a subproblem (A), Task Parallelism with processor allocation proportional to subproblem sizes (B) and Concatenated Parallelism for randomized divide and conquer in quicksort for N = 128K, 512K and 2M random data.

6.5.2 Experimental Results

Figure 6.6 presents a comparison between a task parallel and a concatenated parallel implementation of quicksort. We report results of experiments on sorting 128K, 512K and 2M

random floating point numbers using 4, 8, 16, 32 and 64 processors. Task Parallelism with processor allocation proportional to the problem size [Task Parallelism (B)] performs better than Task Parallelism where half the processors are allocated to each subproblem [Task Parallelism (A)]. Approach (A) has higher imbalance due to which local sorting becomes a significant factor since some processors might have many more elements than others. Concatenated Parallelism has a lower running time than both the other methods. Randomized partitioning strategies can result in unbalanced partitions at each level leading to load imbalance and idling of processors. Load balancing is one advantage that Concatenated Parallelism offers to parallelize randomized divide and conquer methods. For Task Parallelism, processor allocation proportional to subproblem sizes may not always be possible due to topological considerations, or whenever possible may have higher overheads.

6.6 Redistribution Strategies

Subdivision of tasks into subtasks should be stopped as soon as the remaining subtasks can be distributed to individual processors. Redistribution of subtasks to processors occurs at a level when there are at least p subtasks and the size of each has reached a prespecified value so that they can be distributed to processors to be solved sequentially. This constant value determines the "grain size" of the largest task that we have in a pool of tasks ready for redistribution. Redistribution can potentially be done when the cost of subdividing a problem is more than the cost of redistribution. Consider an example where $\frac{\log p}{2}$ iterations of the subdivision have been performed and $\sqrt{p}$ subtasks created. If it is the case that partitioning costs at this stage are higher than redistribution costs then the $\sqrt{p}$ tasks can be redistributed to subpools of processors each of size $\frac{p}{2^m}$, $0 < m \leq \log p$. However, redistribution should only be done when each subgroup of processors can be allocated approximately equal work.

Using $\frac{N}{p}$ as the size of the largest subtask before redistribution this will ensure that no processor gets tasks whose sizes sum up to more than $\frac{2N}{p}$. A load balancing algorithm is used to allocate the subtasks to processors such that each processor gets nearly equal amount of work. There are various load balancing strategies that can be used in this case. We describe two techniques below which are illustrated in Figure 6.7.

1. **Order preserving combinations** – At a stage k, when redistribution is to be performed, order the list of 2^k (0 to $2^k - 1$) subtasks by using their indices in the list. The average work associated with a task on a processor should be $\frac{2^m N}{p}, 0 < m \leq \log p$, for a processor pool of size $\frac{p}{2^m}$. Beginning with the leftmost task entry in the ordered list, we can scan the task list from left to right allocating a task to a processor(pool) until the total work allocated to a processor (pool) is no more than the average value. However, this technique does not lead to the optimal load balance because we can only guarantee that the total work on a processor after the redistribution will not be more than twice the average work. This technique preserves ordering of the subproblems which might be essential in some applications where data ordering is important.

2. **Relative order combinations** – An idea similar to the one used for modified order

maintaining load balance [AfAGR96c] can be used here. At stage k, the 2^k subtasks are sorted in increasing order by the work associated with them. Task allocation to a processor is guided by two pointers, one placed at the start and another at the end of the sorted list. Problems are combined by moving the left pointer to the right and the right pointer to the left, allocating tasks to processors till each processor contains at least the average work. This strategy helps in allocating tasks to processors with total size on each processor as close to average as possible. Larger tasks are combined with small tasks and towards the end smaller tasks are aggregated which would help in not exceeding the average value by much. However, it destroys the ordering of the subtasks which might be of importance in some applications. If this happens at an intermediate level of the partitioning the ordering of tasks can be regained by another redistribution step.

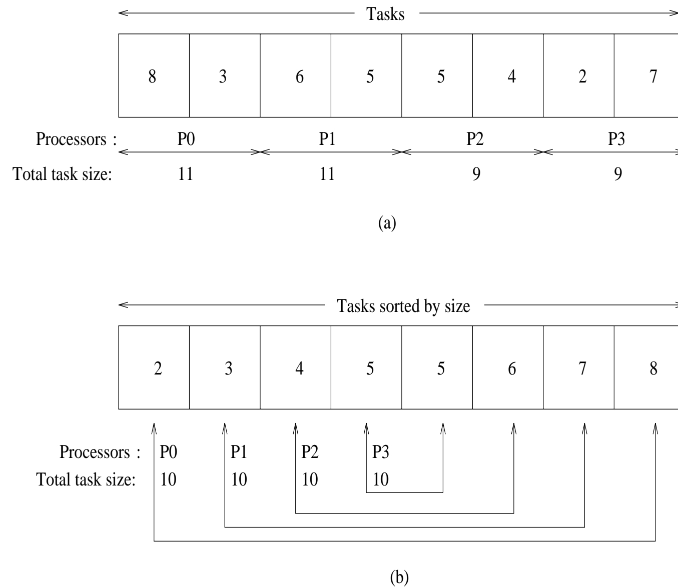


Figure 6.7: Illustration of load balancing during redistribution of tasks (a) Order preserving (b) Relative order combinations. Task allocation to processors is more balanced in (b).

6.7 Conclusions

We have proposed a new strategy called *Concatenated Parallelism* to efficiently parallelize applications resulting in divide and conquer trees. We compare this strategy to the two traditional approaches used in solving such problems – *Task Parallelism* and *Data Parallelism*. Task Parallelism causes significant redistribution of data at every level of the divide and conquer tree. However, it has the advantage that subtasks are uniformly distributed and allocated to smaller groups of processors. Data Parallelism avoids redistribution of data. But, it causes load imbalance and solves smaller sized subtasks using all processors, thus reducing practical efficiency. Concatenated Parallelism attempts to combine the advantages

of both the approaches. It avoids redistribution of data, and by combining the computation and communication of the subtasks, avoids load imbalance and grain-size problems. The minimum grain-size $\frac{N}{p}$, required for effective parallelization of an algorithm typically increases with the value of p. Data Parallelism decreases the grain-size and keeps the number of processors fixed. Concatenate Parallelism maintains the grain-size and keeps the number of processors fixed. However, Task Parallelism maintains the grain-size and decreases the number of processors, thus making the grain-size increasingly more effective. This is the only advantage of Task Parallelism over Concatenated Parallelism.

Concatenated Parallelism always yields better results than Data Parallelism irrespective of the nature of the divide and conquer tree. This is true irrespective of any parameter including the number of children per node, amount of work involved in dividing a task and the distribution of the sizes of the subtasks. It also holds true irrespective of the divide and conquer tree being balanced, unbalanced or randomized.

Concatenated Parallelism can outperform Task Parallelism only when the cost of redistribution of data is significant when compared to the cost of dividing the subtasks. This depends on such parameters as the topology of the parallel computer and the relative values of communication set-up times and unit transmission and computation costs. Certainly, for problems in which the cost of dividing the subtasks is linear, redistribution costs are significant and Concatenated Parallelism is beneficial. However, this may be true for practical values of problem sizes even when the subdivision cost is not linear but a function close to linear. We have shown several important problems for which Concatenated Parallelism has advantages – quicksort, quickhull, construction of quadtrees, octrees and multidimensional binary search trees.

The advantage of Concatenated Parallelism over Task Parallelism (when such an advantage exists) depends upon the nature of the divide and conquer tree. For balanced tree, the redistribution cost is reduced by a factor of $\log p$ on a hypercube while there is no advantages on a mesh. For unbalanced problems, the advantage gained depends upon the effect of imbalance. In the worst case, Task Parallelism fails to provide any speedup at all while Concatenated Parallelism always provides linear speedup. Task Parallelism is sensitive to imbalance where as imbalance has no effect on Concatenated Parallelism. For randomized divide and conquer trees, Concatenated Parallelism takes advantage of the fact that the variance of the sizes of the subtasks at the ith level of the divide and conquer tree reduces with increase in i. Since redistribution is performed only at the last stage, the subtask sizes allocated to individual processors are relatively uniform. Task Parallelism is affected due to high variance in subtasks sizes close to the root of the divide and conquer tree. This can be remedied by using an allocation of processors proportional to the individual subtask sizes but this strategy will not yield ideal results because allocation of processors can not be fractional and allocation that does not respect the topology often leads to congestion problems. It may also be unnatural and hard to program.

There are still several issues that remain to be investigated. There is considerable choice in redistribution strategies for Concatenated Parallelism. Redistribution should be done as soon as a balanced allocation to individual processors is possible, in order to extract the full benefits of Concatenated Parallelism. It would be interesting to investigate provably optimal redistribution strategies. Another interesting avenue to explore is a hybrid approach combining Concatenated Parallelism with Task Parallelism. One strategy is to continue

with Concatenated Parallelism until the communication cost at the next stage of subdivision exceeds redistribution cost. At this stage, processors should be grouped into as many groups of equal size as possible such that a fair redistribution can be done. After the redistribution, the same strategy is recursively applied to each group. Such a strategy can potentially obtain the full benefits of both Concatenated and Task parallelism by dynamically switching between the strategies during the execution of the divide and conquer algorithm. @

Chapter 7

Queries

Several issues need to be taken into account for the choice of an appropriate data structure when queries have to be answered on them. Some of these are summarized below:

- Which of point, polyline and polygon entities can be stored and retrieved efficiently.
- Which search operations can be performed - exact point match, range queries, polygon queries, nearest-neighbor queries etc.
- Is the data structure based on rectangular or non-rectangular sub-division of space, disjoint or non disjoint, regular or irregular.
- Are geometric primitives used for divisions or are arbitrary regular divisions used.
- Are non-zero sized entities, such as lines and polygons, sometimes partitioned or not.
- Is it possible to insert, delete and update entries while the structure remains balanced ?
- Suitability of the data structure for secondary storage. Whether I/O bottleneck can be resolved by efficient distribution or parallel I/O can be used.
- Are geometric primitives stored inside the data structure or are they indexed ?
- What is the worst case performance for building and querying the data structure ? average case ?
- How simple or complicated are the data structure and its algorithms.
- Is input easy or hard to generate on multiple processors ?

The need for insertion and deletion of elements in data structures leads to the choice of either a static, half dynamic or a dynamic data structure. Static data structures are built once for a fixed set of points. Half dynamic data structures allow for insertions and dynamic data structures allow both insertions and deletions. The important criteria in such a choice is for reasonable update time, amount of storage required and the query time for the important queries for that application. The following section gives a brief description of some sample queries that have been identified in the framework for hierarchical applications described in previous chapters.

7.1 Sample spatial queries

We describe the sample spatial queries that are required for applications surveyed in this report. Figure 7.1 illustrates some of these queries.

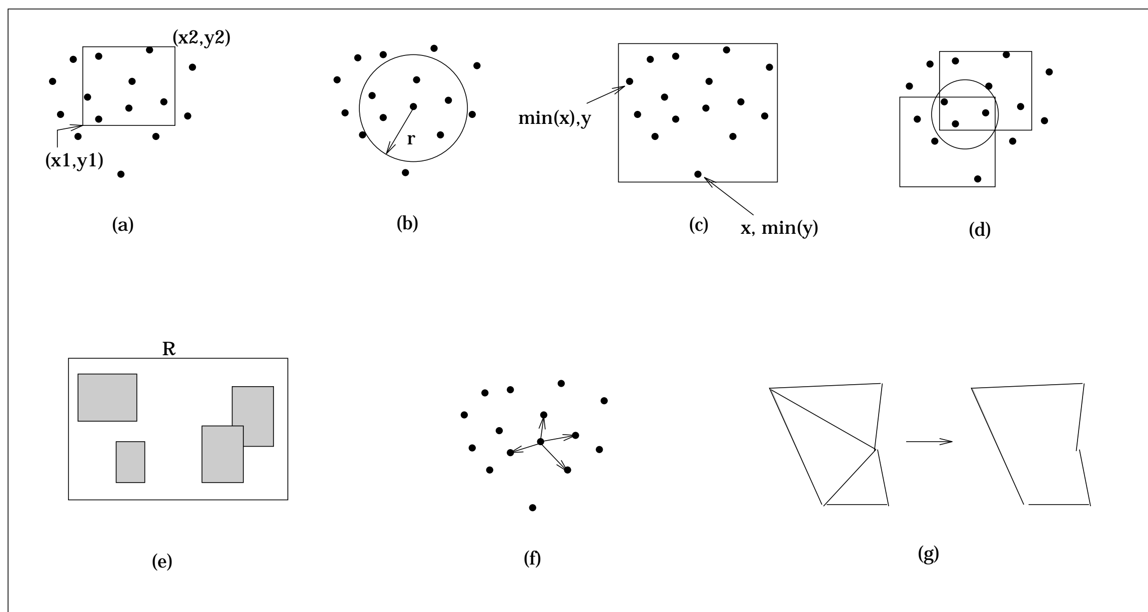


Figure 7.1: (a) Range query (b) Circular region query (c) min(max) query in a range (d) intersection query (e) rectangle containment query (f) 4-nearest neighbors query (g) polygonization

7.1.1 Point Location

Given a set of N points in k dimensions, $A = \{(d_{1i}, d_{2i} \dots d_{ki}) | 0 < i < N\}$, a point location query will be to find an instance of a point $(d_{1j}, d_{2j}, \dots d_{kj})$. Also this can be extended to *partial match* queries, where we are interested in only a subset of the k dimensions. This involves point data structures storing the indices of the point.

7.1.2 Range Query

Given a rectangular region $(p_1, p_2, \dots p_k) \times (q_1, q_2, \dots q_k)$, find all points that lie in the hyperspace defined by the domain space. This query is defined for point data here. This query has extensions into different data types like line, polygon etc.

7.1.3 Circular region query

Given a point (x, y) in 2D-space, find all the points that lie in a radius $r > 0$ from this point.

7.1.4 Minimum/Maximum in a range

Find the points with minimum(maximum) x or y coordinates in a given range $(x_1, x_2) \times (y_1, y_2)$.

7.1.5 Rectangle intersection

Given multiple rectangular regions, find all the points in their intersection.

7.1.6 Rectangle containment

Given a set of rectangles and a rectangle R, find all the rectangles enclosed in R.

7.1.7 k-nearest neighbors

Find k nearest neighboring objects to a given object. An object can be either a point, line or a polygon.

7.1.8 Parallelopiped region query

This is a 3D region query, where an image plane and a volume data is given. We want to find out the region intersected by the rays projected through the 2D image plane into the 3D volume, possibly the image plane is at an angle θ to the volume space.

7.1.9 Polygonization

Determine all closed polygons formed by a collection of planar line segments.

7.1.10 Spatial Joins

Given two data sets, an element from one may be joined with an element from the other if they cover some part of the space that is identical.

Chapter 8

Primitives, Language and Run-time Support

8.1 Primitives required

We describe some of the primitives needed by each of the applications we have studied. The computational phases for any application using treecodes are as follows.

1. Creating a sparse representation, a tree, from dense representation of data.
2. Data partitioning maintaining locality of reference (static partitioning).
3. Retrieving *locally essential data* for computation.
4. Dynamic (and incremental) load balancing to adapt to changes in processor computation load.
5. Incremental updation of locally essential data by a processor to reflect new interactions.

8.1.1 Creating a tree

The primitive *Make_tree()* is used to construct a global representation of data on processors. This can construct the appropriate tree suited for the application. Each processor needs to create a tree representation of the data it contains. This is done by *Create_local_tree()*, which takes dense representation of data on a processor and creates a hierarchical tree by recursive subdivision of space such that each node satisfies a particular constraint.

8.1.2 Data distribution

Data distribution over P processors has to be equitable for load balance. This is achieved by the primitive *Partition_data()*. Data distribution needs to ensure spatial coherence to reduce interprocessor communication. This can be specified by the type of the distribution an application needs. Regular grid distribution and Peano-Hilbert spatial ordering are some examples. There is a portion of the tree at higher levels with incomplete information

about its children. This is completed in the next phase, where higher and more abstract parts of the tree are constructed by exchanging relevant information amongst processors. Each processor now has pointers to all the data, local pointers to the data it owns and remote to data on other processors. Data exchanges can use the primitives *Send_data()* and *Get_data()*.

8.1.3 Fetch locally essential data

Each processor performs computation for the data it owns. Typical interaction calculations require off-processor data that is not currently in the local memory. A prefetch stage of obtaining all off-processor data is the gathering the locally essential data. This is done by the primitive *Build_locally_essential_tree()*. By looking at the geometry of the problem (coordinates of bodies in N-body and voxel coordinates for a ray) each processor can figure out the level of interaction it needs to perform. This step enables each processor to then go and prefetch the needed data, so that computation can proceed without hindrance from communication.

8.1.4 Incremental updates

Incremental_update_of_tree() changes the current tree to reflect the new position of bodies. Using the insight that bodies change positions gradually and not drastically from one iteration to another, most movement of bodies will be to adjacent nodes in the tree. Each processor uses the array of old and new coordinates to reflect the change in the tree. This in turn is used to incrementally adjust the tree. *Increment_locally_essential_data()* is a primitive that updates the locally essential data for each processor as access pattern change due to data assignment to processors. Using a sender oriented protocol, each processor can calculate the data it needs to send to the receiver.

8.1.5 Load Balancing

This movement of bodies changes the load characteristics on each processor. This can be adjusted by using the next primitive, *Dynamic_load_balance()*. Once the tree has been adjusted for the new positions of bodies, the previous run characteristics are used to approximate the load on each processor. The primitive *Peano_hilbert_remapping()* is used to remap the spatial distribution of rays to processors in the Volume rendering application.

8.2 Usage of primitives for Volume Rendering

In this section we discuss how the primitives defined above are used for Volume Rendering. Volume rendering uses hierarchical spatial enumeration of volume to optimize ray traced composition of each pixel. Rays are fired for each pixel that traverse the volume, compositing opacity and color for each slice it passes through. The volume data is represented as a pyramid of hierarchy of volumes. When a ray is traced, it traverses through larger volumes in areas of low opacity and through smaller ones in areas of higher detail. Hence an adaptive ray tracing optimization can be performed quite easily. The primitive *Make_tree()* is used to construct a sparse representation of the volume data which initially is replicated

on all processors. Eventually a parallel distribution of the volume among processors will introduce a parallel tree building algorithm. Rays can be distributed to processors by using the Peano-Hilbert space filling curve which is distributed among processors by doing a prefix scan. This is done in *Partition_data()*. Clearly, portions of volume data have to be fetched from other processors for ray interactions. A prefetch phase makes all the data locally available. The primitive *Build_locally_essential_tree()* does that for volume rendering. The volume data does not change from one iteration to another. What may change though, is the viewing angle. This means the ray now interacts with some additional volume data. An incremental phase of obtaining the new essential tree can be made using the *Increment_locally_essential_data()*. Using the insight that adjacent rays will need to interact with nearly the same volume data, and rays in subsequent frames will trace mostly the same data (Frame coherence), the previous two steps are not very expensive. The final step is to incrementally balance the load among processors when adaptive ray termination is used. Different rays travel different distances in volume data and the initial partitioning may lead to load imbalance. The primitive *Peano_hilbert_remapping()* can accomplish this by adjusting the dividing lines in the Peano-Hilbert sequence to reflect the new load. Adjacent rays only need to be moved to maintain load balance.

8.3 HPF and treecodes

Three major areas of HPF are involved in the implementation of divide and conquer tree codes, as discussed in Section 2 of [For94]. We discuss each one of them in relation with the applications discussed in this report.

8.3.1 Enhanced Mapping

Partitioning of data becomes relevant in irregular problems. When work associated with different elements varies, naive partitioning schemes which allocate equal number of elements to processors might lead to unbalanced computational loads. Locality of data on a processor can reduce communication across processors needed to fetch the required element. Moreover, when dynamic data structures are to be maintained, the partitioning might require adjustment to reflect the changes in processor work loads. Some solutions to these problems have been proposed for irregular problems yet no comprehensive solution exists to address all of the above. Hierarchical problems have not been addressed, except some workarounds that are both inefficient and unelegant.

In hierarchical applications, locality maintaining partitioning will benefit from reduced communication costs, as local data accesses will be fine-grained and off-processor data accesses will be coarse-grained. Construction of k-d trees to $\log p$ levels on p processors, using recursion in parallel, can be seen as a partitioning scheme. $ALIGN$ and $DISTRIBUTE$ directives can be used to map nodes of trees to processors in HPF. Pointer based structures entail a lot of overhead in the run-time system to be partitioned and maintained across processors. A Task Parallel execution, as discussed previously, needs data to be remapped to smaller processor subsets at each step. It is not very clear how this can be done without defeating the compiler's analysis. In the divide and conquer paradigm of hierarchical application, redistribution also means communication in smaller processor subsets minimizing

congestion and maximizing parallelism.

An important optimization for improving communication performance in HPF is to break the execution into a sequence of phases. At the end of each phase, all data referenced by a processor is communicated to that processor, so that the following computation can occur without communication. This is akin to gathering "locally essential data".

8.3.2 Computation Control

Distribution directives allow the user to control the mapping of data elements to the memories of the underlying system. There is no support for mapping th computation to specific processors. Mapping an iteration of an INDEPENDENT loop to its "best" processor might be difficult in presence of indirect array accesses. In such cases allowing the user to specify the mapping of computation to processors can result in better load balance while reducing communication costs. EXECUTE_ON directive has been proposed in [For94] to specify the mappings of INDEPENDENT DO loops, FORALL constructs and statements, and other indexed array assignments.

8.3.3 Communication Optimization

For the applications discussed in this report it is not possible to reuse the communication pattern from iteration to iteration. However, a communication phase at the beginning can gather locally essential data at a processor before the execution of tree traversals. In the construction phase the *concatenated parallel* approach described in Chapter 5 reduces the communication due to data movement to a single round of communication.

8.4 Run time Support on distributed memory machines

For an efficient execution of parallel tree code algorithms, collective communication is required at each iteration. Some of these operations are discussed in Chapter 1. Data redistribution algorithms are required to partition the elements into distinct subsets such that each subset can be allocated to a subset of processors. This requires data movement into two distinct halves, one containing elements lower than a particular partitioning element and one with elements higher than it. Runtime support is required for such an operation to incorporate a redistribution directive into the language.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this report we have studied a class of problems that consists of highly structured computations on sets of subdomains that are coupled hierarchically. The computational relationship between the subdomain is known only at runtime, and may change between computation phases. Parallelization of these applications on distributed memory machines require exploitation of the hierarchical nature both for data distribution, computational load balance as well as maintaining locality to reduce communication overheads. By studying the computational structure of hierarchical applications we observe that sparse data structures like quadtrees, k-d trees are needed for algorithms to reduce data storage sizes as well as to gain asymptotic performance for manipulation and retrieval of data.

We have presented parallel algorithms for selection, an important operation for constructing balanced tree structures. Selection is used extensively in parallel construction of balanced k-d trees. Parallel algorithms for construction of k-d trees are also investigated. We have compared parallel techniques for data structure construction and introduced communication optimizations by reducing data movement. *Concatenated Parallelism* is presented as one such technique and it is shown that for this class of problems it performs better than task parallelism.

We have identified the common spatial queries that are necessary for data access in the computation loop of the applications described in this report. Parallel implementation of these queries on distributed memory machines is the topic of our ongoing research. Lastly we have touched briefly on the issue of primitives and language support for such applications. There are a lot of issues still to be investigated in the optimization of the queries to identify the subset of features that should be incorporated in a high performance language. Till then run-time support for these should be an easier alternative by providing language directives to achieve the desired effect. Redistribution of data and mapping it to subset of processors is an important scenario for divide and conquer algorithms which are the trademark of application using spatial data structures.

9.2 Future Work

The following are interesting areas to investigate in terms of parallel hierarchical methods.

- Application of spatial data structures to large scale databases for clustering data and optimizing disk accesses.
- Optimization techniques for complex queries, such as those involving aggregation and grouping.
- Techniques for supporting multidimensional queries where the data is organized into a "data cube" consisting of a quantity of interest broken into "dimensions" that encapsulate interesting information.
- Optimization techniques to perform parallel I/O for large database queries.
- Investigation of external memory algorithms involving multiple processors and multiple disks for spatial database queries.
- Repositories characterizing storage and management of both data and *metadata* - the information about the structure of the data pose new challenges. Repositories must maintain an evolving set of representations of the same or similar information. Versions, snapshots of an element evolving over time, and configurations, versioned collection of versions need to be maintained efficiently in such systems.

Bibliography

[AA91] D. Chaiken et. al A. Agarwal. The MIT Alewife Machine: A Large-Scale Distributed Memory Multiprocessor. Technical Report MIT/LCS TM-454, Massachusetts Institute of Technology, 1991.

[AfAGR96a] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka. Parallel construction of multi-dimensional binary search trees. *Proc. Intl. Conf. on Supercomputing*, pages 205–212, June 1996.

[AfAGR96b] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka. Practical algorithms for median finding on coarse grained machines. Technical Report SCCS-743, NPAC, Syracuse University, April 1996.

[AfAGR96c] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka. Practical parallel algorithms for selection on coarse-grained parallel computers. *Proc. Intl. Parallel Processing Symp.*, May 1996.

[App85] A. W. Appel. An efficient program for many-body simulation. In *SIAM Journal on Scientific and Statistical Computing*, volume 6, 1985.

[Aup93] L Aupperle. Hierarchical algorithms for illumination. PhD thesis, Princeton University, 1993.

[Bar90] J. Barnes. A modified tree code: Don't laugh; it runs. *Journal of Computational Physics*, 87, 1990.

[BB90] K. Belkhale and P. Banerjee. An approximate algorithm for the partitionable independent task scheduling problem. In *Proc. Intl. Conf. on Parallel Processing*, 1990.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative search. *Comm. of ACM*, 19:509–517, 1975.

[BFMP93] A. Berthomi, B.M. Ferreira, S. Maggs, and C.G. Plaxton. Sorting-based selection algorithms for hypercubic networks. In *Proc.* 7th *International Parallel Processing Symposium*, pages 89–95, 1993.

[BFP+72] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, and R.E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1972.

[BH86] J. Barnes and P. Hut. A hierarchical o(n log n) force calculation algorithm. *Nature*, 1986.

[BH92] B. R. Brooks and M. Hodoscek. Parallelization of charmm for mimd machines. *Chemical Design Automation News*, 7:16, 1992.

[BJ95] D. A. Bader and J. J'aJ'a. Practical parallel algorithms for dynamic data redistribution, median finding and selection. Technical Report CS-TR-3494, University of Maryland, July 1995.

[BK94] Eric A. Brewer and Bradley C. Kuszmaul. How to get good performance from the cm-5 data network. In *Proceedings of the 8th International Parallel Processing Symposium*, 1994.

[Ble90] G.E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge, Massachusetts, 1990.

[BNK92] A. Bar-No and S. Kipnis. Designing broadcasting algorithms for the postal model for message-passing systems. In *Proceedings of 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 13–22, 1992.

[BO84] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, pages 484–512, 1984.

[BR95] S. Bae and S. Ranka. Experimental evaluation of different message paradigms on the cm-5 for irregular problems. In *Frontiers*, 1995.

[CDY95] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the benefits of mixed data and task parallelism. Technical report, University of California, Berkeley, 1995.

[CFH+92] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and J. Saltz. Software support for irregular and loosely synchronous problems. In *Proc. of the Conf. on High Performance Computing for Flight Vehicles*, 1992.

[Cha91] S. Chatterjee. Compiling data-parallel programs for efficient execution of shared-memory multiprocessors. Technical Report CMU-CS-91-189, Carnegie Mellon University, 1991.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E.Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings of 4th ACM Symposium on Principles and Practices of Parallel Programming*, pages 1–12, 1993.

[Cyb89] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[DL87] Willian J. Dally and Chuck L.Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. on Computers*, 36(5):547–553, May 1987.

[ea89] W.J. Dally et. al. The j-machine: A fine-grain concurrent computer. *Proceedings of the IFIP Congress*, pages 1147–1153, August 1989.

[ea95] P. Beazley et. al. Parallel algorithms for short-range molecular dynamics. *World Scientific's Annual Reviews in Computational Physics*, 3, 1995.

[Ede93] D. Edelsohn. Hierarchical tree-structures as adaptive meshes. *International Journal of Modern Physics C*, Vol. 4, No. 5:909–917, 1993.

[FAG83] H. Fuchs, G.D. Abram, and E.D. Grant. Near real-time shaded display of rigid objects. *Computer Graphics*, Vol. 17, No. 3, July 1983.

[FHK+92] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M .Wu. Fortran d language specification. Technical report, High Performance FORTRAN Forum, January 1992.

[FJL+88] G.C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume vol. 1. Prentice Hall, Englewood Cliffs, N.J, 1988.

[For94] High Performance Fortran Forum. Hpf-2 scope of activities and motivating applications. Technical report, Center for Research in Parallel Computing, Rice University, November 1994.

[FR75] R.W. Floyd and R.L. Rivest. Expected time bounds for selection. *Comm. of ACM*, 18:165–172, 1975.

[FST92] A. Feldmann, J. Sgall, and S.H. Teng. Dynamic scheduling on parallel machines. *Foundations of Comp. Sc.*, pages 111–120, 1992.

[GP90] S.A. Green and D.J. Paddon. A highly flexible multiprocessor solution for ray tracing. *Visual Computer*, 6, No. 2:62–73, 1990.

[GR87] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73, 1987.

[GR95] S. Goil and S. Ranka. Dynamic load balancing for raytraced volume rendering on distributed memory machines. In *International Conference on High Performance Computing*, December 1995.

[Gut84] A. Guttman. R-trees:a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 47–57, 1984.

[HDS92] Y. Hwang, R. Das, and J. Saltz. A data-parallel implementation of molecular dynamics programs for distributed memory machines. Technical report, University of Maryland, 1992.

[HS94] E. Hoel and H. Samet. Algorithms for data-parallel spatial operations. Technical Report CS-TR-3230, University of Maryland, February 1994.

[JS94] B. Jawerth and W. Sweldens. An overview of wavelet base multiresolution analyses. *SIAM Review*, 36:377–412, September 1994.

[Kar89] R. Karia. Load balancing of parallel volume rendering with scattered decomposition. Technical report, Australian National University, Canberra, Australia, 1989.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings, 1994.

[Lei85] Charles Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34, No. 10, October 1985.

[Lev88] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, May 1988.

[Lev90a] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9, No. 3:245–261, July 1990.

[Lev90b] M. Levoy. Volume rendering by adaptive refinement. *Visual Computer*, 6, No. 2:2–7, 1990.

[Liu94] P. Liu. *The Parallel Implementation of N-body Algorithms*. PhD thesis, Rutgers University, 1994.

[LL94] P. Lacroute and M. Levoy. Fast volume rendering using a sheat-warp factorization of the viewing transformation. In *SIGGRAPH'94*, August 1994.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessey. The directory based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th Annual International Symposium on computer Architecture*, May 1990.

[LT94] W. Ludwig and P. Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *Proc. Symposium on Discrete Algorithms*, pages 167–176, 1994.

[MPHK93] K. Ma, J.S. Painter, C.D. Hansen, and M.F. Krogh. A distributed parallel algorithm for ray traced volume rendering. *Parallel Rendering Symposium*, October 1993.

[MPS92] C. Montani, R. Perego, and R. Scopigno. Parallel volume visualization on a hypercube architecture. In *Proceedings of the Boston Workshop on Volume Visualization*, October 1992.

[MQ89] S. McCormick and D. Quinlan. Asynchronous multilevel adaptive, methods for solving partial differential equations on multiprocessors : Performance results. *Parallel Computing*, 12, 1989.

[NK93] Lionel M. Ni and Philip K.McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26(2):62–76, February 1993.

[NL92] J. Nieh and M. Levoy. Volume rendering on scalable shared-memory mimd architectures. In *Proceedings of the Boston Workshop on Volume Visualization*, October 1992.

[PB95] M. Parashar and J. Browne. An infrastructure for parallel adaptive mesh-refinement techniques. Technical report, University of Texas, Austin, 1995.

[PS85] F. Preparata and I. Shamos. *Computational Geometry - An Introduction*. Springer-Verlag, 1985.

[RCY94] S. Rajasekharan, W. Chen, and S. Yooseph. *Unifying themes for network selection*. Springer-Verlag Lecture Notes in CS 834, 1994.

[RSA95] S. Ranka, R.V. Shankar, and K.A. Alsabti. Many-to-many communication with bounded traffic. In *Proc. Frontiers of Massively Parallel Computation*, 1995.

[RSB94] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A convex programming approach for exploiting data and functional parallelism on distributed memory multiprocessors. In *Proc. Intl. Conf. on Parallel Processing*, 1994.

[RWS88] S. Ranka, Y. Won, and S. Sahni. Programming a hypercube multicomputer. *IEEE Software*, pages 69–77, September 1988.

[Sam84] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16, No.2, June 1984.

[Sin93] J. P. Singh. *Parallel Hierarchical N-body Methods and thier Implications for Multiprocessors*. PhD thesis, Stanford University, 1993.

[SS92] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–370, 1992.

[SSOG93] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Proc. Principles and Practices of Parallel Programming*, pages 13–22, 1993.

[TWY92] J. Turek, J.L. Wolf, and P.S. Yu. Approximate algorithms for scheduling parallelizable tasks. In *Proc. Symposium on Parallel Algorithms and Architecture*, pages 323–332, 1992.

[vECGS92] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the ISCA '92*, May 1992.

[WS92] M. Warren and J. Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *Proceedings of Supercomputing*, 1992.

[WS93] M. Warren and J. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of Supercomputing*, 1993.

[ZC92] H. Zima and B. Chapman. Vienna fortran - a fortran language extension for distributed memory multiprocessors. *High Performance FORTRAN Forum*, 1992.