Primitives for Problems using Hierarchical Algorithms on Distributed Memory Machines

Sanjay Goil
School of Computer and Information Science and
Northeast Parallel Architectures Center
Syracuse University
Syracuse, NY, 13244-4100
sgoil@npac.syr.edu

Abstract

In scientific computing a class of problems has been identified that consists of highly structured computations on sets of subdomains that are coupled in a irregular manner. The computational relationship between the subdomain is known only at runtime, which can also change between computation phases. This can lead to load imbalance and unreasonable computation overheads on parallel machines. We present an outline of primitives that are required and the runtime support needed to ease the programming tasks involving dynamic computation structures. We study some applications that exhibit different computational structure but are classified under hierarchical methods and suggest primitives that bind them in a common paradigm. This paper describes the structure of the N-body simulation, Volume Rendering and Radiosity applications. The common primitives they require for an efficient implementation on a distributed memory parallel machine are elaborated.

1 Introduction

A class of problems has been discussed here, whose implementation includes the need for irregular data access patterns, irregular and adaptive communication, and adaptive load balancing. Most previous research has concentrated on parallelizing scientific computations with uniform structures whereas efficient parallelizations of dynamic and irregular problems have received much attention only recently. Many high-level languages like Fortran-D [3], Vienna-Fortran [12] support parallel operations on uniform arrays. Some runtime support for irregular structures is however available in PARTI [7], but these approaches take advantage of static data access and communication patterns. These do not provide efficient solutions for dynamically changing data distribution and communication patterns as are seen in the problems we are addressing here. On parallel machines available today the cost of communication is higher than the cost of computation. Irregular problem domains give rise to unpredictable and irregular patterns of communication. To exploit the performance of a parallel machine we need to enhance the computation to communication ratio.

We derive motivation for this work from Salmon's work on the N-Body problem [8]. He has implemented the Barnes-Hut algorithm on the NCUBE and the Intel Touchstone Delta. The various phases of the algorithm require immaculate control on the computation and communication functions. This makes the implementation hard to understand and port on different machines. We present here primitives that can be used by a programmer while hiding the pain in manipulating each byte of data on his own. These ideas have been extended to ray-traced Volume Rendering [6]. Hanrahan [2] has extended the hierarchical structure of the N-body algorithm to develop hierarchical algorithms for radiosity. We investigate the possible approaches that would lead to an efficient implementation on a distributed memory machine, using our primitives.

In the next section we discuss the hierarchical nature of the applications and the need for primitives that bind them in a common paradigm. A software system with runtime support for these primitives on a distributed memory machine can then solve any of these problems. Section 3 introduces the primitives. Section 4 discusses the relevance of these primitives with the Volume rendering application.

2 Applications using hierarchical algorithms

2.1 N-Body simulation

Computational methods to track the motions of bodies that interact with each other are classified as N-body methods. Their use in the areas of astrophysics, semiconductor device simulation, molecular dynamics, plasma physics has received much attention for a long time. The N-body problem computes the state (position and velocity) of N bodies at time T, given an initial state. The common approach is to iteratively calculate the solution 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 *N - 1* bodies. While this method is conceptually simple, vectorizes well, its $\Theta(N^2)$ arithmetic complexity rules it out for large-scale simulations involving millions of bodies.

A point in physical domain requires progressively less information at a lesser frequency from parts of the domain that are farther away from it. Using this fundamental insight Barnes and Hut [1] were the first to propose faster N-Body algorithms which runs in $O(NlogN)$ time. However, parallel implementations have been fairly recent by Salmon and Warren [10]. N-body simulations using adaptive tree data structures are referred to as *treecodes*. The Barnes-Hut algorithm begins by constructing an octree, called the BH-tree, by inserting the bodies into the cluster hierarchy one at a time. First an octree partition of a region in space is computed enclosing the set of bodies. The partition is computed recursively by dividing the original region into eight octants of equal volume until each undivided region contains exactly one body. 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 octree 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.

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. A 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.

On distributed memory machine the data is distributed across the processors. This distribution must be equitable so as to divide the work on the processors equally. 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. Data partitioning must preserve data locality. Typically, communication for fetching off-processor data is much more expensive than a local read. An inappropriate data mapping can increase communication costs and degrade overall performance by adding to the overhead. The two issues of load balancing and data locality can be contradictory, the partitioning must take both into account.

A distinguishing feature of the BH-tree is that it evolves continuously due to ongoing computation. The mapping must be dynamically updated so that it can 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 redone 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. Further, the movement of bodies requires dynamic data mapping.

2.2 Volume Rendering

Volume rendering is a technique for visualizing 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 a colored semitransparent volume. Rays are cast through the volume, obtaining the value of the volume data set and compositing these into the pixel's color and opacity. Since all voxels (volume elements) participate in the generation of each image, rendering time grows linearly with the size of the data set (object space). The principal advantages of these techniques over others are their superior image quality and the ability to generate images without explicitly defining surface geometry. Many datasets contain coherent regions of empty voxels. A voxel is defined as empty if its opacity is zero. Methods for

encoding coherence in volume data include octree hierarchical spatial enumeration, polygonal representation of bounding surfaces and octree representation of bounding surfaces. An optimization to improve performance is to ignore empty voxels while rendering. 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.

To encode volume data using hierarchical spatial enumeration, a hierarchy of volumes is used to create a pyramid. 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, if its value is 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 to advance the ray further on the next iteration than if we had remained at the lower level. However, if the cell being tested contains a one, we move down one level, entering whichever cell encloses the 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.

The main issues for message passing implementation of volume rendering are

1. Managing the naming, replication and fine-grained communication overhead in shared read-only scene data. Processors need to access scene (object space) with fairly unstructured access patterns. Replication of the scene database on every processor is ruled out as being non-scalable.

2. Load balancing on processors needs to be achieved. Prepartitioning the image and the object space intelligently to improve load balancing have been tried out [11]. We propose dividing the Peano-Hilbert spatial representation (Figure 1) of the image to preserve ray coherence and facilitate incremental load balancing [5].

3. Adaptive sampling to reduce computation time gives rise to synchronization management issues. Pixels are shared among neighboring samples regions since the corner pixel values are required for measuring image complexity.

We discuss these in Section 4 with the help of the primitives for a parallel implementation.

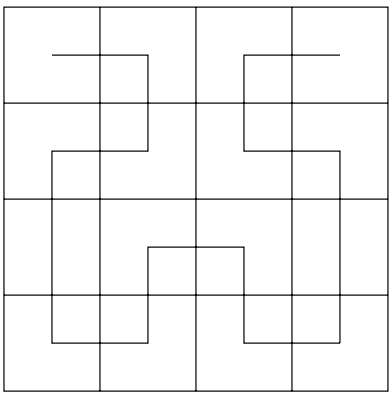

Figure 1: Peano Hilbert Space Filling Curve

2.3 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. 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 input to the algorithm is a set of polygons depicting the scene. These are inserted into a Binary Space Partitioning (BSP) tree [4] 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. 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. 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.

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. Each processor can 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. Alternatively, a single copy of the forest of quadtrees can be maintained in distributed form over the processing nodes. 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.

3 Discussion of Primitives required

We now describe some of the primitives needed by each of the applications we have studied. Table 1 lists the computation phases for the applications described above. The computational phases for any application using treecodes are as follows.

1. Creating a sparse representation, an octree, 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.

3.1 Creating an octree

The primitive *Make_octree()* is used to construct a global representation of data on processors. Each processor needs to create an octree 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.

3.2 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()*.

	Phase	Hierarchical Application		
		N-Body Simulation	Volume Rendering	Radiosity
1.	Tree Construction Tree Update	ORB tree for bodies. Add and delete nodes for load balancing and locally essential data.	MinMax Octree for Volume Data. Add and delete nodes from tree for locally essential data.	Forest of quadtrees for polygons. Add and delete nodes for polygon subdivision
2.	Data Distribution	Distribute bodies to processors	Distribute volume data to processors	Distribute Polygons to processors
3.	Locally essential data	Gather essential cells/bodies for acceleration calculations.	Gather essential voxel data for color/opacity calculation	Cannot be determined beforehand.
4.	Dynamic load balancing	Adjust ORB partition and update tree.	Partition Peano-Hilbert spatial representation of image space	Patch-patch interactions are incrementally redistributed.
5.	Data Locality	Incremental ORB partitioning provides data coherence.	Peano-Hilbert ordering retains spatial proximity for ray coherence.	Processing sibling patches retains data coherence.

Table 1: Computational phases of hierarchical applications

3.3 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 offprocessor 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.

3.4 Incremental updates

Incremental_update_of_octree() changes the current octree 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 octree. Each processor uses the array of old and new coordinates to reflect the change in the octree. This in turn is used to incrementally adjust the octree. *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.

3.5 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 octree 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.

4 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_octree()* 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.

5 Conclusions

We have enumerated the computational structure of some problems that use hierarchical algorithms. A discussion on the primitives for tree-codes, required for an effective solution on distributed memory machines, has been presented. We have identified a class of problems that require similar software support on parallel machines. Hierarchical methods were first used for N-body simulations and were later extended to other problems in Computer Graphics. However, not much effort has been made for a coherent software support system for these methods on parallel machines, notably on distributed memory systems. This work is in a very preliminary stage. Our effort is to develop an understanding of the primitives, develop a runtime support for them and finally provide language support for them to make hierarchical methods easier to program on parallel machines.

6 Acknowledgements

I wish to thank my advisor Prof. Sanjay Ranka for motivation, guidance and many helpful discussions. This work has been funded by National Science Foundation (NSF) under contract number 292-3-38393.

References

[1] Barnes, J. and Hut, P., A hierarchical O(N log N) force calculation algorithm, *Nature*, 324, 1986.

[2] Hanrahan, P., Salzman, D., Aupperle, L., A Rapid Hierarchical Radiosity Algorithm, *Computer Graphics* Vol. 25, No. 4, July 1991.

[3] Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C., Wu, M., Fortran D Language Specification, *High Performance FORTRAN Forum*, January 1992.

[4] Fuchs, H., Abram, G. D., Grant, E. D., Near Real-Time Shaded Display of Rigid Objects, *Computer Graphics*, Vol. 17, No. 3, July 1983.

[5] Goil, S., Software Support for problems using hierarchical algorithms on distributed memory machines, *Technical Report under preparation*, Syracuse University.

[6] Levoy, M., Efficient Ray Tracing of Volume Data, *ACM Transactions on Graphics, Vol. 9, No. 3*, pp 245-261, July 1990.

[7] Choudhary, A., Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Ranka, S., Saltz, J., Software Support for Irregular and Loosely Synchronous Problems, *Proceedings of the Conference on High Performance Computing for Flight Vehicles*, 1992.

[8] Salmon, J., Parallel Hierarchical B-body methods, *PhD thesis*, Caltech, 1990.

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

[10] Warren, M. and Salmon, J., Astrophysical N-body simulations using hierarchical tree data structures, *Proceedings of Supercomputing'92*, 1992.

[11] Green, S. A., Paddon, D. J., A Highly Flexible Multiprocessor Solution for Ray Tracing, *Visual Computer, Vol. 6, No. 2*, pp 62-73, 1990.

[12] Zima, H., Chapman, B., Vienna FORTRAN - A Fortran Language Extension for Distributed Memory Multiprocessors, *High Performance FORTRAN Forum*, 1992