2011

Extreme scale parallel NBody algorithm with event driven constraint based execution model

Chirag Dekate
Louisiana State University and Agricultural and Mechanical College

EXTREME SCALE PARALLEL NBODY ALGORITHM WITH EVENT DRIVEN CONSTRAINT BASED EXECUTION MODEL

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Chirag Dekate
B.S., Louisiana State University, 2002
M.S., Louisiana State University, 2004
May, 2011

Acknowledgments

This work would not be possible without the help and support of my current and former colleagues at the Center for Computation & Technology (Louisiana State University).

I would like to thank all my collaborators from Sandia National Lab, DARPA UHPC Xcaliber project team.

It is a pleasure to thank my advisory committee (Thomas Sterling, J. Ramanujam, Hartmut Kaiser, Jianhua Chen, Sitarama Iyengar, and Yitzak Ram) and the *ParalleX* team: Hartmut Kaiser for his valuable guidance in directing the HPX based implementation of N Body problem, Matthew Anderson for advice and assistance in getting algorithmic correctness, and Bryce Lelbach for his expert advice in C++ programming. Matt was also pivotal in helping me understand and utilize the stencil based framework used in this thesis to represent the Barnes Hut algorithm in the HPX version of the algorithm. Dr. Sterling provided diligent and meticulous guidance which was pivotal to accomplishing this thesis.

I would also like to thank all my collaborators, in particular the following (not in any particular order): Matthew Anderson, Hartmut Kaiser, Maciej Brodowicz, Steve Brandt, Dylan Stark, Alexandre Tabbal, Vinay Amatya, Timur Gilmanov, Daniel Kogler, Bryce Lelbach, and Alex Nagelberg.

This work was supported by funding from the DARPA, NSF, and by the Center for Computation & Technology at LSU. I would like to thank them for their continued efforts in funding path-breaking projects that enable transformational next generation research.

Finally, I would like to thank Shivangi for her constant encouragement and sacrifices for making this thesis possible. I would also like to thank my family for their support throughout this journey.

Table of Contents

List of Tables

List of Figures

Abstract

Traditional scientific applications such as Computational Fluid Dynamics, Partial Differential Equations based numerical methods (like Finite Difference Methods, Finite Element Methods) achieve sufficient efficiency on state of the art high performance computing systems and have been widely studied / implemented using conventional programming models. For emerging application domains such as Graph applications scalability and efficiency is significantly constrained by the conventional systems and their supporting programming models. Furthermore technology trends like multicore, manycore, heterogeneous system architectures are introducing new challenges and possibilities. Emerging technologies are requiring a rethinking of approaches to more effectively expose the underlying parallelism to the applications and the end-users. This thesis explores the space of effective parallel execution of ephemeral graphs that are dynamically generated. The standard particle based simulation, solved using the Barnes-Hut algorithm is chosen to exemplify the dynamic workloads. In this thesis the workloads are expressed using sequential execution semantics, a conventional parallel programming model - shared memory semantics and semantics of an innovative execution model designed for efficient scalable performance towards Exascale computing called ParalleX. The main outcomes of this research are parallel processing of dynamic ephemeral workloads, enabling dynamic load balancing during runtime, and using advanced semantics for exposing parallelism in scaling constrained applications.

Chapter 1
Introduction

1.1 Background

Large class of problems in physics and microbiology can be represented using a particle interaction method commonly known as N-Body methods and computational techniques based on these discretization methods. The science domains utilizing the particle interaction discretization model are limited by the number of particles that can be simulated and the time it takes to execute the computational techniques. Conventional practices have significantly advanced particle interaction based methodologies, however the combined ecosystem of emerging multicore based system architectures and conventional programming models are imposing grave challenges to the continued effectiveness of these methods. This research identifies and addresses these challenges through the hypothesis that emerging system architectures and extreme scale oriented runtime systems can dramatically improve the end-science.

1.1.1 State of High Performance Computing Systems and Programming Models

For several decades, high performance computing systems have been used to solve complex large scale problems across disciplines, often resulting in remarkable scientific breakthroughs. The scale and complexity of high performance computing systems has increased steadily over the last two decades from 59.8 Gigaflop/s in 1993 to 2.57 Petaflop/s in 2010. In terms of system architecture, Cluster based systems comprise more than 82% of the top 500 supercomputing systems [DMSS10] across the world, followed by Massively Parallel Processor systems which comprise almost 17% of the systems. In terms of programming models the Communicating Sequential Processes model [BHR84] is the predominant programming model used by the large scale applications, in the form of Message Passing Interface (MPI) [GLDS96, GL02], to utilize the terascale to petascale par-

allelism offered by the current generation HPC systems. On shared memory systems OpenMP [DM98] is the dominant model of programming. Emerging computer systems based on multicore architectures are creating new opportunities and challenges for High Performance Computing including extending parallelism from large scale supercomputing to common desktop environments. In the context of supercomputing, the multicore processors introduce parallelism within the node resulting in new hybrid programming models such as MPI + OpenMP, being developed to utilize the multi-level parallelism.

1.1.2 State of Large Scale Performance-Oriented Applications

The sustained growth in high performance computing systems, has directly impacted the scale of problems solved using these systems. Over the past 2 decades the peak performance of cutting-edge applications has steadily increased from 450 Megaflop/s [SC88]in 1987 to 2.33 Petaflop/s [RLV+10] in 2010. A majority of the applications that utilize parallel computers have data structures which are conducive to uniform load distribution (e.g. matrix-based), with regular predefined communication patterns, and determinate data accesses. The well defined computational requirements of traditional applications usually map directly on to the prevalent modular system architectures enabling relatively uniform load balancing and resource management using conventional programming models like MPI. Examples of traditional computations include finite difference methods, finite element methods, linear algebra based problems and iterative methods on static matrix structured domains.

1.1.3 Overview of The N-Body Problem

Applications based on graphs and tree data structures, rely on more dynamic, adaptive and irregular computations. This thesis explores an exemplar dynamic tree based application embodied by an N-Body simulation. Systems comprising many particles (N-Body problem) interacting through long-range forces have interested scientists for centuries. N-Body systems comprising 3 or more particles do not have a closed form solution, consequently iterative methods are used to approximate solutions for N-Body problem. The N-Body problem simulates evolution of n particles under

the influence of mutual pairwise interactions through forces such as gravitational pull or electrostatic forces. This thesis focuses on gravitational forces operating on the N-Body systems and the Barnes Hut algorithm to approximating the N-Body solution.

The simplest most accurate algorithm for evolving an N-Body system is using an exhaustive interaction method [PH10] where interactions of each particle with all the other particles in the system are computed. Such an approach while highly accurate provides $O(N^2)$ complexity, which for larger number of bodies (larger N) becomes an intractable problem. Over the last 3-4 decades, several methods designed to reduce the computational complexity while evolving the forces, albeit at an acceptable loss of accuracy, have been demonstrated. These include particle-in-cell [Daw83] (PIC) methods, Fast Multipole Methods (FMM) [CGR88, SL91], and hierarchical tree methods [Her87] among others. A comprehensive coverage of various hierarchical methods and the constraint space is described by Hernquist [Her87]. An overview of some of these methods is presented in Chapter 2

While several approaches to simulating N-Body systems exist, the Barnes Hut algorithm [BH86] is widely used in astrophysical simulations mainly due to its logarithmic computational complexity while generating of results that are within acceptable bounds of accuracy. In the Barnes Hut algorithm the particles are grouped using a hierarchy of cube structures using a recursive algorithm which subdivides the cubes until there is one particle per sub-cube. The Barnes Hut algorithm then uses the adaptive octree data structure to compute center of mass, force on each of the particles. The data structure organization is key to the resultant $O(NlogN)$ computational complexity of the algorithm.

1.1.4 Parallel N-Body Methods

There have been several successful efforts to parallelize the Barnes Hut tree code. Hillis and Barnes implemented [Hil87a] the tree code on Connection Machine-2 [Hil87b] with 65,536 processors. Salmon introduced [Sal91] an innovative domain decomposition method called orthogonal recursive bisection (ORB) that facilitates efficient partitioning of workload in a density agnostic manner

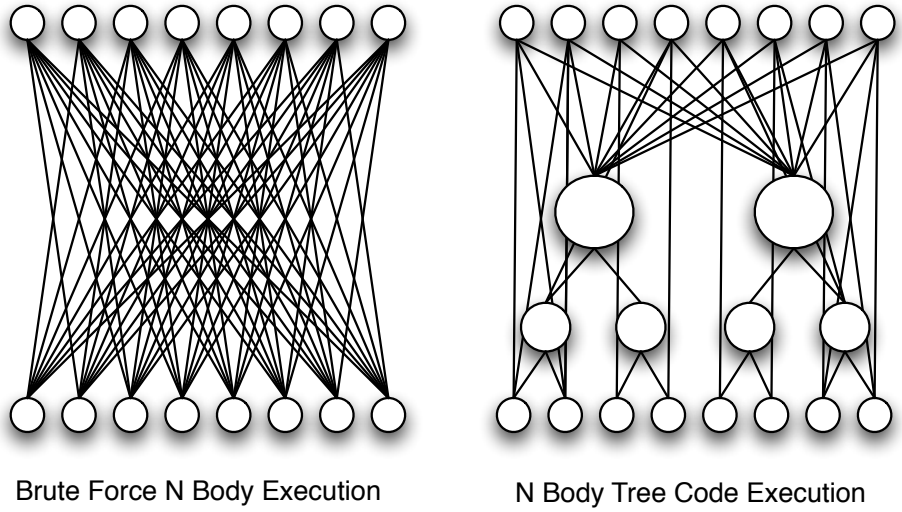


FIGURE 1.1: Execution Patterns of Bruteforce N-Body Problems and Treecode N-Body Problems

over a distributed memory machine. Salmon and Warren [WS93] introduced an algorithm based on parallel hashed octree which allows for efficient (O(1)) access to remote particle data consequently improving scalability and efficiency of the treecode.

P. Singh [Sin93] examined the architectural implications of hierarchical N-Body methods (Barnes Hut) and evaluated techniques to exploit temporal locality to achieve performance improvements. J. Dubinski implemented a modified Salmon / Warren code using the Message Passing Interface (MPI). Bhatt, Liu et, al. [LB94] implemented a MPI and C++ based parallel implementation of Salmon/Warren's improvements of the Barnes Hut algorithm. Makino and Hut have implemented [MHI07] a efficient version tree code on Grape-DR, an accelerator augmented supercomputer providing hardware support for N-Body simulations and enabling very high performance capabilities for problems in the particle based domain. Dinan et al. [DBL+10] have implemented the Barnes Hut algorithm using hybrid UPC-MPI approach targeting parallel implementation of treecode on distributed memory systems and using UPC [EGCSY03, EGS06] to manage the Partitioned Global Address Space [Yel07].

Alternative purpose specific system based solutions such has MTA [CFS99, ABH+03], Anton [SDS+09], and Grape DR [MHI07] have demonstrated that with sufficient system support for graph structures (in their case the hardware level) can provide significant performance and

scalability advantages for these applications. Most of the purpose-specific high performance computing system often achieve performance and scalability for graph applications at the cost of lack of support for standard applications and high price-performance cost.

1.2 Intellectual Problem

This section describes in detail the intellectual problem being addressed in this thesis. This section details the challenges involved in parallel processing of tree based N-Body systems, the goals of this thesis, the hypothesis and objectives of this thesis.

1.2.1 Challenges

This thesis confines the discussion of N-Body problems to tree code algorithmic methods proposed by Barnes Hut [BH86] and parallel tree code approaches, as these methods provide logarithmic computational complexity and the dynamic runtime characteristics which pose challenges to conventional high performance computing systems.

A detailed description of the N-Body problem is provided in Chapter 2. During the simulation the position of the particles change as a result of interaction with other particles, consequently the representative Barnes Hut tree changes dynamically during the computation. The dynamic tree based evolution of the system is atypical to conventional computational applications. High performance computing applications are dominated by simulations where the fundamental data structures are array/matrix that can exploit data locality for performance, through techniques such as row/column striping [dRBC93]. Graph/Tree data structures have non uniform data access patterns which prevent tree based applications from taking advantage of data locality and fully utilizing the parallelism offered by the underlying system. The emergence of multicore processor architecture while providing capabilities for significant intra-node parallelism pose new challenges. Traditional programming techniques when used in isolation fail to expose the underlying parallelism to the high performance computing applications since they cannot address the intra-node parallelism provided by multicore processor architecture. Conventional clusters comprise multicore nodes interconnected by a low latency high bandwidth network such as InfiniBand, and are

usually programmed using MPI. When conventional techniques such as message passing are used in isolation the intra-node parallelism offered by each of the multicore nodes is under utilized, primarily due to limitations of the implementations of MPI. Performance and scalability of graph applications are further affected by high performance computing system architectures that are more suitable for computations with static predictable data access patterns. Systems architectures such as clusters and MPPs have co-evolved with programming models and applications that provide high data locality to computations.

The fundamental challenges to improving performance and scalability of parallel N-Body simulations using the Barnes Hut algorithm, in a system context are:

- **Dynamic Load Balancing** The Barnes Hut algorithm generates a new unique tree for each iteration which is different than the tree used in the previous iterations, consequently conventional load balancing techniques such as striping are ineffective and results in poor load balancing.

- **Variable Workload** In addition to the varying Barnes Hut tree, a fundamental resource management problem encountered while simulating these systems is the variable per particle workload. That is each particle in the system will interact with varying number of particles in the system. Consequently predicting per particle workload in the N-Body simulations is difficult.

- **Data Driven Computation** The Barnes Hut tree algorithm like other graph computations is data driven. The computations performed by the Barnes Hut algorithm are dictated by the vertex and edge structure of the tree rather than directly expressing computations in code. This results in irregular communication patterns and irregular data access patterns limiting ability to parallelize using conventional techniques.

- **Data Locality** The irregular and unstructured computations in dynamic graphs result in poor data locality, consequently resulting in degraded performance on conventional systems which rely on exploitation of data locality for their performance.

1.2.2 Goal

The goal of this research is to enable much larger parallelism in reasonable time or to dramatically reduce execution time of fixed size problems by the application of extreme scale multicore systems to greatly advance science. The goal of this thesis is achieved by developing novel techniques that could improve performance and scalability of the N-Body application,and in general dynamic graph/tree based applications. In order to achieve scalable performance of the N-Body application, the ecosystem and the application would need dynamic characteristics, such as:

- **Goal 1:** The runtime system should be able to handle non-uniform workloads. This ability would allow better utilization of allocated processors and reduce starvation.

- **Goal 2:** The execution model should be able to handle multiple modalities of locality i.e. relocate computation to where the data is located and vice versa as optimally needed by application. This would allow for reduction in overhead where large amounts of data about the bodies would have otherwise move to remote computation localities.

- **Goal 3:** The application should be able to utilize asynchronous communication mechanisms to synchronize between computation. This ability would allow reducing contention for shared objects

1.2.3 Hypothesis

The hypothesis of this dissertation is: Dynamic data-driven semantics and execution techniques in both the application and the supporting ecosystem (runtime system, execution model) can revolutionize management of dynamic graph computations such as Barnes Hut N-Body problem, by dramatically improving scalability, efficiency, and performance in these class of applications. Fol-

lowing are the key hypotheses in relation to the problem area of Barnes Hut N-Body problem and the execution model.

- **Hypothesis 1 - Work Queue:** The key differentiating factor of Barnes Hut N-Body problem from the conventional applications is the varied number of interactions per particle. The resultant non-uniform workload is hard to accurately predict resulting in load balancing schemes which under-utilize allocated resources causing starvation (section 3.3.1). This dissertation attempts to solve this problem by using a work queue that holds all the workload and allows for migrating work where the execution resources are available.

- **Hypothesis 2 - Locality:** Another challenging aspect of the Barnes Hut N-Body problem is that the graph representation while providing algorithmic speedup, provides very poor data locality. Conventional parallelization approaches often rely on good locality semantics i.e. proximity of data dependencies for work currently being executed. The default mode in conventional parallel programming models is to move necessary data to the work and as the processor is treated as the valuable resource and hence the entire ecosystem is designed to keep the processor busy. The poor locality in graphs can be addressed by allowing migration of work to data and vice versa depending on which semantic provides lower overhead and faster performance. The key idea in implementing this technique is that by using such semantics, latency and overhead issues (section 3.3.1) experienced by the application can be reduced.

- **Hypothesis 3 - Asynchronous Control:** A key limiting factor to performance and scalability in conventional parallel systems is synchronous execution semantics such as *fork-join* in OpenMP. These semantics require that all associated computation to a tree-node being evaluated, complete before the tree-node begins execution. This results in poor system utilization for dynamic due to starvation. This dissertation proposes that using asynchronous control mechanisms such as futures, can allow for dynamically activating work when sufficient de-

pendent work units have completed computations. This could facilitate better utilization of the allocated resources and consequently larger scale of problems can be executed more efficiently.

1.2.4 Objectives

In order to test the proposed hypotheses and to satisfy the goals of the dissertation key objectives have been established

- **Objective 1:** To use an extreme-scale oriented execution model that can provide the necessary dynamic mechanisms such as global address space, work-queue execution, futures based synchronization mechanisms required to manage the dynamic graph problems.

- **Objective 2:** To use futures based representation of the Barnes Hut problem, such that the resultant particle interaction lists allow for asynchronous computations. Constraint based — declarative system can free run self adapt

- **Objective 3:** Utilize a work queue based model for execution which allows for efficient utilization of allocated resources and adaptive scheduling of workloads and hide latency.

1.3 Technical Strategy

The technical strategy of used to test the hypothesis involves 3 phases. Phase 1 involves with implementation on conventional architectures, systems and programming model to characterize Barnes Hut N-Body performance and scalability. Phase 1 is achieved by implementing the Barnes Hut N-Body problem using two main techniques namely C++ based sequential implementation and a conventional parallelization techniques using OpenMP. Phase 2 involves using the extreme-scale ParalleX execution model to represent the Barnes Hut N-Body problem to test our hypotheses. Phase 2 is achieved by redesigning the algorithm and adapting the implementation to use the HPX runtime system (an implementation of PX execution model). Phase 3 involves executing various experiments designed towards measuring fine grained aspects of the algorithm such as tree construction, center of mass and force calculations.

- **Sequential Implementation:** The sequential implementation of the Barnes Hut N-Body simulation provides algorithmic clarity, and representation of program logic using the standard C++ language. The results generated from the sequential implementation provide reference standard for validating results obtained from subsequent parallel implementations for the same data set and additionally characterize application performance on a single processor system.

- **OpenMP Implementation:** The parallel implementation utilizes the ubiquitous OpenMP parallel programming libraries that are provided by all latest compilers including GNU gcc/g++ and Intel compilers. OpenMP based parallelization is one of the most common approaches to harnessing concurrency in modern systems. The parallel implementation allows for performance characterization of the algorithms on conventional systems using standard parallelization techniques.

- **HPX Implementation:** The performance characterizations obtained from the above implementations are used to elicit scalability and performance limitations to Barnes Hut N-Body simulations. The hypothesis defined in this thesis is implemented using a reference performance implementation of ParalleX model (HPX). HPX provides several dynamic mechanisms that enable data-directed computation as required by dynamic graph applications. Details of the mechanisms supported by HPX are provided in Chapter 5. The HPX implementation is the key implementation contribution of this thesis, which seeks to address the limitations to current N-Body simulations.

1.4 Thesis Outline

Rest of the thesis is organized as follows:

Chapter 2 describes in detail various approaches to parallelizing N-Body and focuses the discussion on the Barnes Hut algorithmic approach. The Barnes Hut algorithmic approach is explored at a greater depth and constituent components of the algorithm are explored.

Chapter 3 outlines in detail the challenges for parallel Barnes Hut N-Body approaches while using conventional parallelization techniques.This outlines strategies to parallelize the Barnes Hut N-Body problem using conventional parallelism techniques. This chapter also outlines an ideal execution ecosystem (runtime system and algorithmic approaches) that would be required to improve the scalability and performance of the N-Body problem. This chapter highlights the key challenges affecting scalability and efficiency for high performance computing.

Chapter 4 details a novel execution model called ParalleX which address scalability and efficiency challenges posed by conventional system architectures and develops key mechanisms necessary to provide extreme scalability. Essential attributes of an execution model for scalable and efficient high performance computing are discussed followed by a detailed overview of ParalleX Execution model.

Chapter 5 details the execution ecosystem used in validating the hypothesis. Detailed architecture of the performance oriented implementation of the ParalleX execution model called HPX is presented in great detail. In-depth discussion of the algorithmic implementation of Barnes Hut N-Body algorithm using HPX is also presented.

Chapter 6 explores the experimental setup used in obtaining the results. Detailed analysis of the experiments conducted and results obtained are documented. The results obtained are discussed in detail with emphasis on how the measurements affect the hypothesis.

Chapter 7 reconciles the results with the hypothesis and derives key contributions of this thesis. This chapter also highlights the aspects of the applications and systems details that are related yet beyond the scope of the thesis. This chapter also details specifics on future work to be carried out beyond the dissertation.

Chapter 2
Tree Based N-Body Simulations

2.1 Introduction

Some important physical systems can be represented as computational simulations by developing a mathematical model of the phenomenon consisting of a finite number of discrete elements. These mathematical models are usually developed using partial differential equations and numerical techniques. Finite element methods [Cou43], finite difference methods [FW60] and spectral methods [GOS78] are used to recast the mathematical model into a discrete form. Alternatively the physical system can also be approximated using a finite set of "particles" or "bodies" which interact with each other and with externally applied forces (such as gravitational or van der Waals forces). Each of these bodies carry state information (such as their position and velocities) and their interaction allows the entire system to dynamically evolve. These types of "particles" or "bodies" based simulations are often referred to as N-Body simulations. For example, a simple N-Body simulation could comprise N bodies each of which holds state information such as position and velocity vector. Representative models of the force laws such as Newton's gravitational laws (for astrophysical simulations) or van der Waals forces (for molecular dynamics simulations) guide the interaction of the bodies. As the system dynamically evolves the state of the system is modified by particle interactions causing acceleration of the particles resulting in new position and velocities for each of the bodies. There are several ways in which the N-Body simulations can be represented. These methods vary in the nature of interactions, their performance scaling, data structures used to represent the computational space and many such critical factors. The following sections briefly describe some of these methods and their characteristics.

2.1.1 Types of N-Body Systems

N-Body methods can be broadly classified [HE88] into 4 types:

1. Particle-Particle Methods
2. Particle in Cell Methods
3. Particle Mesh Methods
4. Tree Methods

An overview of the each of these methods is presented in the following sub sections with greater emphasis on tree methods.

2.1.1.1 Particle Particle Methods

In Particle-Particle methods [HE88, PH10] the particles interact directly with one another. For each of the particles in the system, forces from the other $N - 1$ particles are calculated. Particle-particle methods are very simple to use as their interaction directly follows a well understood physical interaction. Their simplicity, makes them easy to implement and parallelize. The main drawback with these methods is their time complexity which is on the order of $O(N^2)$ as $N(N - 1)$ pair-wise interactions of the bodies need to be executed. As the number of particles increases the time complexity of the direct approach, $O(N^2)$, imposes a fundamental scalability limitation.

2.1.1.2 Particle in Cell Methods

In the Particle-in-cell methods [Daw83], a regular grid is generated to cover the simulation region and the particles contribute their properties to create a source density which is interpolated at the grid points. A fast Poisson solver based elliptical partial differential equation is used to calculate the potentials at the grid point. The calculated potentials are used to compute the force and to interpolate the particle positions.

2.1.1.3 Particle Mesh Methods

In Particle Mesh methods [HE88] the particle interactions are expressed as a solution of a differential (Poisson) equation discretized onto an auxiliary mesh. Particles interact with each other indirectly through a mean field, essentially softening the gravitational interactions at small scales. Potentials and forces for each of the particles are calculated using finite difference approximations

and are interpolated to the mesh. The time complexity of this class of methods is on the order of $O(N)$ and the FFT complexity is of the order $O(N_g log(N_g))$. Particle mesh methods are limited by the mesh size, which introduces inaccuracies since the forces are computed based on interpolations to the mesh. In simulations involving large number of bodies, these inaccuracies can introduce problems with short-range interactions between systems such as two-particle binary systems. The main drawback of this class of methods is that the methods produce approximate solutions, often generating relative errors.

2.1.1.4 Tree Methods

In Tree based N-Body methods, particles are organized into a hierarchy of clusters, representing the physical system using a tree data structure. Several different approaches to effect Tree based N-Body methods covering diverse scaling, data structure interaction and mathematical models etc., these include Appel [App85], Greengard [GR97], Barnes Hut [BH86], and more. In general tree based methods provide either $O(NlogN)$ or $O(N)$ time complexities depending on the method used. In most cases tree based methods use either binary tree or octree data structures. Depending on the approach used the tree methods can have different interaction characteristics such as body-body, body-node or node-node. A detailed classification on the basis of above facets and more is discussed in detail by Salmon [Sal91, HE88].

2.1.2 Tree Based N-Body Simulation

While there exist several approaches to using tree based methods for discretization of the N-Body problem, the Barnes Hut algorithm [BH86] provides characteristics necessary to model dynamic workloads. Following are the main reasons for selecting Barnes Hut formalism.

- **Tree Data Structure** The tree data structure in Barnes Hut algorithm is significantly different than those used in other tree based methods. Each node in the Barnes Hut tree corresponds to a cube (square in 2D) in physical space. Each node has 8 child nodes / cubes (4 in 2D case) corresponding to eight smaller cubes obtained by sub-dividing the larger cube in half along the 3 coordinate axes (X,Y,Z). (section 2.2.1)

- **Adaptive Data Structure** The tree data structure in Barnes Hut algorithm is adaptive. In this context, "adaptive" refers to the ability of a data structure to grow naturally to more levels in the region with high particle density. By definition the Barnes Hut tree is constructed in such a manner that each cube/square can contain atmost one body. Hence as high particle densities are encountered the algorithm subdivides adaptively. This results in a tree that changes during every iteration when new position vectors for the particles are computed.(section 2.2.1)

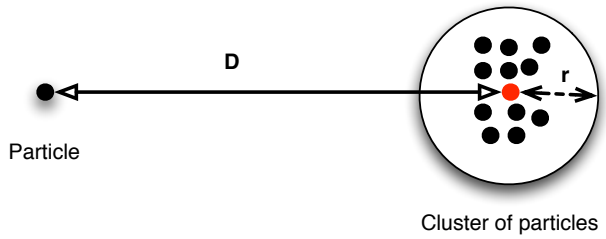
- **Computing Acceleration** In the Barnes Hut algorithm, accelerations are computed only for the bodies and not for the internal nodes of the tree. The internal nodes of the tree are static objects used as data structures for computing forces on the bodies. (section 2.2.3)

- **Computational Complexity** The tree construction and force calculation phases are distinct, providing algorithmic clarity and making the parallelization of Barnes Hut algorithm easier. The algorithmic complexity of $O(NlogN)$ facilitated by the adaptive octree formalism makes the problem of simulating large number of bodies.

The rest of the chapter is organized as follows: Section 2.2.1 contains a detailed description of quadtree/octree data structure which is pivotal to effectively representing the spatial distribution of the particle system. Section 2.2 provides the description of the steps of the Barnes Hut algorithm.

2.2 Barnes Hut Algorithm

Barnes Hut algorithm [BH86] is widely used in the astrophysical community as it is the simplest hierarchical N-Body algorithm. The Barnes Hut algorithm exploits the same idea used by all tree codes, where the forces on a body from a remote cluster of bodies can be approximated by treating the remote cluster as a single body. The accuracy of these approximations (see fig 2.1) depends on the distance (D) of the cluster from the body and the radius (r) of the cluster of particles. Remote clusters of bodies are treated as a single body only if D is greater than r/θ, consequently the parameter θ controls the error of the approximation. The complete algorithm for a Barnes Hut simulation is listed in figure 2.2.



Legend:

D - Distance of particle from the Cluster
r - radius of the cluster of particles
- center of mass the particle cluster

FIGURE 2.1: Barnes Hut Approximation Criteria

Sequential Barnes Hut Algorithm

```
For each iteration following steps are executed:
  1) Create the Barnes Hut tree
  2) Calculate the Center of Mass
  3) for(i = 1 to num_bodies)    //loop over all bodies
    3.1) check if distance of body(i) from remote cell is > r/theta
        if yes : treat remote cell as one body and use the
        center of mass to calculate force
    3.2) Recurse deeper into the subcells and repeat 3.1
           end for
  4) Calculate new position and velocities for each body
       using the force calculated in step 3
  5) Output current iteration data for visualization
  6) If the stopping criteria has been reached then stop simulation else
                  use latest position and velocities to start from
                  step 1)
```

FIGURE 2.2: Barnes Hut Algorithm

The key steps in the Barnes Hut N-Body algorithm are:

- Creation of the Barnes Hut tree data structure
- Center of mass calculation for each of the vertices in the Barnes Hut tree.
- Force calculation using the Barnes Hut approximation for each body in the system

In this section each of the above steps is explored in greater detail.

2.2.1 Barnes Hut Tree Construction

The foundational attribute of Barnes Hut N-Body algorithm and its performance is the underlying quadtree (2D) and octree (3D) data structure that is used to represent the physical space. All the major steps (physical system representation, center of mass and force calculations) require creation and utilization of the tree data-structure. The tree data structure facilitates recursion which makes the programmability of Barnes Hut algorithm easier.

2.2.1.1 Quadtree Construction

In order to create a Barnes Hut tree for 2D particle distribution, the computational region is initially represented as a square encompassing all the bodies involved in the simulation. This box is recursively subdivided into 4 quadrants of equal area until each undivided box contains exactly one body. The Barnes Hut algorithm for constructing a quadtree for 2D distribution consists of inserting each of the bodies into a Barnes Hut quadtree data structure until the body recurses down the quadtree to a leaf position where it is the sole resident. If during insertion a body reaches an existing leaf node where a previous body resides, then that node is divided into four child nodes until each of the two bodies can be located in bounding boxes corresponding with their coordinates. The position x-coordinate and y-coordinate comparisons used to decide the path of insertion of a body in a 2D Barnes Hut Tree.

2.2.1.2 Octree Construction

In order to create a Barnes Hut tree for 3D particle distributions, the computational region is initially represented as a cube encompassing all the bodies. This bounding cube is recursively sub-

divided into 8 octants of equal volume until each undivided cube contains exactly one body. The Barnes Hut algorithm for constructing an octree (3D distribution) similar to the quadtree approach. The approach consists of inserting each of the bodies into an octree data structure until the body recurses down the octree where it is the sole resident. If during the insertion, a body reaches an existing leaf where a previous body resides, then that node is subdivided into 8 child nodes until the two bodies can be located in individual bounding subcubes corresponding with their coordinates. The position x-coordinate, y-coordinate and z-coordinate comparisons are used to decide the path of insertion of the body in a 3D Barnes Hut tree.

The computational region is represented in a bounding cube which becomes the root node of the octree. Just as in a quad tree data structure, for each cube the octree data structure is recursively subdivided along the 3 coordinate axes into 8 equally sized sub-cubes as seen in Fig. 2.3. Each node in the tree refers to a sub-cube and the edges represent the parent-child relationship sub-cubes.

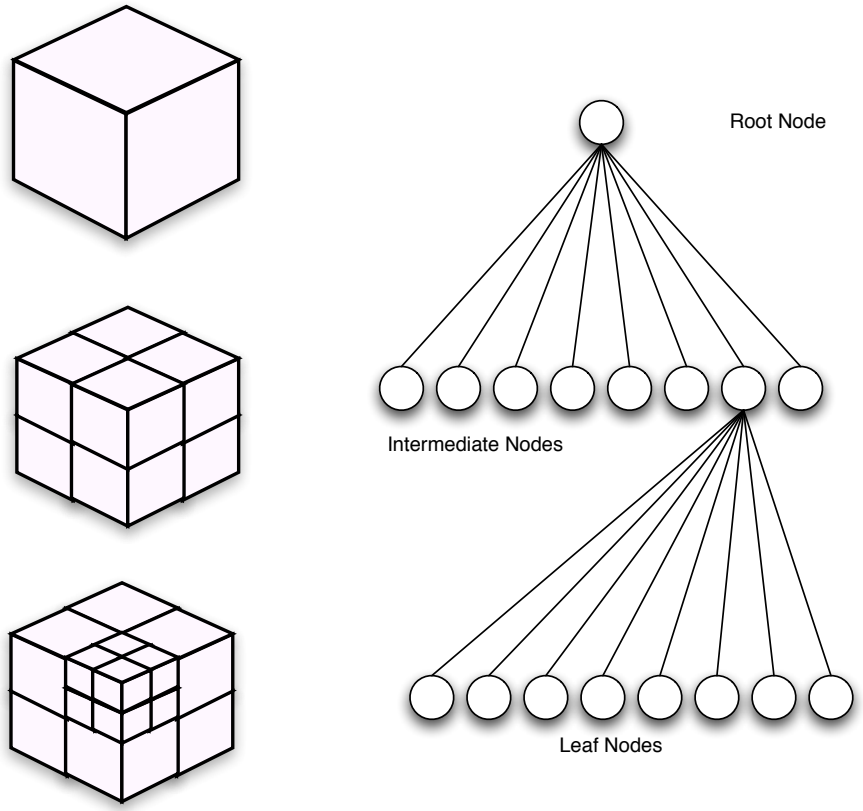


FIGURE 2.3: Octree

In order to effectively simulate a physical system, the data structures need to represent the spatial layout and the state information of each element of the system. The astrophysical data sets used for these simulations (positions of celestial bodies in 2D / 3D space) contain non uniform distribution of bodies. The non uniform distribution of data across the computational space creates challenges in accurate representation of the spatial layout. Barnes Hut algorithm handles this by a simple yet powerful adaptive construction of the tree data structure. Each cube/square in the Barnes Hut tree data structure can contain only one particle. The algorithm enforces this rule by subdividing cells, containing two or more particles, equally along the coordinate axes. If the resultant cell still contains more than one particle then that particular cell is further subdivided until the resultant cells contain only one particle. Figure 2.4 provides an exemplar adaptive subdivision. Using the input conditions of a physical system, comprising the positions of each of the bodies, an adaptive tree can be constructed using the algorithm listed in figure 2.5.

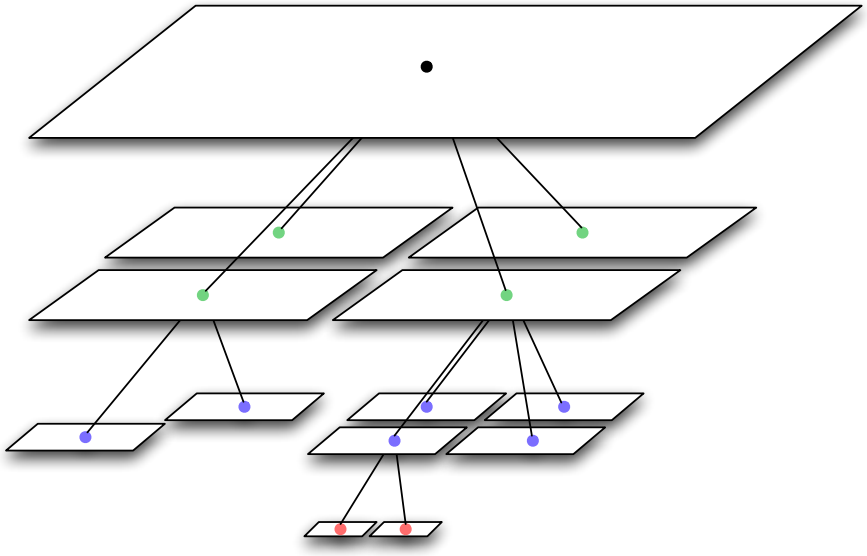


FIGURE 2.4: Adaptive Tree Representation

The resultant Barnes Hut tree consists of bodies stored at terminal nodes and internal nodes are primarily data structures. Each of the terminal nodes store complete state information of a body comprising, the mass of the body, position (in 2D or 3D), velocity vector (2D or 3D), and accelera-

Barnes Hut Tree Creation Algorithm

```
treeCreate(num_bodies) //provide the number of bodies in the system
  for(i = 1 to num_bodies)//loop over all bodies
     InsertBody(i, root)       //insert the i-th body
  end for

  InsertBody(i, node)       // try to insert the i-th body at the 'node'
    if subtree rooted at 'node' contains > 1 body
      determine which child 'chld' of the 'node', body i lies in
      InsertBody(i,chld)
    else if the subtree rooted at 'node' already contains a body
      copy the existing body in temp
      create a new subtree at the current position called chld-tree
      InsertBody(temp, chld-tree)
      InsertBody(i, chld-tree)
    else if the subtree rooted at 'node' is empty
      store the body at node n
  endif
```

FIGURE 2.5: Algorithm for creating an adaptive Barnes-Hut tree

tion. The intermediate nodes temporarily hold the aggregate information of their child nodes mass, center of mass and a position vector pointing to the geometric center of the cube/square.

2.2.2 Calculating Center of Mass and Total Mass

The Center of Mass (CM) of the Barnes Hut tree (and the intermediate nodes in the tree) is computed as a vector with X, Y and Z coordinates. The center of mass and total mass of a Barnes Hut tree can be calculated by performing a post order traversal of the Barnes Hut tree. Each node in the Barnes Hut tree holds the values for the mass and center-of-mass. For each of the leaf nodes the value of mass and total mass of the node are same. For the intermediate nodes the mass is computed as a sum of the masses of each of the individual children (c(i)). If the current node is a terminal node then the CM is calculated as :

$$CM.X = (currentNode.X * mass)/mass = currentNode.X$$

$$CM.Y = (currentNode.Y * mass)/mass = currentNode.Y$$

$$CM.Z = (currentNode.Z * mass)/mass = currentNode.Z$$

If the current node is an intermediate node then the center of mass is computed using the following formalism.

$$CM.X = \frac{1}{total\_mass} * \sum_{i=1}^{n} chld[i].X * chld[i].mass$$

$$CM.Y = \frac{1}{total\_mass} * \sum_{i=1}^{n} chld[i].Y * chld[i].mass$$

$$CM.Z = \frac{1}{total\_mass} * \sum_{i=1}^{n} chld[i].Z * chld[i].mass$$

The resultant Barnes Hut tree consists of mass and center of mass of bodies stored at the terminal nodes. The intermediate nodes store the mass and center of mass relative to the number of children. The algorithm used to calculate center of mass for the Barnes Hut N-Body simulation is provided in figure 2.6.

Center of Mass Calculation Algorithm

```
1  for(i = 1 to num_bodies)    //loop over all bodies
2     ComputeCM(root)      //insert the i-th body
3  end for
4
5  ComputeCM(node)            // try to insert the i-th body at the 'node'
6      if subtree rooted at 'node' contains  1 body
7          the mass and center of mass of the body are identical
8          node.mass = node.mass
9          node.cm.x = (node.x * mass) / mass
10         node.cm.y = (node.y * mass) / mass
11         node.cm.z = (node.z * mass) / mass
12         return (node.mass, node.cm)
13     else
14         for all children c(i) of node (i = 1,2,3,...,8)
15              (mass(i), cm(i)) = ComputeCM(c(i))
16         end for
17         node.mass = mass(1) + mass(2) +...+ mass(8)
18         node.cm = (mass(1)*cm(1) + mass(2)*cm(2)
19                              +...+ mass(8).cm(8)) / node.mass
20         return (node.mass, node.cm)
21      end if
```

FIGURE 2.6: Algorithm for calculating center of mass

2.2.3 Force Calculation

The force calculation using the Barnes Hut algorithm uses distance based approximations to reduce the number of particles evaluated. The ratio relating a particle to a remote group of bodies bounded by a box is computed:

$$D/r = \frac{sizeofbox}{distancefromparticletoCMofbox}.$$

If the distance ratio is < a user defined value θ then the force is calculated using the formula

$$Force = G * m * m_{boxcm} * (\frac{x_{boxcm}-x}{r^3}, \frac{y_{boxcm}-y}{r^3}, \frac{z_{boxcm}-z}{r^3})$$

where r is the distance from the particle to the center of mass of the box computed using

$$r = \sqrt{(x_{boxcm} - x)^2 + (y_{boxcm} - y)^2 + (z_{boxcm} - z)^2}$$

If the distance ratio is > the user defined θ then the force is calculated using the formula

$$Force = G * m * m_{cm} * (\frac{x_{cm}-x}{r^3}, \frac{y_{cm}-y}{r^3}, \frac{z_{cm}-z}{r^3})$$

where r is the distance from the particle to the center of mass of the particle in the box computed using

$$r = \sqrt{(x_{cm} - x)^2 + (y_{cm} - y)^2 + (z_{cm} - z)^2}$$

The algorithm for the Barnes Hut force calculation is listed in figure 2.7. At the end of the above step each particle has a new force, acceleration and velocity computed. Position for each of the particles is modified based on the force computed and new positions are updated. This is done by using a leapfrog (Verlet Integration) method [HE88, HMM95]. The leapfrog integration method is a common integration technique used in integrating the equations of motion in N-Body systems. Standard implementations of the leapfrog method involve time-interleaving the positions and velocities at alternate points in time. The governing equations for the leapfrog integration method in the Barnes Hut Method can be represented as follows:

$$r_1 = r_0 + v_{1/2}\delta t$$

$$v_{3/2} = v_{1/2} + a_1\delta t$$

Force Calculation Algorithm

```
for(i = 1 to num_bodies)    //loop over all bodies
    Force(i) = ComputeForce(i, root)
    //Compute force between i-th body and all other bodies
end for

ComputeForce(i, node)
 // Compute force on ith particle due to all the particles
 // in the box at 'node'
 if subtree rooted at 'node' contains  1 body
       Force is computed using above listed formula 2
 else
       r = distance from particle i to CM of box at 'node'
       D = size of box at 'node'
       if D/r < theta
            Force is computed using above listed formula 1
       else
            for all children c of n
               f = f + TreeForce(i,c)
            end for
        end if
   end if
```

FIGURE 2.7: Algorithm for calculating Force

where r is the position vector, $v = dr/dt$ is the velocity (change in position over time) and $a = dv/dt$ is the acceleration (change in velocity over time).

The Barnes Hut tree is then recomputed and the tree creation, center of mass calculation and force calculation steps are repeated until the exit criteria is reached (eg. number of iterations).

2.2.4 Performance Factors - Barnes Hut Tree

Tree Creation The time complexity of tree creation in Barnes Hut algorithm depends on the distribution of the bodies in the computational space. The cost of inserting a particle is directly proportional to the distance from the root to the level at which the particle is placed. If the particles are clustered closely together the complexity is likely to be higher as the number of sub-boxes and subtrees created increases proportionally to the cluster. If the distribution of the particles is uniform across the computational space then the complexity of creating the tree is $O(NlogN)$.

Center of Mass Calculation The time complexity of calculating the center of mass of bodies in the N-Body tree is directly proportional to the number of bodies in the tree. Generally the measure for computational complexity for quad tree based data structures is $O(NlogN)$.

Force Calculation The time complexity of calculating the force of the bodies in the Barnes Hut tree is directly proportional to the level at which the particle is stored. If the distribution of particles is uniform across the computational space then the complexity is $O(NlogN)$.

2.3 Parallel Barnes Hut Methods

The Barnes Hut method has been successfully parallelized using several techniques [WS93, Sal91, LB94, Sin93, Dub96, MHI07] as discussed in section 1.1.4. The key challenges in parallelizing the Barnes Hut simulation comprise domain decomposition of the Barnes Hut tree across the allocated memory resources and load balancing the workloads allocated processors .

2.3.1 Parallel Barnes Hut Method Using Shared Memory Systems

In the context of shared memory semantics, parallelization of the tree is straightforward. Due to the system architecture the shared memory space can be addressed by all processors using a

supporting shared memory programming model such as OpenMP. Parallelization of Barnes Hut tree in this context involves creation of the tree in shared memory space and providing controlled write access to prevent contention and race conditions [Sin93]. The force computation in a Barnes Hut simulation requires read only access to the tree with local acceleration updates, the force computation can be parallelized by providing each processor access to a distinct portion of the N-Body system. Given p processors in a shared memory system, to solve an N body system evolution problem using parallel Barnes Hut approach, each processor would have a workload of N/p as seen in figure 2.8. A key aspect to note is that while the number of particles might be easy to distribute

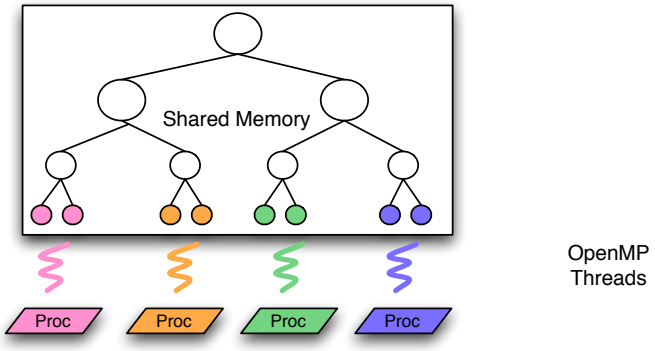


FIGURE 2.8: OpenMP Shared Memory Parallelization of Barnes Hut Tree

across the allocated processors, each of the particles using the Barnes Hut formalism interacts with varied number of particles. That is per particle workload in the Barnes Hut formalism will vary, for large N-Body systems, the variable workload when processed using shared memory programming models such as OpenMP result in suboptimal utilization of allocated resources.

2.3.2 Parallel Barnes Hut Method Using Distributed Memory Systems

In the context of distributed memory semantics, parallelization of the tree is a little more complex. One approach towards parallelization of Barnes Hut is to create a complete tree in the local memory space of each of the allocated processors. Using this method each of the processors calculates forces for a section of the N-Body system and has full access to the Barnes Hut tree (as shown in figure 2.9). However the key limitation in this approach is that the scale of the problem that can be simulated is constrained to the number of bodies that can be created in the local memory space.

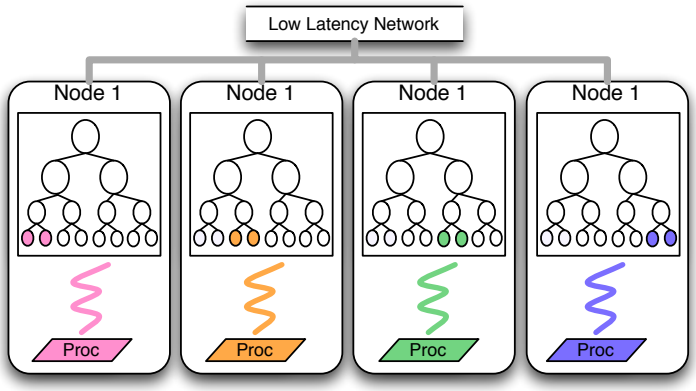


FIGURE 2.9: Distributed Memory Processing of Barnes Hut Tree

A better and more scalable approach towards parallelization of the Barnes Hut algorithm involves partitioning of the tree across the distributed nodes. Several successful implementations of distributing the tree data structure across the allocated resources have been successfully demonstrated [WS93, LB94, Dub96]. Fundamental to the success of this approach is the domain decomposition of a complex non-uniform structure. Common approaches towards decomposition involve creating interaction list for each of the particles in the N-Body system. This per-particle list comprises the essential subset of tree such that when forces are computed all data dependencies are resolved locally using orthogonal recursive bisection (ORB [Sal91]) and creation of locally essential trees (LET [WS93]). Salmon and Warren's approach also involved associating each particle with a unique identifier which was resolvable using simple hash functions. Using Morton key ordering or Hilbert-peano key ordering provided ability for the simulation to efficiently utilize message passing based synchronization semantics and consequently the allocated resources more efficiently. Figure 2.10 provides a notional representation of a tree that is partitioned across each of the allocated nodes.

2.3.2.1 Parallelization Using Warren-Salmon approach

The Warren-Salmon approach utilizes Orthogonal Recursive Bisection (ORB) technique [Sal91] to spatially partition the computational domain. The large square comprising all the particles in the simulation are subdivided into two sub-rectangles each of which contains approximately equal

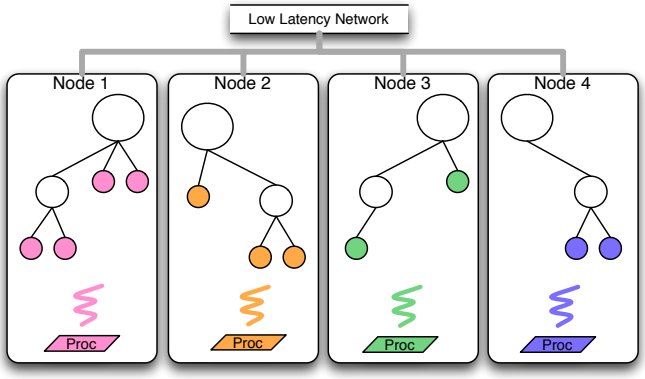


FIGURE 2.10: Domain Decomposition of The Barnes Hut Tree on a Distributed Memory System

number of bodies. Each of these rectangles is assigned to a processor. During the simulation when bodies move across processor domains, work imbalances can result. Consequently for densely distributed particles the ORB is recomputed at the end of every iteration. Rather than requiring each of the allocated processor to host the global octree structure, Warren Salmon approach involves creation of locally essential trees (LET). The key idea in the construction of LETs is that for each particle all the particles that satisfy a Multiple Acceptability Criteria (MAC), a modified form of Barnes Hut approximation criteria $D/r < \theta$, is utilized MAC is used to develop a subset of the global octree required by all the particles in the processor's computational domain. This approach ensures that data locality dependencies of the local pool of particles are satisfied reducing communication overhead during the simulation.

This research is inspired by the distributed tree approach by Warren and Salmon [Sal91, WS93]. In this research rather than decomposing the tree across the allocated resources, the entire Barnes Hut tree is created in the global address space (AGAS) where each of the nodes has a globally unique identifier. Workload distribution is achieved by creating interaction lists for each particle and representing each computation in a work-queue model which is scheduled to operate across each of the allocated compute resources. This is done by creating interaction lists for each particle by using Barnes Hut approximation criteria $D/r < \theta$. Futures [BH77, RHH85] based synchronization is used to create asynchrony within each time step (iteration) of the N-Body simulation. Details on the research implementation is provided in Chapter 5

Chapter 3
Reference Implementations of Barnes Hut Algorithm

The Barnes Hut algorithm can be implemented using several different techniques depending on the system architecture on which the simulation is executed and the supporting execution ecosystem (comprising programming model, runtime systems and systems organization). In this research the following implementations of the Barnes Hut N-Body algorithm have been developed using conventional techniques.

- **Sequential Implementation** The sequential version of Barnes Hut N-Body algorithm is implemented using the C++ programming language. The sequential Barnes Hut implementation provides characterization of the algorithm on a sequential / serial processor using no implicit or explicit parallelism. The results produced from this implementation are used to validate the accuracy of the parallel implementations, since parallelization can sometimes introduce errors in the computational results. The sequential implementation also provides a clear formal description of the Barnes Hut algorithm. A detailed description of the implementation is provided in section 3.1

- **OpenMP Implementation** The parallel version of Barnes Hut N-Body algorithm is implemented on a shared memory system using the OpenMP programming model. The OpenMP programming model is the critical conventional means of providing parallelism for applications in a shared memory systems environment. The OpenMP parallelized implementation of Barnes Hut N-Body algorithm provides a reference parallel implementation using conventional parallelization techniques for comparing the research implementation of the algorithm using HPX (high performance ParalleX). The measurements obtained from the OpenMP implementation of the algorithm provide a platform for comparing effectiveness of conven-

tional parallelism techniques to the proposed techniques of the HPX. A detailed description of the OpenMP implementation of the Barnes Hut algorithm is provided in section 3.2

The goal of this section of the thesis is to characterize the Barnes Hut N-Body algorithm using conventional execution techniques and derive limitations of these approaches from the implementations of the algorithm. Section 3.3 provides details of the challenges to scalability and performance of Barnes Hut N-Body simulation on the systems selected. Section 3.3.2 details the desired characteristics in a runtime system that provides desired performance and scalability for dynamic graph applications such as the N-Body simulation.

3.1 Sequential Implementation of Barnes Hut Algorithm

The sequential version of the Barnes Hut algorithm is implemented using the C++ language with the following goals

- Formal representation of the algorithm using programming language semantics
- Results generated are used to validate parallel implementations of the Barnes Hut algorithm
- Characterization of algorithm on a serial processor

This section provides a detailed description of the serial implementation including the execution flow, data structures used, functional description of the program, and the execution setup.

3.1.1 Execution Flow of the Sequential Code

The execution flow of the sequential implementation maps directly on to the Barnes Hut algorithm described in chapter 2. In addition to the steps of creating the Barnes Hut tree, calculation of the center of mass, and the computation of force, key implementation details such as input and output file generation and visualization of results are added. The key steps in the sequential implementation of Barnes Hut algorithm are

1. Reading in input values containing position, initial velocity, number of bodies etc from the input file, which allows for increasing the number of bodies without having to recompile the program.

2. Calculating the bounding box and the center position of the bounding box containing all the N bodies to determine the position of the root of the Barnes Hut tree.

3. Creating a tree data structure that maps the spatial position to a corresponding location in the tree. The default tree construction algorithm is configured for 3D simulations resulting in an octree representation of the spatial distribution of the particles.

4. Calculating the center of mass by post order traversal of the octree data structure.

5. Computing forces on each of the particles using the Barnes-Hut method, where the particle interacts with all entities (bodies or nodes in the Barnes Hut tree) matching the Barnes Hut approximation criteria of $D/r < \theta$, where θ is a user defined value obtained from the input file.

6. Using leapfrog integration to compute the changed position and velocities of the bodies in preparation for the next iteration.

Detailed description for each of the above steps, and the functions used are given below.

3.1.2 Data Structures

There are two main data structures used in the program: the leaf nodes which contain the body and the intermediate node which hold the values for the center position of the bounding boxes.

The IntrTreeNode data structure comprises a node_type which identifies it as a CELL, a position vector which stores the x,y and z positional coordinates, a mass variable which stores the weighted mass of its children, a pointer to the parent node in the tree and a vector of 8 children which form the building blocks of the octree data structure. Each of 8 children of an IntrTreeNode point to either a body, another IntrTreeNode or point to null. The 8 children correspond to the 8 sub-cubes

corresponding to the 3D coordinate system when the cube is equally divided across the x,y,z coordinates. The initial position of the IntrTreeNode is set to the center of the cube that it represents. The TreeLeaf data structure comprises a node_type identifier which identifies it as a PAR, a position vector which stores the x,y, and z positional coordinates, a mass variable which stores the mass of the body, a velocity vector and an acceleration vector. The main difference between the two data structures is that TreeLeaf data structure has a velocity vector and an acceleration vector which the IntrTreeNode does not. This is because only the bodies involved in the system move, the IntrTreeNodes are used to represent pseudo-particles (a cluster of particles) and do not move.

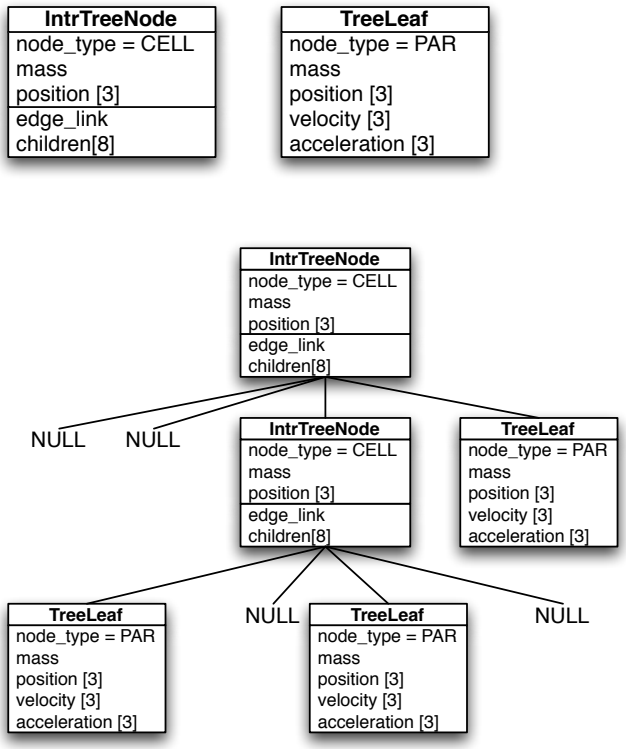


FIGURE 3.1: Barnes Hut Data Structures Used in the Sequential Implementation

3.1.3 Functional Description

The sequential implementation of the Barnes Hut N-Body algorithm is a relatively simple implementation and comprises 6 key functions: *main(), computeRootPos(), treeNodeInsert(), calculateCM(), calculateForce(), moveParticles().* Each of these functions performs a critical activity.

- *main()* function is the entry point in the program and calls all other functions at appropriate times. This function is described in detail in section 3.1.3.1.

- *processInput()* function processes the input parameter file to load critical simulation values such as number of iterations, number of bodies, and position, velocity vectors of the bodies. This function is described in detail in section 3.1.3.2.

- *computeRootPos()* function calculates the bounding region of the computational domain and determines the root position of the Barnes Hut tree. This function is described in detail in section 3.1.3.3.

- *treeNodeInsert()* function constructs the Barnes Hut tree by inserting the body one at a time into the tree and organizing the tree according to spatial distribution of the particles. This function is described in detail in section 3.1.3.4.

- *computeCM()* function uses the Barnes Hut tree to calculate the center of mass at all nodes in the tree. These center of mass values are used in computing the force exerted by the system on each of the particles. This function is described in detail in section 3.1.3.5.

- *calculateForce()* function uses the center of mass calculations and the Barnes Hut tree structure to apply the Barnes Hut approximation criteria and calculate the force exerted by the system on each of the particles. This function is described in detail in section 3.1.3.6.

- *moveParticles()* function uses the acceleration calculated by the *calculateForce()* function and applies the leapfrog/verlet integrator to advance the position of the bodies and calculate new velocities of the particles involved. This function is described in detail in section 3.1.3.7.

3.1.3.1 main() Function

The *main()* function is the key driver function in the program. This function calls all the other functions and organizes the computation. The main function checks the command line arguments and parses the command-line arguments to read correct input files. Once the input file name to process

is known, the main function then calls the *processInput()* function. This function provides *main()* with simulation related information such as number of iterations, number of bodies and pertinent information about the bodies such as the position and velocity vectors. This information is loaded on to corresponding data structures and the next function *computeRootPos()* is invoked. The *computeRootPos()* function uses the position information of each of the bodies in the system to derive a computational bounding box and to calculate the center position of the computational domain. This function generates the center position where the root of the Barnes Hut tree is created. The main function then calls the *treeNodeInsert()* function to cycle through each of the bodies in the system and insert them into the Barnes Hut octree. Once the tree is fully populated with the bodies, the main function then calls the *calculateCM()* function which performs a post order traversal of the octree and calculates center of mass for each of the intermediate nodes and the leaf nodes in the Barnes Hut tree. The main function then calls the *calculateForce()* function which uses the Barnes Hut approximation criteria ($D/r < \theta$) to determine the subset of the tree that each particle interacts with and force effected by the remote particle or pseudo-particle to the body under consideration is computed. The value of θ is provided by the *processInput()* function which reads it as a user defined parameter from the input file. Upon successful processing of all the bodies, the main function calls the *treeReuse()* function to clear the contents of the tree and utilize the allocated memory space for the next iteration. This allows the program to re-purpose allocated memory rather than reallocate the tree every time. Finally the main function calls the *moveParticle()* function which uses a leapfrog integrator to calculate the new position and velocities for each of the bodies. The *main()* function then repeats the above 6 steps in a for-loop until the exit condition of maximum number of iterations is satisfied.

3.1.3.2 processInput() Function

The *processInput()* function takes in a string *input_file* as an argument. Using the *input_file* string argument the *processInput()* function creates a input filestream to read the input data. The function reads in the first line into the *num_bodies* variable which defines the number of bodies in the system.

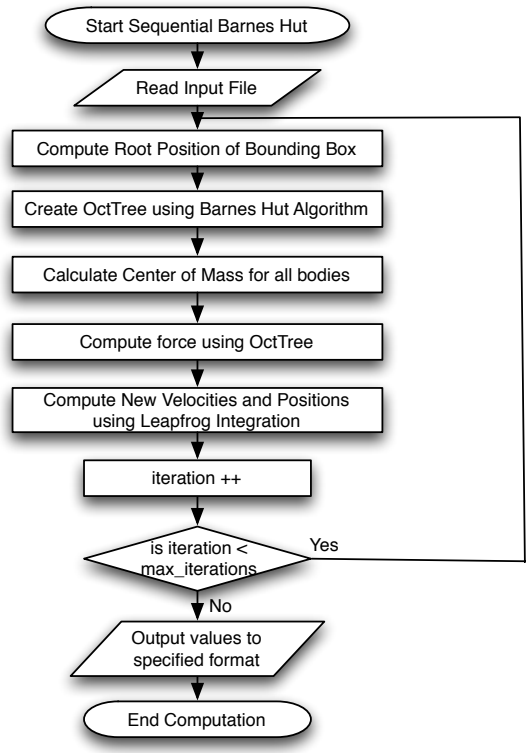


FIGURE 3.2: Flowchart of Barnes Hut Sequential Algorithm

The function reads in the second line into the *num_iterations* variable which defines the number of iterations to be computed. The function reads in the third line into the *dtime* variable which defines the integration timestep for the Verlet (leap frog) integrator. The fourth and the fifth lines are read in as the *eps* and the *tolerance* variables. Remaining lines correspond with the particle information comprising: mass, position and velocity. The function creates a for-loop to read the data into the TreeLeaf data structure created for each of the bodies. At the end of this function each of the bodies has a fully qualified mass, position vectors and velocity vectors. The function closes the input file buffer and returns control to the *main()*.

3.1.3.3 computeRootPos() Function

The *computeRootPos()* function takes in 3 arguments: the number of bodies, box size and the centerPosition array. The function initiates a for loop that cycles through all the particles in the Barnes Hut N-Body system and uses the position vector of each of the particles to compute the maximum (X, Y, Z) coordinates and the minimum (X, Y, Z) coordinates. These maximum and minimum

coordinates determine the bounds of the computational region. The function uses the maximum and minimum coordinate values to calculate the center point of the computational domain. This information of the bounding box and center of the computational domain are passed backed to the *main()* function.

3.1.3.4 treeNodeInsert() Function

The *treeNodeInsert()* function takes in 2 arguments: the particle to be inserted and the sub-box size. The function compares the position of the particle to be inserted (P_i) to the position of the current intermediate node. Based on the relative position of P_i, the corresponding octant into which the particle is to be inserted is chosen. Three possible conditions can occur when an octant is chosen:

- The node in the tree where the body is being inserted is empty. In which case the body is inserted at the computed position.
- The node in the tree where the body is being inserted is occupied by a Intermediate node. In which case the function recursively cycles down into the branches of the and inserts the body at the correct leaf location.
- The node in the tree where the body is being inserted is occupied by another body. In which case the function copies the existing body into a temporary buffer variable, creates an intermediate node at that position and inserts the previous body and the new body using the intermediate node as a starting point.

Once the particle is inserted the control is returned back to the main function. Once all the particles are processed through the *treeNodeInsert()* function, all the particles become part of the global Barnes Hut tree. This tree is then used to compute the center of mass and gravitational forces in the system.

3.1.3.5 computeCM() Function

The *computeCM()* function is a recursive function that computes the center of mass of each of the bodies and the intermediate nodes in the Barnes Hut N-Body system. The function performs a post

order traversal of the Barnes Hut tree generated by the *treeNodeInsert()* function. The function expands down the tree until it reaches the leaf nodes. The function then calculates the center of mass of the leaf nodes containing the bodies, which is the same as the position of the bodies (weighted mass cancels out for the bodies). The function then collapses upwards and calculates the center of mass at the intermediate nodes using the following formula.

$$CM = \frac{1}{total\_mass} * \sum_{i=1}^{n} pos[i] * mass[i]$$

The center of mass calculated by the above formula is a weighted average of the particles contained within the sub cell / cube. The algorithm used to compute the center of mass is described in section 2.2.2. Once the function successfully completes, each of the nodes (both intermediate and leaf) is populated with center of mass values and the control is returned back to the main function.

3.1.3.6 calculateForce() Function

The main function then calls the *calculateForce()* function to compute the force effected on each particle by the rest of the system. For each particle the calculateForce() function uses the Barnes Hut approximation criteria of $D/r < \theta$ to check if the remote cluster is close enough from the impacted body to warrant the unrolling of the subtree and recursing through the next level in the tree. If the remote cluster is far enough away that the constituent bodies of the cluster have low individual impact on the force experienced by the body, then the function uses the center of mass of the cluster to compute the force on the particle, effectively treating the cluster as a pseudo particle. If the remote cluster is close to the impacted body then the function recursively traverses down to the next level of granularity and the above process is repeated. If the *node_type* encountered by the function in the next level is a body rather than an intermediate cell, then the function uses the body data to compute the force effected on the remote body. If the distance ratio is greater than the user defined θ then the force is calculated using the formula

$$Force = G * m * m_{cm} * (\frac{x_{cm}-x}{r^3}, \frac{y_{cm}-y}{r^3}, \frac{z_{cm}-z}{r^3})$$

where r is the distance from the particle to the center of mass of the particle in the box computed using

$$r = \sqrt{(x_{cm} - x)^2 + (y_{cm} - y)^2 + (z_{cm} - z)^2}$$

The detailed algorithm used by the function to calculate the forces is listed in section 2.2.3. The function returns control back to the *main()*, once all the bodies impacting the body under consideration have been considered and resultant acceleration is accounted for.

3.1.3.7 moveParticles() Function

The *moveParticles()* function uses the acceleration computed by *calculateForce()* to calculate the new position and velocity of the particles. This is done by using the Verlet or the LeapFrog integration technique. Standard implementations of the leapfrog method involve time-interleaving the positions and velocities at alternate points in time. The governing equations for the leapfrog integration method in the Barnes Hut Method can be represented as follows:

$$V_{dt/2} = a * (\delta t/2)$$

$$V_{1/2} = V_t + V_{dt/2}$$

$$r_1 = r_0 + V_{1/2} * \delta t$$

$$v_{3/2} = v_{1/2} + (a * \delta t)$$

where r is the position vector, $v = dr/dt$ is the velocity (change in position over time) and $a = dv/dt$ is the acceleration (change in velocity over time). Once the new position and velocity are calculated the control is returned back to the *main()* function. The *main()* function then recomputes the tree, center of mass, force calculation, and the move generation until the exit criteria of number of iterations is reached.

3.1.4 Execution Setup

The following section highlights the execution setup and the supporting files

Example Input File

```
5                  //number of bodies
25                 //number of iterations
0.025              //dtime - integration time step
0.05               //density smoothing length
0.5                //tolerance
1.0E-03 5.231E-03 3.36E-01 -4.14E-01 -3.59E-01 2.06E-01 6.32E-01
1.0E-03 3.46E-02 -2.76E-02 3.18E-01 3.49E-02 -7.53E-01 -2.56E-01
1.0E-03 1.173E00 -2.29E-01 2.61E-01 7.25E-02 -2.54E-01 -1.22E-01
1.0E-03 4.599E-01 1.93E-01 2.2E-01 -3.47E-01 -8.38E-01 -1.17E-01
1.0E-03 -7.10E-01 4.99E-01 1.65E00 1.05E-01  4.79E-01  1.07E-02
```

FIGURE 3.3: Input File Format

3.1.4.1 Input File Format

The input files for the Barnes Hut N-Body simulations are organized to facilitate user configuration of various parameters so that the program does not have to be recompiled to account for changes in parameters. The number of bodies, iteration, θ and integration time-step can be reconfigured by editing the input file. The first line consists the number of bodies. The second line contains the number of iterations. The third contains the integration time step. The fourth contains the density smoothing length. The fifth line contains the tolerance value. The remaining lines contain particle information with the first column being the mass of the body, columns 2,3,4 contain the X, Y, Z position of the particle and 5, 6, 7 contain the velocity of the particle. An example of the 5 body input file is provided in figure 3.3

3.1.4.2 Output File Format

The output file is formatted in a similar form to the input file where the first line consists of the number of bodies and the second contains the iteration number. The first three columns of the ensuing data contain the position vectors (x, y, z coordinates) of the N bodies, and the second 3 columns of the ensuing data contain the velocity of the particles. An example of the output file for a 5 body simulation is presented in figure 3.4

Example Output File

```
5                   //number of bodies
1                   //iteration number
0.0343185 -0.0467009 0.305785 0.0343896 -0.750564 -0.259804
1.3191 -0.231906 0.260379 0.0720048 -0.252621 -0.125223
-0.00359628 0.343426 -0.401016 -0.35236 0.206633 0.632994
0.447417 0.172157 0.224146 -0.342964 -0.838932 -0.186177
-0.698271 0.511935 1.65175 0.109564 0.477784 0.0101666
```

FIGURE 3.4: Output File Format

3.1.4.3 Visualization Routines

Basic visualization is performed using GNUplot. For the output format described above the visualization routines generate images depicting position of particles at each iteration. Following are some of the output visualizations generated for simulations.

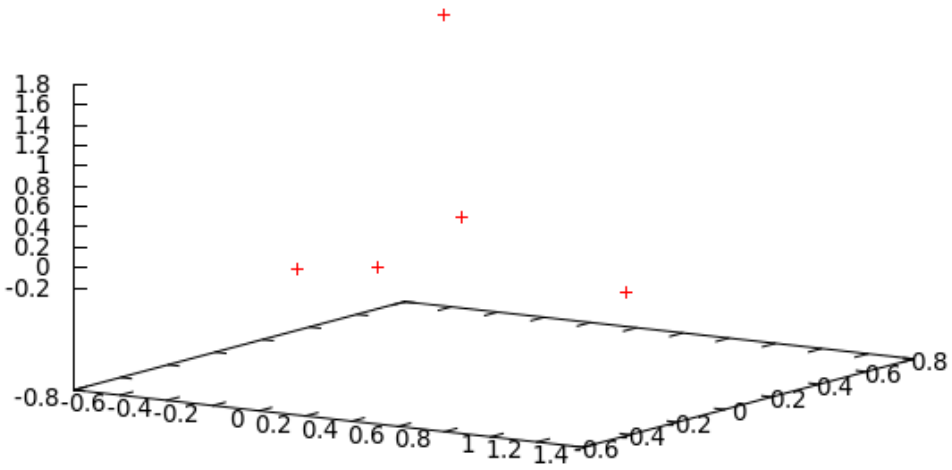


FIGURE 3.5: 5 body run visualization, initialized using the Plummer model

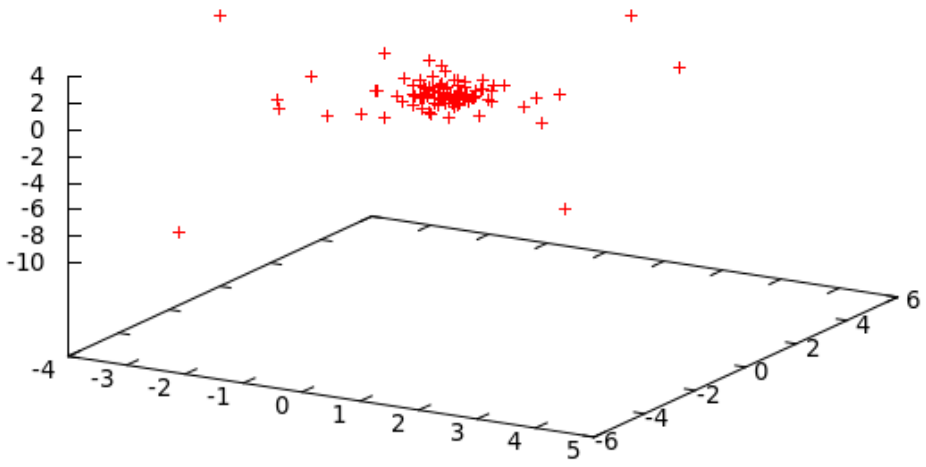


FIGURE 3.6: 100 body run visualization, initialized using the Plummer model

Rudimentary support for movies is provided by an awk based script which cycles through the GNUplot visualizations of the output to create a movie-like effect. The visualization routines were derived from "The Art of Computational Science" [PH10].

3.1.5 Characterization of Sequential Implementation

The sequential implementation was executed for different set of input files of 100 bodies, 1,000 bodies, 10,000 bodies, and 100,000 bodies to determine the part of the program that takes most time to execute. The profiling of the output was done using GNU gprof where the application was compiled with $-pg$ linker flags to generate profiling mappings to the compiled executable. The resultant raw output was processed using gprof to generate data on percentage distribution in time of various functions. Table 3.1 shows that the force computation routine comprises more than 98% of overall time of the sequential implementations across an increasing input data size. Based on the data, the force calculation routine of the Barnes Hut is the most time consuming routine in the program. The parallelization approaches (in both OpenMP and HPX) focus on parallelizing the

TABLE 3.1: Characterization of Sequential Barnes Hut Implementation

Name of Function	100 body %	1000 body %	10,000 body %	100,000 body %
Force Calculation	99.97	98.46	99.23	99.42
Tree Creation	0.01	0.41	0.29	0.25
Move Particles	0.01	0.01	0.03	0.08
Center of Mass Calc.	0.01	0.41	0.16	0.12

force calculation with the goal of reducing the overall computation time. Detailed characterization of the sequential Barnes Hut N-Body implementation is discussed in Chapter 6.

3.2 OpenMP Implementation of Barnes Hut Algorithm

The parallel version of the Barnes Hut algorithm is implemented using the OpenMP for the following reasons

- OpenMP is one of the primary conventional means of achieving parallelism in applications
- The OpenMP programming model provides a fair comparison to single locality shared memory based runs performed using High Performance ParalleX
- The shared memory programming model provides performance and algorithm parallelism characterization of the Barnes Hut using one of the main conventional approaches to parallelism

This section provides a detailed description of the shared memory systems architecture, the OpenMP programming model that is used to program shared memory systems, the OpenMP based parallel implementation of Barnes Hut algorithm, and challenges to scalability of Barnes Hut algorithm on shared memory systems.

3.2.1 Background: Shared Memory System Architecture

Shared memory architecture is one of the major system architectures in high performance computing. In the shared memory model each node comprises several banks of processors each of which has hardware support for system wide access to banks of memory. While performing write

access on a shared data space memory corruption could occur, for this reason most shared memory systems are equipped with coherency protocols that write-protect the shared memory space.

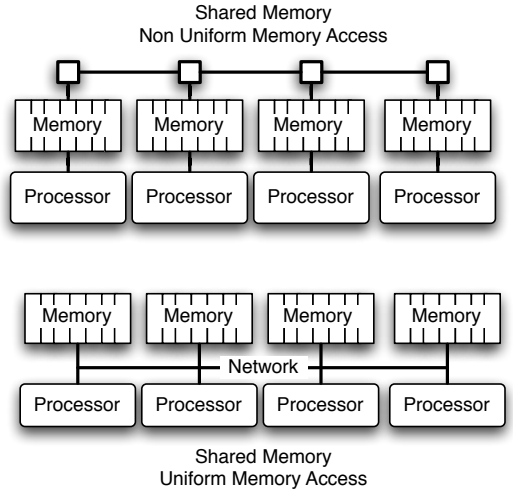


FIGURE 3.7: Shared Memory System Architecture

3.2.2 Programming Model: OpenMP

The primary model for programming a shared memory system is OpenMP [DM98, CDK+01]. OpenMP uses the fork-join model to create new threads and synchronize control among the threads. Since shared memory system provide hardware support for system wide access to memory, all executing threads can access the data transparently without having to use message-passing based network oriented data synchronization primitives. When multiple processors operate on a shared data

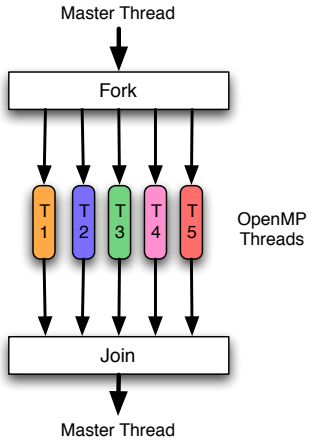


FIGURE 3.8: OpenMP Fork-Join Model

space in write mode, memory data corruption could result. OpenMP provides several primitives that help protect the data to varying levels. Additionally OpenMP also supports critical sections semantics which limits the access to a particular section of the memory to one control thread at a time. The number of threads is limited to the number of processors available in the system. Most OpenMP based applications are parallelized using the following steps:

1. A number of processors(threads) are assigned to solve the problem.

2. The data structure representing the computational space is populated with domain knowledge (for eg. particle mass, position and velocity in Barnes Hut algorithm)

3. The workload / data structure is partitioned distributed over the number of threads available using standard load balancing techniques.

4. Same program runs on each of the different threads and operates on the allocated portion of the data hosted in the shared memory space.

5. No explicit message passing based synchronization is needed as the entire dataset is resident in the shared memory space. However blocking fork and join operations are needed for synchronization across to ensure orderly progress of each iteration and data consistency. Additional data coherency protocols are needed to ensure that the shared memory space does not lead to corrupted data due to write conflicts.

6. If the exit condition is reached then all processors abort; if not, the last two steps are repeated until the exit condition is reached.

3.2.3 Barnes Hut N-Body OpenMP Implementation

The OpenMP version of the Barnes Hut Algorithm builds upon the sequential version of the Barnes Hut algorithm. The input file processing, creation of the Barnes Hut tree, computing the center of mass and the move generation function are exactly the same as the serial version. The key difference is that the force computation for-loop is parallelized using OpenMP. The force computation

over the N-Body system is composed of independent computations. This allows us to parallelize

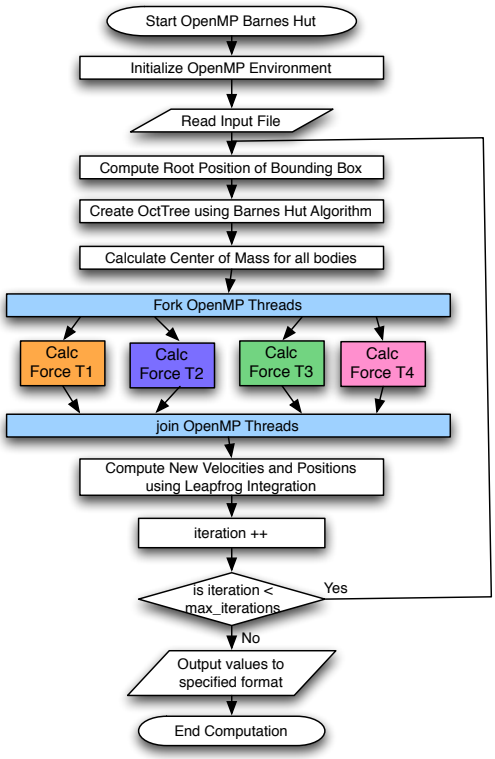


FIGURE 3.9: Flowchart of Barnes Hut OpenMP implementation

the force computation using the *parallel for* construct of OpenMP. Under the fork-join model of OpenMP the runtime system utilizes the **$OMP_NUM_THREADS** environment variable to determine the number of OpenMP threads to fork. Based on this information the OpenMP runtime system automatically parallelizes the workload across the number of OpenMP threads. In the case of for-loop parallelization OpenMP divides the work such that each processor has a chunk of the array allocated. For example, in a 4 thread, 100 N-Body system simulation, each OpenMP thread would be allocated 25 distinct bodies to work on. Each thread processes its workload and updates are made to the shared octree data structure. This is feasible in shared memory systems since all the threads operate on the same block of memory, enabling each of the OpenMP threads complete access to the Barnes Hut tree. In the N-Body OpenMP implementation each of the body updates its own acceleration (data space), hence special data protection mechanisms which hinder OpenMP program performance, such as critical sections, are not needed. The move generation routine which

computes the new position and velocities of the particles is computed using the same technique as the serial version of the program. The flow chart for the execution flow of the Barnes Hut OpenMP program is given in the Figure 3.9.

3.2.4 Characterization Using OpenMP Implementation

The OpenMP parallelization of the Barnes Hut algorithm focused on the force calculation routine. For a fixed data set of 100,000 bodies, the run-times for various functions in the serial and parallel versions were compared. The UNIX gettimeofday() routine was used to obtain microsecond-level granularity. By increasing the number of OpenMP threads to 8, the force calculation routine completes in 1/5th time of the sequential run (see Table 3.2). Detailed characterization of the sequential Barnes Hut N-Body implementation is discussed in Chapter 6.

TABLE 3.2: Characterization of OpenMP Barnes Hut Implementation

Function	OpenMP time (ms)	Serial time (ms)
FileRead	506.00	480.57
CalcRootPos	75.51	75.52
TreeCreate	346.16	331.18
ForceCalc	29,552.70	144,367.00
Move Bodies	176.14	127.18

3.3 Challenges in Using OpenMP for Scalability

OpenMP provides a convenient easily programmable means of harnessing loop parallelism in applications that can benefit from such parallelism approaches. The key issue to note in OpenMP is that each of the thread executes the same set of instructions on a distinct data set. For a simulation comprising N bodies utilizing p OpenMP threads for parallelism, each of the OpenMP threads has an approximate workload of N/p. In the context of the Barnes Hut N-Body simulation each OpenMP thread operates on a N/p of the total bodies in the system. A more uniform workload algorithm such as the exhaustive $O(N^2)$ can utilize the loop parallelism approach more effectively as each thread is operating on a predictable constant work load. Consequently each of the processors allocated to the parallelization would have the same workload and would be efficiently utilized.

However in Barnes Hut N-Body algorithm each of the bodies interacts with a varying subset of bodies (see fig 3.10). Consequently the workload per body in a Barnes Hut simulation varies. Even though OpenMP loop parallelism allows for effective parallelization of the pool of bodies, each of the OpenMP threads operates on a different workload. For large N-Body systems where each of the bodies interacts with a varying subset of the tree, resulting in variable workload on a per body basis, conventional parallelism techniques result in less than efficient utilization of the underlying parallelization causing some of the OpenMP threads to finish execution sooner than the rest. For extreme scale systems inefficiency of load balancing severely constrains the scalability of the simulations.

```
p : 0 list  :  1 2 4 5 7 8 10 11 15 17 18 19 21 22 26 27 28 30 31 32 35
p : 1 list  :  0 2 4 5 7 8 10 11 27 28 30 31 32 35
p : 2 list  :  0 1 4 5 9 12 15 17 18 19 21 22 29 36
p : 3 list  :  0 1 2 5 7 8 10 11 15 17 18 19 21 22 26 27 28 30 31 32 35
p : 4 list  :  0 1 2 4 9 12 15 17 18 19 21 22 29 30 31 32 35
p : 5 list  :  0 1 2 4 5 8 10 11 13 14 17 18 19 21 22 26 27 28 30 31 32 35
p : 6 list  :  0 1 2 4 5 7 10 11 13 14 17 18 19 21
p : 7 list  :  0 1 2 18 19 21 22 26 27 28 30 31 32 35
p : 8 list  :  0 1 2 4 5 7 8 10 15 17 18 19 21 22 24 25 27 28 30 31 32 33 34
p : 9 list  :  0 1 2 4 30 31 32 35
p : 10 list :  0 1 2 4 5 7 8 10 21 22 26 27 28 30 31 32 35
p : 11 list :  0 1 2 4 5 9 10 11 15 18 19 21 22 24 25 27 28 30 31 32 33 34
p : 12 list :  0 1 2 4 5 9 12 15 17 19 21 22 24 25 27 28 30 31 32 35
p : 13 list :  0 1 2 4 5 9 10 11 15 17  24 25 27 28 30 31 32 33 34
p : 14 list :  0 1 2 4 5 9 10 27 28 30 31 32 33 34
p : 15 list :  0 1 2 4 5 9 12 15 17 18 19 21 26 27 28 30 31 32 35
p : 16 list :  0 1 2 4 5 9 12 15 17 18 19 21 22 25 27 28 30 31 32 33 34
p : 17 list :  0 1 12 15 17 18 19 21 22 24 27 28 30 31 32 33 34
p : 18 list :  0 1 2 6 9 12 15 17 18 19 21 22 24 25 28 30 31 32 33 34
p : 19 list :  0 1 2 4 5 9 19 21 22 24 25 27 30 31 32 33 34
p : 20 list :  0 1 2 4 5 9 10 11 15 17 18 19 21 22 24 25 27 28 31 32 33 34
p : 21 list :  0 1 2 4 5 9 12 15 17 18 19 21 22 26 27 28 30 32 33 34
p : 22 list :  0 1 2 4 5 9 10 11 15 17 18 19 21 22 24 25 27 28 30 31 33 34
p : 23 list :  0 1 2  22 24 25 27 28 30 31 32 34
p : 24 list :  0 1 2 4 5 9 12 15 17 18  28 30 31 32 33
```

FIGURE 3.10: Exemplar Interaction List for a 25 Body Barnes Hut Simulation

3.3.1 Factors Affecting Scalable N-Body Computations

In order to provide scalability and performance for Barnes Hut N-Body computations critical challenges will need to be addressed. Four major factors contribute to performance degradation with respect to ideal scalability must be addressed by any future architecture. These factors are: Starvation, Latency, Overhead, and Waiting (“SLOW”).

- **Starvation** is wasted processor cycles due to insufficiency of work at execution sites. This can be as a result of too little work because of inadequate workload parallelism or because of poor load balancing such that even if there is enough concurrency, the tasks to be performed are not evenly distributed across all the elements of the system leaving some parts with nothing to do while ironically other parts are overburdened. Starvation is not orthogonal to other factors of performance degradation because, the exposure of enough parallelism is in part limited by overhead, while load balancing can be hindered by latency. In the context of Barnes Hut N-Body simulations the per particle varied workload results in improper load balancing when conventional parallelization techniques are used resulting in some processors having too much work while others having too little (see Table 3.3). In the table for

TABLE 3.3: Workload Imbalance in OpenMP Barnes Hut Implementation

OpenMP TID	10000 Bodies (ms)	100000 Bodies (ms)
T0	5272.59	79218.80
T1	5330.93	79243.50
T2	5330.94	79285.30
T3	5331.11	79287.20
T4	5301.91	79294.20
T5	5302.36	79257.70
T6	5318.99	79219.60
T7	5331.02	79190.10

the 10,000 body simulation run, threads 0, 4, 5, 6 have distinctly different workloads when compared with the rest of the threads. For the 100,000 body simulation each of the thread has varying workloads while thread 7 has the least workload and therefore finishes executing faster than the other threads resulting in starvation for processor 7.

- **Latency** is the time distance often measured in cycles for the conveyance of information from one point in the system to another. If an executing element is blocked in its operation while a remote access or service request is being transferred through communication channels, then the latency of the transfer can directly impact total system performance.

- **Overhead** is the critical time path work required to manage the program concurrency, work that would not be done for a purely sequential version of the code. For strict scaling where the goal is to reduce the execution of a fixed size workload, overhead imposes a fundamental upper limit for fixed workload problems, as it determines the granularity of concurrent tasks that can be efficiently exploited.

- **Waiting** can be due to delays caused by contention for shared resources and can result in hot spots such as memory bank conflicts or network routing bottlenecks. Dynamic adaptive techniques for working around the points of contention and avoid the resulting waiting and lost cycles are essential for all but the most predictable problems.

3.3.2 Goals for Extreme Scale N-Body Simulation

In order to leverage the multi-billion-way parallelism offered at Exascale new ways of managing computation effectively will be needed with the implicit goal of ameliorating the effects of performance degradation factors. To manage non uniform workloads of Barnes Hut algorithm and other more dynamic graph applications new ways of synchronization and mechanisms to better expose the underlying parallelism to the applications will be needed. The new model of computation capable of enabling performance at Exascale will be dramatically different from the current models, while exhibiting certain shared properties of user programmability and application suitability. The objectives of the new model are briefly described:

- **Goal 1:** A new model to serve the future HPC community must provide a discipline to govern future scalable system architectures, programming models and methods, and runtime system software. This model is a cross-cutting structure that influences the functionality and interrelationships of all system layers. Implicit in the model is a decision chain that establishes by intent or happenstance the means by which the time and place of every operation is determined through contributions of all layers of the system.

- **Goal 2:** A new model of computation must incorporate latency hiding as an intrinsic property to attack latency, a major factor of inefficiency. Prior models have dealt with latency largely through avoidance and the expectation of success of cache based systems. System wide latencies cost tens of thousands of cycles and consequently strong mitigation strategies need to be implemented. Among these are the general class of methods known as "latency hiding" that permits the overlap of computation with communication.

- **Goal 3:** A new model must provide a framework for efficient fine grained synchronization and scheduling. On the order of approximately a billion-way parallelism will be required to deliver the anticipated 1 Exaflops peak performance and an additional one to two orders of magnitude of parallelism to hide system-wide latency for 1 Exaflops sustained performance. This degree of concurrency is 1000X greater than the largest systems of today, hence requiring new ways of exposing and exploiting application parallelism for performance, efficiency, scalability, and even programmability.

- **Goal 4:** A new model must provide light-weight synchronization mechanisms to enable performance oriented scalability. Lightweight synchronization reduces overhead of control to allow effective use of fine-grain parallelism. Often, although not always, this results in the availability of more concurrency than would otherwise be possible. It also reduces the over constraining operational characteristics of global barrier making possible superior scheduling of tasks and improved resource utilization.

- **Goal 5:** The new execution model must embody a framework that permits runtime system conditional information to be incorporated in the decision making process. Not all task scheduling decisions can be made optimally at compile time or load time, requiring instead information about the runtime status of the system performing the application. On the fly dispatch can dynamically adapt to unpredictable circumstances resulting from intermediate

results of the computation and from low-level system behavior outside the control of the computation (e.g., network congestion).

- **Goal 6:** The new execution model must provide a framework for power management and reliability of operation in the presence of faults. Both factors will limit the scale of such future systems and will require innovative approaches represented by aspects of the selected model of computation. Integration of a new strategy for dealing with faults including detection, isolation, and recovery, in to the execution model itself will allow all parts of the system hierarchy to contribute to satisfying these stringent requirements.

Chapter 4

ParalleX Execution Model: Enabling Data Driven Computing

Continued progress in Barnes Hut N-Body simulations and dynamic graph application domains will require new methods of exploiting parallelism from advanced technologies to increase scalability. As technologies advance towards enabling Exascale computing, major challenges of scalability, efficiency, power and programmability will need to be addressed. New models of computation will be needed to provide the governing principles for co-design and operation of the cooperating system layers from programming models, through system software to process or architectures. The following sections describe proposed model of execution, ParalleX, to address these challenges.

The ParalleX execution model is derived with the goal of specifying the next execution paradigm essential to exploitation of future technology advances and architectures in the near term as well as to guide co-design of architecture and programming models in conjunction with supporting system software in the long term. ParalleX is intended to catalyze innovation in system structure, operation, and application to realize practical Exascale processing capability by the end of this decade and beyond.

The four principal properties exhibited by ParalleX to this end are:

1. Exposure of intrinsic parallelism especially from meta-data to meet the concurrency needs of scalability by systems in the next decade,

2. Intrinsic system-wide latency hiding for superior time and power efficiency,

3. Dynamic adaptive resource management for greater efficiency by exploiting runtime information, and

4. Global name space to reduce the semantic gap between application requirements and system functionality both to enhance programmability and to improve overall system utilization and efficiency.

ParalleX provides an experimental conceptual framework to explore these goals and their realization through the concepts identified above to inform the community development of a future shared execution model. The evolution of ParalleX has been, in part, motivated and driven by the needs of the emerging class of applications based on dynamic directed graphs. Such applications include scientific algorithms like adaptive mesh refinement, particle mesh methods, multi-scale finite element models and informatics related applications such as graph explorations (eg chess [Sha50, KM75]). They also include knowledge-oriented applications for future web search engines, declarative user interfaces, and machine intelligence. The strategy of ParalleX is to replace the conventional static Communicating Sequential Processes model [BHR84] of computation with one based on a dynamic multiple-threaded work-queue processing model discussed above. ParalleX also incorporates message-driven computation through Parcels, an advanced form of active messages.

This section discusses the ParalleX execution model in detail. ParalleX can be viewed as a combination of ParalleX Elements, and ParalleX Semantic Mechanisms. ParalleX Elements are the fundamental building blocks of the execution model comprising ParalleX Threads, ParalleX Parcels, Localities and ParalleX Processes. ParalleX mechanisms are complex interactions enabled by the structure, semantics of the ParalleX elements, and the dynamics involved in the execution model (see table 4.1.

4.1 ParalleX Elements

ParalleX Elements are the fundamental building blocks managed by the mechanisms. The elements comprise: ParalleX Threads, Parcels, Processes, LCOs, and Localities.

TABLE 4.1: ParalleX Concepts Summary

ParalleX Elements	ParalleX Mechanisms	Emergent Properties
ParalleX Threads	Multi-Threaded Work Queue	Percolation
Parcels	Message Driven	Security
Local Control Objects	Synchronization	Adaptive Scheduling
ParalleX Processes	Active Global Address Space	Self Aware
Localities	Overlap Computation and Comm.	Split Phase Transactions

4.1.1 Locality

Like many models dealing with distributed concurrent computing, ParalleX recognizes a contiguous local physical domain. In the case of ParalleX, a Locality is the locus of resources that can be guaranteed to operate synchronously and for which hardware can guarantee compound atomic operation on local data elements. Within a locality, all functionality is bounded in space and time permitting scheduling strategies to be applied and mechanisms to build structures that prevent certain pathological cases like race conditions, can be built. The locality also manages intra-locality latencies and exposes diverse temporal locality attributes. A locality in ParalleX is a first-class object, meaning that each locality in a ParalleX system has a globally unique identifier associated with it and can be addressed throughout the user global address space. The locality comprises heterogeneous processing units for example, a combination of homogeneous multicore processing elements and GPGPU accelerators. As an approximation the classical node (comprising processors, memory banks, network interface cards etc) can be considered to form a locality in ParalleX.

4.1.2 ParalleX Threads

The fundamental element that accomplishes computational work is, as is referred to in many programming models, the complex or more commonly “thread”. Like more conventional threads in other models, ParalleX confines the execution of a thread to a single localized physical domain, conventionally referred to as a “node” and here as “locality” in ParalleX terminology. A thread is a locus of local control and data for execution of a set of instructions.

Multi-threading is the concurrent execution of multiple threads by the local resource ensemble supporting thread execution. The relaxed definition of thread in the context of multi-threading

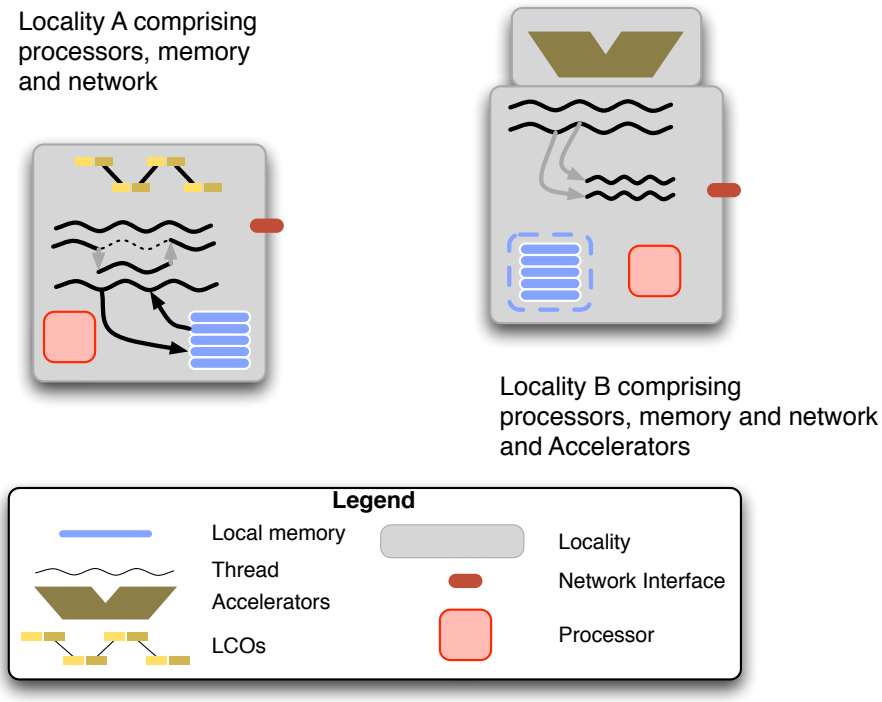


FIGURE 4.1: Two Exemplar Localities as Defined by ParalleX

requires only that one operation must be issued in a given cycle by any one of the multiple concurrent threads supported by the local multi-threaded execution resource ensemble. While conventional use of the term "thread" assumes purely sequential execution of thread instructions, actual practice in modern computer architecture attempts concurrency of thread instruction execution through a number of mechanisms. The flow control representation schema of a thread is restricted in definition only to the point that the principle of collocation is satisfied; i.e. an operation is not performed until its constituent information elements are relatively local with respect to the coupled physical resources supporting the execution. For this model, a thread is stationary; it does not move or migrate among execution resources. Specifically, a thread, once instantiated, does not traverse between domains of physical resources. This guarantees certain performance properties attributed to relaxed local thread execution as described above. The creation of a thread is a product of a communicated synchronization event, potentially influenced, at least in part, by one or more events occurring in non-local resource domains. A thread terminates when it can no longer continue execution within the physical and logical resources of the relaxed local domain in which

it resides. Alternatively, a thread can also suspend activity when remote data fetch operations or data dependencies on local operations exist. This behavior prevents a ParalleX thread from blocking a valuable resource (CPU) from being utilized for other work in the work queue. A ParalleX thread could also contain a continuation [AJ89], which specifies actions once the current thread executes its workload. Differing from other thread models such as Pthreads [Alf94], ParalleX

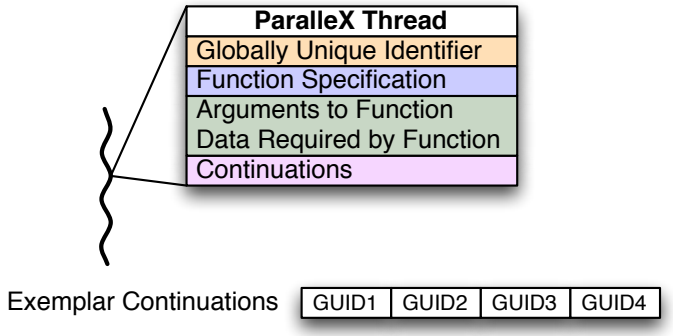


FIGURE 4.2: Schematics of a Thread as defined by ParalleX

threads are ephemeral: they can be instantiated at any time and terminate likewise at any time. They are heterogeneous in that threads of many different forms (code) can be instantiated at the same time with no restrictions. Both these properties distinguish ParalleX threads from those of UPC [EGCSY03, EGS06] for example. A particularly important distinguishing aspect of ParalleX threads is that they are first class objects; they are named as is any other variable or object and may be manipulated within the limitations of their type class from any part of the allocated system. ParalleX threads may be as short as a compound atomic operation on a given data structure element or as long as a conventional process that lasts the entire duration of the user program. The intended mode of operation however is for short lived threads that operate exclusively on private or local data within a single locality and then pass on the flow of control to a remote locality where the transaction of execution is to continue based on data placement. However it is not always optimal to move work to the data and sometimes a more conventional gather operation requiring one or more data elements from one or more remote localities is the better approach requiring a thread to be suspended. To achieve this a ParalleX thread which is unable to proceed in its execution

due to incomplete control state transitions (e.g., a required intermediate value is still pending) is "depleted" (it has no more work it can do) or suspended in the common terminology and is transformed in to a new kind of object with the same name. This is one of a general class of lightweight objects that includes control state referred to in ParalleX vocabulary as a "Local Control Object" or "LCO" described in section 4.1.4. In doing so the ParalleX thread saves its requisite private state and relinquishes the physical resources for other active threads in the locality. A ParalleX thread differs in one last way from more conventional threads in that typically a thread is assumed to be sequential in its instruction issue. ParalleX incorporates an intra-thread instruction precedent constraint based flow control schema for intermediate values not unlike static dataflow [Den80]. This eliminates anti-dependencies at least for intermediate (intra-thread) values allowing generalized application to a diverse set of processor core instruction set architectures; thus facilitating the use of heterogeneous computing systems. ParalleX threads support parallelism at two levels: concurrent threads and operation level parallelism within a thread through the dataflow control.

4.1.3 ParalleX Parcels

ParalleX embodies a powerful, general, and dynamic method of message-driven computation called Parcels that allows the invocation of threads on remote localities through a class of message that carries information about an action to be performed as well as some data that may be involved in the remote computation. Parcels (PARallel Control ELements) are a form of active message [ECGS92] that extend the semantics of parallel execution to the asynchronous domain for scalability and efficiency. The parcel supports message-driven computation allowing work to be moved to the data rather than the data always gathered to the localized site of the work. Parcels reduce both bandwidth requirement and hide latency for greater efficiency and scalability. Parcels extend the effectiveness of the work-queue model to operate across localities, not just within a single locality, as would be the case without them. A parcel carries four kinds of basic information that enables message-driven computation: destination, action specification, argument values, and continuations (see fig 4.3). The destination is the global address of the entity upon which the parcel is to be ap-

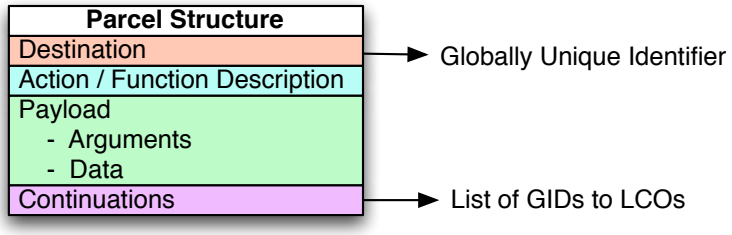


FIGURE 4.3: Parcel Structure in ParalleX

plied. It can be as simple as the virtual address of a scalar variable or as complicated as the virtual name of an entire process. It can also be a physical location such as a low-level counter associated with a piece of hardware in a locality. The action specification defines the task to be performed at the site of the destination object. Actions triggered by the incidence of a parcel are of a diversity of forms. They may constitute RDMA (remote direct memory access) for data movement across the system. Simple actions may be performed atomically such as the simplest Test-and-Set or more complex compound atomic operations on LCOs. A number of actions are thread related. These may be updates to thread state like loading a register and the control state associated with the load. They often cause a new thread to be instantiated. Or, they may manipulate a thread in its entirety such as affecting its scheduling sequence or killing it. Actions may be performed directly on the system hardware such as setting control bits and reading status registers.

Parcels can carry argument values as payload that will be used in the computations performed. For large data movements this is the principal purpose of the parcel. For most other cases, this is just a small part of the entire functionality. Under certain circumstances, the type of the values carried is “instruction”, that is it provides details of an action to be performed beyond the capability of the action field itself. The final form of basic information carried by a parcel is the “continuation” ; a means of expressing what is to happen after the parcel action has completed. This may be as simple as a return-to-sender typical of many models. But ParalleX parcels are far more sophisticated and tie in to the pan-system web of LCOs that make up much of the distributed control state of the system parallel execution. Continuations often are references to a LCO or list of LCOs within the system, modifying the system control state to reflect the completion of the parcel specified action.

It is interesting to note that parcels are among the few entities within the ParalleX framework that are not first class objects. This is an example of how ParalleX manages policies, operational invariants that must be satisfied but for which great latitude is provided the implementer in how this is to be achieved.

4.1.4 Local Control Objects

One of the most powerful aspects of the ParalleX model is the way it represents and manages global flow control. Typically, distributed applications employ the construct of the global barrier to manage phased execution in what is referred to as the Bulk Synchronous Protocol or BSP model [Val90]. Simply, BSP uses a compute-synchronize-communicate phased management approach where the exchange of data via messages only occurs after all processes have finished their respective work for the current cycle. A global barrier is a synchronization object that ensures the strictness of this ordering requiring all processes to check in to the barrier upon completion of their work and to not proceed with their communication phase until notified to do so by the barrier (which makes certain that all processes have finished). For many applications and systems this has proven an effective approach. But this coarse grained parallelism precludes the more fine grain and adaptive parallelism required for extended scalability.

Local Control Objects (LCOs) are event driven conditional structures that under specific satisfaction criteria will cause a thread or some other action to be instantiated. Events are either direct accesses by threads within the same locality and process, or through parcel messages outside these restricted domains. Local Control Objects improve efficiency by avoiding barriers and permitting highly dynamic flow control. Among the simplest form of an LCO is a mutex [CHP71]. But the more interesting forms include the dataflow template and the futures [BH77] construct (initially specified as part of the Actors model[HB78] and used as a key element of Multilisp [RHH85]). A future is a promise of a returned value, which may be any first class object (see fig **??**). If the value itself is not required such as manipulating the metadata surrounding it, then computation may proceed. Only when the actual value is required and not just its placeholder will further computation

be blocked. But futures are far more powerful. They can coordinate anonymous producer-consumer computation such as histogramming and can also coordinate access to a shared structure such as a vertex of a graph. As mentioned above, a depleted thread is an LCO. Although derived for very different purposes, it turns out that the semantics and structure of a depleted thread is within the scope of the general class of LCO objects. One example of this usage is in a search tree (such as used in the game of Chess) where the vertices of the graph are depleted threads. LCOs provide lightweight synchronization reducing the overhead of synchronization and removing over constraining global barriers. This can be particularly important where the inner loops of computations due to nonlinearities exhibit highly non-uniform execution times. Barriers will call long idle times for some processor cores while effectively employed LCOs can permit overlapping of successive phases of computation to achieve far greater utilization than would otherwise be possible with BSP-based approaches. LCOs may greatly increase the total average parallelism available and therefore the potential scalability of future systems based on the ParalleX model of computation while dynamically accelerating asynchrony of operation.

4.1.5 Processes

Conventionally, a process is a form of task or action comprising many specified steps or instructions. A process generally runs on a processor core and perhaps communicates with other processes via messages in the system I/O distributed name space. ParalleX view processes as contexts containing names entities including data constants and variables, concurrent threads both active and deleted, local control objects, and other child processes. Processes also enable "symmetry" of semantics that allow a user to perform remotely, the activities that an application would perform locally. The important property of a ParalleX process that distinguishes it from conventional process oriented models is that it can occupy many localities (remember these are system nodes) at the same time using the combined resources of the collection of assigned localities to perform the process active threads and storing the process data. Like threads, processes are ephemeral. A process is instantiated from a combination of a procedure specification (or just "procedure"), a set of

operand values, and allocated physical localities at any point in time of the execution of a parent process and may be terminated either by an internal "end" statement of one of its internal threads or by a "kill" statement from one of its parent process threads.

A ParalleX process is both imperative and object oriented. It can be instantiated (as just stated) through the use of argument values in a class sequence. Alternatively or in addition (both are valid), a method in a process may be called by a thread from a sibling, cousin, or parent process. In so doing, a thread within the called process is instantiated from the method to process data within the context of the called process. As a consequence, a process can serve in diverse modalities as distinct as a pure function (value in, value out) or a pure abstract data structure. It also enables dynamic resource management throughout the execution of the incorporating application. It should be noted that when a job is created by an operating system in response to a user request, an outermost process is activated. This master-process is called "main" (from the obvious source) and is assigned all localities and their internal resources that the application may employ during processing. "Main" is the exception to the rule; it is not ephemeral but must be created at the beginning of a program execution and remain active through to the end of the program. It has implicit child processes for general interaction with the system operating system and for specific I/O functions but is also a child process of the operating system.

4.1.6 Active Global Address Space

ParalleX supports a shared memory model that distinguishes it from the conventional distributed memory MPP and cluster system architectures, but is also different from the cache-coherent shared memory systems such as SMPs. It incorporates a strategy among the class of "Global Address Space" models sometimes referred to as put-get models. The ParalleX approach, referred to as "Active Global Address Space" or AGAS differs from the more widely employed model called Partitioned Global Address Space in one specific and important way. With PGAS [Bon02] a virtual object is restricted scope of allocation within the confines of a particular partition. This is because its virtual name includes information about its physical location; in particular its resident physical

partition. The value of this is that it makes address translation and request routing very efficient. But it is restrictive on the runtime distribution of global objects. If for any reason one absolutely had to move a selected object to another partition, its virtual name would have to be altered to coincide with its new physical location. ParalleX, at the risk of additional overhead, relaxes this constraint. It permits virtual objects to be moved across the system between localities while retaining their original virtual addresses. This is the "active" aspect of AGAS versus the passive or static nature of the "partitioned" aspect of PGAS. This is critical for dynamic directed graphs and for load balancing as well as system reconfiguration.

4.2 ParalleX Mechanisms

The real power comes from combining constituent ParalleX elements such as ParalleX Threads, Parcels, LCOs, and Processes using dynamic mechanisms that take advantage of various features to deliver an innovative execution model. threads with these mechanisms, namely Work-queue Scheduling, Split-phase transactions, and message driven semantics are described in greater detail below.

4.2.1 Work-queue Scheduling

To achieve higher efficiencies than conventionally realized, increased utilization of the primary computing resources is required. Low utility is a consequence of blocking on some statically scheduled computation waiting for a result, service, or resource. Higher utilizations may be achieved by decoupling the blocking of a computational task from the use of the physical resources. An important opportunity for achieving this is to employ the work-queue strategy for resource scheduling. The work-queue model has an execution resource such as a processor core work on a piece of computation until it is either completed or blocked. In either case, a new piece of pending computation is immediately scheduled on the same resource assuming there is sufficient parallelism to keep an input stream active. Separating the physical hardware from the abstract computation combined with dynamic resource management opens a diversity of opportunities for more efficient use of

systems and improved scalability. One way of using a work-queue strategy is that of split-phase transactions.

4.2.2 Split Phase Transactions

A transaction is a set of interdependent actions on shared values. Processes and threads on conventional systems are examples of such transactions. But on distributed systems a transaction may have its working data set spread across multiple nodes or localities. Such transactions may be divided into phases, each phase operating on the data of only a single node, thus making all accesses local within a particular transaction phase. If flow control is transferred to the node housing the data of the respective transaction phase, then all accesses of the transaction performed may be relatively local. This can dramatically reduce the average access latency time. The only distributed activity is the actual transfer of control between successive phases to the next node. This class of operation is referred to as"split-phase transactions" and is a powerful way for new models of computation to improve system operational efficiency.

4.2.3 Message-Driven Computation

To enable the above and other important concepts, a transition from conventional message passing semantics and mechanisms to an alternative form in support of message-driven computing provides an opportunity. Message-driven computing has been a subject of study and experimentation for three decades. The Actors model [HB78], Dataflow [AN90, Den80], the J-Machine [NWD93], Active Messages [ECGS92], and most recently Percolation [GT97] have studied the possible forms and usage of messages that cause actions to occur at remote sites within a distributed system. Most commonly referred to as "Active Messages", this capability allows computing work to be moved to the remote site rather than requiring that all remote data be gathered and moved to the site of fixed flow control (as is usually done). Message-driven computing provides a means of managing system asynchronous operation if combined with multithreading on top of a work-queue scheduling model. Together they provide an important means of hiding system-wide latency and adapting to contention for shared resources sometimes referred to as "hot spots". Such messages, in

addition to carrying a payload of data also incorporate a representation of an action to be performed at the designated remote site. This action specifier may provide an identifier for a function or method thereby enabling full generality of remote action as defined by user programs. The use of message-driven computation has found application in the domain of "grid" computing involving widely distributed systems and the use of very coarse-grained procedures through RPC actions or remote procedure calls.

4.2.4 The ParalleX Model: Assembling The Parts

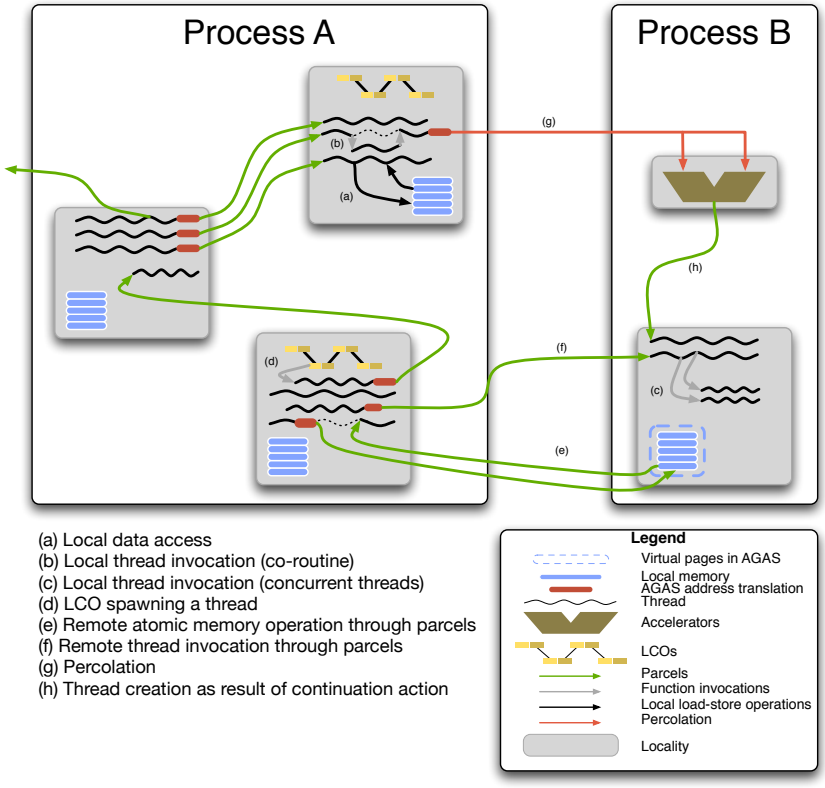


FIGURE 4.4: ParalleX Mechanisms

This diagram illustrates many of the key semantic constructs and mechanisms of the ParalleX (PX) model of computation. The shaded squares represent physical localities. All operations within a locality are synchronous. All operations between localities are asynchronous. A process is a context that defines data and tasks to be performed on the localities to which it is assigned. A process may be allocated multiple localities; these are not in any way mutually local to each other.

More than one process can share a locality (not shown). Process A is allocated three localities while Process B is on one locality but also uses a separate physical execution unit. Processes can incorporate child processes (not shown). Processes are illustrated by large rectangular transparent boxes.

Basic tasks in the ParalleX model are conducted by "threads" which are partially ordered instructions working on a potentially infinite set of registers. This convenient abstraction minimizes the number of anti-dependencies to permit easy back end compilation to diverse ISA and data paths for different cores. In the diagram, the threads are represented by black worm-like "squiggly" lines. While a process may span multiple localities, a thread by definition resides on only one locality at a time and usually (but not necessarily) on a single locality throughout its lifetime. A thread may be suspended but the ideal thread is relatively lightweight, works only on local data and terminates quickly. The effect of a thread may be manifest in several ways: 1) it changes local mutable state, 2) it creates child threads (or processes) that outlive the parent thread, or 3) it creates parcels that invoke remote threads (or processes) or change remote mutable state. Threads can be manipulated by other threads potentially of other processes. Threads can be suspended to form objects called "depleted threads". Threads perform almost all the work of the program. The exception is the parcel that directly effects remote state without invoking a thread.

Parcels are illustrated by the green lines that convey effect between localities. Parcels support semantics at remote localities that threads can perform directly within their resident locality. These are creating remote threads or processes, changing remote state through compound atomic operations, or manipulating threads or processes directly. A special form of the parcel is used for a methodology called percolation. Percolation moves the entire work foot print of a thread to a remote physical execution site. It is illustrated by the red lines between thread and functional unit. By foot print it is meant to include the instructions of the thread, the stack frame of the thread, the input arguments used for the thread, and any other information required to fully instantiate the thread on the remote execution site. Percolation is used to make efficient use of relatively precious

resources like accelerators, GPUs, FPGAs, or expensive streaming processors. It does so by assuming the overhead and hiding the latencies that such a unit would otherwise have to incur. Thus percolation supports pre-staging of execution information. Percolation is also used to dynamically allocate new resources within a system to an application program.

A critical element to the management and coordination of global parallel flow control is the Local Control Object or "LCO" shown as small yellow/brown-green boxes. An LCO is a synchronization object in the object-oriented sense. It has associated with it both state and methods that operate upon the state. The purpose of the LCO is to coordinate global flow control by means of a number of different types of synchronization mechanisms from simple mutex elements to sophisticated dataflow templates and futures constructs. An LCO is accessed like any other object locally by a thread or remotely by a parcel possibly changing its state. An LCO includes one or more criteria which if satisfied will result in the instantiation of a thread. LCOs can also buffer incoming requests until such a condition can be satisfied. This may include the availability of physical resources. The LCO is the embodiment of a continuation. The continuation field of a parcel with some exceptions is a reference to one or more LCOs.

4.3 Processing of Dynamic Directed Graphs Using ParalleX

A ParalleX runtime system provides several key mechanisms that would enable processing of dynamic directed graphs. The mapping of key enabling ParalleX runtime characteristics to the requirements for directed graph processing are listed in Table 4.2.

In ParalleX, threads and LCOs are first class entities, additionally ParalleX threads also support suspending threads and resuming them via local or remote invocations. This combination of elements and mechanisms is one of the key building blocks in enabling data driven computing.

ParalleX semantics also allow for direct creation of suspended threads which are also globally addressable in AGAS. This allows for a large number of potential exploratory paths to be created during runtime allowing the computation to decide the exploratory paths it can take based on the resources available.

The ParalleX work-queue model enables "split-phase transaction processing", which in applications where sufficient fine grained parallelism exists can result in better utilization of resources by overlapping computation and communication. It also provides dynamic adaptive task scheduling through its asynchronous flow control inter-node exchange strategy with local task switching determined by the next available action to be performed. This asynchrony of global flow control has more impact than merely covering variation in network latency; it dynamically adapts to changes in execution that is data dependent, such as dynamically evolving directed graphs.

Additionally the ParalleX runtime system supports offloading key system operations such as thread scheduling and AGAS management to FPGAs where available through percolation. Using the same mechanisms ParalleX also allows offloading of core application logic to special purpose accelerators such as GPGPUs. One example of this is offloading of chess move generation to a special purpose accelerator there by making available the core CPU resources for other key computations in the chess game. This has the potential to improve the overall performance of the applications.

Combination of these key driving elements and mechanisms, that expose the underlying resources to the computation using an interface allows the computation to determine its best course of action based on resource constraints, makes ParalleX a very compelling alternative to conventional models. For these reasons ParalleX has been chosen as the runtime system enabler where the scheduling policies that govern scalability of directed graphs will be implemented.

4.4 ParalleX Related Works

Charm++ [KK93] as a runtime system has many similarities to the HPX runtime system in its underlying goals and some semantics. The Charm++ model of computation is similar to the Actor-Model [HB78] (based on Chares), but extends it by providing a variety of other data sharing mechanisms (branch-office chares, monotonic variables, accumulators, etc.). Charm++ uses an IDL to generate code, which is used by C++ code to generate the parallel programs. Charm++ has a notable sizable user community that includes bio-molecular code (NAMD [KBB+98]), rocket simulations, cosmology (ChaNGa [JGM+08]), and quantum chemistry (OpenAtom [BBK+08]).

TABLE 4.2: Overview of ParalleX mechanisms that enable dynamic directed graph processing.

Dynamic Directed Graph Requirements	**ParalleX Enablers**
Data-driven computing	Parcels carry data, target address, action, continuation Allows work to be moved to data Enables migration of flow control Supports asynchronous event driven execution
Support for fine-grained parallelism	Work queue model enables split phase transactions i.e overlapping computation with communication Threads support dataflow on transient values
Dynamic thread support	multithreaded light weight entities First class objects Ability to directly create and address suspended May be compiled to different types of processors
Global memory access	PX communication using AGAS and Parcels non cache coherent global memory space supports copy semantics and affinity relationships

While both systems support migrating objects in a global address space (AGAS), and both use a form of active messaging, HPX, prefers to supply synchronization at a lower level than the Actor-Model in which events received by an object are ordered. Instead, it provides a wide variety of synchronization mechanisms to handle parallel event processing on the same object, namely: mutexes, semaphores, futures, dataflow controls, etc. Moreover, the form of active messages used by HPX (Parcels) provides a slightly richer semantics in that they support continuation lists. ParalleX model represents synergistic combination of cross-cutting elements across the vertical systems stack, while the Charm++ as a runtime system exposes parallelism through a horizontal layer in the systems stack.

Cilk++[cil09] supports light multithreading, simple synchronization and work-stealing model of thread scheduling. Cilk++ is designed to work within an SMP, and not for distributed computing. The threading model implemented in Cilk++ is probably the fastest and lowest overhead multi-threading model in comparison with standard threading techniques and represents a performance goal for the future ParalleX runtime. Unlike Cilk++, ParalleX supports multiple localities and has an advanced GAS system.

UPC [EGCSY03, EGS06], CAF [NR98], and Titanium [YSP+98] all fall under the category of PGAS programming models. They provide a language solution to enable shared memory programming where objects are statically mapped. In ParalleX (as in Charm++) physical and virtual name spaces are decoupled, enabling adaptive resource management and load balancing. UPC implements multi-threading partially; it supports static threads and an SPMD model. Its primary strength is the lightweight global memory access (PGAS). Within UPC, one can implement some lightweight synchronization objects, although this only will manage blocking and unblocking of existing threads. However, it is not able to adopt:

- Dynamic threads
- Message-driven computation
- Latency hiding, except through locality management by programmers
- Parallel processes for virtual to physical memory mapping,
- True MIMD or diverse threads, where different processors can be executing substantially different tasks.

Chapel [CCZ07] is a PGAS language, but does not make the distinction between local and remote access explicit at the source level. Instead, it provides key abstractions to help the system make the decisions between local and remote access, as well as associated optimizations, at run time.
Linda [DS92] programs coordinate operations by means of atomic insertions, reads, and evaluations of data in a global tuple space. Processes do not need to be specifically aware of each other, removing the need for complex message passing code. Implementations of Linda are available in a number of languages and it should be possible to provide one in ParalleX for a certain class of applications .

Chapter 5

High Performance ParalleX Barnes Hut Implementation

This chapter presents the High Performance ParalleX runtime software system which serves as a ParalleX reference implementation. It is used to develop the research implementation of the Barnes Hut N-Body algorithm. The first half of this chapter describes the key implementation details of the High Performance ParalleX runtime system. The rest of the chapter discusses the implementation of the Barnes Hut N-Body algorithm using the HPX runtime system.

5.1 High Performance ParalleX Runtime System

The High Performance ParalleX (HPX) runtime system provides the experimental platform that embodies the key semantic principles and mechanisms of the postulated ParalleX Model. HPX 3, the current version, leverages the experience obtained from developing earlier versions of runtime systems for parallel execution models. This C++ implementation represents a first attempt to develop a comprehensive API for a parallel runtime system supporting the ParalleX model. HPX provides a target for the development and implementation of the XPI specification which defines a low level API for ParalleX applications, very much as MPI [GL02, GLDS96] defines an API for communicating sequential processes [BHR84].

The HPX runtime system is devised as an alternative to conventional programming models, such as MPI, while attempting to overcome their limitations such as: global barriers, insufficient and coarse grained parallelism, and poor latency hiding capabilities. HPX provides a multi-threaded, message-driven, split-phase transaction, distributed shared memory programming model using Futures [BH77] and Dataflow [Den80] based synchronization on large distributed system architectures. This chapter describes the HPX runtime system in detail and in the ensuing sections explores the research implementation of Barnes Hut N-Body algorithm using HPX.

5.1.1 General HPX Design

The design objectives of the HPX library are derived from the ParalleX execution model. The most important objective of HPX is to design a state-of-the-art parallel runtime system providing a solid foundation for ParalleX applications while remaining as efficient, as portable, and as modular as possible. This efficiency and modularity of the implementation is central to the design, and dominates the overall architecture of the library (see Figure 5.1). The figure shows a block diagram of the current architecture of the HPX implementation. The HPX architecture incorporates the necessary modules and expresses an API to create, connect, manage and delete any ParalleX object from an application that togeather comprise all the features necessary for a runtime system like HPX to provide resource management.

The current implementation of HPX provides the infrastructure for the following ParalleX concepts:

- First class Threads and thread management
- Parcel communication and parcel management
- LCO‘s (local control objects) for lightweight synchronization
- AGAS (active global address space), including a global garbage collection scheme

The following sections will describe design considerations and implementation specific details for those elements.

5.1.2 The Active Global Address Space

The active global address space (AGAS) provides dynamic migration of virtually advanced objects in support of dynamic applications. The AGAS simplifies application development as it removes the dependency of codes on static data distribution and enables seamless load balancing of application and system data.

The abstraction of localities is introduced as a means of defining a border between controlled synchronous (intra-locality) and fully asynchronous (inter-locality) operations. Different localities

may expose entirely different temporal locality properties. The current HPX implementation interprets a locality to be equivalent to a "node" in a conventional system which is usually a cache coherent multicore SM (symmetric MultiProcessor). Intra-locality data access means that the application mainly accesses local memory, while inter-locality data access and data movement depends on the properties of the system network.

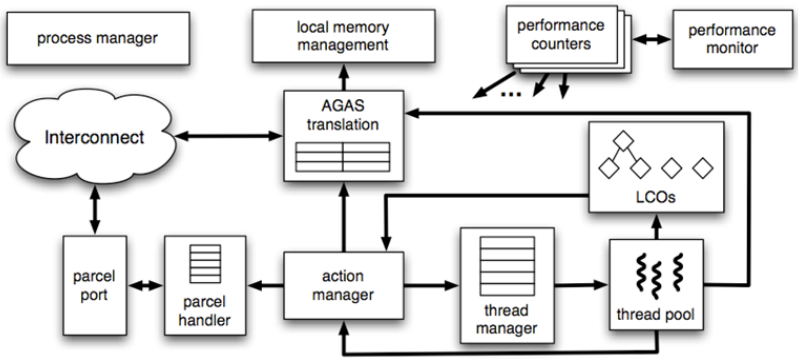


FIGURE 5.1: HPX Architecture

The Active Global Address Space (AGAS) assigns global names (identifiers, unstructured 128 bit integers) to all entities managed by HPX. The AGAS overcomes disadvantages of PGAS in prior systems such as X10 [CGS+05], Chapel [CCZ07], and UPC [EGCSY03, EGS06] by allowing objects to move in virtual space without having to rename the object or change the identifiers. AGAS provides a mechanism of resolving these global identifiers into the corresponding local virtual addresses (LVA's). LVA's comprise: the locality id, the type of the entity being referred to, and its local memory address. Moving an entity to a different locality updates the LVA-global identifier mapping, keeping all references to the moved item valid. The current implementation is based on centralized database storing the mappings which is accessible over the local system network. A local caching policy is implemented to reduce bottlenecks and to minimize the number of required network round trips. While this reflects the correct semantics of ParalleX, it does not provide the efficiency and scalability that will ultimately be required.

Other optimization strategies have been utilized, with the aim of minimizing the required database accesses. For instance, the current implementation allows for the autonomous creation of the glob-

ally unique id's in the locality where the entity is initially located. It supports memory pooling of objects of similar type which avoids additional overhead. The modular system architecture makes it possible to replace the client-server architecture in the future with a more scalable solution without modifying any of the other parts of the runtime system.

HPX implements a reference count based garbage collection scheme for all objects referenced by global names. HPX uses AGAS to store global reference counts for all global names. Additionally, HPX maintains a local reference count on each locality from which the object is addressed. The local reference counts keep track of copies of the global names as long as they are made locally. Whenever a global name is transferred to a different locality the global reference count is incremented. Deleting any last local copy of a global name on a locality results in decrementing the global reference count. A global reference count of zero results in freeing the reference object. This reference counting garbage collection is fully automatic and does not require any special handling in user code. It is based on C++ language features allowing to do automatic tracking of object construction, destruction, and copying.

5.1.3 HPX-Threads

HPX-threads are not based on operating system threads (Pthreads) but are implemented as lightweight user level threads. They are first class objects with immutable global names enabling remote thread management. Allowed thread states are: *initialized*, *pending*, *running*, *suspended*, and *terminated*. HPX thread semantics also allow the state of the thread to be changed remotely. Although moving a thread is theoretically possible, HPX-threads reside entirely within any single locality at a time. The preferred method of moving work is to send a parcel which creates a new thread continuing to work on the task at hand. Constraining threads to a locality is more efficient, especially on heterogeneous systems where moving threads is a very expensive operation. HPX threads maintain a thread state, an execution frame, and a (operation specific) set of registers (see figure 5.2). HPX-threads are cooperatively scheduled in user mode by a thread manager on top of a static OS-thread

(e.g., Pthreads) per core. The HPX-threads can be scheduled without a kernel transition, which provides is critical to efficiency and scalability for sustained performance.

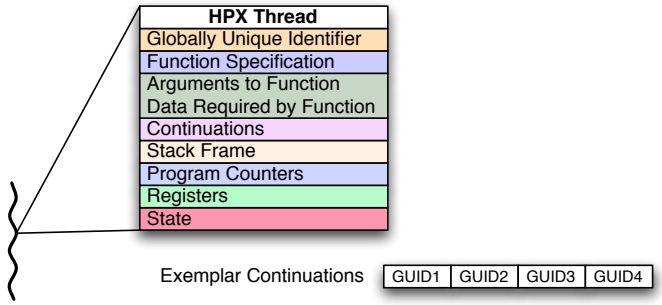


FIGURE 5.2: HPX Thread Structure

5.1.4 Thread Management

The HPX-thread manager is modular and can implement a work queue based execution model similar to prior systems (Cilk++ [cil09], TBB [Int10, tbb09], PPL [Mic10]). These are cooperatively scheduled in user mode by a custom thread manager on top of one operating system thread (e.g., Pthread) per available system core. A set of specially designed synchronization primitives (such as semaphores, mutexes, etc.) allows HPX threads to synchronize in a way similar to conventional threads. Currently the HPX thread manager is implemented in 3 key modalities.

- Global Thread Queue
- Local Thread Queue
- Local Priority Thread Queue

5.1.4.1 Global Thread Queue

In this modality of the HPX Thread Manager, all the cores in the locality access a global queue. The global queue operates using a first-in first-out (FIFO) scheduling policy where threads are executed in the order in which they enter the global queue (see figure 5.3). This is the simplest of queues; however, this approach is not scalable beyond 4 to 6 cores. Contention for the shared

global queue affects performance of the thread manager when implemented on a locality with more than 6 operating system threads due to contention for this shared logical resource.

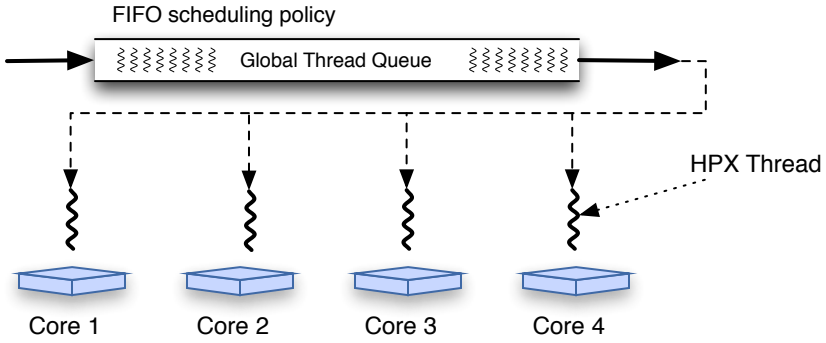


FIGURE 5.3: HPX Thread Manager Managing a Global Queue

5.1.4.2 Local Queue

In this modality of operation the HPX Thread manager creates a local queue for each of the cores in the system. Each operating system thread operating on each core is assigned user threads from the local queue specific to that OS thread (see figure 5.4). The Current implementation supports a FIFO scheduling policy on each core. Rudimentary support for work stealing is also implemented whereby an operating system thread operating on a core that has exhausted the local work queue can "steal" or acquire work from the front of the local queue of a neighboring core. Work stealing from the front of a neighboring queue can create undesirable side effects.

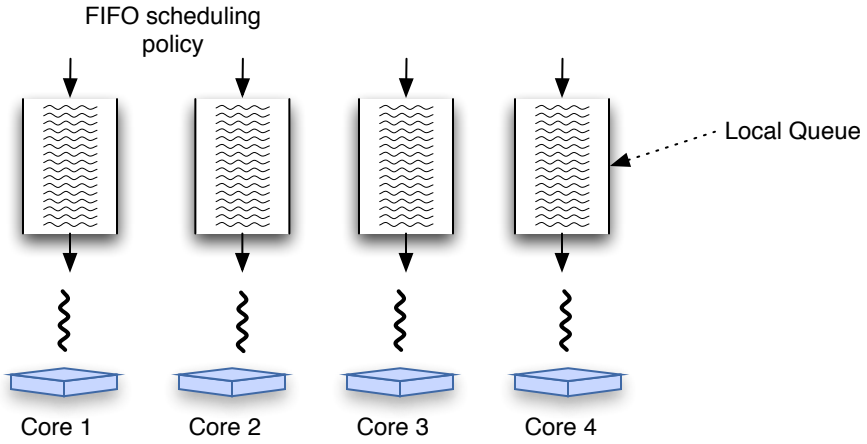


FIGURE 5.4: HPX Thread Manager Managing 4 Local Queues

5.1.4.3 Local Priority Queue

This modality of operation of the HPX Thread manager allows execution of high priority threads in a timely manner. Currently this mode of operation is an extension of the Local Queue approach. The main difference is that one of the cores has an additional high priority queue (see figure 5.5). Initial designs of the high priority queue were as a global priority queue; however, measurements demonstrated that the global shared priority queue becomes a source of contention as the number of cores increase, thereby decreasing performance of the overall system. In the current version of implementation the core hosting the high priority queue periodically checks the queue for high priority workload and makes the compute resource available for any threads in the high priority queue by suspending processing of the ordinary local queue.

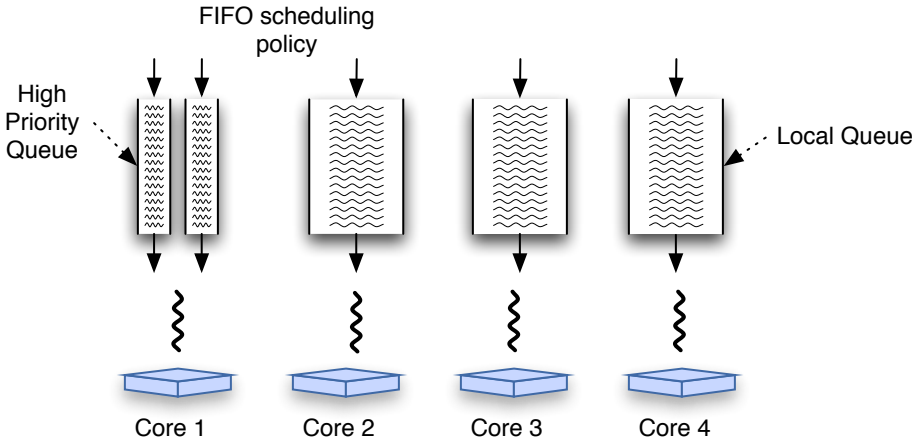


FIGURE 5.5: HPX Thread Manager Managing Local Priority Queues

5.1.5 Parcel Transport and Parcel Management

Any inter-locality messaging in HPX is based on Parcels an advanced form of active messages. In the HPX implementation, parcels are represented as polymorphic C++ objects. Parcels are serialized and de-serialized using a sophisticated serialization scheme which facilitates the binding of actions to execute at compile time. This highly optimal implementation removes the need to look up the function to invoke based on the action description stored in the parcel.

Parcels are sent to their destination using a parcel transport layer. Currently, it establishes asynchronous P2P connections between the source and destination localities based on the system network. The parcel transport layer not only buffers incoming and outgoing parcels, but it automatically resolves the global destination address of a parcel. Additionally, it either dispatches the incoming parcels to the local thread manager or forwards them in case the destination entity has been moved to a different locality.

In preparation to send a parcel from one locality to another the host locality parcel handler receives a parcel object from the entity that wants to send a parcel. The parcel handler creates a serialized parcel whereby all the dependent data is bundled along with the parcel. Serialization of the parcel allows for work to be moved completely along with its data to a remote destination. At the receiving locality the parcel is received using standard TCP/IP protocols in the case of Ethernet based interconnect networks. The action manager at the receiving locality receives the de-serialized parcel and creates HPX threads out of the specifications (see figure 5.6).

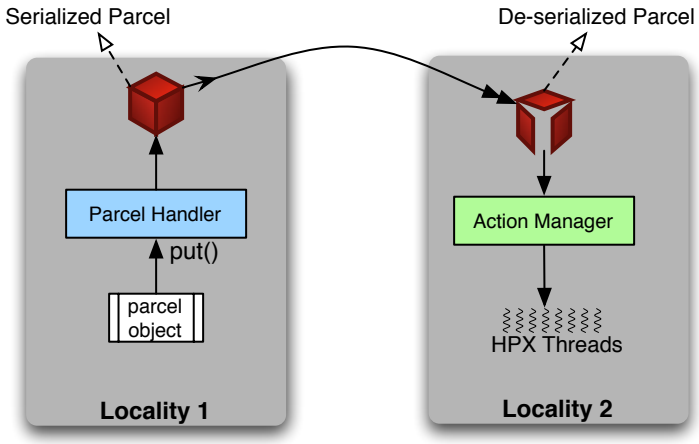


FIGURE 5.6: Parcel Management in HPX

5.1.6 LCO's - Local Control Objects

A local control object is an abstraction of different synchronization functionalities for event driven HPX-thread creation, protection of data structures from race conditions and automatic event driven on-the-fly scheduling of work. Its goal is to let every single function proceed as far as possible in

the computation limited only by its data precedence constraints. Every object which may create (or reactivate) a PX-thread as a result of some action exposes the necessary functionality of a LCO. LCO's are used to organize flow control. Parcel and thread continuations are implemented as lists of LCO's enabling remote thread creation or reactivation combined with the delivery of a result or an error condition after the remote operation has been executed.

HPX provides specialized implementations of a full set of synchronization primitives usable to cooperatively block a PX-thread while informing the thread manager that other work can be run on the OS-thread (core). The thread manager can then make a scheduling decision to execute other work. HPX-threads require these special, cooperative synchronization primitives as it is not efficient to use synchronization primitives as provided by the operating system because these are designed for OS-threads.

The current implementation of HPX supports several LCOs including mutexes, semaphores, conditions, semaphores, full-empty bits, futures, dataflow, work-queue, and barriers.

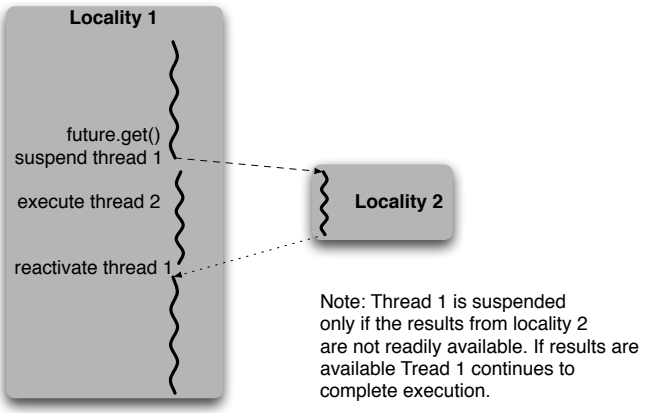


FIGURE 5.7: Futures LCO in HPX

5.1.6.1 Futures LCO

In HPX, a futures LCO refers to an object that acts as a proxy for a result that is initially not known, usually because the computation of its value has not yet completed. The Futures LCO synchronizes the access to this value by optionally suspending the requesting thread until the value is available

(see figure 5.7). A list of threads pending on the future and extended as further references are made. Upon return of the computed value the pending threads are continued.

5.1.6.2 Barrier LCO and DataFlow

The barrier LCO in HPX provides synchronization primitives very similar to the semantics of MPI_Barrier() in MPI. In HPX, the barrier LCO holds a group of HPX threads until all the threads in the thread pool participating in the synchronization are ready to execute. Once all the threads are ready to execute the barrier LCO allows all the threads to continue executing the next step in the algorithms (see figure 5.8). A special case of the barrier is the Dataflow template [Den80] that is applied to a single succeeding thread and which holds operand values.

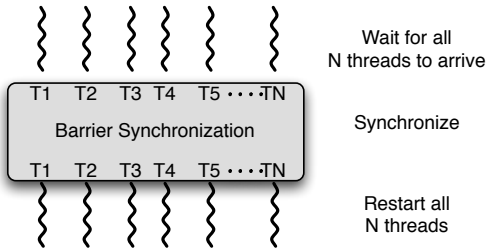


FIGURE 5.8: Barrier LCO in HPX

5.1.6.3 Work-Queue LCO

HPX also implements the novel semantics of work-queue as an LCO. The work-queue LCO can take in a HPX thread or a group of HPX thread and create one or many HPX threads in response (see figure 5.9) . One possible use of the Work queue LCO is to queue work items or HPX threads waiting for a resource to become available.

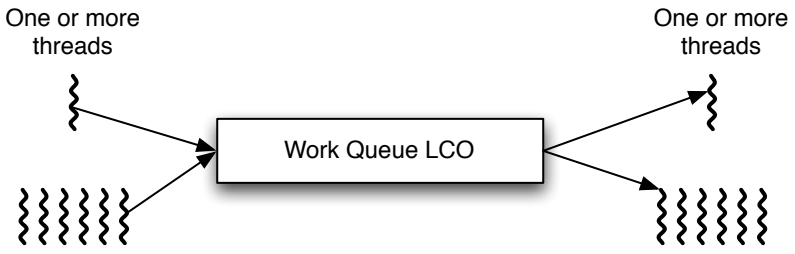


FIGURE 5.9: Barrier LCO in HPX

LCOs are a core part of the HPX implementation and the ParalleX model that provide runtime level support for dynamic execution characteristics. HPX is not limited to these LCOs and can easily be extended to provide more when use-cases require creation of specific LCOs.

5.1.7 HPX Runtime System

The HPX runtime system as implemented is modular. It provides support for policies at multiple levels. For example, if for some workloads FIFO thread queue doesn't provide the required performance, HPX provides an interface to specify user defined policies to change the scheduling policy of the thread manager. Implementation of the policies can be done without affecting the core of HPX or even without modifying the entities involved. The HPX runtime system also provides interfaces to allow developing new LCO based synchronization mechanisms. While the AGAS transport layer is under active development, the modular implementation of HPX allows users to write multi-locality code so that these codes can take advantage of multi-locality support when a future high performance AGAS becomes available.

5.2 HPX Implementation of Barnes Hut Algorithm

Scaling of conventional dynamic computations such as Barnes Hut N-Body applications and the adaptive mesh refinement (AMR) based applications is severely constrained in some cases. The system architectures and the supporting conventional models of parallelism expose the application management and execution to the factors that impede performance and scalability (Starvation, Latency, Overhead, Contention). In order to improve the efficiency and scalability of dynamic applications new modalities of execution models that better expose the underlying parallelism will need to be explored. HPX described in the above sections is an implementation of one such model of execution (ParalleX).

High Performance ParalleX exposes semantics that allow dynamic load balancing and utilization of advanced synchronization primitives among other advanced techniques to overcome limitations of current system architectures. HPX provides synchronization primitives such as the futures LCO which better expose the underlying parallelism using overlapping of computation with communica-

tion based on application demands. HPX semantics support runtime specification of HPX threads, allowing applications to define the HPX thread granularity at runtime. Right combinations of these semantics allow for a rich set of semantics that enable dynamic workload management and better exploration of parallelism both within each iteration and across iterations.

This research implementation of the Barnes Hut N-Body algorithm is implemented using the HPX runtime system. The HPX version of the Barnes Hut algorithm utilizes several attributes of the ParalleX execution model as implemented by HPX to provide asynchronous parallelism within each iteration of the Barnes Hut algorithm. As the force computation function of the algorithm is the most computationally intensive part of the HPX version of the Barnes Hut algorithm, it parallelizes the force computations using dynamic parallelism techniques offered by the HPX runtime system. This section describes in detail the research implementation of the HPX code and additional techniques used to elicit performance from the underlying systems. The HPX version of the Barnes Hut algorithm is implemented using the C++ language and the HPX runtime system with the following goals

- Parallelizing force calculation step using HPX
- Creating interaction list and co-locating related work to provide data locality
- Utilizing futures based asynchrony and work queue based computation for force calculations to provide balanced loads to the processors.
- Providing support for controlling the granularity of each HPX Thread to determine the best workload characteristics for fast, scalable processing of the algorithm.

5.2.1 Parallelizing Barnes Hut Algorithm Using HPX

Inherent in the Barnes Hut algorithm is a global barrier at the tree construction stage of every iteration, where the algorithm requires positional information of all particles in the system. The research implementation explores opportunities to exploit parallelism within each iteration. The

computationally intensive force calculation is the key focus for parallelization using the HPX runtime system. The main challenges in parallelizing the force computation are:

- exposing fine grained control and parallelism at the particle interaction level and devising methodology to express each interaction as a work unit
- the number of interactions for each particle varies across iterations, consequently new ways of providing dynamic load balancing are required to manage efficient execution of the fine grained computations.

The technical approach in tacking the above challenges involves exploiting a dataflow based framework for executing the Barnes Hut N-Body simulation. The dataflow approach allows representation of each of the interactions as a work unit and helps take advantage of a proven approach to providing scalability for scaling constrained applications. Inherent in the HPX based dataflow approach is the use of futures for resolving data dependencies which provides asynchronous access to data required for computing each dataflow stencil. Two modes of granularity support are enabled: workload based HPX thread allocation which allows for managing the size of HPX threads and increasing the number of threads based on the workload, and alternatively fixed number of HPX threads based workload allocation where the number of HPX threads is fixed and the variable workload size adapts to match the resource allocations. These methodologies provide a path for implementing dynamic workload management and load balancing.

The most computationally intensive part of the Barnes Hut algorithm is the force computation stage. The HPX parallelization efforts and the research implementation focuses on parallelizing the force computation stage of the Barnes Hut algorithm. This is done by representing the computational work as a stencil based work flow. The fully populated Barnes Hut tree is flattened into an input row vector stencil, comprising all the tree nodes including the particles and the intermediate tree nodes. A complimentary output row vector stencil representing the result buffer is created to store the results from the force calculation stage. Each stencil in both the input and output stencil

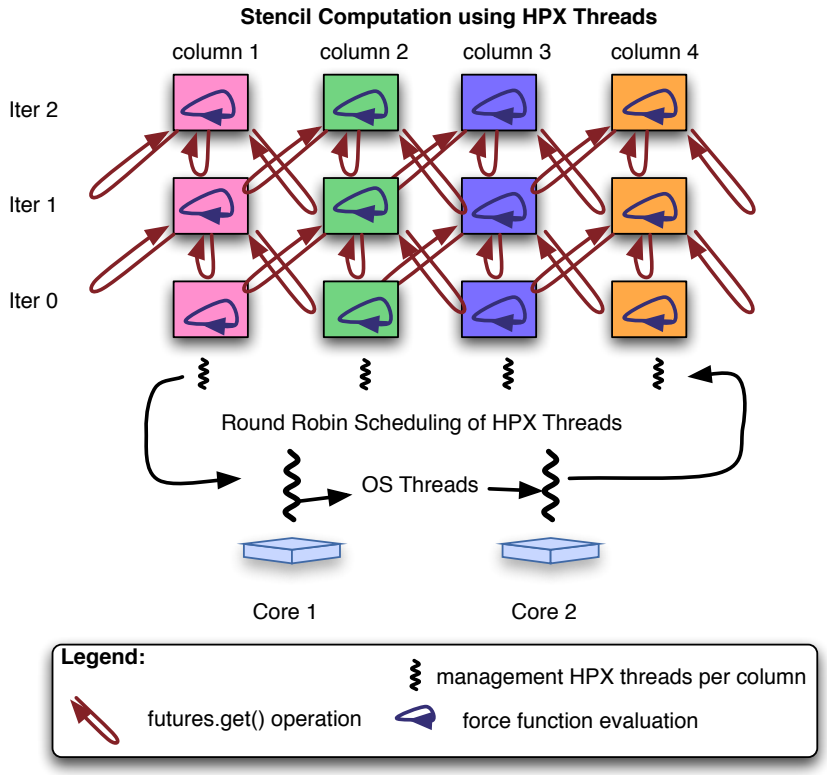


FIGURE 5.10: HPX runtime system management of the Barnes Hut Algorithm Workload

rows has a GUID associated with it. Each stencil element in the input row maps to itself in the output stencil row. Additionally, for each stencil in the output stencil row that is of the particle type, connections are made to the input rows. The connections are made on the basis of Barnes Hut approximation criteria of $D/r < \theta$. In this context, making a connection implies that each of output row stencil elements holds a vector of input row stencil element GUIDs that provide the necessary data for computing for each particle in the output stencil elements.

Each column in this connected network of input and output stencil has a HPX thread for managing the internal force computations. This fully connected network is depicted in the diagram 5.10. When executed in parallel a global pool of these management HPX threads are created and made available to the allocated resources. For each of the operating system thread managing each of the cores in the parallel system, HPX user-level threads are allocated from the global pool in a round-robin manner. The user also has the ability to map the HPX threads to preferred local queues in the HPX system.

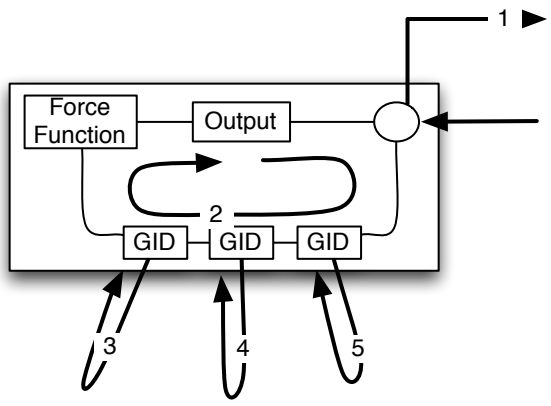


FIGURE 5.11: Stencil computation using HPX

Each HPX thread managing a column triggers the dataflow for every stencil element allocated. This work flow is depicted in the figure 5.11. The management thread is depicted as the arrow numbered 1 in the diagram. When the computation flow for each stencil element is triggered, a new HPX thread for the stencil is created. This thread is identified as Thread 2 in the diagram. This stencil-specific HPX thread performs a "futures.get()" operation on each of the GUIDs for the input stencils that the particle interacts with. For each futures.get() operation a new HPX thread is created to manage the asynchronous data accesses (3,4,5). For each input stencil data that is available the local management thread (2) computes the force function and stores the intermediate result in a result buffer. For each stencil evaluation a total of $(2+n)$ threads are required (1 column manager thread + 1 local computation thread + n futures.get() created threads). Once the current stencil is completely evaluated, the next stencil is loaded and the same workflow is repeated.

The futures based asynchrony in representing the interconnections provide rich semantics for exploiting parallelism available within each iteration. HPX semantics and the implementation also allow specifying granularity of each execution block, thus allowing for co-locating related computations, to provide implicit data locality. The following sections present the implementation details of the HPX implementation of the Barnes Hut algorithm.

5.2.2 Execution Flow of The HPX Barnes Hut Code

The execution flow of the HPX implementation of the algorithm is different from the sequential and OpenMP implementations. The key changes in the implementation include creating a tagged tree data structure, flattening the tree to create interaction lists, using the interaction list to provide granularity support, and calculating the forces using the HPX runtime system. The key steps in the HPX implementation of Barnes Hut algorithm are

1. Reading in input values containing position, initial velocity, number of bodies etc from the input file, which allows for increasing the number of bodies without having to recompile the program.

2. Calculating the bounding box and the center position of the bounding box containing all the N bodies to determine the position of the root of the Barnes Hut tree.

3. Creating a tree data structure that maps the spatial position to a corresponding location in the tree. The default tree construction algorithm is configured for 3D simulations resulting in an octree representation of the spatial distribution of the particles.

4. Calculating the center of mass by post order traversal of the octree data structure.

5. Tagging both leaf and intermediate nodes in the Barnes Hut tree data structure

6. Creating a ragged array representation of the interaction list which represents the subset tree interaction for each particle involved in the simulation.

7. Adding support for granularity where the granularity of each thread can be modified to a user defined value. Creating meta particles of the size specified by granularity and developing interaction list of meta particles.

8. Compute forces using static dataflow LCOs and futures LCO for synchronization and using leapfrog integration to compute the changed position and velocities of the bodies in preparation for the next iteration.

5.2.3 Data Structures

The data structures used in the HPX version of the Barnes Hut N-Body code are similar to the sequential and OpenMP versions of the code. The key difference is that the HPX version of the data structures have an added identifier called "tag" which facilitates creation of tagged tree data structure (see figure 5.12).

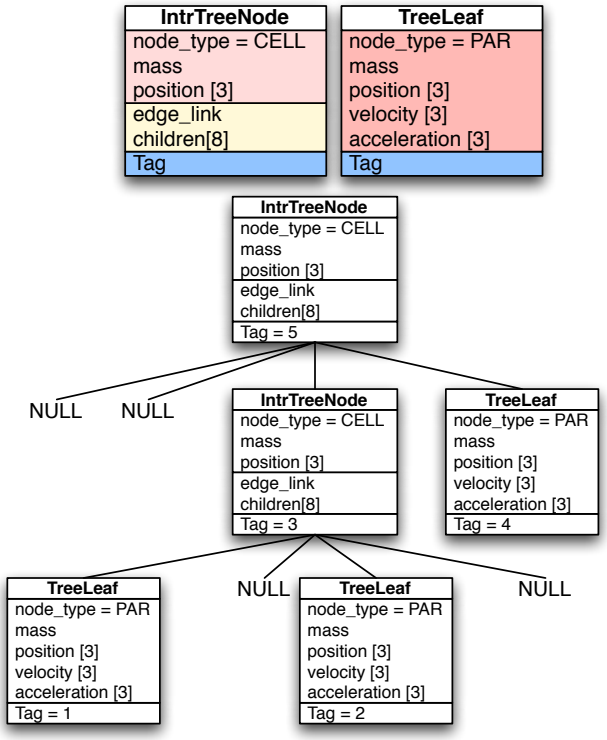


FIGURE 5.12: Data structures used in HPX version of Barnes Hut N-Body implementation

5.2.4 Functional Description

The HPX implementation of the Barnes Hut N-Body algorithm is significantly different from the sequential and OpenMP versions of the program and incorporates the following key functions: *main, computeRootPos, treeNodeInsert, calculateCM, tagTree, buildBodies, interList, init_execute, execute*. Each of these functions performs a critical activity.

- *main()* function is the entry point in the program and calls a wrapped function *hpx_main()* to invoke the Barnes Hut algorithm. The purpose of main is to setup the necessary environment

to run HPX primitives such as AGAS, and parse command line options. This function is described in detail in section 5.2.4.1.

- *hpx_main()* function is the entry point in the Barnes Hut algorithm execution and calls all other functions at appropriate times. This function is described in detail in section 5.2.4.2.

- *processInput()* function processes the input parameter file to load critical simulation values such as number of iterations, number of bodies, and position, velocity vectors of the bodies. This function is described in detail in section 5.2.4.3.

- *computeRootPos()* function calculates the bounding region of the computational domain and determines the root position of the Barnes Hut tree. This function is described in detail in section 5.2.4.4.

- *treeNodeInsert()* function constructs the Barnes Hut tree by inserting the body one at a time into the tree and organizing the tree according to spatial distribution of the particles. This function is described in detail in section 5.2.4.5.

- *computeCM()* function uses the Barnes Hut tree to calculate the center of mass at all nodes in the tree. These center of mass values are used in computing the force exerted by the system on each of the particles. This function is described in detail in section 5.2.4.6.

- *tagTree()* function performs a post-order traversal of the tree and tags each of the leaf and intermediate nodes with a unique identifier in sequence. This function is described in detail in section 5.2.4.7

- *buildBodies()* function traverses the tagged Barnes Hut tree and creates a mapped vector data structure that contains information of all the nodes in the tree. This function is described in detail in section 5.2.4.8.

- *interList()* function performs a post order traversal of the tree and creates an interaction list for all bodies in the system. For each body the function traverses the tree and matches intermediate nodes or bodies that match the Barnes Hut approximation criteria. This function is described in detail in section 5.2.4.9

- *bilist()* generates a granularity based meta-particle (called PX particle) description and generates an interaction list between the PX particles. This function is described in detail in section 5.2.4.10

- *init_execute()* function passes the granularity-balanced interaction list, and particle information to the HPX based routines that use dynamic techniques to schedule the computations. This function is described in detail in section 5.2.4.11

- *generate_initial_data()* function uses the meta-particle interaction to populate stencil values in preparation for initiating a simulation run. This function is described in detail in section 5.2.4.12

- *prep_ports()* function establishes connections between the static data flow stencil elements so that the allocated HPX thread can utilize the synchronization mechanisms to effectively process stencil values. This function is described in detail in section 5.2.4.13

- *eval()* function is the key function where the force computation routines and move generation routines are specified. Each HPX thread uses the *eval()* function to determine the order of processing and the set of instructions to be executed for computing forces using static data flow and futures based mechanisms. This function is described in detail in section 5.2.4.14

- *moveParticles()* function uses the acceleration calculated by the *init_execute()* function and applies the leapfrog/verlet integrator to advance the position of the bodies and calculate new velocities of the particles involved. This function is described in detail in section 5.2.4.15.

5.2.4.1 main() Function

The *main()* function in the HPX version of Barnes Hut is the key function that organizes the HPX environment for executing the application code. The *main()* function sets up the local host to be an AGAS host and initializes the runtime system in console mode which is the default mode of operation for the runtime system. The function then parses the command line arguments that are passed to the program. The current version of the program requires adding runtime flag *"-r"* which initiates the AGAS server, and *"-t N"* where *-t* instructs the runtime system to use N operating system threads. The argument list has also been customized to accept the following flags: *"-g M"* which allows the user to customize the granularity of the HPX threads where *M* specifies the size of the workload to be executed by each thread, and *"-p parameterFile"*, which allows a user specified parameter file to be passed in. An example of the parameter file is provided in figure 5.13 which allows the user to specify the type of scheduler, input data file for N-Body simulation and the type of scheduler to use. The *main()* function parses the parameter file to set the input data file, granularity of the computations and the type of scheduler. Currently two options can be used for the scheduler: the global scheduler, and the parallel scheduler (local queue). The default setup

Example Parameter File

```
1 [nbody]
2 loglevel     = 2
3 scheduler = 1
4 input_file = 5_file
```

FIGURE 5.13: Parameter file for the HPX Barnes Hut program

uses the parallel or the local queue scheduler. The *hpx_main()* function is invoked using the user specified configuration (or the default configuration).

5.2.4.2 hpx_main() Function

The *hpx_main()* function is the entry point into the Barnes Hut algorithm logic and performs all the algorithm related operations such as creation of Barnes Hut tree, calculation of center of mass,

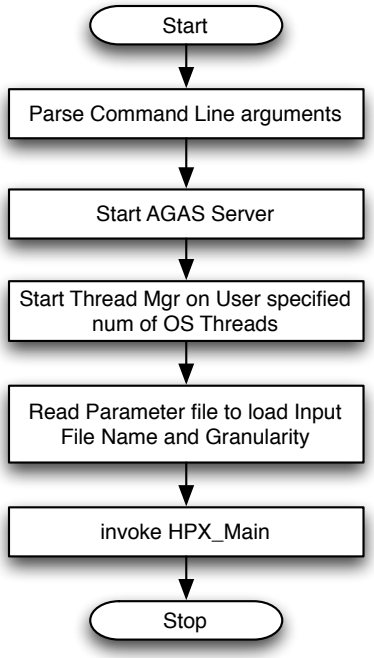


FIGURE 5.14: Sequence flow of the main function in HPX Barnes Hut code

and calculating forces. The function uses the parameters parsed by the *main()* function and opens the input file specified by the user. The input file format is the same for all the three versions of the Barnes Hut implementations (sequential, OpenMP and HPX). The input file is parsed, all the particle information (mass, position and velocity and smoothing values) are read from the file and are populated into the vector that stores particle information using the *processInput()* function. The following functions are repeated for the number of iterations specified by the input file: *computeRootPos(), treeNodeInsert() computeCM(), tagTree(), buildBodies(), interList(), bilist() and init_execute()* (force calculation). The *computeRootPos()* function is invoked to compute the maximum domain bounds of the system and center position of the system. Using the *treeNodeInsert()* function each of the particles in the system are inserted into the intermediateNode created at the root position and a Barnes Hut tree is created as described in section 5.2.4.5. The *hpx_main()* function then invokes the *computeCM()* function which performs a post-order traversal of the tree and computes the center of mass for each node in the Barnes Hut tree. Then the *tagTree()* function is called which performs a post order traversal of the tree and tags each of the nodes in the tree with

a unique numerical tag reflecting the order in which the nodes are visited. The tagged tree is then flattened into a vector of particles and intermediate nodes using the *buildBodies()* function which also maps the particle information to the flattened vector. The flattened tree vector is required to provide mapping to the stencil based parallel processing framework used in the force calculation step. The *interList()* function is then invoked to create a comprehensive interaction list for each particle. For each particle in the system the interaction list comprises the tag identifiers of bodies and intermediate nodes with which it interacts (determined by the Barnes Hut approximation criteria $D/r < \theta$). This interaction list is further processed by the *bilist()* function that creates interaction lists for meta particles called PX particles. The PX particles are made up of N actual bodies where N is the granularity size. The only exception to this case is when the number of particles in the system are not evenly divisible by the granularity size In which case the remaining extra particles (which are less than N) are assigned to the last PX particle. The *bilist()* function uses the interaction list created by the *iList()* function to develop an interaction list for the PX particles (meta particles). This balanced interaction list is then passed on to *init_execute()* function which uses the balanced interaction list to set up a static data flow based computational workflow. Each of the PX threads requested by the user is assigned a part of the stencil using the local work queues created by the HPX thread manager. Each thread operates on the stencils allocated, using the *eval()* function (called by the *init_execute()* function, to calculate the force and move generation on each stencil. The synchronization and computation steps are elaborated in detail in section 5.2.4.14. The *hpx_main()* function then gathers the results for each iteration and updates the particle vector with new positions and velocities. When the "maximum number of iterations" criteria is met, the *hpx_main()* function shuts down the supporting infrastructures such as the AGAS server, thread manager and makes the resource available for further use.

5.2.4.3 processInput() Function

The *processInput()* function in the HPX version is exactly the same as the *processInput()* function as implemented in the sequential and OpenMP versions of the code. The function takes in a string

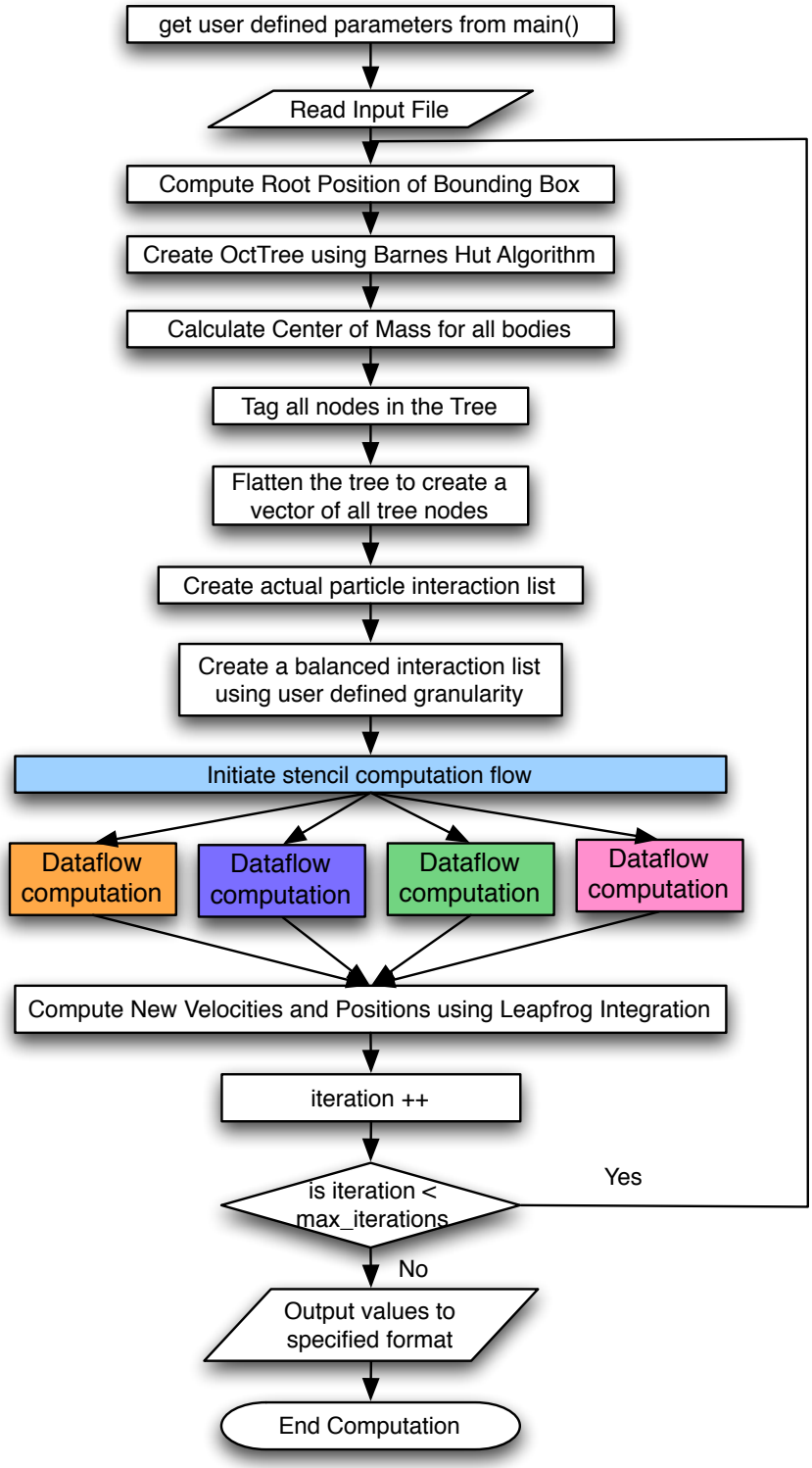


FIGURE 5.15: Sequence flow of the hpx_main function in HPX Barnes Hut code

input_file as an argument. Using the input_file string argument the *processInput()* function creates an input filestream to read the input data. The function reads in the first line into the *num_bodies* variable which defines the number of bodies in the system. The function reads in the second line into the *num_iterations* variable which defines the number of iterations to be computed. The function reads in the third line into the *dtime* variable which defines the integration timestep for the Verlet (leap frog) integrator. The fourth and the fifth line are read in as the *eps* and the *tolerance* variables. Remaining lines correspond to the particle information including: mass, position and velocity. The function creates a for-loop to read the data into the TreeLeaf data structure created for each of the bodies. At the end of this function each of the bodies has a fully qualified mass, position vectors and, velocity vectors. The function closes the input file buffer and returns control to the *main()*. This function does not assign values to the "Tag" portion of the data structure other than initializing the them to zero.

5.2.4.4 computeRootPos() Function

The *computeRootPos()* function in the HPX version is exactly same as the *computeRootPos()* function of the sequential and OpenMP versions of the code. The function takes in 3 arguments: the number of bodies, box size and the centerPosition array. The function initiates a for loop that cycles through all the particles in the Barnes Hut N-Body system and uses the position vector of each of the particles to compute the maximum (X, Y, Z) coordinates and the minimum (X, Y, Z) coordinates. This maximum and minimum coordinates determine the bounds of the computational region. The function uses the maximum and minimum coordinate values to calculate the center point of the computational domain. This information of the bounding box and center of the computational domain are passed backed to the main function.

5.2.4.5 treeNodeInsert() Function

The *treeNodeInsert()* function in the HPX version of the code is exactly the same as the *treeNodeInsert()* function of the sequential and OpenMP versions of the code. The *treeNodeInsert()* function takes in 2 arguments: the particle to be inserted and the sub-box size. The function com-

pares the position of the particle to be inserted (P_i) to the position of the current intermediate node. Based on the relative position of P_i, the corresponding octant to insert the particle into is chosen. 3 possible conditions can occur when an octant is chosen:

- the node in the tree where the body is being inserted is empty. In which case the body is inserted at the computed position.
- the node in the tree where the body is being inserted is occupied by a Intermediate node. In which case the function recursively cycles down into the branches of the and inserts the body at the correct leaf location.
- the node in the tree where the body is being inserted is occupied by another body. In which case the function copies the existing body into a temporary buffer variable, creates an intermediate node at that position and inserts the previous body and the new body using the intermediate node as a starting point.

Once the particle is inserted the control is returned back to the main function. Once all the particles are processed through the treeNodeInsert() function, all the particles become part of the global Barnes Hut tree. This tree is then used to compute the center of mass and gravitational forces in the system.

5.2.4.6 computeCM() Function

The *computeCM()* function in the HPX version is exactly the same as the *computeCM()* function of the sequential and OpenMP versions of the code. The *computeCM()* is a recursive function that computes the center of mass of each of the bodies and the intermediate nodes in the Barnes Hut N-Body system. The function performs a post order traversal of the Barnes Hut tree generated by the *treeNodeInsert()* function. The function expands down the tree until it reaches the leaf nodes. The function then calculates the center of mass of the leaf nodes containing the bodies, which is the same as the position of the bodies (weighted mass cancels out for the bodies). The function then

collapses upwards and calculates the center of mass at the intermediate nodes using the following formula.

$$CM = \frac{1}{total_{mass}} * \sum_{i=1}^{n} pos[i] * mass[i]$$

The center of mass calculated by the above formula is a weighted average of the particles contained within the sub cell / cube. Once the function successfully completes, each of the nodes (both intermediate and leaf) is populated with center of mass values and control is returned back to the main function.

5.2.4.7 tagTree() Function

The *tagTree()* function is implemented specifically for the HPX version of the code and does not exist in the serial or OpenMP versions of the code. This function performs a post-order traversal of the Barnes Hut tree and updates the tags on each of the nodes in the order they are visited. Once this function completes successfully the result is a fully tagged Barnes Hut tree data structure as shown in figure 5.16.

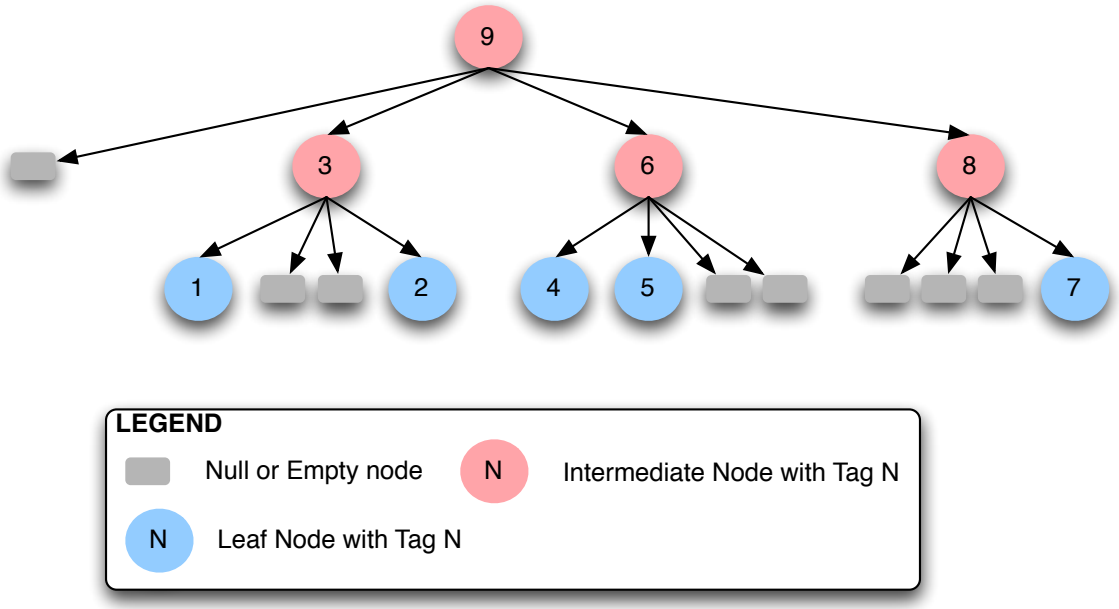


FIGURE 5.16: Tagged Tree data structure created in HPX version of Barnes Hut N-Body implementation

5.2.4.8 buildBodies() Function

The *buildBodies()* function is unique to the HPX version of the Barnes Hut N-Body implementation. This function performs a post order traversal of the Barnes Hut tree and flattens the tree into a tagged vector data structure. The function also maps the particle information to cells of particle type in addition to mapping intermediate node information to the cells of the intermediate-node type. Upon successful completion of this function the tagged vector containing all the nodes and particles in the system are represented in an easily accessible manner to create interaction lists. Characterization of the tagged vector of all nodes in the system is shown in figure 5.17

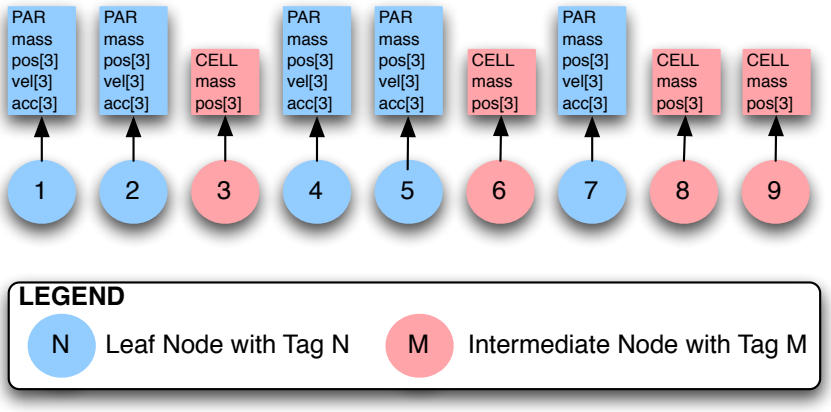


FIGURE 5.17: Flattened tree vector data structure created in HPX version of Barnes Hut N-Body implementation

5.2.4.9 interList() Function

The *interList()* function is unique to the HPX version of the Barnes Hut N-Body implementation. The purpose of this function is to create an interaction list of particles and intermediate nodes that satisfy the Barnes Hut approximation criteria. For every particle in the system, this function traverses the tree and identifies intermediate nodes and particles with which the particle under consideration interacts. This is determined by the Barnes Hut approximation criteria $D/r < \theta$ which uses the position of the particle under consideration and the position of the remote node (if the remote node is close enough to take the remote particle into account for force calculations).

The interaction list generated is a "ragged-array" data structure because each particle interacts with a varying number of nodes. An example interaction list is shown in figure 3.10

5.2.4.10 bilist() Function

The *bilist()* function provides granularity support for controlling the the workload for each PX thread. This is done by creating meta particles or PX particles, where each of the PX particles contains N actual particles. The Function performs the division as follows: for a system of N bodies and for a user specified granularity of g, N/g number of PX particles are created of g actual particles. If N is not evenly divisible by g then the remaining actual particles are stored in the (N/g) + 1 particle which contains less than g number of actual particles.

Once the particles are created the function uses the interaction list of actual particles created by the *interList()* function to create the interaction list of PX particles called *bilist* (balanced interaction list). The process of creating the interaction list of PX particles is as follows: The interaction list of actual particles is processed in batch mode for g particles at a time (since g is the size of PX particles), the size is adjusted for the final PX particle which may contain either g particles or less-than g particles if it is storing extra particles. The change in size is handled by the algorithm adaptively. For the first meta particle the function loops through each of the actual particles, and based on the mapping to interacting nodes given by the interaction list, the function cycles through each tag of the interacting node to determine the meta particle interaction which is done by dividing the interacting node's tag by g. Once all the particles and their corresponding interaction lists are processed the resultant interaction pattern represents the interaction of PX particles.

5.2.4.11 init_execute() Function

The *init_execute()* function was written as part of the adaptive mesh refinement (AMR) framework developed to improve scalability of three dimensional AMR codes [TAB+10]. The function has been redesigned to provide a suitable framework for executing the Barnes Hut N-Body codes using HPX. The function uses the distributed_factory component to create a pool of stencil components each of which has a globally unique identifier (GUID) allocated by the AGAS server. The size

of this pool is defined by the number of PX particles in the system. The function then initializes the number of rows of each stencil to two. The key idea behind the two rows is that two rows of PX particles are created with row_0 identifying the input PX particles and row_1 identifying the output PX particles. The mapping between the two is created by the *prep_ports()* function which results in a static data flow like computational structure. The maximum number of interactions can be configured by the application developer to any value. However, larger numbers of interactions result in more memory usage and very long compile times. The next function to be called is the *prep_ports()* function which uses the balanced interaction list generated by the *bilist()* function to develop mappings of the input row and output row. The function then calls the *init_stencil()* function for each of the rows to initialize the stencil values. Then *get_output_ports()* function is called to get the pool of GUIDs into a vector. The *connect_input_ports()* function connects the output GUIDs obtained from the *get_output_ports()* to connect the stencil inputs with the stencil output ports. Finally the function calls the *execute()* function to perform a user defined function which in this case is the force calculation and move generation computations on a per PX thread basis.

5.2.4.12 generate_initial_data() Function

The *generate_initial_data()* is called implicitly by the underlying functions. The main goal of this function is to populate each of the PX particles in the stencil representation with the correct mass, position and velocity coordinates. For each PX particle the function determines the number of actual particles and for each of the actual particle the function calculates the global index. Using the global index value the particle information for the actual particle is extracted and stored into the stencils corresponding to each of the PX particle.

5.2.4.13 prep_ports() Function

The *prep_ports()* function is invoked by the *init_execute()* function to develop mappings of the input stencil row and the output stencil row. The balanced interaction list of PX particles generated by the *bilist()* function is used to develop this mapping.For PX particle in the interaction list, the

function determines the interaction list and uses the per particle interaction to connect the input stencil rows to the output stencil rows. This is done by indicating the destination GUID for each of the input and output ports to the output and input stencil row elements (columns). The result of this operation is a connected row of stencils, that is used by the execute function to evaluate each HPX thread (as shown in figure 5.18).

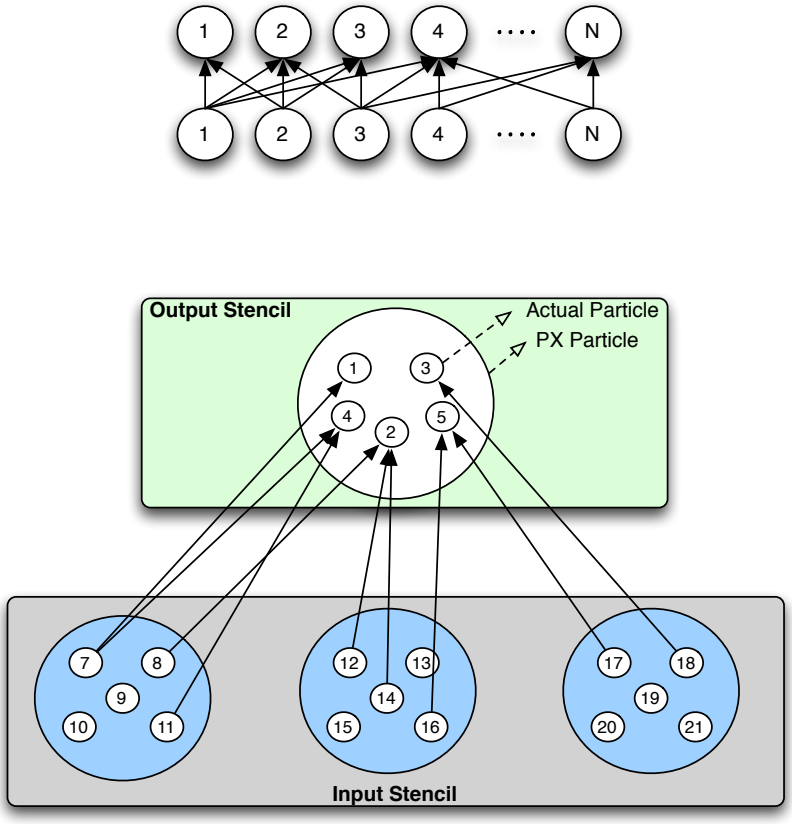


FIGURE 5.18: Interactions between PX particles and the role played by actual particles

5.2.4.14 eval() Function

The *eval()* function defines the code that each PX thread executes. The function that is executed by each thread is as follows. For each of the output stencil elements the function first determines the number of actual particles contained within the PX particle. For each of the actual particles within the output PX particle stencil, the global index is computed. The function similarly determines the number of actual particles in each of the interacting input PX particle stencils. For each of the actual particles enclosed within the input PX particle stencil, the global index is calculated.

Since force is experienced only by the actual bodies and not the intermediate nodes (which purely serve as pseudo-particles for representing the cluster), the force calculation and position updates are carried out only for the particles and not intermediate nodes. The force is calculated using the formula

$$Force = G * m * m_{cm} * (\frac{x_{cm}-x}{r^3}, \frac{y_{cm}-y}{r^3}, \frac{z_{cm}-z}{r^3})$$

where r is the distance from the particle to the center of mass of the particle in the box computed using

$$r = \sqrt{(x_{cm} - x)^2 + (y_{cm} - y)^2 + (z_{cm} - z)^2}$$

Each stencil computation is executed in the following manner:

1. The input stencil elements are instantiated using Futures LCOs.

2. Each HPX thread evaluating the data flow computation performs a *futures.get()* operation using the GUID of the input stencil elements to get the position, velocity and mass from each of the interacting elements asynchronously.

3. For each input stencil,whose input value is available the above set of steps as outlined in the *eval()* function are evaluated.

4. A write lock controls write access to a result buffer where all the results from the computations are stored.

5. At each of the receiving output, a *futures.get()* operation is initiated for all GUIDs of the input computations results stored in the result buffer.

6. The connections between the output stencils and the input stencils are created by the *prep_ports()* function.

7. For each expected input connections to the result buffer, a read lock is placed to control read access to the result buffer to ensure that the result value is written to before a read operation is executed.

These sets of interactions are depicted in figure 5.19 When all of the input stencils have been evaluated and the output stencil has read all of the values from the result buffer, the next data flow computation is executed until each of the stencils managed by the HPX thread has been evaluated.

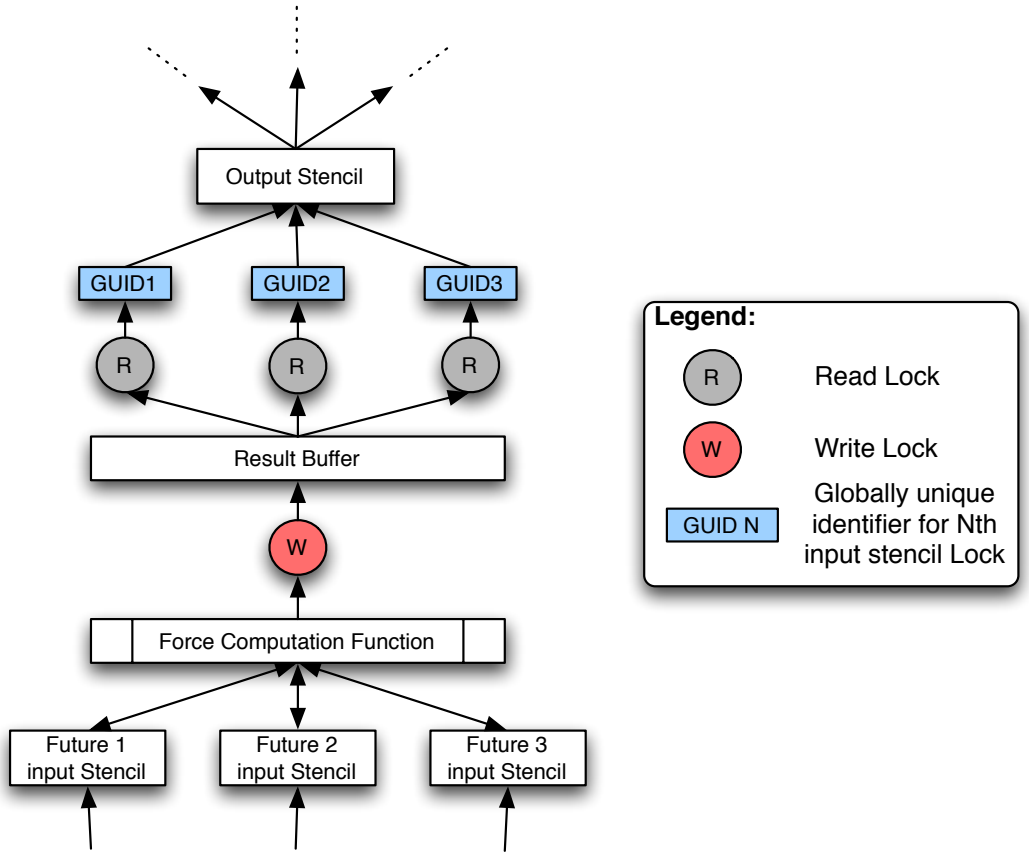


FIGURE 5.19: The dataflow computational approach to asynchronous computation of forces

5.2.4.15 movePartícles() Function

The *moveParticles()* function uses the acceleration computed by *init_execute()* to calculate the new position and velocity of the particles. This is done by using the Verlet or the LeapFrog integration technique. Standard implementations of the leapfrog method involve time-interleaving the posi-

tions and velocities at alternate points in time. The governing equations for the leapfrog integration method in the Barnes Hut Method can be represented as follows:

$$V_{dt/2} = a * (\delta t/2)$$

$$V_{1/2} = V_t + V_{dt/2}$$

$$r_1 = r_0 + V_{1/2} * \delta t$$

$$v_{3/2} = v_{1/2} + (a * \delta t)$$

where r is the position vector, $v = dr/dt$ is the velocity (change in position over time) and $a = dv/dt$ is the acceleration (change in velocity over time). Once the new position and velocity are calculated the control is returned back to the *main()* function. The *hpx_main()* function then recomputes the tree, center of mass, force calculation, and the move generation until the exit criteria of number of iterations is satisfied. Detailed performance characterizations of the HPX version of the Barnes Hut algorithm are presented in Chapter 6

Chapter 6
Experiments and Results

Experiments highlighted in this thesis have been constructed around the sequential, OpenMP and the HPX implementations of the Barnes Hut N-Body algorithm. Several experiments have been conducted to test the parallel semantics of the ParalleX execution model in enhancing the scalability for dynamic graph applications. This chapter outlines the experimental setup, design of experiments, results and analysis of the experiments.

6.1 Experimental Setup

Systems Used for Experimentation The experiments in this thesis have been executed on a 32 core shared memory multiprocessor system (Pollux). The Pollux system has 8 sockets each with dedicated memory. The sockets (processors) use a shared hyper threading framework to access memory across the system. Each of socket has a Quad-core 2.7 GHz AMD Opteron processor (family 8384), with 512 KB cache size. Each of the sockets in the system is associated with 8 GB dual channel DDR2 RAM. The system has an aggregate memory of 64 GB that can be addressed by each of the 32 cores in the system. The main OS used by the shared memory system is 64 bit Linux kernel version (2.6.18).

Software Stack All programs used in the experiment on the Pollux system have been compiled using the standard GNU g++ compilers version 4.4.2. The HPX runtime system is currently under active development. For the purposes of this experimentation, the HPX version 3 is used. The OpenMP linking is done through the version of OpenMP primitives available in the GNU g++ compilers version 4.4.2. Other system software such as Intel compilers, performance measurement tools etc have also been installed on the system.

6.2 Sequential Barnes Hut N-Body Experiments

The sequential Barnes Hut N-Body implementation is compiled using the GNU g++ compiler and executed on the standard Linux environment. The purpose of the sequential experiments is to determine the timings of various functions of the algorithm and to identify the parts of the algorithm that dominate the overall execution times as well as to analyze their suitability for parallelization. In order to do this the sequential program is run for 25 iterations using different problem sizes, varying from 10 to 5,000,000 particles. The program is profiled using GNU gprof utility to develop execution characteristics. GNU gprof analyzes the execution sequence of a program to provide information such as function level performance.

6.2.1 Sequential Barnes Hut N-Body Experiments

The sequential version of the program was executed in the standard environment for varying problem sizes from 10 to 5 million bodies. Barnes Hut algorithm was used to evolve the system for 25 iterations. . GNU gprof profiling utility was used to profile and characterize the sequential Barnes Hut N-Body implementation. The table 6.1 tabulates the result from the experiment and shows the overall time spent in milliseconds in various functions of the program. the function names are listed in the first column of the table.

TABLE 6.1: 25 iterations Sequential Results: Result gprof data from a series of runs using the sequential version of Barnes Hut algorithm. All data listed below is presented in milliseconds

Functions	**10**	**100**	**1000**	**10,000**	**100,000**	**500,000**	**1,000,000**	**5,000,000**
RootPos()	0.03	0.12	1.08	13.93	431.41	2397.98	5012.95	31180.90
TreeCreate()	0.04	0.75	9.48	107.71	1542.93	6816.46	14234.70	83018.80
CenterMass()	0.03	0.31	2.96	35.95	716.25	3662.53	7531.25	45199.00
ForceCalc()	0.40	30.73	1541.10	42218.20	727381	1864280	4874810	24,451,700
LeapFrog()	0.03	0.20	2	24.24	632.06	3519.70	7364.18	43681.40

6.2.2 Implications of Sequential Profiling

From the data listed in Table 6.1, the function that dominates the overall execution time in the sequential version of the program is the force calculation function. For each of the bodies in the N-Body system, the program uses the Barnes Hut tree data structure and the Barnes Hut approxi-

mation criteria and calculates the force impacted by part of system on the body. The order of computation complexity of traversing the Barnes Hut tree is $O(NlogN)$. The dominance of the force calculation function in the overall execution time of the program is maintained as the problem size increases.

6.3 OpenMP Barnes Hut N-Body Experiments

The experiment design for the OpenMP simulations involved studying variations in the following key parameters:

- For the experiments conducted in this thesis, the number of OpenMP threads (which is the same as operating systems threads) is varied. The number of threads is controlled using the environment variable "$OMP_NUM_THREADS" The set of values of this parameter is 2, 4, 8, 16, 28.

- Another parameter in the experiment is the number of bodies used for simulation. The problem sizes used in the experiments are as follows (number of bodies): 100, 1000, 10,000, 100,000.

- The final parameter is the number of iterations of the computation. These are varied between 1 and 100 iterations. For accurate comparison of the overhead costs of the parallel programming environment, 1 step runs are developed and compared with the HPX version of the program. The higher order iterations provide insights into the correlation of workload and parallelism.

For each of the above combinations of independent parameters the time to execute the force computation portion of the Barnes Hut N-Body algorithm is captured using the standard Unix "gettimeofday()" based timer. The purpose of these sets of experiments is to measure the performance provided by conventional parallelization of the force computation in the OpenMP based Barnes Hut N-Body implementation. The parallelism for an OpenMP application can be controlled by

varying the number of threads available to the program. The other two variants; problem size and number of iterations, are used to determine the effect of parallelism for different workloads. Using these three parameters the parallelism in an OpenMP based Barnes Hut N-Body implementation can be characterized.

6.3.1 OpenMP Barnes Hut N-Body Experiments

The OpenMP version of the program was executed on the shared memory system previously described. The parallelism is controlled by changing the environment variable $OMP_NUM_THREADS which sets the number of OS threads allocated for an OpenMP program. The set of values of this parameter employed for the experiment is : 2,4,8,16,28.

6.3.1.1 1 step

A series of runs for a single step of OpenMP version of the Barnes Hut implementation were executed for 3 problem sizes 1000, 10,000, and 100,000 for different levels of parallelism: 2, 4, 8, 16, 28. The results are tabulated in table 6.2. The columns in the table correspond to the number of OpenMP threads used and the rows in the table correspond to the problem sizes of the three systems used in these simulations. Since only one iteration was being measured, the 100 body system was not considered as the granularity of measurement time was too fine grained to be registered correctly using the "gettimeofday()" function in UNIX.

TABLE 6.2: 1 Step simulation results: Result data for various problem sizes evolved using the OpenMP version of Barnes Hut algorithm. All measurement data is presented in milliseconds

	2	**4**	**8**	**16**	**28**
1000	20.68	13.54	7.51	27.39	111.84
10,000	528.84	311.54	161.58	98.00	182.03
100,000	9230.14	4939.49	2624.10	1333.00	1538.70

6.3.1.2 10 Steps

Several runs for different problem sizes (100, 1000, 10,000, 100,000) were carried out for a fixed number of iterations (10 steps) and different levels of parallelism: 2, 4, 8, 16, 28. The results are

tabulated in table 6.3. This set of runs is used to understand the correlation between the increased workload and parallelism. The workload is varied by increasing the number of iterations.

TABLE 6.3: 10 Steps simulation results: Result data for various problem sizes evolved using the OpenMP version of Barnes Hut algorithm. All measurement data is presented in milliseconds

	2	**4**	**8**	**16**	**28**
100	5.07	3.42	2.51	1.83	1.86
1000	206.94	132.40	72.38	45.44	27.79
10,000	5255.27	3096.91	1621.49	826.02	512.25
100,000	91975.50	48911.20	26831.80	13284.00	7719.02

6.3.1.3 100 Steps

The number of iterations is then increased to 100 and the time to execute the force calculation function was measured using the same "gettimeofday()" based timers. Different problem sizes are used in this experiment ranging from 100, 1000, 10,000, and 100,000. Each of these problem sizes is executed using different gradients of OpenMP threads, ranging from 2, 4, 8, 16, 28. The results obtained from these experiments are tabulated in Table 6.4 Following are preliminary numbers for the problem size and the number of OS threads combinations.

TABLE 6.4: 100 Steps simulation results: Result data for various problem sizes evolved using the OpenMP version of Barnes Hut algorithm. All measurement data is presented in milliseconds

	2	**4**	**8**	**16**	**28**
100	44.26	26.57	20.50	20.46	19.56
1000	2,057.64	1,292.21	690.22	393.50	217.80
10,000	52,234.30	31,023.40	16,756.20	8,790.57	5,086.30
100,000	909,186	481,626	254,242	131,726	76,715.50

6.3.2 Implications of OpenMP based simulation results

In the 10 step run of the simulation, when the problem size is fixed and parallelism is increased by adding twice the number of OpenMP threads a speed up of 1.5 to 1.8 is consistently achieved. These effects are observed in both data presented in table 6.3 and figure 6.1. When the number of steps is increased to a 100 steps, the resultant times obtained for the OpenMP force calculations (see table 6.4) are approximately ten times the timing measurements for the 10 steps OpenMP

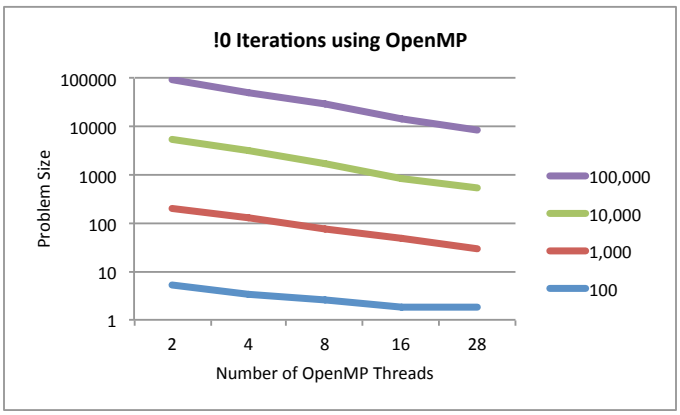


FIGURE 6.1: Scaling results for 10 step of the OpenMP version of Barnes Hut implementation

based force calculations (see table 6.3). The speedup factor of 1.5 - 1.8 is maintained through the increased problem size simulations with increase in OpenMP threads.

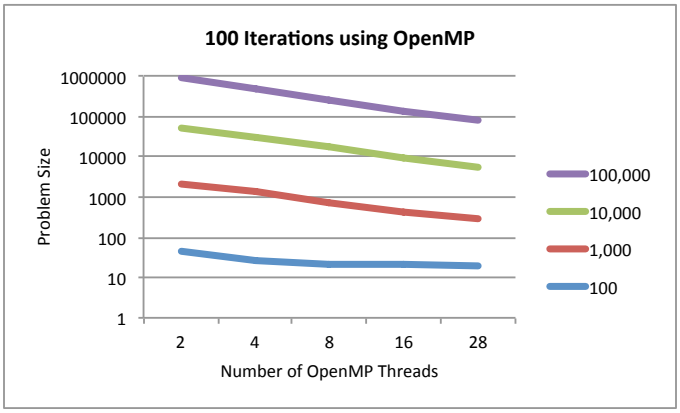


FIGURE 6.2: Scaling results for 100 steps of the OpenMP version of Barnes Hut implementation

6.4 HPX Barnes Hut Experiments

The HPX implementation of Barnes Hut N-Body provides several control parameters. The key control parameters utilized for the experiments are as follows:

- **-t** which allows specification of the number of Operating System threads to use
- **-g** which allows specification of the workload granularity per HPX thread. This option cannot be used with -z option since there cannot be 2 different control states managing granularity.

- **-z** which allows fixing the number of HPX threads to use for the simulation. This option cannot be used with the -g option since there cannot be 2 different control states managing granularity.

Changing the number of operating system threads in HPX simulations is similar to varying the number of threads using the "$OMP_NUM_THREADS" in the OpenMP simulations. For the experiments conducted in this research, the problem sizes are varied between 1000, 10,000 and 100,000. For each of the problem sizes, the number of operating system threads are varied and the grain sizes are varied to determine the correlation of the parallelism to the workload granularity. For each of the combinations the time taken to execute the force calculation loop is measured.

6.4.1 HPX Barnes Hut N-Body Experiments

Several experiments were performed to measure the performance of the algorithm using HPX with limited overhead in using the data flow stencil approach. The parallelism was controlled using two main parameters: the number of operating systems threads and the workload granularity (which directly impacts the number of HPX threads used). As the granularity of the workloads decreases (the work per thread is made more fine-grained) the number of HPX threads increase and the work done by each of the HPX threads decreases. The number of operating system threads, controls the explicit parallelism such that each of the threads gets the activities to be performed from the pool of HPX threads created. Since the change in workload can significantly impact the performance of the application, more exhaustive experiments are needed to determine the impact of workload on the parallelism achieved. Listed below are measurements of the force calculation function with the two parameters; number of operating systems threads and the workload granularity, varied.

6.4.1.1 1 Step

Several single step runs were carried out by keeping the problem size constant and varying the number of operating system threads and the work load granularity. Since each of the problem sizes have different granularities the results are tabulated and discussed separately for each of the problem sizes.

Table 6.5 presents the time required to execute the force calculation is measured in milliseconds. The granularities are varied such that twice the number of PX particles are created for each change in granularity. The granularity to be used is determined by the size of the flattened Barnes Hut tree data structure. The maximum size of stencil elements obtained after flattening the tree is 14856 stencil elements. If the maximum number of stencil data elements is used then the resultant interaction is captured in a single PX particle. Consequently there is no parallelism that can be gained from such a particle distribution since the entire workload is expressed as a single work unit. The granularities are varied among 7428, 3174, 1857, 928, 464, 232, 200 actual elements of the flattened Barnes Hut tree structure (number of total stencil elements) resulting in 2, 4, 8, 17, 33, 57, and 75 PX particles. The number of HPX threads created is directly proportional to the number of PX particles and is inversely proportional to the grain size. This is because as the granularity decreases the number of actual particles stored in the PX particles decrease, consequently more PX particles are needed to capture all the particles.

TABLE 6.5: 1 Step HPX Results: Result data a problem size of 10,000 particles using the HPX version. All measurement data is presented in milliseconds

	2	**4**	**8**	**16**	**28**
7428	413.20	405.46	406.12	416.37	420.60
3174	506.73	352.60	302.05	294.69	299.24
1857	577.76	358.13	191.30	202.69	196.67
928	538.25	315.36	191.79	139.32	149.51
464	537.97	287.81	178.96	155.21	139.93
232	590.79	354.33	254.79	250.95	223.05
200	1212.37	1037.99	871.70	768.91	874.13

As observed in figure 6.3 the timing measurements show the impact of granularity on parallelism provided and the consequential speedup obtained. For the granularity of 200 stencil elements or 75 PX particles the work load size is too small, consequently the management overheads caused by the runtime system cause degraded performance. Consequently the data point of 200 stencil element shows as an outlier. The data set characterized by a granularity of 7428 stencil data elements or 2 PX particles shows constant execution time even as more parallelism resources in the form of

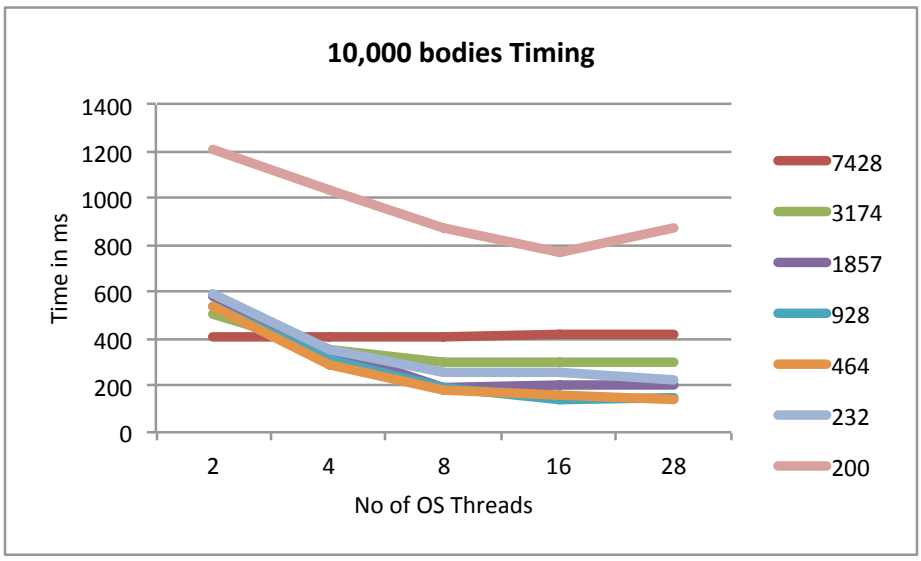


FIGURE 6.3: Timing in the 10,000 body problem using HPX version of Barnes Hut N-Body problem

increased number of OS threads are added. This is because at maximum 2 OS threads are needed to evaluate the 2 PX particle ensemble. Even if 4, 8, 16, or 28 OS threads are added a maximum of 2 OS threads will be used for the computations. As the granularity PX particles decrease and the number of PX particles increase the PX particles are able to better pack the work queue resulting in better scalability with the increase in parallelism (OS threads).

Similar experiments are also carried out for a problem size of 100,000 bodies and the simulation results are tabulated in the table 6.6. As observed in the simulations with a problem size of 10,000, the performance of the force calculation routine improves with the combined effect of decreasing granularity and increasing parallelism. This trend is captured in the figure 6.4. The granularities are varied among 74153, 37076, 18538, 9269, 4634, 2317, 1158 actual elements of the flattened Barnes Hut tree structure (number of total stencil elements) resulting in 2, 4, 8, 16, 32, 64, and 128 PX particles. As observed in figure 6.4 the timing measurements show the impact of granularity on parallelism provided and the resultant speedup. The data point of 200 stencil element shows as an outlier. The data set characterized by a granularity of 74153 stencil data elements or 2 PX particles shows constant execution time even as more parallelism resources in the form of increased number of OS threads are added. This is because at maximum 2 OS threads are needed to evaluate the 2

TABLE 6.6: 1 Step HPX Results: Result data a problem size of 100,000 particles using the HPX version. All measurement data is presented in milliseconds

	2	4	8	16	28
74153	7350.46	7120.50	7254.58	7287.76	6830.52
37076	8455.33	5641.84	5478.74	5279.44	5164.02
18538	9063.87	5754.77	3212.94	3390.33	3093.54
9269	9857.23	5485.39	3183.79	1792.67	1834.43
4634	9136.28	4735.02	2578.09	1496.10	1415.78
2317	9100.13	4661.99	2565.59	1535.59	1212.78
1158	9217.61	5550.23	3630.82	2811.31	2494.09

PX particle ensemble. Even if 4, 8, 16, or 28 OS threads are added a maximum of 2 OS threads will be used for the computations. As the granularity PX particles decrease and the number of PX particles increase the PX particles are able to better pack the work queue resulting in better scalability with the increase in parallelism (OS threads). The large problem size of 100,000 bodies implies that finer granularities (for example 128 PX particles) have larger workload (1158 actual stencil elements) hence continue to provide scaling as parallelism is added.

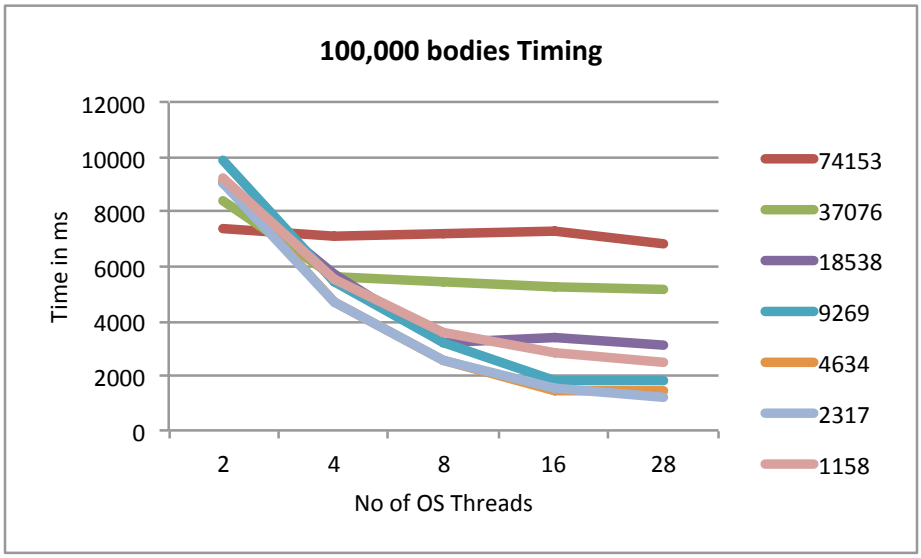


FIGURE 6.4: Timing in the 100,000 body problem using HPX version of Barnes Hut N-Body problem

6.4.2 10 Iterations

Similar set of experiments were carried out with the problem sizes of 10,000 and 100,000 particles to understand the impact of increased iterations on granularity and parallelism. For each of the problem sizes, granularity and parallelism combinations the time taken to execute the force calculation function was measured.

TABLE 6.7: 10 iteration for a problem size of 10,000 bodies. Result data from a series of runs using the HPX version. All measurement data is presented in milliseconds

	2	**4**	**8**	**16**	**28**
7428	4062.88	4056.97	4059.17	4055.47	3947.94
3174	5153.08	3243.98	2886.97	2835.08	2894.68
1857	5588.17	3307.42	2932.51	1984.37	2010.09
928	5297.27	2998.28	1857.74	1172.82	1200.80
464	5317.26	2865.47	1786.89	1302.33	1085.58
262	5863.54	3533.17	2407.86	1979.07	1819.57
200	11063.9	9092.45	7825.03	7294.74	7174.03

Table 6.7 shows the results from the measurements for problem size of 10,000 bodies for 10 iterations and appropriately sized granularities. The pattern of 10x increase in the aggregate force calculation times of the 10,000 bodies over the 1 iteration runs is observed. Additionally for larger workloads such as the 10,000 body simulations the optimal performance of the parallel evaluation of force function shifts to lower granularities provided there is increased parallelism. It is important to note that the improved performance in the 10,000 body simulations is maintained, when the number of iterations is increased. This behavior is observed and captured in figure 6.5.

6.5 Implications

The correlation between the granularity of the workload, number of OS threads and the overall problem size, with respect to the execution times of the force calculation function can be approximated by a multiprocessor scheduling problem. In the case of the HPX version of the Barnes Hut N-Body algorithm, as the granularity of the workload increases, the computations become more coarse grained resulting in inefficient load balancing. Consequently adding parallelism to coarse

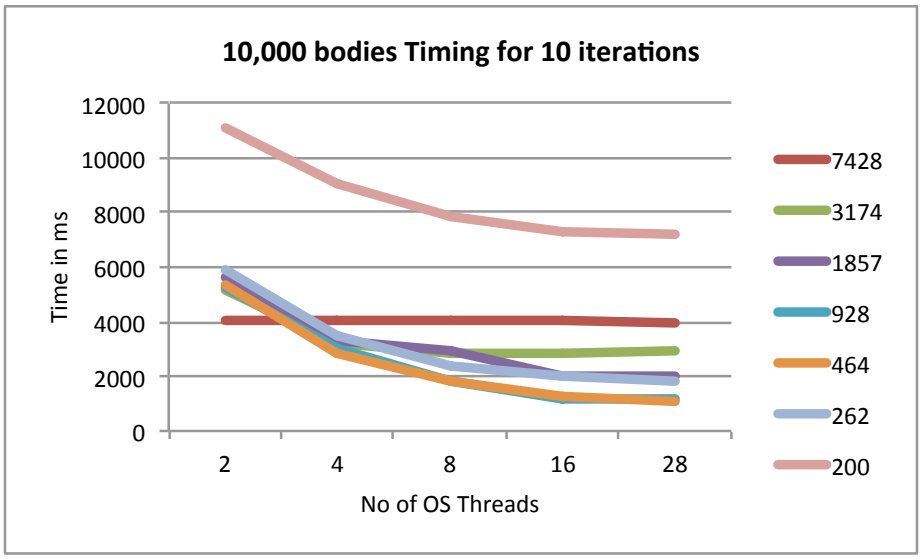


FIGURE 6.5: Timing in the 10000 body problem using HPX version of Barnes Hut N-Body problem for multiple iterations

grained workloads doesn't provide much scalability. As the granularity decreases, the workload becomes more fine grained and hence is more amenable to efficient load balancing. In the case of fine granular workloads, adding parallelism by increasing the number of threads results in easily packing the queues of the workload with sufficient work, providing efficient load balancing and extreme scaling. The main caveat for the fine grained load balancing is that potential for overheads to dominate the computations is higher. The optimal load balancing can be achieved when the policies are finely tuned for specific problem sizes and provide a fair balance between managing overhead and workloads of the right granularity.

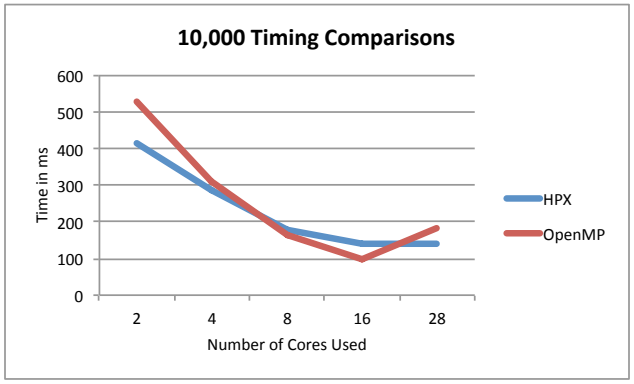


FIGURE 6.6: Comparison of timing in 10000 body problem using HPX and OpenMP versions of Barnes Hut N-Body problem for multiple iterations

Comparative timing studies of the best performance across granularities of HPX versus the complementary OpenMP version of the problem for the same number of operating system threads demonstrates on par results of both approaches. In some cases HPX version of the Barnes Hut N-Body algorithm outperforms the OpenMP version of the program. This is observed in both problem sizes of 10,000 (see fig 6.6) and 100,000 (see fig 6.7) bodies.

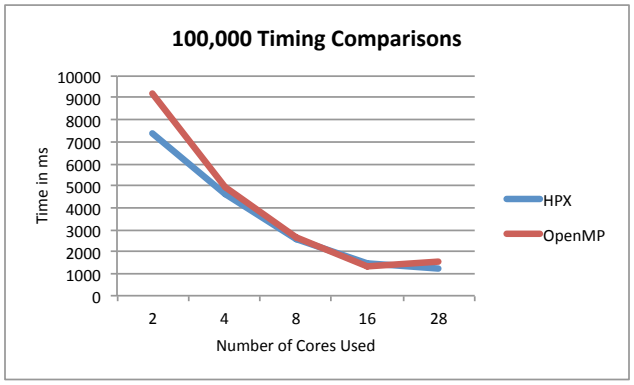


FIGURE 6.7: Comparison of timing in 100000 body problem using HPX and OpenMP versions of Barnes Hut N-Body problem for multiple iterations

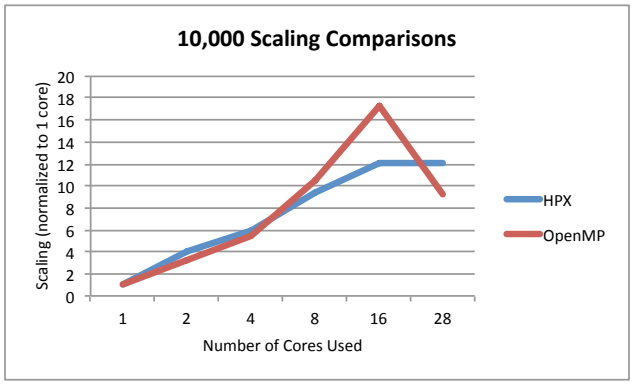


FIGURE 6.8: Comparison of Scaling in 10000 body problem using HPX and OpenMP versions of Barnes Hut N-Body problem for multiple iterations

Comparative scaling studies of the best performance (measured in fastest time to execute the parallelized force calculation function) across granularities of HPX versus the complementary OpenMP version of the problem for the same number of operating system threads demonstrates that the HPX version of Barnes Hut has comparable scaling to the OpenMP version of Barnes Hut as the problem size increases. In some cases HPX version of the Barnes Hut N-Body algorithm

outscales the OpenMP version of the program. This is observed in both problem sizes of 10,000 (see fig 6.8) and 100,000 (see fig 6.9) bodies.

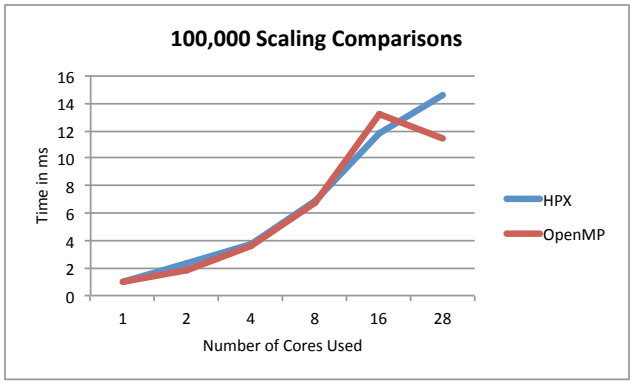


FIGURE 6.9: Comparison of Scaling in 100000 body problem using HPX and OpenMP versions of Barnes Hut N-Body problem for multiple iterations

These experiments demonstrate the relationship between the problem size, granularity of computations and parallelism obtained. Larger problem sizes result in more parallelism consequently better scaling and performance when developed using the ParalleX context. Fixed granularity computations do not always provide the best performance. The experiments conducted in this research demonstrate that granularities when tailored for problem sizes and parallelism can provide the best performance.

These experiments also demonstrate the correctness of ParalleX semantics when used to parallelize the Barnes Hut N-Body problems. The results obtained from the ParalleX version of the Barnes Hut algorithm are consistent irrespective of the number of iterations used or the problem size. The performance, scaling and the consistent results validate the viability of ParalleX semantics for variable data structure problems such as the Barnes Hut N-Body problem.

Chapter 7
Conclusions and Future Works

This thesis explores the space of parallel execution and management of dynamic data structure applications, with the goal of providing a framework for enabling extreme scalability in dynamic applications. The research described in this thesis demonstrates viability of dynamic data driven computations enabled by the ParalleX execution model. With the overall goal of providing extreme scalability to scaling constrained applications especially those involving trees and graphs, this thesis provides demonstration of techniques to exploit parallelism within each iteration of the Barnes Hut algorithm. However providing further scaling future versions of the implementation will need to break the global barriers. This last chapter highlights the key intellectual contributions of this thesis, the technical challenges faced and finally the next steps in evolving these techniques to an advanced level of applicability.

7.1 Intellectual Contributions

The HPX Barnes Hut implementation, which is the core part of this thesis is the first implementation of a tree based scientific computation using the ParalleX model of execution. This research is a first important step towards providing extreme scalability to dynamic graph data structure based applications. Following are the key intellectual contributions of this thesis.

- **Semantic Correctness** This thesis demonstrates the semantic correctness of the ParalleX semantics. Several ParalleX mechanisms have been implemented to enable exploitation of the parallelism within each iteration. The results obtained using these approaches are correct and on par with conventional approaches to providing parallelism. The technical approach and the results obtained, successfully establish the correctness of the dynamic mechanisms used to expose parallelism within each iteration of the Barnes Hut N-Body problem.

- **Enabling Parallelism** Parallelism fundamental to the accomplishment of this thesis is the definition, identification, and exposure of new forms and functional characteristics of the semantics of parallelism through the exploitation of which dramatic increase in scalability is to be achieved. This parallelism comes not from the conventional methods of control fork-join parallelism or data SIMD parallelism but the dynamic exposure of inherent parallelism of the data structure meta-data through advanced techniques of data-driven control, merging control and data parallelism into a single composite form of concurrency.

- **Dynamic Load Balancing** This thesis demonstrates that complex applications especially those involving dynamically evolving data structures can benefit from dynamic load balancing through runtime allocation of resources to tasks and data. Applications such as the Barnes Hut N-Body algorithm where the controlling tree data structure evolves with every iteration, do not fully expose the parallelism inherent within the algorithm at compile time. Conventional parallelism approaches often do not provide the necessary control mechanisms to perform dynamic load balancing. While loop parallelism techniques such as OpenMP provide dynamic scheduling for the coarse grained workload, the ParalleX execution model as implemented by the HPX provides multilevel parallelism. The Barnes Hut HPX implementation takes advantage of the ParalleX model semantics to demonstrate the potential dynamic runtime balancing of the workload. In the Barnes Hut HPX implementation, the workload dynamically evolves during the simulation, as new particle-particle, particle - intermediate node interactions are created at each iteration. The simple dyanamic load balancing methodologies account for this variance, and load balance the new interaction lists to equally divide the work over the allocated resources. The HPX implementation provides further load balancing features through "work-stealing" in threads that allow the runtime system to handle any un-addressed load balancing issues. Using these two techniques, this thesis has successfully demonstrated the effect of dynamic load balancing on reducing factors of performance degradation, such as starvation, by utilizing all the allocated resources to their fullest capa-

bilities. Alternative policies for load balancing, when implemented can allow applications to take advantage of variable grain size to provide a workload distribution that provides the best performance.

- **Data Locality and Granularity Control** This thesis demonstrates that dynamic applications, where the work load evolves during application execution, can benefit from support for mechanisms that allow for controlling the granularity of the workload to be executed on each of the allocated resources. In the Barnes Hut N-Body algorithm, the application utilizes HPX mechanisms to change the granularity of the workload executed by HPX threads during runtime, allowing for dynamic optimization of workloads. Dynamic applications especially tree based and graph based simulations, have poor data locality. In the Barnes Hut HPX algorithm, the granularity control is implemented such that closely located particles are clustered together, providing high data locality and performance improvements. This is critical to the migration of the effects of overhead which are limited to software implementation on conventional computing platforms.

- **Data Driven Computations** The HPX Barnes Hut implementations exploits futures based asynchrony and AGAS based globally unique identifiers to facilitate on-demand data driven force computation. Conventional parallelism approaches rely on access to data through either system architectures such as shared memory where the data resident in a shared memory can be accessed by all computing resources, or through message passing where in many cases critical path of execution is blocked until remote data can be fetched. In the HPX Barnes Hut implementation futures based data resolution allows for, local force computations to continue executing for available data while the "get" operation is performed. In the case of remote fetches, which cannot be satisfied immediately, the executing HPX thread suspends operation allowing for another stencil element to continue force calculation. This on-demand

data access provides asynchrony without blocking the critical execution path, there by allowing the application to fully exploit the underlying resources.

7.2 Technical Challenges

One of the key impediments in developing a performance oriented application using the extreme scale execution model implementation was that the application development required direct interfacing with the underlying software parallelism structures. For complex application such direct interfacing with the parallelism primitives would limit adoptability and applicability of the runtime system. In terms of the Barnes Hut problem, a more usable programming model interface could better enable expression of parallelism thereby providing more opportunities to exploit the capabilities of the underlying execution model. ParalleX Programming Interface, XPI, currently under development could offer one such interface to enabling abstraction from the runtime system.

The execution environment of the HPX based system comprises mostly conventional architectures and systems (shared memory and distributed memory). The execution model semantics are geared towards providing scalability for next generation extreme scale architectures such as Xcaliber, while providing an upgrade path using a performance oriented runtime system for current generation systems. ParalleX based applications would perform better on next generation architectures, since current generation system structures are fundamentally handicapped by incremental approaches taken by core architectures and software systems. This is particularly true for the software implementations of support mechanisms that impose undue temporal overheads. Demonstrating the productivity gained for dynamic applications and enabling new extreme scale dynamic graph applications, would allow for much needed new ideas to emerge in the computer architecture and systems organizations.

7.3 Future Works

The core contribution of this thesis involves implementing futures based asynchrony within each iteration of the Barnes Hut N-Body simulation. In order to gain extreme scalability new tree based

algorithms need to be explored. This section of the thesis describes a few such approaches with the goal of improving scalability of tree based N-Body simulations.

7.3.1 Optimization the HPX Barnes Hut Implementation

The current implementations of Barnes Hut algorithm can be updated to provide a better approximation criteria than what is currently being used $D/r < \theta$. Salmon and Warren presented a more accurate approximation criteria that takes into account the relative particle positions to the intermediate node. This stability provides a more accurate representation of the force that a remote cluster has on the particle under consideration. A key update in the future works would be to implement a better approximation criterion. The current implementation of the Barnes Hut algorithm does not account for multipole moments. Providing multipole capabilities to the force computation function can provide more accurate acceleration and force calculations. Another important update for a future version of the implementations would be to add multipole capabilities; however doing so can potentially increase the workload at the force computation stage. Analysis and experimentation would need to be performed to find the right balance of computational accuracy and performance requirements. The HPX version of the Barnes Hut N-Body code can be optimized further to combine the tree tagging operations with the computation of center of mass. Further algorithmic updates such as computing the balanced interaction list in tandem with the interaction list of the actual particles can be carried out. These enhancements can provide for minor improvements in performance, as they are not computationally intensive. The performance of the current implementation of HPX Barnes Hut force calculation function can be improved by developing new approaches for processing PX particles. The current implementation requires iterating through the PX particles, and then mapping the indices with an interaction list to determine interaction. This process introduces a huge performance overhead. A different data structure, which has moderate item insertion cost and very low access cost, such as a hash-map could be considered for managing look-ups of actual particle interaction. Such an approach would dramatically reduce the overheads and provide better performance characteristics than the conventional parallelism approaches. Pro-

gramming models other than the stencil based approach to using HPX could be explored to represent the tree based dynamic algorithm, facilitating better exposure of parallelism at critical areas.

7.3.2 Distributed Shared Memory

The current version of the HPX based Barnes Hut N-Body implementation has been written with support for multiple localities. However all the simulations have been carried out within a single locality environment, which relies on a shared memory system architecture where the entire shared memory system, is considered to be a single locality. This is because the HPX runtime system is currently under active development and supports single locality simulations effectively. The active global address space in HPX is being rewritten to provide robust multi-locality support. For the simulations to exploit cluster-like conventional architectures where each of the compute nodes forms its own locality, would require multi-locality support. With a robust active global address space and multi-locality support in place, the HPX Barnes Hut N-Body algorithm can utilize the conventional distributed memory systems more effectively. Experiments with a distributed memory system architectures and the HPX Barnes Hut N-Body algorithm can then be carried out to demonstrate scalability of dynamic techniques over multiple localities. A key milestone in the evolving experiment space of HPX based Barnes Hut N-Body simulations would be to better understand the scalability comparisons against an MPI version of the Barnes Hut N-Body implementation in a distributed memory environment.

7.3.3 Breaking The Global Barriers

The Barnes Hut algorithm approach for evolving N-Body systems involves creation of a global tree data structure, computing center of mass, calculating forces and finally applying forces to the bodies and generating new position and velocity vectors for each of the bodies. Implicit in the Barnes Hut algorithm is a global barrier where at the beginning of each iteration a new tree is created. In order to create the new tree, it is imperative in the algorithm that the new positions of the bodies be updated before the tree creation process starts. This requirement of an updated

global state of the system at the beginning of each iteration, constrains the exploitable parallelism to a per-iteration basis.

In order to take full advantage of the parallelism offered by ParalleX, new approaches will be needed to relax the constraint of knowing the full state of the system a-priori. One approach towards doing this could include using statistical / predictive approaches to estimate the new position of particles. This approach relies on the fact that gravitational N-Body systems are slowly evolving systems and (depending on the value of δt) new positions of the particles are close to their positions in prior iterations. Breaking this global barrier can allow a future Barnes Hut N-Body simulations to exploit parallelism across multiple iterations simultaneously. This approach could dramatically increase the scalability and efficiency of future N-Body applications.

Results [TAB+10] from simulation of another scientific application Adaptive Mesh Refinement [BO84], demonstrate that breaking global barriers for dynamic applications can have upto an order of magnitude improvement in parallelism, and can lead to superior load balancing over conventional techniques. The one dimensional AMR implementation using HPX demonstrates improvement in scalability of 5 - 10 % depending on simulation parameters. When 2-3 levels of refinement are used the HPX AMR implementation outperforms and out-scales the MPI implementation. The 3D AMR HPX application that is currently being implemented shows promise of even greater scalability and performance over current generation programming models.

7.3.4 Broader Applicability to Molecular Dynamics

The core kernels simulated in the Barnes Hut N-Body approach are very similar to other applications such as protein folding and molecular dynamics simulations. Applications of a more advanced form from the current HPX Barnes Hut N-Body parallel implementations can in principle apply to these new domains resulting in significantly better scalability and performance for a substantial High Performance Computing application base. In the space of molecular dynamics the problem size (number of particles) does not grow substantially with simulation scales on the order of tens of thousands to a million particles depending on the nature of system that is simulated. Since the

problem size remains constant across simulations, the molecular dynamics simulations are strong scaling applications. Molecular dynamics simulations often involve multi-scale potentials (force fields) to account for short range interactions and long-range interactions. The forces in molecular dynamics simulations are very different to those in astrophysics. These two forces occur randomly during simulation and hence can result in workload imbalance across the simulation. Consequently conventional programming models cause scalability limitations. These kinds of problems can benefit from the dynamic load balancing and asynchronous environment provided by HPX and the underlying ParalleX execution model upon which it is based.

Bibliography

[ABH+03] Wendell Anderson, Preston Briggs, C. Stephen Hellberg, Daryl W. Hess, Alexei Khokhlov, Marco Lanzagorta, and Robert Rosenberg. Early experience with scientific programs on the cray mta-2. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 46–, Washington, DC, USA, 2003. IEEE Computer Society.

[AJ89] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 293–302, New York, NY, USA, 1989. ACM.

[Alf94] Robert A. Alfieri. An efficient kernel-based implementation of posix threads. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 5–5, Berkeley, CA, USA, 1994. USENIX Association.

[AN90] K. Arvind and Rishiyur S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3):300–318, 1990.

[App85] Andrew W. Appel. An efficient program for many body simulation. *SIAM Journal on Scientific and Statistical Computing*, 6(1):85–103, 1985.

[BBK+08] E. Bohm, A. Bhatele, L. V. Kalé, M. E. Tuckerman, S. Kumar, J. A. Gunnels, and G. J. Martyna. Fine-grained parallelization of the car-parrinello ab initio molecular dynamics method on the ibm blue gene/l supercomputer. *IBM J. Res. Dev.*, 52(1/2):159–175, 2008.

[BH77] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[BH86] Joshua Barnes and Piet Hit. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[BO84] M J Berger and J Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comp. Phys.*, 53:484, 1984.

[Bon02] Dan Bonachea. Gasnet specification, v1.1. Technical report, Berkeley, CA, USA, 2002.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[CDK+01] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[CFS99] Larry Carter, John Feo, and Allan Snavely. Performance and programming experience on the tera mta. In *In SIAM Conference on Parallel Processing*, 1999.

[CGR88] J. Carrier, L. Greengard, and V. Rokhlin. A Fast Adaptive Multipole Algorithm for Particle Simulations. *SIAM Journal on Scientific and Statistical Computing*, 9(4):669–686, 1988.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[cil09] Cilk++: Parallelism for the masses. 2009.

[Cou43] Richard L. Courant. Variational Methods for the Solution of Problems of Equilibrium and Vibration. *Bulletin of the American Mathematical Society*, 49:1–23, 1943.

[Daw83] John M. Dawson. Particle simulation of plasmas. *Reviews of Modern Physics*, 55(2):403–447, April 1983.

[DBL+10] James Dinan, Pavan Balaji, Ewing Lusk, P. Sadayappan, and Rajeev Thakur. Hybrid parallel programming with mpi and unified parallel c. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 177–186, New York, NY, USA, 2010. ACM.

[Den80] J. B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[DMSS10] JJ Dongarra, HW Meuer, E Strohmaier, and HD Simon. The top 500 list, July 2010. `http://www.top500.org/`.

[dRBC93] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel i/o via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21:31–38, December 1993.

[DS92] A. Deshpande and M. Schultz. Efficient parallel programming with linda. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 238–244, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Dub96] John Dubinski. A parallel tree code. *New Astronomy*, 1(2):133 – 147, 1996.

[ECGS92] T.v. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: A mechanism for integrated communication and computation. *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pages 256–266, 1992.

[EGCSY03] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[EGS06] Tarek El-Ghazawi and Lauren Smith. Upc: unified parallel c. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 27, New York, NY, USA, 2006. ACM.

[FW60] G. E. Forsythe and W. R. Wasow. *Finite-difference methods for partial differential equations*. 1960.

[GL02] William Gropp and Ewing Lusk. Advanced topics in mpi programming. pages 199–235, 2002.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6):789–828, 1996.

[GOS78] David Gottlieb, Steven A. Orszag, and G. A. Sod. Numerical analysis of spectral methods: Theory and application. *Journal of Applied Mechanics*, 45(4):969–970, 1978.

[GR97] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 135:280–292, August 1997.

[GT97] Guang R. Gao and Kevin B. Theobald. The htmt program execution model. Technical report, In Workshop on Multithreaded Execution, Architecture and Compilation (in conjunction with HPCA-4), Las Vegas, 1997.

[HB78] C. Hewitt and H. G. Baker. Actors and continuous functionals. Technical report, Cambridge, MA, USA, 1978.

[HE88] R. W. Hockney and J. W. Eastwood. *Computer simulation using particles*. 1988.

[Her87] L. Hernquist. Hierarchical n-body methods. In *Particle Methods in Fluid Dynamics and Plasma Physics*, 1987.

[Hil87a] W. D. Hillis. Programming a highly parallel computer. *Nature*, 326:27–30, mar 1987.

[Hil87b] W. Daniel Hillis. The connection machine. *Sci. Am.*, 256:108–115, June 1987.

[HMM95] P. Hut, J. Makino, and S. McMillan. Building a better leapfrog. *Astrophysical Journal*, 443:L93–L96, apr 1995.

[Int10] Intel. Intel thread building blocks 3.0, 2010. http://www.threadingbuildingblocks.org.

[JGM+08] P. Jetley, F. Gioachin, C. Mendes, L.V. Kale, and T. Quinn. Massively parallel cosmological simulations with changa. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, April 2008.

[KBB+98] Laxmikant V. Kalé, Milind A. Bhandarkar, Robert Brunner, N. Krawetz, J. Philips, and A. Shinozaki. Namd: A case study in multilingual parallel programming. In *LCPC '97: Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, pages 367–381, London, UK, 1998. Springer-Verlag.

[KK93] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, 1993.

[KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artif. Intell.*, 6(4):293–326, 1975.

[LB94] Pangfeng Liu and Sandeep N. Bhatt. Experiences with parallel n-body simulation. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, SPAA '94, pages 122–131, New York, NY, USA, 1994. ACM.

[MHI07] Junichiro Makino, Kei Hiraki, and Mary Inaba. Grape-dr: 2-pflops massively-parallel computer with 512-core, 512-gflops processor chips for scientific computing. In *SC*, page 18, 2007.

[Mic10] Microsoft. Microsoft parallel pattern library, 2010.

[NR98] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[NWD93] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The j-machine multicomputer: an architectural evaluation. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 224–235, New York, NY, USA, 1993. ACM.

[PH10] Jun Makino Piet Hut. Moving stars around. `http://http://www.artcompsci.org//`, 2010. [Online; accessed 19-Feb-2010].

[RHH85] Jr. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[RLV+10] Abtin Rahimian, Ilya Lashuk, Shravan Veerapaneni, Aparna Chandramowlishwaran, Dhairya Malhotra, Logan Moon, Rahul Sampath, Aashay Shringarpure, Jeffrey Vetter, Richard Vuduc, Denis Zorin, and George Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. *SC Conference*, 0:1–11, 2010.

[Sal91] John K. Salmon. *Parallel hierarchical N-body methods*. PhD thesis, Pasadena, CA, USA, 1991.

[SC88] A.J. Semtner and R. M. Chervin. A simulation of the global ocean circulation with resolved eddies. *Journal of Geophysical Research*, 93:15502–15522, December 1988.

[SDS+09] David E. Shaw, Ron O. Dror, John K. Salmon, J. P. Grossman, Kenneth M. Mackenzie, Joseph A. Bank, Cliff Young, Martin M. Deneroff, Brannon Batson, Kevin J. Bowers, Edmond Chow, Michael P. Eastwood, Douglas J. Ierardi, John L. Klepeis, Jeffrey S. Kuskin, Richard H. Larson, Kresten Lindorff-Larsen, Paul Maragakis, Mark A. Moraes, Stefano Piana, Yibing Shan, and Brian Towles. Millisecond-scale molecular dynamics simulations on anton. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 39:1–39:11, New York, NY, USA, 2009. ACM.

[Sha50] C Shannon. Programming a computer for playing chess. *Philosophical Magazine*, (7):41–314, 1950.

[Sin93] Jaswinder Pal Singh. *Parallel hierarchical N-body methods and their implications for multiprocessors*. PhD thesis, Stanford University, Stanford, CA, USA, 1993. UMI Order No. GAX93-17818.

[SL91] K. E. Schmidt and M. A. Lee. Implementing the fast multipole method in three dimensions. *Journal of Statistical Physics*, 63:1223–1235, jun 1991.

[TAB+10] Alex Tabbal, Matthew Anderson, Maciej Brodowicz, Hartmut Kaiser, and Thomas Sterling. An application driven analysis of the parallex runtime system. *Performance Modeling, Benchmarking and Simulation of High performance Computing Systems, Workshop at Super Computing Conference, New Orleans*, 2010.

[tbb09] Tbb: Intel thread building blocks, 2009.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[WS93] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 12–21, New York, NY, USA, 1993. ACM.

[Yel07] Yelick, K. and Bonachea, D. and Chen, WY. and Colella, P. and Datta, K. and Duell, J. and Graham, SL. and Hargrove, P. and Hilfinger, P. and Husbands, P. Productivity and performance using partitioned global address space languages. In *Intl. Conf. on Symbolic and Algebraic Computation, Proceedings of the 2007 international workshop on Parallel symbolic computation*, London, Ontario, Canada, 2007.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance java dialect. In *In ACM*, pages 10–11. ACM Press, 1998.

Vita

Chirag Dekate was born in Bhopal, India, in 1981. He completed his B.S in 2002 from Louisiana State University, Baton Rouge, majoring in Computer Science. He completed his Master of Science in System Science in 2004 from Louisiana State University, Baton Rouge. He co-authored 10 peer-reviewed articles and participated in over 5 major technical demonstrations. He contributed to 3 research projects and reviewed articles for 2 conferences.

Chirag Dekate is also an high performance computing researcher at the Center for Computation and Technology. His research interests are in understanding and developing extreme scale high performance computing software systems, specifically runtime management systems for new innovative high performance computing applications.

His most significant publications are:

- R. Murphy, T. Sterling, and C. Dekate: *"Advanced Architectures and Execution Models to Support Green Computing"*, in Computing in Science and Engineering. Volume 12, Issue 6, November 2010, pp 38-47.

- T. Sterling, and C. Dekate: *"Enabing Exascale Through ParalleX Paradigm"*, in Parallel Computational Fluid Dynamics: Recent Advances and Future Directions. 21st International Conference on Parallel Computational Fluid Dynamics 2010, pp 3-18.

- T. Sterling, and C. Dekate: *"Productivity in High-Performance Computing"*, in Advances in Computers, Elsevier, Volume 72, High Performance Computing, 2008, pp 101-134.