EXTENDING PARATREET, A FRAMEWORK FOR SPATIAL TREE BASED ALGORITHMS, WITH GPU KERNELS AND LOAD BALANCERS

BY

SIMENG LIU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Adviser:

Professor Laxmikant Kale

ABSTRACT

Improving the performance of iterative, computationally heavy applications with frequent memory access is challenging and exciting. This thesis shows the performance improvement efforts of the paraTreeT library. ParaTreeT is a parallel tree toolkit inspired by the N-body simulation problem to model and investigate the dynamic motion of astronomical bodies given a set of initial conditions. ParaTreeT provides a generic parallel tree traversal framework targeting high scalability and programmability. The inputs from the user are partitioned and decomposed into leaf nodes of a chosen tree structure. The interactions among particles are done through traversals of a global tree. Users apply their custom structs of user data and define the tree type into which the data is partitioned, as well as the partition algorithm used. In addition, the library is extendable with custom traversal algorithms.

ParaTreeT has achieved better central processing unit (CPU) performance compared to its predecessor ChaNGa by providing tree data using a shared-memory cache model, as well as separating the data computation functionality from its spatial tree representation. ParaTreeT demonstrated a 2-3x speedup over ChaNGa using up to 256 nodes (21504 threads) on the Summit's POWER9 machine.

This thesis focuses on improving the performance of the ParaTreeT framework through two approaches.

The first approach is to implement load balancing strategies to speed up the iterative code with growing load imbalance. This thesis presents different load balancing algorithms and efforts to make them scalable. With theoretical runtime analysis of each algorithm, together with scaling experiments, the thesis identifies the scaling bottleneck and scalability of each algorithm.

The second approach is to add general-purpose computing on graphics processing units (GPGPU) kernels to offload highly parallel computationally intensive work from CPU hosts to graphics processing units (GPUs). This thesis identifies the computationally heavy blocks of the CPU implementation and proposes multiple GPU kernels to speed up the computation. The thesis also analyzes the overhead of the GPU implementation, with discussion of plans for future improvements.

To my parents Junping Liu and Jun Cao,
to my boyfriend Mark Kraman,
and my angel Meadow,
for their love, support, and company.
To Professor Laxmilant Kale for his support and guidance.

ACKNOWLEDGMENTS

Foremost, I would like to thank my advisor Professor Laxmikant Kale for his sponsorship of my master's study and research.

In addition, I would like to appreciate the major developer of the paraTreeT library, Joseph Hutter, the advisor Thomas Quinn and developers Jaemin Choi and Justin Szaday for their great work and help. I also thank the developers and maintainers of Charm++.

I acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing high performance computing (HPC) resources that have contributed to the research results reported in this paper. URL: http://www.tacc.utexas.edu.

This work was supported by the extreme science and engineering discovery environment (XSEDE) supported by National Science Foundation grant number ACI-1548562.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 PROGRAMMING MODEL OF CHARM++

Charm++[1] is a parallel programming language built on top of C++. Compared with the widely used MPI[2] which is process-centric, Charm++ adopts the programming model of migratable objects. The basic unit of Charm++ is a *chare* which is similar to an actor. A *chare* is typically a C++ class with a collection of logically related functions. Users can define functions in a *chare* as *entries* to make it remotely invocable. *Entry* functions can be called by other *chare* objects potentially remotely in an asynchronous way. *Chares* can group into a *chare array* to leverage the object-oriented natural of C++ classes with overdecomposition. To perform overdecomposition, Charm++ typically partitions the domain of the problem into finer grain sizes than the available processing units. In this way, multiple *chares* shares one processing unit. With a combination of asynchronous execution and a runtime scheduler, Charm++ provides a natural overlap of communications and computations. There is not a fixed mapping between *chares* and processing units. All *chare* objects are migratable. Users can specify variables to be packed with one object such that with variable unpacking, there is no effect of the change of location. Charm++ [3] has supported several diverse science and engineering applications with its message-driven execution model with migratable objects and an adaptive runtime.

1.2 THE CHARM++ LIBRARY WITH BUILT-IN LOAD BALANCING FRAMEWORK

Load balancing (LB) is a common technique to speed up iterative applications with multiple actors. The computation and communication behavior in the previous iteration of each *chare* gives us a good estimation of the future. With the migratable objects and overdecomposition scheme adopted by Charm++, load balancing can be achieved by optimizing the placement of *chares* to processing units. A better placement can impose less communication traffic or smooth the accumulated computation on each processor. Charm++ [4] provides a generic load balancing framework to achieve measurement based optimization. The Charm++ runtime tracks the execution and communication of each *entry* function call for each *chare* in a load balancing database. When the load balancing step is called, the Charm++ will apply a load balancing strategy to the collected *chare* data to generate a new placement mapping for each *chare*. Runtime orchestrated dynamic load balancing can

adapt to runtime changes of actor grain size and evolving load distribution at each iteration.

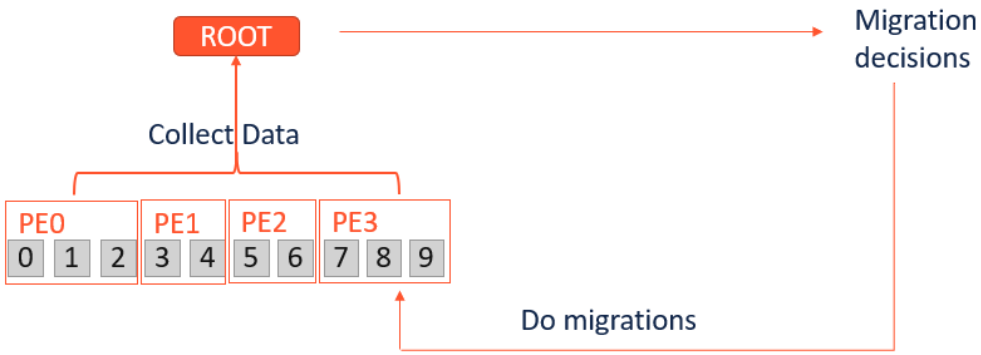


Figure 1.1: Diagram of the workflow of centralized LB.

A load balancing strategy has three critical parts, which are data collection, decision algorithm, and object migration. The load balancing framework supports multiple strategies. Figure 1.1 shows the workflow of the *central* strategy. Object data from all PEs will be grouped together to a root PE via a global reduction. The root PE will then execute the strategy algorithm on the collection of global data to generate migration decisions. As a final step, the LB framework will migrate objects according to the decisions. Distributed strategies, on the other hand, groups object data to local PEs. All PEs will execute the strategy algorithm in parallel and send migration decisions of local objects. The current integrated load balancing framework is described in [5] in detail.

An algorithm can work with any strategies adopting the data collection and algorithm execution model. To describe a specific load balancing strategy, for example, a *central* strategy coupled with a refinement decision algorithm, we will call it a CentralRefineLB. A greedy decision algorithm extending the distributed strategy will be called a DistributedGreedyLB.

Decision algorithms working with the *central* strategy tend to be easy to program but suffer from poor scalability. In contrast, distributed decision algorithms are harder to program but can have better scalability if done nicely. Diffusion [6] based distributed load balancing algorithm is a typical example to show the efficiency and scalability of distributed decision algorithms.

1.3 THE CHARM++ LIBRARY WITH GPGPU SUPPORT

CUDA kernels can be injected into Charm++ programs without additional support. However, with the overdecomposition and asynchronous message-driven execution, synchronization between host and device may be expensive. The Charm++ runtime scheduler will be

blocked from handling incoming messages or scheduling work from other *chares* sharing the same PE.

It is a good practice for each *chare* to have their own CUDA stream to schedule memory transformation or computation kernels asynchronously. Since kernel launches and data transfer enqueued on the same stream will be executed in order, we can ensure dependencies with ease. Choi [7] showed that we could achieve better overlap of computation and communication with CUDA kernels by having multiple streams per *chare*. Choi created a stream with higher priority to deal with all communication related operations while having a stream with lower priority doing computation kernels only. With multiple streams in each *chare*, computation kernels can run concurrently with communication kernels without dependencies. CUDA events need to be added to enforce dependencies between streams.

CUDA enables the invocation of host functions based on the completion of device kernels. Users may want to use cudaLaunchHostFunc to enqueue a host function in the CUDA stream so that once the previous kernel finishes, a host function will be launched. In this case, the CPU thread generated by the CUDA runtime will not be managed by the Charm++ runtime. To support such a use case, *Hybrid API* (HAPI) calls are implemented. Users can call *hapiAddCallback* with a CUDA stream and a Charm++ callback object. Once the Charm++ runtime detects the completion of the recorded CUDA events on the given stream, the callback object will be scheduled for execution. The NAMD [8] application running on top of hetergenous systems shows that the efficient CPU and GPU kernels can scale well from thousands of CPU cores and thousands of GPU cores.

1.4 THE PARATREET LIBRARY

ParaTreeT [9] is a generic library for tree-based algorithms which was initially motivated by the N-body simulation problem and built for generic applications. For a universe containing n particles, calculating interactions between all pairs of particles imposes an algorithm with the complexity of $\mathcal{O}(n^2)$. In 1986, Barnes and Hut [10] published an algorithm with $\mathcal{O}(nlog(n))$ complexity which soon became the standard for N-body simulation. The Barnes and Hut algorithm decomposes particles into a hierarchical spatial tree (usually an octree) to capture the spatial relationship among particles and do interactions among particles via traversing along the tree. The pseudocode of the algorithm is in Listing 1.1:

Listing 1.1: Pseudocode of the BarnesHut algorithm

```
1 void BarnesHut(particle, node){
2     if (far_away(particle, node)){
```

```
3           calculateGravity(particle, node);
4       } else if (hasChild(node)){
5           for (child_node : node.children()){
6               BarnesHut(particle, child_node);}
7       } else {
8           for (p : node.particles){
9               calculateGravity(particle, p);}
10      }
11  }
```

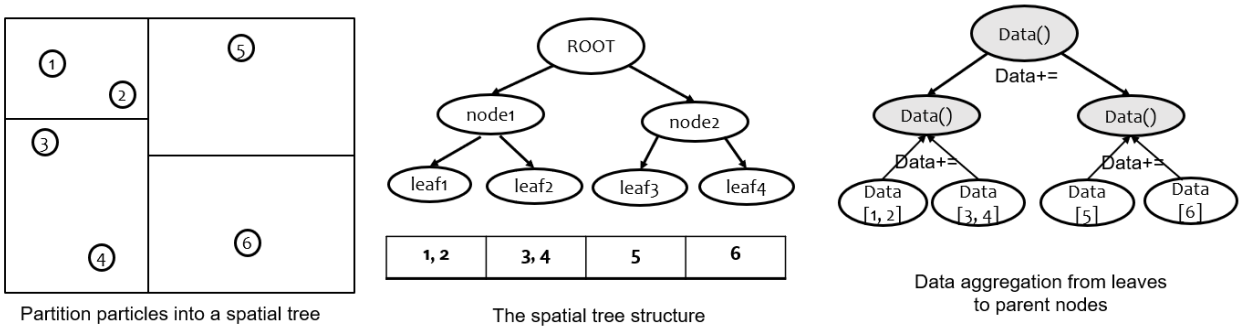


Figure 1.2: The process of building subtrees. The left graph shows a spatial partitions of particles into blocks. Each block becomes a leaf node in the tree shown in the middle graph. Internal nodes are constructed to connect the leafs to a root. The right graph shows data aggregation from leaves to the root.

ParaTreeT created four abstractions components to model the problem. *Trees* and *Data* are the fundamental concepts. *Traversal* and *Visitor* build on top of the basic one. Trees are built by partitioning the input particles from the top down following a specified tree type. The left and middle graphs in Figure 1.2 demonstrate a tree build of six particles into a tree with four leaves. Data is subject to application specific properties. Users can define a custom data class with required operators.

ParaTreeT creates a Charm++ chare array named *Subtree* to represent the spatial structure of the tree. Each Subtree *chare* will obtain a vector of leaves and internal nodes between the leaves and the global root. From leaves to the root, Subtrees aggregate data and populate internal nodes represented by gray nodes in the right graph in Figure 1.2.

On top of the concept of tree and data, *Traversal* defines an algorithm to visit each node in the tree structure and *Visitors* define the action to be performed at each step of the traversal. Multipole expansion can be used to approximate the gravitational force of a group of bodies. With higher powers of the distance, the total gravitational force of the group is proportional to the total mass divided by the square of the distance. Therefore, we define

the boolean function `open()` to determine if we want to keep traversing the children of an internal node or stop the traversal and contribute the gravitational force of all descendant particles with estimations using the aggregated data. We define function `node()` to calculate the gravitational forces between an internal node and a target node using estimation with the aggregated data. We define `leaf()` function to calculate gravitational forces between a source leaf node and a target node by interacting with every pair of particles.

We can now rewrite the single tree traversal Barnes-Hut algorithm to dual-tree traversals [11]. The pseudocode of the algorithm is shown in Listing 1.2

Listing 1.2: Pseudocode of the dual-tree traversal algorithm

```
1 void node(target_node, source_node){
2     for (particle in target_node){
3         update = calculateGravityWithEstimation(particle, source_node);
4         update(particle, update);
5     }
6 }
7
8 void leaf(target_node, source_node){
9     for (target_particle in target_node){
10        update = 0;
11        for (source_particle in source_node){
12            update += calculateGravity(target_particle, source_particle);
13        }
14        update(particle, update);
15    }
16 }
17
18 void DualTreeTraversal(target_node, source_node){
19    if (open(target_node, source_node)){
20        node(target_node, source_node);
21    } else if (hasChild(node)){
22        for (child_node : node.children()){
23            DualTreeTraversal(target_node, child_node);
24        }
25    } else {
26        leaf(target_node, source_node);
27    }
28 }
```

In paraTreeT, the global tree is distributed among a Charm++ *chare array* where each *chare* has a vector of leaf nodes. Each *chare* can do traversals simultaneously with local leaf nodes being the target nodes following a traversal of the global tree. When the source

node locates remotely, we need to access remote *chares* to bring the node data to the local processor. ParaTreeT [9] proposed a shared-memory cache model for tree data to insert the remote data to local caches to reduce communication overhead and duplication. Interactions involving remote nodes are buffered and executed later.

1.4.1 N-body Simultaitons on GPUs

A survey about used hardware of the history over the last five decades of N-body simultaitons is shown in Bédorf and Zwart [12]. Stock and Gharakhani [13] proposed to generate interaction lists on CPUs and accelerate the computations of interactions with GPU. Belleman et al.[14] uses GPU to calculate force and potentials while using CPU to do prediction and correction. Hamada et al.[15] proposed a solution that CPUs generate tree walks while GPU kernels executes multiple walks in parallel. Jetley et al.[16] integrates GPU kernels to compute interaction pairs in parallel with modified ChaNGa CPU code. Bédorf et al.[17] presented a sparse octree gravitational N-body code that runs on one GPU. It can achieve 20x speed up compare to CPU implementations. Liu et al.[18] moves the tree traversal to GPUs together with interaction computation.

CHAPTER 2: LOAD BALANCING STRATEGIES AND OPTIMIZATIONS

In this section, we present a few LB strategies that are triggered periodically during the execution of the paraTreeT program to detect and fix the unbalanced object distribution during execution.

For runtime analysis of load balancing algorithms, we define related parameters as follows:
Let α, β, and γ represent latency, bandwidth, and computation cost respectively.
Let P represent the number of PEs.
Let V represent the number of objects.
Let C represent any constant.
We will always analyze the average case of the runtime. With overdecomposition, we will always have more objects than the number of PEs. The ratio between V and P is a constant which is taken as an input in the paraTreeT program.

2.1 ORB BASED STRATEGY

Orthogonal Recursive Bisection (ORB) [19, 20] is an algorithm that recursively splits a space into two halves such that the elements in each half are approximately equal in some property

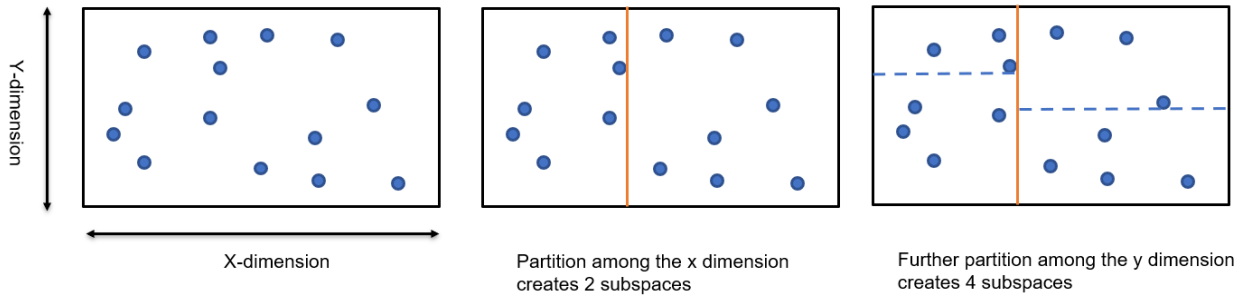


Figure 2.1: Partition of the input space into four blocks using the ORB algorithm.

The left graph shows the input space. The middle graph shows a partition along the x dimension. The right graph shows two partitions along the y dimensions with the halves spaces created from the last step.

Figure 2.1 shows an example of a 2D space with 14 elements and one partition along the x dimension and 2 partitions among the y dimension.

The ORB partition groups spatially nearby elements and provides spatial locality. When using the ORB algorithm to redistribute objects to processing elements, we select a splitting

coordinate such that the resulting partitions have an approximately equal sum of loads for the elements contained within.

Load balancing with ORB based algorithms has shown great performance with spatial inputs that execute short range interactions [21]. Fleissner et al.[22] partitions particles with ORB based algorithms, and use sampling to reduce the execution time of the partition step. In this section 2.1, we show centralized and distributed ORB based load balancing approaches integrated to the Charm++ load balancing framework.

2.1.1 Centralized ORB

With a centralized load balancing strategy, we first need to aggregate all object data to one PE via a global reduction. With the global array of objects, we apply the ORB algorithm to the longest dimension. We first need to sort all objects with 3D coordinates along with the increasing order of the chosen longest dimension. Next, we loop through the sorted array to find the splitting coordinate on the chosen dimension where the left and right halves will have even loads. We then repeat the algorithm on the splitted halves.

We now present the runtime analysis of the centralized ORB algorithm in Table 2.1. We first analysis the basic executions in the algorithm and their costs:

Table 2.1: Core steps of the centralized ORB algorithm.

S1	**Collection** Collect n object data from k PEs to a root via a reduction	$\mathcal{O}(log(k)(\alpha + n\beta))$
S2	**Sort** Quick sort an array of n objects with respect to a selected dimension	$\mathcal{O}(nlog(n))$
S3	**Solve** Find the even load point of the array of n objects	$\mathcal{O}(n)$

At the beginning of the load balancing phase, we first make a global reduction to collect object data (S1). Once we obtain an array of data for all V objects, we start to partition this array using the ORB algorithm into P parts.

We start with partitioning the whole array of length V, apply S2 and S3 on it, which makes a runtime of $\mathcal{O}(Vlog(V) + V) = \mathcal{O}(Vlog(V))$. Next, we partition the left and right halves to split the whole array into four parts. Assuming the split is even, this step takes a runtime of $2 * \mathcal{O}(\frac{V}{2}log(\frac{V}{2}))$. To achieve a P-way split, we need to apply 1 ORB on an array of V, 2 ORBs on arrays of size $V/2$, 4 ORBs on arrays of size $V/4$,..., $P/2$ ORBs on arrays

of size $V/(P/2)$. The calculation of the total runtime of ORB (S2 + S3) executions is shown below.

$$
\begin{aligned}
&\mathcal{O}(Vlog(V)) + 2 * \mathcal{O}(\frac{V}{2}log(\frac{V}{2})) + 4 * \mathcal{O}(\frac{V}{4}log(\frac{V}{4})) + ... + \frac{P}{2} * \mathcal{O}(\frac{V}{\frac{P}{2}}log(\frac{V}{\frac{P}{2}})) \\
&= \mathcal{O}(Vlog(V)) + \mathcal{O}(Vlog(\frac{V}{2})) + \mathcal{O}(Vlog(\frac{V}{4})) + ... + \mathcal{O}(Vlog(\frac{V}{\frac{P}{2}})) \\
&= \mathcal{O}(V(log(V)(log(P) - 1))) - log(2) - log(4) - ... - log(\frac{P}{2}) \\
&= \mathcal{O}(V(log(V)(log(P) - 1))) - 1 - 2 - ... - (log(P) - 1) \\
&= \mathcal{O}(V(log(V)(log(P) - 1))) - log(P)(log(P) - 1)/2 \\
&since V > P, \\
&= \mathcal{O}(Vlog(V)log(P))
\end{aligned}
\tag{2.1}
$$

Therefore, the total runtime of the centralized ORB is

$$\mathcal{O}(log(P)\alpha + Vlog(P)\beta + Vlog(V)log(P)\gamma) \tag{2.2}$$

2.1.2 Distributed ORB

In distributed load balancing strategies, object data are stored locally. To measure the variation of the current load distribution and collect the boundary of the global universe, we ask each PE to contribute their local sum of load and local universe boundary. We can measure the degree of unbalanced loads by the ratio of the maximum PE load and the average. If the unbalance ratio is smaller than a defined tolerance, we have a balanced placement of loads among PEs already and need to do nothing more. If we have an unbalanced load placement, we will apply the distributed ORB algorithm.

We are given a subspace to partition together with a group of PEs and objects on them to redistribute. Our goal is to split the subspace into the number of PE chunks and map objects in each chunk on a PE. The leader PE broadcasts information about the subspace and the chosen dimension to partition with to all PEs in the group along a spanning tree. (Shown in phase 1 of Figure 2.2) When a member PE receives the information, it will decompose the subspace along the chosen dimension into evenly spaced bins. The bin number is predefined. The member PE will loop through its collection of objects, accumulate their loads into matching bins, and accumulate counts of objects to each bin. After PE binning, the leader PE accumulates binning results among all group PEs via a reduction. (Shown in phase 2 of

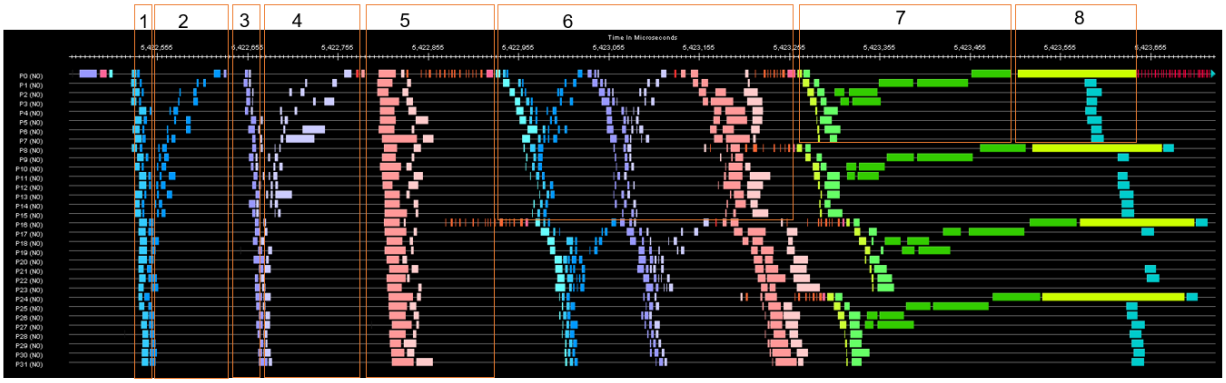


Figure 2.2: Projection trace of distributed ORB load balancing strategy with 32 PEs.

Each color represents a function.
Phase 1 applied S1 (binning initialization) with a global broadcast of the subspace and the chosen dimension to partition with. Once each PE receives the broadcast, it will do S2 (local binning).
Phase 2 applied S3 (aggregation) with a global reduction of binning results. At the root, P0 will do S4 (evaluation).
Phase 3 is a global broadcast that asks all PEs to contribute their object data with the optimal bin which applies to the first half of S5 (collection).
4 aggregates object data from all PEs via a reduction and applied the latter half of S5 (collection).
Phase 5 represents S6 (permutation). PE0-15 will send all local objects whose expected PE will be among PE16-31.
Phase 6 repeats S1-6 among PE0-15.
Phase 7 accumulates objects from PE0-7 to PE0 which is applied in the first half of s7 (centralize solve).
Phase 8 applied the ORB algorithm on the collection of objects which is the second half of s7 (centralize solve).

Figure 2.2) The leader then loops through the bins to find the optimal one whose boundary coordinates along the chosen dimension makes the evenest partition. If the partition satisfies the predefined tolerance, we successfully obtained a coordinate that separates all objects into two parts mapping to two halves of the current PE group. We move objects on each PE that should belong to the other half (shown in the phase 5 of Figure 2.2) and then do decomposition recursively on both halves (shown in block 6 of Figure 2.2).

If the optimal bin boundary does not satisfy the predefined tolerance and the optimal bin has more than a threshold number of objects, we need to repeat the binning process with a smaller subspace derived from the optimal bin. If the optimal bin has no more than a threshold number of objects, we will collect the information of objects in this bin directly. (Shown in phases 3 and 4 of Figure 2.2) We then loop through the collected objects to find

the coordinate that produces the evenest split of loads and move PE objects accordingly.

If the subspace for partition has a small number of objects, we can avoid unnecessary and time consuming recursions by collecting all objects on the group PE into a leader to solve the decomposition in a centralized way described in the previous section. (Shown in phases 7 and 8 of Figure 2.2)

The distributed ORB algorithm is complicated with conditions. We present the pseudocode in Listing 2.1 for the algorithm to help clarify the algorithm.

Listing 2.1: Pseudocode of the distributed ORB algorithm

```
1 void loadBinning(subuniverse, dimension){
2     for (obj : local_objects){
3         contribute object load to load bins;
4         contribute a count to count bins;
5     }
6     //send load and count bins to the leader PE via reduction;
7     contribute(local_bins);
8     //at the leader PE, call processBinning();
9 }
10
11 void processBinning(subuniverse, PEs, dimension, accumulated_bins, nObjects){
12     splitter <- the optimal spitting coordinate defined by bin boundaries
13     if (splitter fails tolerance){
14         if (nObjects < THRESHOLD_B){
15             gather all objects in the optimal bin to the leader PE;
16             update the splitter;
17         } else {
18             call loadBinning(optimal_bin_universe, dimension) on PEs;
19         }
20     }
21     // the current partition is completed
22     permute objects with respect to the partition;
23     left_universe <- left half of the subuniverse;
24     left_PEs <- left half of the PEs;
25     left_dim <- longest dimension of the left_universe;
26     left_nobj <- number of objects in the left_universe;
27
28     DistributedOrb(left_universe, left_PEs, left_dim, left_nobj);
29
30     // repeat for the right half
31     // The variables are made for the right half.
32     DistributedOrb(right_universe, right_PEs, right_dim, right_nobj);
33 }
```

```
34
35 void DistributedOrb(subuniverse, PEs, dimension, nObjects){
36     if (nObjets < THRESHOLD_A){
37         gatherObjects(PEs);
38         centralizedOrb();
39     } else {
40         call loadBinning(subuniverse, dimension) on PEs;
41     }
42 }
```

We now present the runtime analysis of the distributed ORB algorithm in Table 2.2. We first analyze the basic executions in the algorithm and their costs:

Table 2.2: Core steps of the distributed ORB algorithm.

S1	**Binning initialization** Broadcast the dimension coordinates of the sub-universe and the chosen dimension to partition with among k processing elements	$\mathcal{O}(log(k)(\alpha+\beta))$
S2	**Local binning** Loop through local objects and do binning among k PEs and n objects	$\mathcal{O}(\frac{n}{k}\gamma)$
S3	**Aggregation** Run reduction of histogram binning of object loads and sizes within the sub-universe among k PEs. (Bin size is a constant)	$\mathcal{O}(log(k)(\alpha+C\beta))$
S4	**Evaluation** Check if the optimal bin splitter satisfy the error tolerance	$\mathcal{O}(C\gamma)$
S5	**Collection** Collect coordinates of objects in the optimal bin directly among k PEs with a broadcast followed by a reduction. Then, find the optimal splitting dimension among the collected data. (The number of gathered object is bounded by a constant.)	$\mathcal{O}(log(k)(\alpha+C\beta)+C\gamma))$

Table 2.2 (continued)

S6	**Permutation** Move object data relevant to future partitions among k PEs and n objects. Pair a PE on the right half with one on the left half and exchange between the pair. (Assume half objects per processing unit need to be moved)	$\mathcal{O}(\alpha + \frac{n}{2k}\beta))$
S7	**Centralized solve** Gather n objects among k PEs to a root PE. Apply the centralized ORB algorithm with n objects and k PEs	$\mathcal{O}(log(k)(\alpha + n\beta) + nlog(n)log(k))\gamma$

To calculate the runtime of the distruibuted ORB algorithm, we start with applying S1 (Binning initialization), S2 (Local binning), S3 (Aggregation), S4 (Evaluation), S5 (Collection), S6 (Permutation) among all P processing elements and V objects, which makes a runtime of $\mathcal{O}(log(P)\alpha + (log(P) + \frac{V}{2P})\beta + C\gamma)$. In fact, S1-4 could happen multiple times before reaching S5 (Collection). Since we force an upper limit of the repetitions, the runtime stays the same. In practice, the value of $\frac{V}{P}$ is a constant, we can derive the runtime function to $\mathcal{O}(log(P)(\alpha + \beta) + C\gamma)$. If there are less than a certain threshold amount of objects in the half of processing elements in the previous iteration, we apply S7 (Centralize solve) and end the computation. At this time, we are dealing with no more than a constant threshold number of objects and processing elements since there is a constant ratio of objects per PE. If there are more than a certain threshold amount of objects in the half of processing elements in the previous iteration, repeating from the binning.

To derive the runtime formula for the whole process:

$$\begin{aligned}&\mathcal{O}(log(P)\alpha + log(P)\beta + C\gamma) + \mathcal{O}(log(\frac{P}{2})\alpha + log(\frac{P}{2})\beta + C\gamma) + \mathcal{O}(log(\frac{P}{4})\alpha + log(\frac{P}{4})\beta + C)\\ &+ ... + \mathcal{O}(log(C)\alpha + log(C)\beta + C\gamma)\\ &+ \mathcal{O}(log(C)\alpha + log(C)C\beta + Clog(C)log(C)\gamma)\end{aligned} \tag{2.3}$$

Therefore, the total runtime of the distributed ORB is

$$\mathcal{O}(log(P)(log(P)(\alpha + \beta) + C\gamma)) \tag{2.4}$$

2.1.3 Scaling Comparison with Simulation

With the complicated Distributed ORB algorithm, we developed a simulation program to simulate PEs with chares. The program will estimate communication runtime with the input α, β, and P (number of PEs) parameters, and wall time is used for computation. In the simulation, each PE keeps a local timer. When a message is passed from PE_i to PE_j, PE_i will attach a timer to the message as its current local timer plus a communication time estimation calculated with the predefined α, β, and P. When PE_j receives the message and the attached timer from the message is larger than the local timer, PE_j will update its local timer to the attached one from the message.

This simulation program provides three major benefits. First of all, the simulation reduces the time required for larger runs involving thousands of PEs. We can now simulate any scale we want on one node without waiting for a large job to go over the scheduling queue of large clusters. Secondly, the simulation helps us understand the scaling behavior of the algorithm with flexible parameters of α, β, and P. For the result shown later, we represent a core as a PE. In reality, we can make each PE represent a computation node with an inter-node α value. We can choose to partition objects onto nodes using the distributed ORB algorithm and further partition objects in a node onto each core with other strategies. Last but not least, the simulation speeds up the development process. The fast execution on any scale helps us to discover bugs and edge cases with increasing scales. In addition, the scaling behavior helps us to diagnose the code blocks with poor scalability and make improvements.

Here we present the simulation results of centralized and distributed OrbLB strategies to better understand the comparison of the runtime analysis.

In the practice of the paraTreeT program, the ratio of chare numbers on each PE is configurable and a constant. Therefore, We can apply $\frac{V}{P} = C$ to remove the V term in the expressions. The runtime of centralized OrbLB can be derived as $\mathcal{O}(log(P)\alpha + Plog(P)\beta + Plog(P)log(P)\gamma$. The runtime of distributed OrbLB can be derived as $\mathcal{O}(log(P)log(P)\alpha + log(P)log(P)\beta + log(P)\gamma)$.

The dominant terms are $Plog(P)log(P)\gamma$ for the centralized OrbLB and $log(P)log(P)\alpha$ for the distributed one. When $P\gamma < \alpha$, the centralized algorithm runs faster. When $P\gamma > \alpha$ with large P, the distributed algorithm runs faster.

Figure 2.3 shows the simulated runtime of centralized and distributed OrbLB with various α options. β is fixed to 1 byte per nanosecond. γ is measured as walltime of function duration. From figure 2.3 we can see that when α is small being 10 and 100 microseconds, the distributed algorithm scales better than the centralized one. When α is large being 1000 microseconds, centralized algorithm behaves better up to 1024 PEs and the distributed

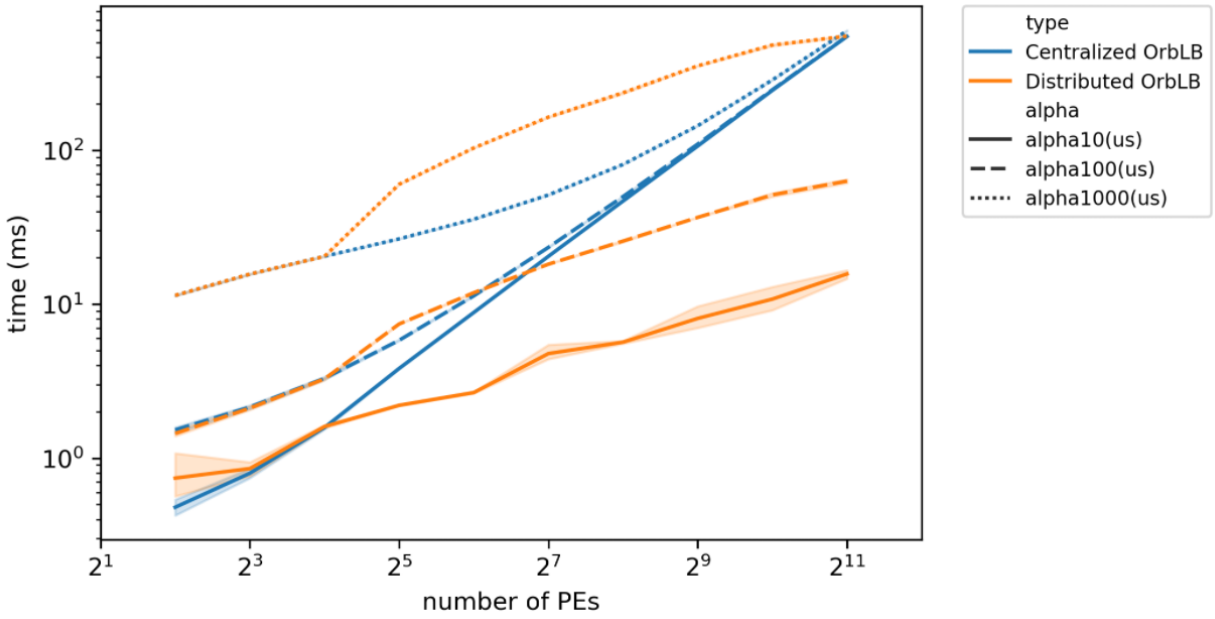


Figure 2.3: The runtime simulation of centralized and distributed OrbLB with various α estimations up to 2048 PEs and 32937 objects (about 16 objects per PE). The β is 1 byte/-nanosecond. The γ is measured as walltime.

algorithm beats the centralized one with 2048 PEs.

In conclusion, we show that the distributed ORB algorithm has better scalability than the centralized one.

2.2 PREFIX BASED STRATEGY

Prefix based strategies sort all objects according to their global indexing order. The sorted array of objects will be decomposed in the number of PE chucks with an even accumulated load. In the strategy, each PE will have a subarray of objects with continuous indices and preserve the characteristics of the decomposition property. For example, if the objects are partitioned in the order of where their coordinates fit on the Space Filling Curves (SFC), the redistribution will not change it.

Aluru et al.[23] and Harlacher et al.[24] balance loads for unstructured meshes using prefix sums based on their relative standings on the space filling curves. In this section 2.2 we present centralized and distributed prefix based load balancing strategies integrated to the Charm++ load balancing framework.

2.2.1 Centralized Prefix

Once a global array of objects are collected at the root PE, we sort the objects according to their indexing order. Next, we loop through the array to calculate the total load and average load per PE. Then, we loop through the array again to assign objects to PEs based on their prefix sum. The runtime of the algorithm is

$$\mathcal{O}(log(P)\alpha + Vlog(P)\beta + Vlog(V)\gamma) \tag{2.5}$$

2.2.2 Distributed Prefix

In the distributed prefix algorithm, we first sort the local objects with respect to their indexing order and calculate the local sum. Next, we calculate global prefix of each PE using the recursive doubling algorithm [25]. The runtime of this step is $\mathcal{O}(log(P)\alpha + log(P)\beta + \frac{V}{P}log(\frac{V}{P})\gamma)$. Given the total global load and the prefix sum of each PE, one PE loops through its sorted array of objects to assign it to PEs based on its global prefix sum. In practice, $\frac{V}{P}$ is bounded by a constant. Therefore, the total runtime can be derived as

$$\mathcal{O}(log(P)\alpha + P\beta + C\gamma) \tag{2.6}$$

2.3 EVALUATION OF LB STRATEGIES

We apply the four different LB strategies mentioned above to the paraTreeT program on the Stamepede2 cluster[26]. Five data points are taken for each configuration. The mean of the data points is shown with lines and a 95% confidence interval is shown in shadow.

Figure 2.4 shows the total LB time for each strategy scaling from 1 node to 128 nodes. On each node, we use 46 PEs. For this experiment, one PE maps to a physical core on the node. The $\frac{V}{P}$ ratio is around 16 on average for all the runs. From Figure 2.4, we can see both the distributed strategies scale better than the centralized ones.

There are two main components to the total runtime. The first part is initialization defined by the time after entering the LB phase and before applying strategy algorithms. For centralized strategies, all objects on each PE will be gathered to a root via a reduction, imposing a runtime of $\mathcal{O}(log(P)(\alpha + V\beta))$. For distributed strategies, all objects will be kept local and thus impose a runtime of $\mathcal{O}(1)$. Figure 2.5 shows the LB initialization time. We can see that data gathering is expensive on a large scale.

The second component is the application of algorithms to make decisions to map objects to PEs. In this step, centralized strategies are given an array with all objects to work with

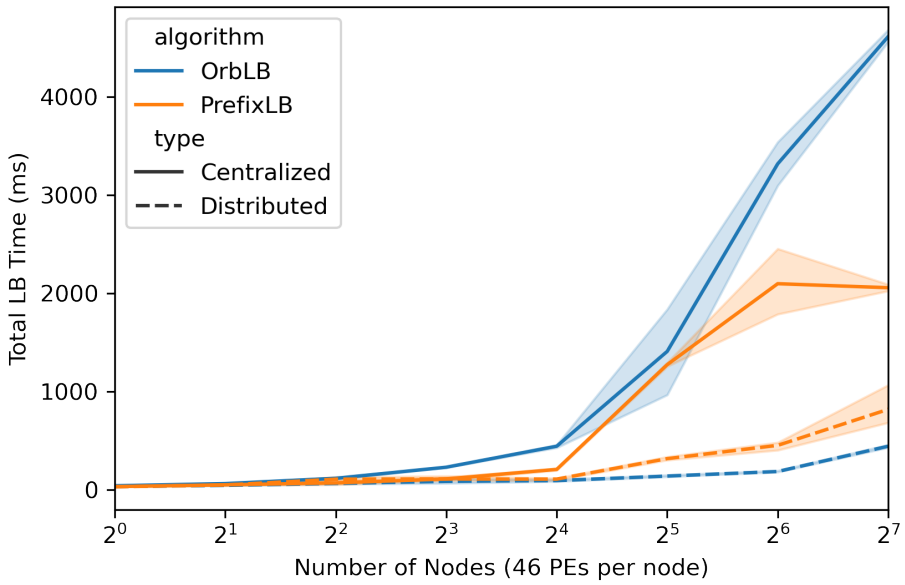


Figure 2.4: Total LB time of each strategy scaling from 1 node (46 PEs) to 128 nodes (5888 PEs) with an average of 16 objects per PE.

and only the root PE will execute the algorithm. For distributed strategies, all PEs will participate in the algorithm with their local objects. From Figure 2.6, we can see that the centralized ORB algorithm scales worse than the distributed ORB. The centralized prefix algorithm scales better than the distributed prefix.

Looking back to the total LB runtime in Figure 2.4, the runtime scalability of centralized strategies will always suffer from the initialization step where object data are gathered at a root. Although distributed strategies tend to be more complicated, if the algorithm itself scales well, the development effort will pay back. In fact, we have tried three major versions of the distributed ORB algorithm to come up with the current one with the best scalability so far. The simulator mentioned in section 2.3 in this chapter helps us with a quick diagnosis of the scaling behavior of code blocks and fast improvement.

2.4 LIMITATION OF LOAD BALANCING STRATEGIES AND FUTURE WORK

The runtime of the load balancing strategies increases with more PEs. However, applications tend to run faster per iteration with more PEs. The different scaling patterns of the load balancing strategies and application execution time makes load balancing more expensive on larger scale runs and could hurt performance. From our observation, the unevenness

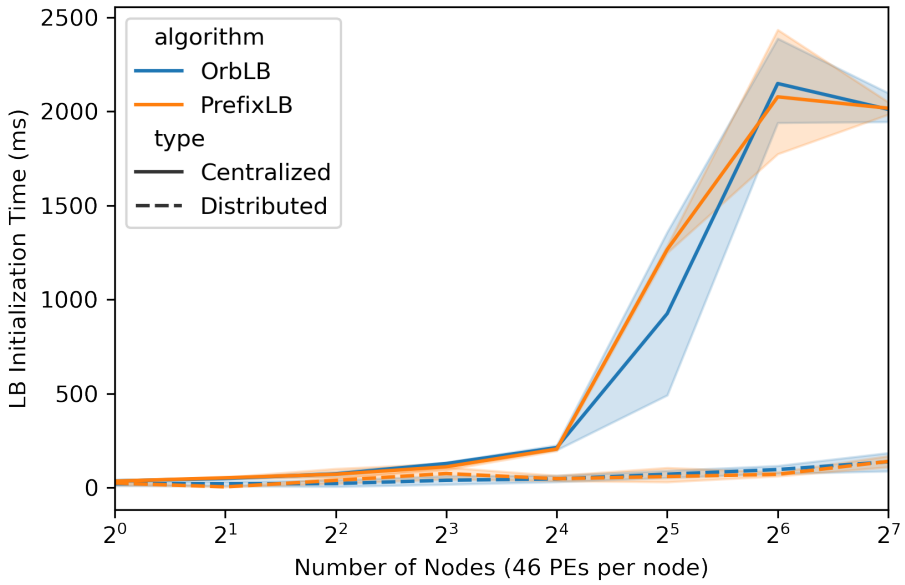


Figure 2.5: The initialization time of each LB strategy. scaling from 1 node (46 PEs) to 128 nodes (5888 PEs) with an average of 16 objects per PE.

grows slower with larger scale runs. The user can reduce the frequency of the load balancing steps to smooth load distribution during the execution with less overhead.

The basic unit of PE is a CPU core among the above experiments. We can reduce the parallel runtime of distributed load balancing strategies by reducing the number of PEs involved in the algorithm which reduces the depth of the spanning tree for broadcasts and reduction. For example, with the same number of compute nodes and cores per node, it is faster to partition objects to the node level than the core level using the same algorithm. We can add hierarchies to the load balancing frameworks to achieve better flexibility and reduce overhead.

For example, we can first partition the objects into each node and then apply a different algorithm to further partition node objects to cores. We hope to achieve better performance with distributed ORB strategy by partitioning objects into nodes using the distributed ORB algorithm and applying a different algorithm on the node level. The node level algorithm could be ORB with some fuzziness to trade spatial locality with more even loads per core. The ORB algorithm can allocate a super object on a core whose load is 10x or even larger than the average load per object. If this super object is on the edge of partition decisions, for whichever PE that takes this object, the total load per PE could be far above the average. If we allow exchanging the super object with some small objects, we can avoid placing the

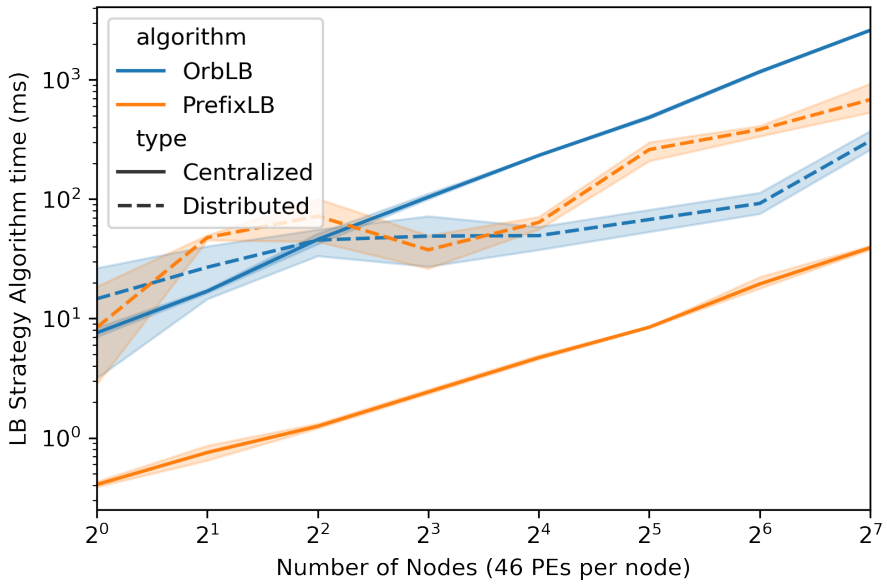


Figure 2.6: The strategy time of each LB strategy. scaling from 1 node (46 PEs) to 128 nodes (5888 PEs) with an average of 16 objects per PE. The y-axis is in log scale.

super object on the edge and can achieve a smoother load distribution with a minor sacrifice of spatial locality.

The super objects could impose a greater challenge if one object has a load that is larger than the average accumulated load per PE. In this case, even if we put only one super object on a PE, this PE could still be the PE with the largest load, which results in a large max by average ratio for PE loads. Bak et al. [27] proposed a solution to adapt user-defined code to break large objects to smaller migratable pieces to reduce the impact of super objects.

CHAPTER 3: INTEGRATION OF GPU KERNELS TO THE HOST WORKFLOW

Recall from section 1.4, at each traversal step, when we need to recurse on the child nodes of the current source node, we need to access each child node as the new source node with a local pointer or a remote request. When we do not need to do recursion, we need to execute `node()` or `leaf()` functions which are computation heavy.

The `node()` function takes a target node and source node. It will loop through all particles in the target node to calculate gravitational forces with the aggregated data at the source node to produce updates for each particle. If each node has no more than p particles, the runtime of the `node()` function is $\mathcal{O}(p)$.

The `leaf()` function also takes a target node and source node. For each particle in the target node, it will loop through the particles in the source node to accumulate the gravitational forces update to the target particle and execute the update at the end. The runtime of this function is $\mathcal{O}(p^2)$.

Before presenting the modifications for GPU kernels, let us first walk through the workflow of the cpu implementation.

3.1 WORKFOW OF THE CPU IMPLEMENTATION

ParaTreeT has a Subtree-Partition model where particle decomposition can be done in two different manners, taking charge of different functionalities and working in sync. The traditional N-body framework has only one decomposition type to serve the data hierarchically and distribute computation loads among processors. ChaNGa [28] for example uses *TreePieces* to manage segments of all particles in the input space. However, the traditional decomposition enforces cubical decomposition of inputs which could potentially create imbalanced computation distribution. Hutter et al. [9] showed performance improvement with an octree decomposition to serve the spatial structure (i.e., memory) and SFC decomposition to distributed loads (i.e., computation). *Subtree* is the chare array serving a hierarchical spatial view of the input and handling memory requests. We define *tree type* to be the decomposition type of Subtree. *Partition* is the chare array organizing computations. We define *decomposition type* to be the partition type of Partition. The user can choose to give the same decomposition type for Subtree and Partition. In that case, paraTreeT will work similarly to the traditional approach. Figure 3.1 presents an example of the Subtree-Partition model with Subtree partitions the universe in three pieces while Partition decomposes the inputs into four units. The graph at the bottom shows one Subtree chare and two Partition

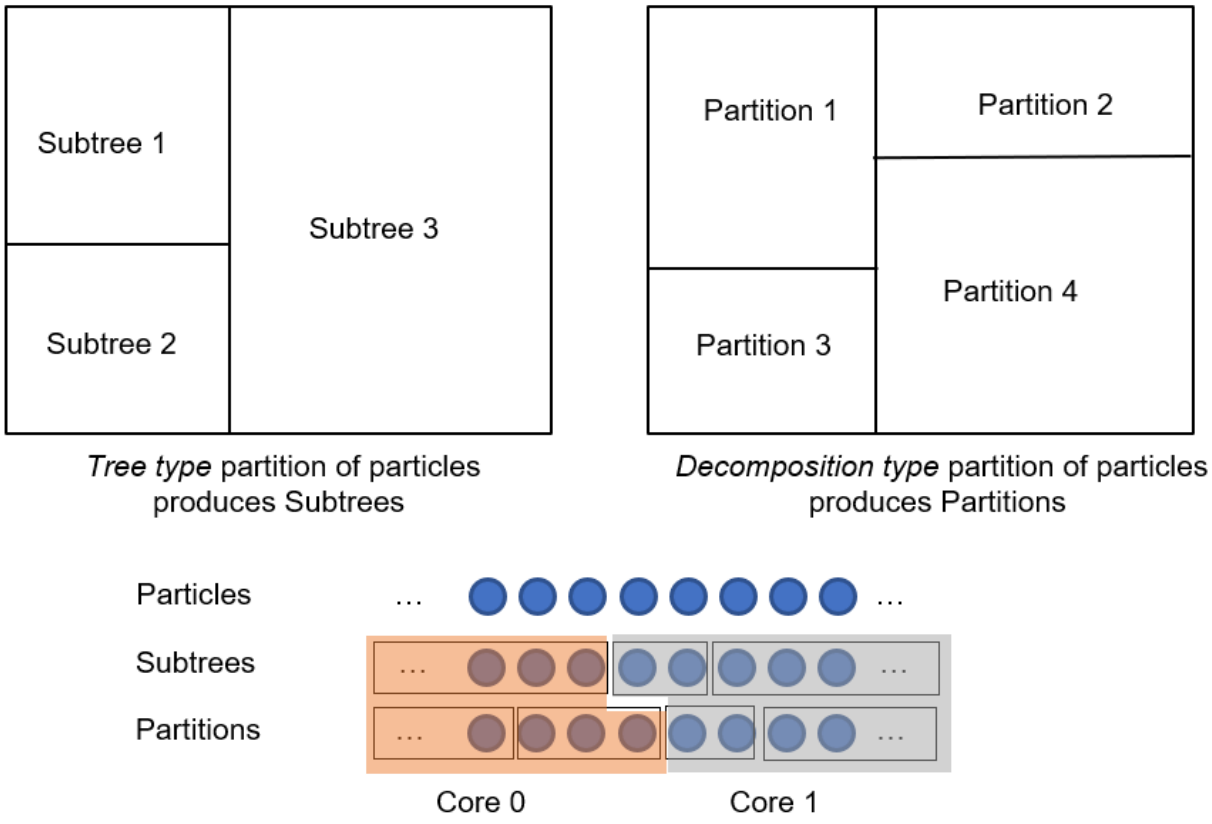


Figure 3.1: Subtree and Partition have different decompositions of the input space. This graph shows a case with three Subtree chares and four Partition chares distributed on two cores.

chares located on core 0 and the rest located on core 1.

3.1.1 Input Decomposition

ParaTreeT reads the user input of initial conditions of all particles in the universe in parallel, growing the universe boundary box while adding particles. The read-in particles will then be overdecomposed into fine-grained chunks and be sent to Subtrees and Partitions. With a space-filling curve (SFC) based algorithm, all particles are assigned a key with respect to their coordination in the universe and be sorted according to their keys and be decomposed into continuous blocks. The particles assigned to the same block will have locality in regard to the SFC order. Particles can also be decomposed by applying the K-D tree algorithm. We can recursively partition the universe into blocks with even particle sizes. With the K-D tree decomposition, particles in the same block have locality respecting their coordinates. The type of decomposition should be chosen carefully with regard to the particle distribution in

the input universe to minimize communication and reduce load variation.

3.1.2 Tree Build and Data Aggregation

Each decomposed input block will be sent to a Subtree chare to build the spatial tree from the top down. Particles in a Subtree will further be partitioned recursively following the defined tree type. The goal of the tree build step is to partition particles into leaf nodes of the spatial tree and construct internal nodes to connect the global root to all local nodes. User can define the maximum number of particles a leave can hold to enforce a upper bound of the `leaf()` and `node()` functions. The left and middle graphs in Figure 1.2 shows this top-down process of tree build.

After each Subtree chare has its local share of the global spatial tree, it will traversal the local tree bottom-up to populate the tree with data aggregation. The right graph in Figure 1.2 shows such aggregation. Internal nodes aggregate data of all its direct and indirect children by applying the algorithm defined by the `Data::operator+=` method. The aggregated data at internal nodes will help with gravity computation with estimation between a target node and a far-away source node. The aggregated data at the source node will be used to contribute updates to particles in the target nodes. With this estimation technique, we no longer need to traversal through the piece of the global tree with the source node being the subroot.

After tree build and data aggregation, Subtrees will send particles to their related Partitions. If the Subtree and Partition follow the same type of decomposition, a particle in Subtree with index i should be located to the Partition at index i. Leaf nodes in a Subtree can be sent to its pairing Partition without changes. If Subtree and Partition apply different decomposition algorithms, particles in a leaf node might be sent to different Partition chares.

3.1.3 Tree Traversal with Vistors

Users can define the custom Traversal class to implement different algorithms of the tree traversal. Each Partition chare performs the traversal algorithm concurrently to visit each node in the global tree and update local particles by applying the Visitor functions. Let's take a look at the example of the builtin *TransposedDownTraverser*. At each node of the global tree, if the node does not locate on the local process, we will invoke a remote request to bring data of the remote node locally. The interactions related to remote nodes will be queued up and performed after the interactions among local nodes. When the source node is a leaf, we will apply the `leaf()` function defined in section 1.4 to loop through particles in

the source node and particles in the target node to compute gravitational forces for all pairs of the particles. When the source node is not a leaf and it fails the open criteria, we will apply the `node()` function defined in section 1.4 to compute gravitational forces between the aggregated data of the source node and all particles in the target node. If the source node passes the open criteria, we keep traversing the children of it.

The pseudocode for the traversal algorithm is shown in Listing 3.1:

Listing 3.1: Pseudocode of the recursive tree traversal algorithm

```
1 void TransposedDownTraverser::recurse(source_node, active_target_nodes){
2     vector<Node> new_active_target_nodes;
3     if source_node is a leaf:
4         for (target_node : active_target_nodes){
5             leaf(target_node, source_node);
6         }
7     else if source_node is an internal node:
8         for (target_node : active_target_nodes){
9             if (open(target_node, source_node)){
10                new_active_target_nodes.push_back(target_node);
11            } else {
12                node(target_node, source_node);
13            }
14        }
15    else if source_node locates on remote process:
16        request source_node on remote processes
17
18    if (! new_active_target_nodes.empty()){
19        for (child : source_node.children()){
20            recurse(child, new_active_target_nodes);
21        }
22    }
23 }
24
25 recurse(global_root, local_leaves_in_Partition);
```

We define *local interactions* to be the computation done between local source and target nodes. *Remote interactions* refer to the computation done between remote source nodes and local target nodes. The initial tree traversal will perform all local interactions as soon as they are invoked. In the meantime, remote nodes will be requested. After the competition of the initial traversal and all the local interactions, we will deal with the remote interactions accumulated. For each remote source node, we will traverse the piece of the global tree with the source node being the root. Note that traversal on tree pieces rooted at a remote node

can invoke requests of new nodes from remote processes.

3.1.4 Post Traversal Updates and Reset

We are now at the last phase in an iteration. Particles updates their local fields with respect to the accumulated gravitational forces done by remote and local interactions. Subtrees and Partitions need to be reset to prepare for the next iteration. The load balancing step might be invoked at the end of each iteration to detect and fix an unbalanced load distribution.

3.2 WORKFLOW OF THE GPU IMPLEMENTATION

We name the GPU kernel in substitution of the `node()` function `nodeOnDevice()` and name the `leafOnDevice()` kernel to replace the `leaf()` function. We need to provide a list of interactions for each invocation of GPU kernels. To avoid duplicated memory copies of the local node and particle data feeding to each GPU kernel, we decide to serialize and allocate a global device copy of the local node and particle data per process. At each invocation of computation kernels, we can just pass in indexing data for each node or particle to access the global device storage to save memory bandwidth.

We now talk about modifications of the CPU implementation for adding GPU supports.

3.2.1 Tree Build and Data Aggregation

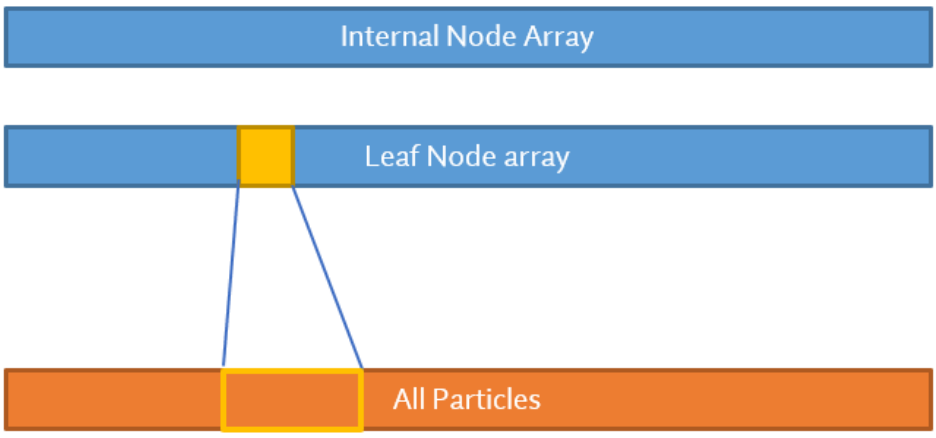


Figure 3.2: The device memory of local nodes and particles.

ParaTreeT allocates one *CacheManager* chare per process to maintain a unique copy of node and particle data. During the tree build process, each Subtree chare will generate internal and leaf nodes while registering unique pointers for them at the local CacheManager. Each CacheMananger will have pointers of all local nodes and particles once all Subtrees in the process completes the tree build process. We can then serialize all nodes and particles and copy the memory to the device while Subtree chares are doing data aggregation.

Figure 3.2 shows the structure of device memory of local nodes and particles. The top two bars represent vectors of internal nodes and leaf nodes respectively. Computations involving internal nodes only need aggregated data which we add to the struct. The `leaf()` function loops through all particles in the leaf node to calculate gravitational forces. To address particles belonging to a leaf node, we will allocate a continuous block of memory in the particle array for particles in one leaf node. Struct for the leaf node tracks the starting index of its particles in the vector and the size of particles in the node. Remote nodes and particles will be added on-demand later during traversals. Since `cudaMalloc` has implicit synchronization to allocate memory on the host, we create memory buffers at the initialization time of the CacheManager class to reduce thread blocking on the CPU side. We use `cudaMemcpyAsync` to avoid blocking the host execution while data transfer is in progress.

3.2.2 Tree Traversal with Visitors

In the current design of our implementation, we ask CPUs to do tree traversal, buffer the interactions and invoke GPU kernels in bulk. Appending pairs of interactions could be memory heavy as the list goes long with no memory reuse. We allocate enough capacity to the memory buffer base on input size to avoid expanding the buffer size while collecting interaction pairs and excessive memory copies. To copy host data on the device, CUDA first needs to allocate pinned memory pages to copy the host buffer to the pinned memory. After that, pinned memory will be copied to the device. To avoid this two-step memory process, we use `cudaMallocHost` to allocate pinned memory where we will store the interaction lists directly. Pinned memory enables asynchronous data transfers between the host and device.

Figure 3.3 shows a plot of binning of the duration of `cudaMemcpyAsync` calls with memory allocated using `cudaMalloc` and `cudaMallocHost`. We can see that the memory copy duration with pinned memory centers around 20 milliseconds. The duration with unpinned memory centers around 40 milliseconds and has more instances with higher duration than the case with pinned memory. Memory copying with unpinned memory takes longer since the host needs to request pinned memory pages before returning from the asynchronous call.

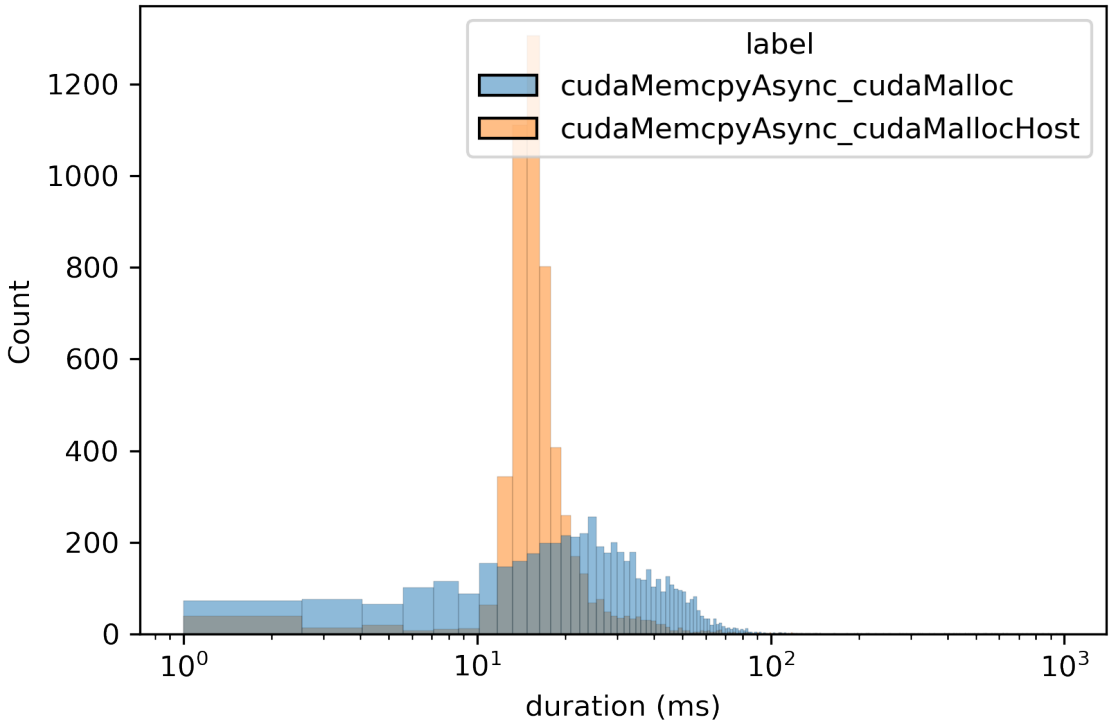


Figure 3.3: Plot of binning of the duration of `cudaMemcpyAsync` calls with memory allocated using `cudaMalloc` and `cudaMallocHost`.

Frequent memory requests on the system can impose a large overhead.

Using pinned memory could potentially hurt system performance as it consumes available memory resources for paging. Each Partition chares will create two CUDA streams with one dedicated to leaf interactions and one serving the node interactions. Keeping separate streams per Partition increases the runtime concurrency of CUDA kernels and improves performance. At each CUDA computation kernel, updates on the particles in the target node will be written to the global GPU memory using atomic operations to avoid data racing.

3.2.3 Post Traversal Updates and Reset

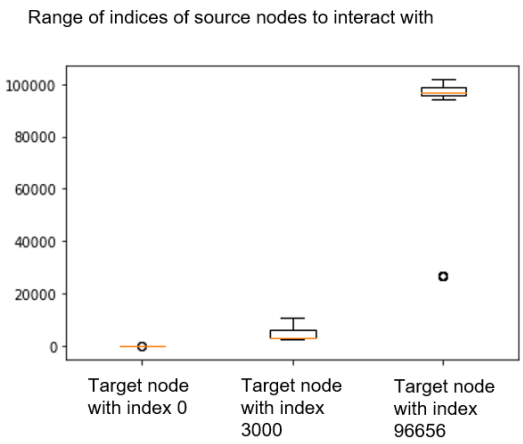
After the competition of interactions, the GPU memory of local particles needs to be copied back to the host. We can reduce the copy back of the particle memory if the mapping between nodes particles as well as the relationship between nodes and their location in Subtrees and Partitions stay constant. In this case, the host only needs to wait for all

streams to finish their queued jobs to start the next iteration. Note that, even if we can reuse the GPU memory across iterations, we cannot reuse the interaction lists from the last iteration. The movement of particle coordinates can change the testing result of the open criteria between two nodes in consecutive iterations. We do not need to deallocate GPU and CPU buffers as we can reuse them. At the last iteration of the program, we need to clean up memory allocations and reset the GPU device.

CHAPTER 4: GPU KERNEL EXPERIMENTS AND EVALUATIONS

All experiments are done on a Dell R730 server with 2x Intel Xeon E5-2698v4 (20-core @ 2.20GHz-3.60GHz) CPUs and 2x NVIDIA Tesla K80 GPUs per node. We choose an input with three million particles.

We first identify the characteristics of the formation of interaction lists.



Target node index	Indices of source nodes to interact with	Interaction list size
0	0 1 2 3 4 11 15 16 17 30 31 32 33 34 35 36 39 40 41 42 43 44 45 51 52 115 116	27
3000	2495 2496 2945 2953 2954 2955 2956 2959 2960 2970 2971 2974 2975 2976 2977 2978 2979 2980 2981 2982 2983 2984 2985 2986 2987 2988 2989 2997 2998 2999 3000 3001 3005 3006 3007 3008 3009 3010 3017 3018 3019 3020 3021 3095 3096 3097 3098 3099 3100 3101 3102 3103 3104 3105 3106 3107 3108 3109 3116 3168 3173 3174 3175 3176 3179 3182 3183 3184 3185 3186 3187 3193 3194 3195 3198 3199 3200 3201 3202 3203 3204 3205 3206 3207 3208 3209 3210 3211 3215 3216 3217 3218 3219 3220 3248 3249 3260 3261 3262 3297 3300 3301 3302 3306 3307 3319 3320 3331 3340 3341 3342 3345 3348 3349 3350 3351 3352 3353 3354 3355 3358 3416 3417 3439 3457 3458 3468 3469 3505 6092 6136 6137 6759 6760 6761 6770 6780 6790 6793 6800 6801 6806 6823 6824 6827 6841 6975 6978 6979 6980 6986 6987 6990 6995 7025 7026 7027 7028 7029 7030 7031 7032 7037 7038 7041 7050 7054 7055 7056 7059 7060 7064 7077 7085 10833 10865	176
96656	26564 26584 26585 26586 26618 26619 26620 26621 26624 26625 26626 26627 26628 26629 26635 26636 26637 94041 94666 96171 96184 96185 96186 96198 96199 96200 96545 96554 96555 96556 96557 96558 96561 96572 96573 96589 96590 96599 96600 96601 96625 96634 96635 96636 96637 96638 96639 96640 96653 96654 96655 96656 98639 98640 98641 98653 98642 98643 98656 98657 101851 101852 101853 98635 98632 98609 98631 98636 98637 98638 98644 98648 98649 98645 98623 98633	76

Figure 4.1: Leaf interaction lists of three target nodes. The right table lists all source node indices that the target node interacts with and the size of the interaction list. The left graph is the box plot of the interaction lists for three target nodes.

Figure 4.1 presents three leaf interaction lists corresponding to three target nodes. The right table shows the raw data. The left graph is a box plot of the raw data. Recall that we serialize local nodes with indices to address them on the GPU device. In this experiment, we collect interaction lists for doing the `leaf()` computation for each leaf nodes in the process. We found that the size of interactions varies depending on the positions of the leaf nodes. Boundary nodes with few nearby neighbors tend to have short lists and nodes in the center surrounded by many nodes tend to form long lists.

4.1 APPROACH 1: 1-D KERNELS

In this approach, each GPU thread takes care of a pair of source and target nodes. During the tree traversal, we append the pair of nodes doing `leaf()` or `node()` computations to

separate lists.

Let *buffer size* (represented by N) be the size which when a list collects N pairs, it will invoke asynchronous memory kernel to copy the interaction list to the device and perform computation kernels in bulk. Recall the tree traversal process in section 3.1.3, when visiting a source node, pairs of interaction between this source node and all leaf nodes in a Partition chare will be appended if needed. Therefore, for continuous pairs of interactions, they are likely to have different target nodes while sharing the same source node. The cache line size of the GPU is 128 bytes, which can hold around 10 node records. As Harris [29] suggested, global memory access can be done efficiently if threads in a warp have coalescing memory accesses. For the 32 threads in a warp, ideally, access to the target nodes can coalesce in four cache lines. Threads in a warp are likely to access the same source node and thus cannot take advantage of the cache line size and cannot reduce attainable memory bandwidth with memory coalescing.

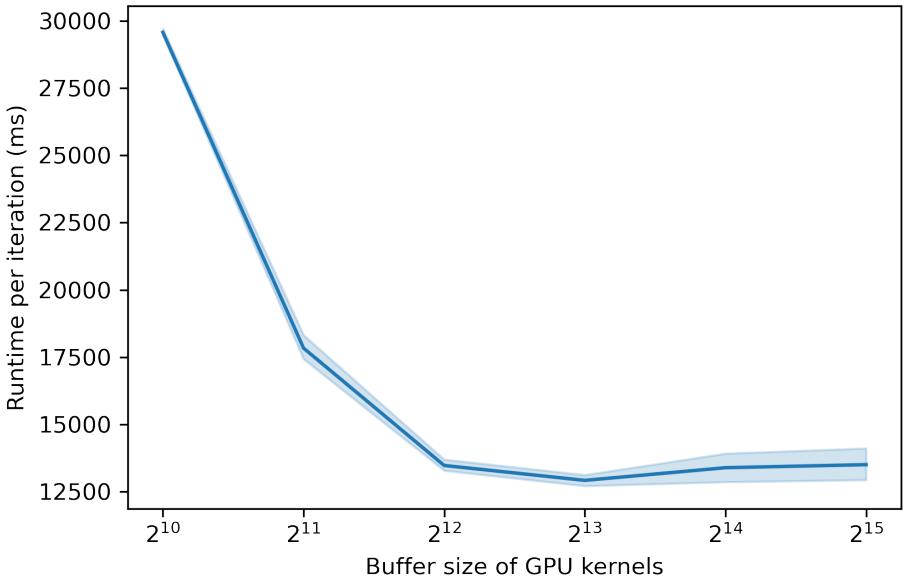


Figure 4.2: Runtime per iteration with various GPU kernel buffer sizes (N). In this experiment, the blockDim of CPU kernels is held constant. Buffer size being 8192 provides the best performance.

In this approach, we can tune the performance of the end-to-end runtime of all computations by changing the buffer size (N). Figure 4.2 shows experiments with various buffer sizes. We can see that buffer size being 8192 provides the best performance. With a smaller buffer size, we will trigger more launches of the GPU kernels. On the CPU side, with a constant host to device memory transformation bandwidth, the copy time of the input list

of interaction pairs grows proportionally to the buffer size. The host overhead of the computation kernel launch is fixed regardless of the buffer size. A smaller buffer size allocates fewer threads per kernel launch and enables parallel execution of more kernels from different streams on the device. In this experiment, we always allocate 256 threads in a block and enough grids to cover the list of buffer size. For our testing hardware, there can be at most 32 streams working concurrently on a device.

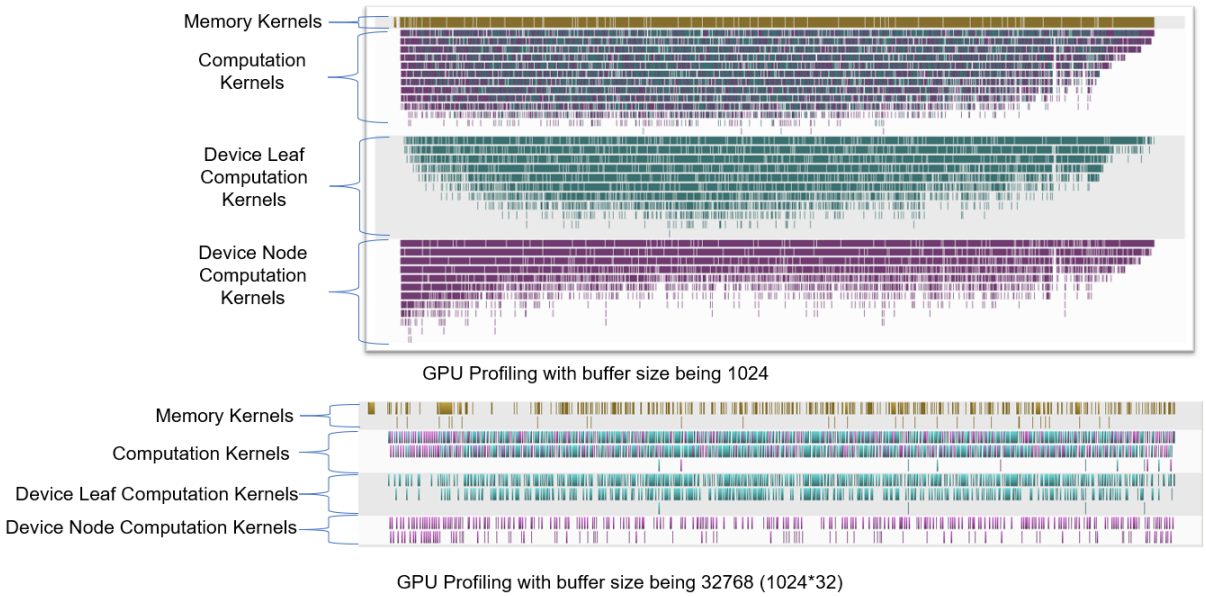


Figure 4.3: Profiles of GPU kernels with different buffer size. The x-axes of the two plots are not to scale. The top graph has buffer size being 1024. Each computation kernel will allocate one grid and one block with 1024 threads in total. The bottom graph has buffer size being 32768. Each computation kernel will allocate 32 grids of blocks with 1024 threads each.

Figure 4.3 shows two profiles of GPU kernels. For both runs, there are 31 Partition chares each having two CUDA streams with one doing leaf interactions and one doing node ones. The graph on top has the buffer size being 1024. One block with 1024 threads can cover the entire input interaction list. The graph on the bottom has the buffer size being 32768. Each computation kernel launch needs to allocate 32 blocks to cover the input list. With small kernels, the top graph shows a maximum of 13-way parallelism with computation kernels. Kernels from ten steam can run concurrently the most time. In the meanwhile, at the peak, thirteen streams can be scheduled to execute in parallel. With larger kernels, the degree of parallelism is reduced to two on average and three at the maximum. Despite the higher level of parallelism with a smaller buffer size, the runtime of the whole program is more than four times longer than the run with a bigger buffer size. The overhead of around 32X more

launches outweighs the benefits of parallelism.

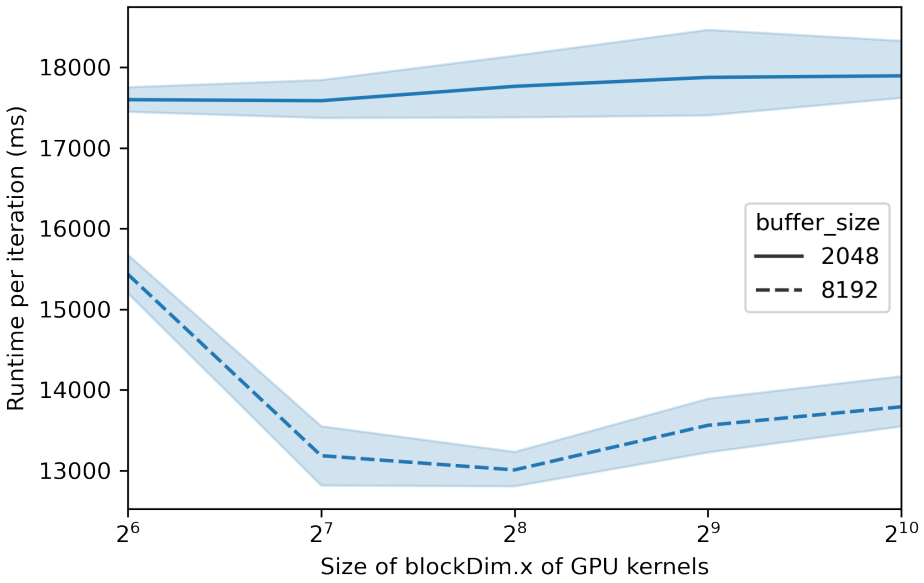


Figure 4.4: Runtime per iteration with various number of threads per GPU block. In this experiment, we set buffer size (N) to be 2048 and 8192. The configuration with buffer size being 8192 and blockDim.x being 256 has the best performance.

We then hold the buffer size constant and look at the impact of the number of threads per block. For a GPU kernel, with more threads per block, we need fewer grids. Figure 4.4 shows the impact of tunning blockDim.x for kernel launches. We can see that an allocation of 256 threads per block has the best performance with buffer size being 8192.

Figure 4.5 shows the scatter plot of the duration between consecutive device kernel launches and the execution time of the device kernels. The figure shows that the execution time in a kernel is mostly shorter than the time for input collection.

In the 1-D kernel, each thread needs to perform as many atomic writes as the number of particles in the target node. The atomic write step alone takes around 20% of the total GPU computation time. To minimize atomic operations, the second approach, 2-D kernels is proposed.

4.2 APPROACH 2: 2-D KERNELS

This approach is designed with the objective to reduce global atomic operations. To do so, we want each particle in the target node to interact with multiple source nodes while

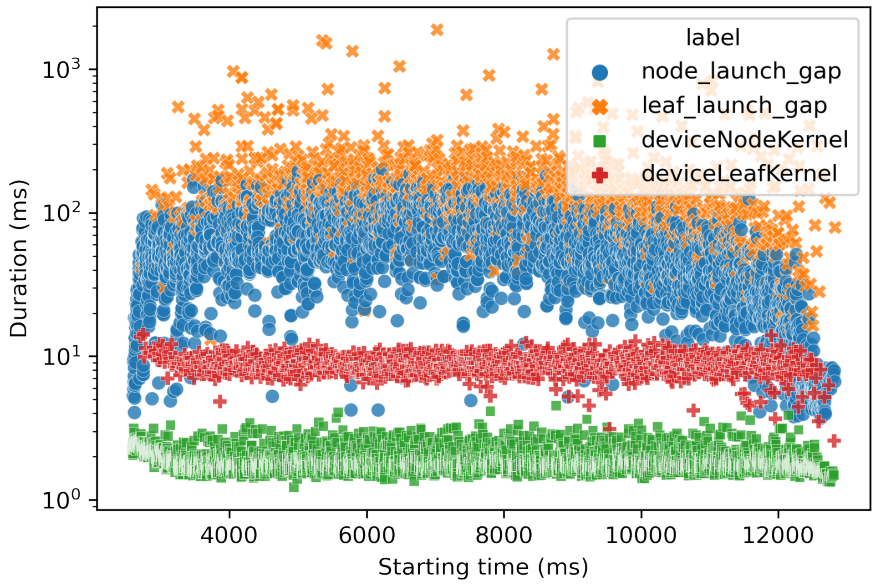


Figure 4.5: Scatter plot of device computation kernel duration as well as the time spent for input collection before each launch. The x axis is the starting time of events. Node_launch_gap is the duration between two consecutive deviceNodeKernel launches. Leaf_launch_gap is the duration between two consecutive deviceLeafKernel launches.

accumulating partial updates and conduct one atomic write to the global memory at the end.

Figure 4.6 presents the kernel design of the second approach with 2-D threads in blocks. Let BX, BY be the x and y dimension size of a block. Each thread with x dimension being 0 will be the leading thread among the ones with the same y index. Each thread with y dimension being 0 will be the leading thread among the ones with the same x index. At the beginning of the computation kernel, all leading threads in the y dimension will load a particle in the target node to shared memory. Each leading thread in the x dimension will load one source node to the shared memory. After a block synchronization, each thread will calculate the gravitational force between one particle in a target node and a source node. With device leaf computation, a thread will loop through all particles in the source node and accumulate particle to particle updates. With device node computation, a thread will calculate the update between a particle and aggregated data information in internal node. After another synchronization, the leading threads in the y dimension will collect all updates with respect to the same particles and perform an atomic update on the global memory.

In the second approach, we are able to reduce the number of atomic writes by a factor of BX by leveraging the shared memory and the 2-D thread layout. With the 2-D kernel,

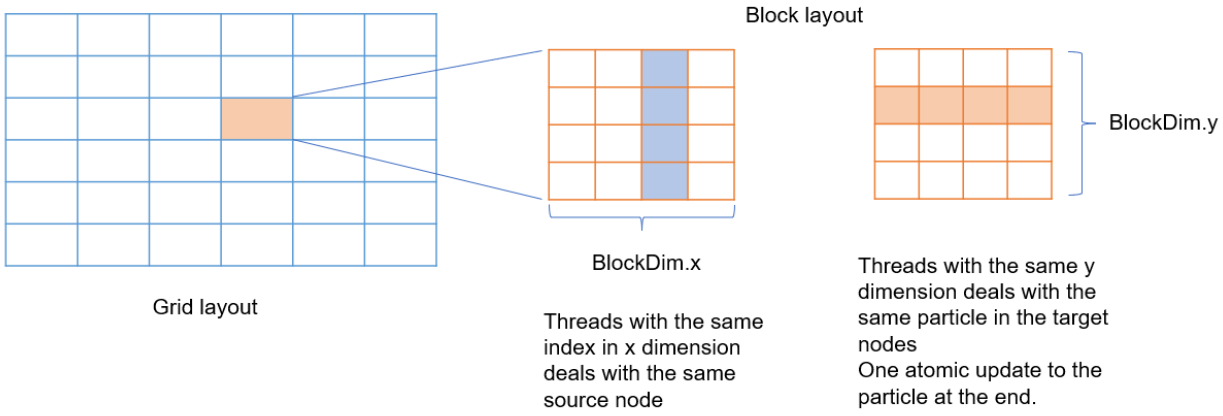


Figure 4.6: Diagram of the 2-D kernel structure. For each block with 2-D threads, threads with the same x index will share one source node. Threads with the same y dimensions share the same particle in the target node. Each block produces updates between one target node and multiple source nodes.

BY has to be the size of maximum particles in a leaf node which is configurable. $BX * BY$ cannot exceed 1024 which is the maximum number of threads per block.

To tune the performance of this approach, we can change BX, BY and the buffer size of interaction lists. For the 2-D kernel, we need to group source nodes in chunks of size BX with one target node. During the tree traversal introduced in section 3.1.3, we will have K entries to collect interactions where K is the number of leaf nodes in a Partition chare. Once at an entry, there are BX number of source nodes appended, we will create one entry in the kernel input buffer to add the target node index to the array of target nodes and add BX source nodes to the source node array. The collect buffer will be cleared after adding information to the kernel input buffer.

The buffer size of the kernel input buffer is defined as the size of the source nodes. We can change buffer size to invoke more computation at each launch. However, with a larger buffer size, it takes longer to accumulate enough inputs.

Figure 4.7 shows the average iteration time with various configurations of BY and buffer size. We can see that $BY = 32$ is the optimal configuration with buffer size being 512 and 1024. The configuration with $BY = 32$ and buffer size being 1024 has the shortest average interaction time being 18.247 seconds.

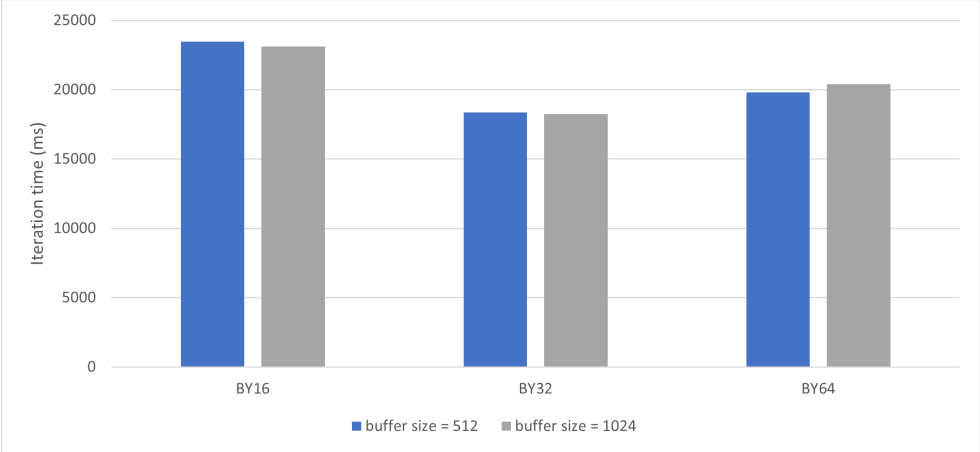


Figure 4.7: Average interaction time with various configurations of BY and buffer size. (BX is fixed to 16.)

4.3 FUTURE WORK

From Figure 4.1 we can see that adjacent target nodes may share the same source nodes in their interaction lists. In addition, nearby source nodes are likely to be listed in the iteration lists continuously. Instead of sending the complete interaction list to GPUs, we can reduce the memory transformation bandwidth by encoding the input interaction lists to group continuous sequential source node indices as well as reducing duplication of the same source node sequence in interaction lists of adjacent target nodes. In addition, we can offload more CPU work to GPUs. For example, Liu et al. [18] offloaded the local tree walk process to GPU by simulating tree walks with stacks.

CHAPTER 5: CONCLUSIONS

This thesis introduced two methods to accelerate the ParaTreeT library. The load balancing approach demonstrates strategies with good scalability that smooth the load distributed among processing units. Load balancing offers optimization of per iteration runtime during the execution adopting the runtime changes of the computation and communication behavior of chares. The design of load balancing algorithms needs insights into the characteristics of the input as well as the iterative evolving trends. The load balancers introduced above are able of reducing 20% of the runtime.

There is still room for improvement on top of our proposed load balancing algorithms. One direction is to apply different algorithms on different layers of control. We can perform one algorithm to distribute objects on computation nodes followed by another algorithm doing an intra-node placement. In this way, inter-node algorithms will have the dominant runtime factor being communication while intra-node algorithms are likely to be computation heavy. Another direction is to add fuzziness to sacrifice some sanity of partition rules in exchange of more evened load distribution.

Chapters 3 and 4 introduce the workflow of CPU implementation to describe the modification for supporting GPU kernel additions. The effort of minimizing host overhead includes the creation of memory buffers, usage of asynchronous multistreaming kernels as well as reduction of unnecessary kernel launches. Interaction between target and source nodes is highly parallelizable and computation heavy, therefore, two kernel designs are proposed by the thesis to leverage GPU devices.

The approach with 1-D kernels has the advantage of a fast collection of inputs but imposes a large amount of global atomic operations. To reduce the amount of global atomic writes, the approach with 2-D kernels utilizes the shared memory to increment target particle reuse and reduces global operations by a constant factor. With experiments exploring the configuration space, the thesis points out the best performant configuration.

The GPU kernels can be further improved by encoding inputs to GPU kernels to reduce memory copying overhead. Modification of CPU implementation is needed to increase device data reuse across iterations.

REFERENCES

[1] B. Acun et al., "Parallel Programming with Migratable Objects: Charm++ in Practice," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[2] M. Snir et al., *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd ed. Cambridge, MA, USA: MIT Press, 1998.

[3] L. V. Kale and A. Bhatele, Eds., *Parallel Science and Engineering Applications: The Charm++ Approach*. Taylor & Francis Group, CRC Press, Nov. 2013.

[4] R. K. Brunner and L. V. Kalé, "Handling application-induced load imbalance using parallel objects," in *Parallel and Distributed Computing for Symbolic and Irregular Applications*. World Scientific Publishing, 2000, pp. 167–181.

[5] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[6] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279–301, 1989. [Online]. Available: https://www.sciencedirect.com/science/article/pii/074373158990021X

[7] J. Choi, D. F. Richards, and L. V. Kale, "Achieving computation-communication overlap with overdecomposition on gpu systems," in *2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, 2020, pp. 1–10.

[8] J. C. Phillips, D. J. Hardy, J. D. C. Maia, J. E. Stone, J. V. Ribeiro, R. C. Bernardi, R. Buch, G. Fiorin, J. HÃ©nin, W. Jiang, R. McGreevy, M. C. R. Melo, B. K. Radak, R. D. Skeel, A. Singharoy, Y. Wang, B. Roux, A. Aksimentiev, Z. Luthey-Schulten, L. V. KalÃ©, K. Schulten, C. Chipot, and E. Tajkhorshid, "Scalable molecular dynamics on cpu and gpu architectures with namd," *The Journal of Chemical Physics*, vol. 153, no. 4, p. 044130, 2020. [Online]. Available: https://doi.org/10.1063/5.0014475

[9] J. Hutter, J. Szaday, J. Choi, S. Wallace, S. Liu, L. Kale, and T. Quinn, "ParaTreeT: A Fast, General Framework for Spatial Tree Traversal," in *Proceedings of the IEEE International Parallel Distributed Processing Symposium*, 2022.

[10] J. Barnes and P. Hut, "A hierarchical o (n log n) force-calculation algorithm," *nature*, vol. 324, no. 6096, pp. 446–449, 1986.

[11] A. G. Gray and A. W. Moore, " 'N-Body' Problems in Statistical Learning," in *Proceedings of the 13th International Conference on Neural Information Processing Systems*, ser. NIPS'00. Cambridge, MA, USA: MIT Press, 2000, p. 500–506.

[12] J. Bedorf and S. Portegies Zwart, "A pilgrimage to gravity on gpus," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 201–216, 2012.

[13] M. Stock and A. Gharakhani, "Toward efficient gpu-accelerated n-body simulations," in *46th AIAA aerospace sciences meeting and exhibit*, 2008, p. 608.

[14] R. G. Belleman, J. Bédorf, and S. F. P. Zwart, "High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda," *New Astronomy*, vol. 13, no. 2, pp. 103–112, 2008.

[15] T. Hamada, K. Nitadori, K. Benkrid, Y. Ohno, G. Morimoto, T. Masada, Y. Shibata, K. Oguri, and M. Taiji, "A novel multiple-walk parallel algorithm for the barnes–hut treecode on gpus–towards cost effective, high performance n-body simulation," *Computer science-research and development*, vol. 24, no. 1, pp. 21–31, 2009.

[16] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, "Scaling hierarchical n-body simulations on gpu clusters," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[17] J. Bédorf, E. Gaburov, and S. P. Zwart, "A sparse octree gravitational n-body code that runs entirely on the gpu processor," *Journal of Computational Physics*, vol. 231, no. 7, pp. 2825–2839, 2012.

[18] J. Liu, M. Robson, T. Quinn, and M. Kulkarni, "Efficient gpu tree walks for effective distributed n-body simulations," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 24–34.

[19] R. D. Williams, "Performance of dynamic load balancing algorithms for unstructured mesh calculations," *Concurrency: Practice and experience*, vol. 3, no. 5, pp. 457–481, 1991.

[20] H. D. Simon and S.-H. Teng, "How good is recursive bisection?" *SIAM Journal on Scientific Computing*, vol. 18, no. 5, pp. 1436–1445, 1997.

[21] J. Dubinski, "A parallel tree code," *arXiv preprint astro-ph/9603097*, 1996.

[22] F. Fleissner and P. Eberhard, "Parallel load-balanced simulation for short-range interaction particle methods with hierarchical particle grouping based on orthogonal recursive bisection," *International Journal for Numerical Methods in Engineering*, vol. 74, no. 4, pp. 531–553, 2008.

[23] S. Aluru and F. E. Sevilgen, "Parallel domain decomposition and load balancing using space-filling curves," in *Proceedings fourth international conference on high-performance computing*. IEEE, 1997, pp. 230–235.

[24] D. F. Harlacher, H. Klimach, S. Roller, C. Siebert, and F. Wolf, "Dynamic load balancing for unstructured meshes on space-filling curves," in *2012 IEEE 26th international parallel and distributed processing symposium workshops & PhD forum*. IEEE, 2012, pp. 1661–1669.

[25] M.-H. Fan, C.-H. Huang, Y.-C. Chung, J.-S. Liu, and J.-Z. Lee, "A programming methodology for designing parallel prefix algorithms," in *International Conference on Parallel Processing, 2001.* IEEE, 2001, pp. 463–470.

[26] "Texas advanced computing center (tacc)," The University of Texas at Austin. [Online]. Available: http://www.tacc.utexas.edu/

[27] S. Bak, H. Menon, S. White, M. Diener, and L. Kale, "Multi-level load balancing with an integrated runtime approach," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, pp. 31–40.

[28] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. Quinn, "Massively parallel cosmological simulations with changa," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–12.

[29] M. Harris, "How to access global memory efficiently in cuda c/c++ kernels," Aug 2020. [Online]. Available: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/