A MapReduce Style Framework for Searches and Computations on Parallel Tree Structures

Abstract

Cloud computing enables a user to access vast amounts computational power by providing a framework consisting of some basic primitives for the user to write. For parallel computing, such a platform is very helpful to a user for writing applications without dealing with the complexity due to parallelism. MapReduce is one such paradigm which provides parallelism to data processing for applications written through two primitives. With a similar thought, in this paper we propose a very simple and generalized framework for parallel tree structures, to build search and computational applications easily on them by defining basic primitives sequentially. The framework performs the computations using the user defined primitives on the tree structure across the cluster in parallel.

1 Introduction

The MapReduce programming paradigm [9], first proposed by Google in 2004, has gained immense popularity due to its high level abstraction, simplicity and ease of use. The continuing explosive growth raw data in the past decade, in virtually every sphere of human activity, necessitates large scale data-intensive and compute-intensive processing. Making this possible for some applications with much ease, the MapReduce model provides a very simple abstraction for data distribution and processing that can be used in applications dealing with list-based data sets. Inspired by the map and reduce primitive functions commonly used in many functional programming languages, MapReduce provides processing of large amounts of data in parallel automatically on massively parallel clusters. Major large scale MapReduce model based system implementations include the implementation used at Google [9], and the open-source Apache Hadoop project [3]. Because of its versatility and ease of use, the MapReduce model has recently been ported to emerging architectures, like graphics processors (GPUs) [18, 5], the IBM Cell processor [8], and general-purpose multi-core CPUs [21]. Researchers have also adapted and enhanced the basic MapReduce model to address more specific data processing domains, like relational data queries [27], machine learning on multi-cores [6], and .NET-based distributed computing [20]. Such functional style programming models with high level abstractions are important for Cloud-Computing, as they provide vast computing and storage resources to the user without the need for architecture awareness or dealing with parallelism and data distribution, thereby greatly reducing the programming effort in building applications.

MapReduce models proposed so far deal with list-based raw data, where processing of individual data entities are quite independent of each other. A vast number of data and compute intensive applications do not fit this simple model. A particularly important subclass of applications involve trees, widely used in many high-performance computational and search applications. These applications involve huge amounts of data on a tree structure which need to be distributed across many processors, making parallel processing on them inevitable. Structured documents represented using markup languages, like SGML and its derivatives such as XML, have a tree based representation. Vast amounts of archives of such documents need sophisticated query processing, and can be efficiently performed by exploiting their tree structure. Skillicorn [24] models operations on such structured text using parametrized tree homomorphism functions on binary trees. Many of the simpler search applications are based on *search trees*, like binary trees and B-trees, which are designed to make data retrieval fast and efficient. Spatial trees [22] are employed for geometric modeling, graphics and image processing [7]. In high-performance scientific computing, hierarchical spatial trees are intensively utilized: For example, in spatial databases, like the Sloan Digital Sky Survey [25], and in N-body problems, like the astrophysical simulations and galaxy dynamics [26, 12], molecular dynamics [4], and computational electro-magnetics [17].

Parallel implementations of these applications require significant programming effort by the user. Therefore, motivated by Google's MapReduce paradigm, we propose a simple search and computational framework for parallel tree structures. The MapReduce model deals with data that lack dependencies, and hence computations are easily captured. Incorporating operations on trees into a generalized framework is not as easy due to the many possibilities of data dependencies in the structure. Our goal in developing this model is to generalize computations and searches for various kinds of trees, including dependencies among data, and provide a simplified set of functions for the user to define in order to implement many different applications which employ trees.

The rest of the paper is organized as follows: In Section 2 we propose and describe our framework. We then provide a set of some frequently used operations on trees cast to our framework in Section 3. We briefly discuss the parallel algorithms which can be used to realize this framework in Section 4, and then provide some example applications in Section 5 which can be implemented using the framework. We end the paper with a discussion on future directions and conclusions in Sections 6 and 7, respectively.

2 The Proposed Framework

We assume that a rooted tree exists in some, possibly distributed, tree representation. Each node of the tree consists of tree topology information and the application data. The tree topology information fully defines the tree through parent and children relations, along with their ordering. We provide two basic functions in our framework, `treeSearch` and `treeCompute`, targeting applications involving search on a tree, and computations on each node of the tree, respectively. These are described in the following subsections.

In the following we use the letters u and v to denote nodes of the tree. We represent the application data for a node u by a tuple $u = \langle k_u, X_u \rangle$, where the key k_u is a node identifier, and X_u is the data stored at u. The data types of k_u and X_u are defined by the user. Also, we use the arrow "$\longmapsto$" to represent a function provided by the framework to be used by the user, and the arrow "$\longrightarrow$" to represent a function to be written by the user. The user defined functions provide the essential search or compute operations for the application.

2.1 Tree Search

Given a list of search items, the tree search operation returns the search results as a list of *node-lists*. Let $\mathcal{K}$ denote a single search item. The following is the definition form of the `treeSearch` function:

$$\texttt{treeSearch}(\text{list}(\mathcal{K})) \longmapsto \text{list}(\text{list}(v)) \tag{1}$$

where, $\text{list}(\mathcal{K}) = (\mathcal{K}_1, \mathcal{K}_2, ..., \mathcal{K}_n)$ is a list of n search items, and $\text{list}(\text{list}(v)) = (\text{list}(v_1), \text{list}(v_2), ..., \text{list}(v_n))$ is the corresponding list of the result nodes. It employs a user defined function `select`.

2.1.1 Select function

The `treeSearch` function supports searches that follow top-down search paths on the tree, where the search starts at the root of the tree and recursively moves down to the descendant nodes. For each node encountered in the search paths, one or more child nodes are chosen to be a part of the search. The user specifies the search property through the `select` function that defines, given a search item, which children of a node to descend to. It has the following function construct:

$$\texttt{select}(u, \mathcal{K}) \longrightarrow \text{list}(v) \mid \texttt{FOUND} \tag{2}$$

where, $\mathcal{K}$ is a search item, $\text{list}(v)$ is the list of the desired subset of children of the node u for the search to descend to, and `FOUND` specifies if u contains the desired search item and should be a part of the result. Note that an empty $\text{list}(v)$ would mean that the search path should stop at u.

2.2 Tree Compute

For computational applications on a tree structure, our framework provides the `treeCompute` function. This function is applied to all the nodes of the tree. The following is its definition form:

$$\texttt{treeCompute}(u) \longmapsto u' \tag{3}$$

where, $u' = \langle k_u, X'_u \rangle$ represents the updated values at node u after the computed results are stored.

The user supplies two functions: `generate` and `combine`. Consider a particular node u of the tree, and let any other node v be referred to as a *remote* node from the perspective of the *local* node u. Then the constructs of these two user defined functions are as follows.

$$\texttt{generate}(u) \longrightarrow \langle \text{list}(v), \texttt{DEPENDENCY} \rangle \tag{4}$$

$$\texttt{combine}(u, v) \longrightarrow u' \tag{5}$$

2.2.1 Generate function

The `generate` function (Equation 4) specifies the set of remote nodes, *compute-list*, to be involved in computing the new information at node u. This compute-list is generated by starting from u and using the parent and child information, made accessible through the following functions.

$$\texttt{parent}(u) \longmapsto v \tag{6}$$

$$\texttt{children}(u) \longmapsto \text{list}(v) \tag{7}$$

List iterators can be used by the user to iterate through any intermediate list of nodes, and apply conditions on their inclusion in the final compute-list.

Equations 6 and 7 define `parent` and `children` for a single node, but they extend naturally to a list of nodes: when `parent`, or `children`, is applied on a list of nodes, it translates to the function being applied to each of the nodes in the list and their result nodes concatenated in order, with removal of duplicates, to form the compute-list.

Since the `generate` function is applied individually to each node of the tree, *cascading* data dependencies may be created among the nodes, where, a node v_1 is present in the compute-list of node v_2, v_2 is present in compute-list of node v_3, and so on forming chains of dependencies. Unless these chains form cycles, the user can specify by setting the `DEPENDENCY` flag, in the output of `generate` function, if these dependencies among the nodes should be respected for computations or not.

2.2.2 Combine function

The `combine` function defines the computations at node u using the nodes v of the compute-list obtained from the `generate` function. The data at node u and a node v are combined to obtain new values at u. It is applied for each node v in the compute-list to the node u in an arbitrary order: $v_1, v_2, ..., v_n$, where n is the size of the compute-list, $v_i \in \text{list}(v)$ and $1 \leq i \leq n$. The operation defined for the nodes u and v is, therefore, required to be associative. This arbitrary order of application of `combine` results in a series of intermediate values at u: $u'_1, u'_2, ..., u'_n$, where u'_i is the result of $\texttt{combine}(u'_{i-1}, v_i)$, $1 \leq i \leq n$, $u'_0 = u$, and $u'_n = u'$. Hence, the final computed results are obtained as u', completing the `treeCompute` function.

3 Casting Tree Operations to the Framework

In this section we describe how to easily cast some frequently used operations on trees into our framework.

3.1 Point Search in Search Tree

Let us consider the point search operation on a given binary search tree. Point search on this tree progresses downwards from the root, where at each node in the search path, either left, or right child node is taken depending if the search key is smaller, or larger than the key at the node, respectively. To perform binary search using the `treeSearch` function, the search item is defined to be a key, $\mathcal{K} = k$, and the `select` function is then defined as given below.

```
select($u, \mathcal{K}$) :
 1: if $\mathcal{K} = k_u$ then
 2:   return FOUND
 3: end if
 4: $childList \leftarrow$ children($u$)
 5: if $\mathcal{K} < k_u$ then
 6:   return $childList[0]$
 7: else if $\mathcal{K} > k_u$ then
 8:   return $childList[1]$
 9: else
10:   return NULL
11: end if
```

3.2 Range Search on Spatial Tree

Let us consider a spatial tree, where each node defines a region in space, and the leaf nodes store the data points contained in their corresponding regions. For a range search on such a tree, the search item is defined to be the query range, represented by a pair of points: $\mathcal{K} = \langle q_a, q_b \rangle$. Similar to the point search, the range search proceeds downwards from the root, taking all those children of a node whose corresponding regions intersect with the query range. The `select` function can, therefore, be defined as given below.

```
select($u, \mathcal{K}$) :
 1: $childList \leftarrow$ children($u$)
 2: if $childList =$ NULL then
 3:   if $k_u$ is intersects the range $\mathcal{K}$ then
 4:     return FOUND
 5:   else
 6:     return NULL
 7:   end if
 8: end if
 9: $returnList \leftarrow$ NULL
10: for all $v \in childList$ do
11:   if $k_v$ intersects with $\mathcal{K}$ then
12:     add $v$ to $returnList$
13:   end if
14: end for
15: return $returnList$
```

3.3 Local Computations

The simplest compute operation that can be performed on a tree is local computations on each node of the tree without any interactions with other nodes. The `treeCompute` can be used for such a purpose by simply defining the `generate` function to return the local node itself, and the computations in the `combine` function:

$$\texttt{generate}(u) : \textbf{return } \langle u, \texttt{FALSE} \rangle$$

$$\texttt{combine}(u, u) : \textbf{return } \text{compute}(u)$$

3.4 Upward Tree Accumulation

Upward tree accumulation is defined as aggregation of data on each node of a tree from all its descendants, and is performed at a node by aggregating the final data from all its children nodes. Since the accumulated values of all the children of a node u are required before the values can be computed at u, this operation propagates upwards starting from the leaves, to the root. To implement this operation, define the compute-list as the list of children of a node with the `DEPENDENCY` flag set to `TRUE` in the `generate` function. The `combine` function defines the cumulative aggregation operation of data, represented by $\oplus$ below, from each child node:

$$\begin{aligned}&\texttt{generate}(u) : \textbf{return}\ \langle \texttt{children}(u), \texttt{TRUE}\rangle\\&\texttt{combine}(u, v) : \textbf{return}\ (X_u \oplus X_v)\end{aligned}$$

3.5 Downward Tree Accumulation

In contrast to upward tree accumulation, a downward tree accumulation is defined as the the aggregation of data for each node of a tree from all its ancestors, and is performed at a node by aggregating the final data from its parent. Hence, this operation propagates downwards starting at the root node to the leaf nodes. In order to implement this operation, the `generate` function is defined to return the parent node and `DEPENDENCY` as `TRUE`, while the data aggregation operation, $\oplus$, is defined in the `combine` function:

$$\begin{aligned}&\texttt{generate}(u) : \textbf{return}\ \langle \texttt{parent}(u), \texttt{TRUE}\rangle\\&\texttt{combine}(u, v) : \textbf{return}\ (X_u \oplus X_v)\end{aligned}$$

3.6 Nodes within a Distance Range

Consider a hierarchical spatial tree (e.g. a quadtree) built on data points in space, where the leaf nodes contain data points. Every node of the tree represents a region in the space. Suppose that we need to perform computations at each node of the tree using all nodes at the same level which are within a distance range $[d_1, d_2]$ from the center of the node, where $0 \leq d_1 < d_2$. The distances d_1 and d_2 represent a shell around each node. Let us call this shell S_u for the node u. Therefore, the `generate`(u) function needs to return the list of nodes which are at the same level as u and intersect with S_u. A straight-forward way to accomplish this is as follows. Moving upwards in the tree from u, using the `parent` function, the ancestor of u is accessed which fully contains S_u. Then moving downwards to its descendants one level at a time, they are checked if they intersect with S_u. If they do, they are retained for next iteration, till the remaining nodes are either leaves, or at the same level as u. The following pseudocode for the `generate` function defines this operation:

```
generate($u$) :
  $i \leftarrow 0$, $par \leftarrow u$
  $temp \leftarrow$ NULL, $nodeList \leftarrow$ NULL
  while $par$ does not fully contain $S_u$ and is not the root do
    $par \leftarrow$ parent($par$)
    $i \leftarrow i + 1$
  end while
  $temp \leftarrow par$
  for $j = 1$ to $i$ do
    $temp \leftarrow$ children($temp$)
    for all $v \in temp$ do
      if $v$ and $S_u$ do not intersect then
        remove $v$ from $temp$
      else if $v$ is a leaf node then
        add $v$ to $nodeList$ and remove from $temp$
      end if
    end for
    end for
    $nodeList \leftarrow temp + nodeList$
    return $\langle nodeList, \texttt{FALSE} \rangle$
```

4 Building the Framework

In our framework, we assume that the tree already exists, distributed across a number of processors in a cluster. A number of parallel tree construction algorithms have been developed over the years for various kinds of trees, for example, see [1, 16] for parallel construction of spatial trees and multi-dimensional binary search trees on coarse-grained distributed memory parallel computers. Libraries for these parallel algorithms exist and can be used to build the required tree. Parallel algorithms have also been developed for basic operations on tree structures. In this section we briefly discuss some of these parallel algorithms for tree operations which can be utilized to build our framework.

4.1 Tree Search

In a general search problem, called the multisearch, given m search items need to be processed in parallel on a data structure. In our framework, we consider multisearch for a search tree, where m queries need to be processed in parallel on the tree. Our framework supports those tree searches where the query processing starts at the root of the tree and descends down, one level at a time, along a set of search paths defined during the processing, according to the search strategy of the tree. During search of multiple items, a particular node may occur in the search paths of more than one query. For example, the root node appears in all the search paths. In a parallel system, the nodes of the search tree are distributed across processors, and such congestion creates bottlenecks in the performance. Hence, proper distribution of processing among the processors is needed.

Dehne *et al.* [11] provide a technique to process multiple searches in parallel on an ordered h-level graph for a hypercube multiprocessor. Also, Atallah *et al.* [2] provide parallel multisearch techniques on hierarchical directed acyclic graphs for mesh-connected parallel computers. The k-ary search trees form a subset of both ordered h-level graphs and directed hierarchical DAGs, where the direction of edges in the search tree is from a node to its child. Therefore, these methods can be adapted to our framework to process multiple search queries on the search trees in parallel.

Note that search operations on tree based structured documents, not involving search trees, can be implemented using the tree compute function. This is because in such documents, the data is not organized according to a search property, rather it is stored as in the document attaining the tree structure due to the document representation, and hence the search needs to be applied on all nodes of the tree structure.

4.2 Tree Compute

For computations on the tree, the user invokes `treeCompute` function after defining `generate` and `combine` primitives. The framework then executes it individually on all nodes of the tree. First, the compute-list for each node is constructed using the `generate` function. A call to `parent`, or `children`, function for a node u may need to return a node v which is either present on the same processor as u, or on another processor. In the latter case, the target processor is determined from the parent and child information, and a list of node requests is created for each node. Such lists for all the nodes within a single processor are gathered together to form a request list for that processor. The processors then perform a single send and receive communication round to send the requests and obtain the required nodes from other processors. There are as many communication rounds as the number of calls to `parent` and `children` functions.

Once the compute-lists are created, the `combine` function is applied to each node in the list in an arbitrary order, since preserving an order hurts concurrency. Depending on the `DEPENDENCY` flag as set by the user, there arise two possible cases:

1. `DEPENDENCY` = `FALSE`. This is the simple case. Since dependencies are not to be respected, the computations on each node are performed, while storing a copy of the old values. Using a send-receive

round of communication among the processors to send the request and receive the values from the nodes in the compute-list, the `combine` function is applied on them.

2. `DEPENDENCY` = `TRUE`. Since dependencies among the various nodes need to be taken care of, the values at a particular node cannot be computed unless the final values at all the nodes in its compute-list are available. We discuss this case in more detail below.

The tree topology information in the framework also maintains the *level* information for each node. This specifies the level of the node in the tree, with the root being at level 0. Consider a node u. To respect the dependencies, there are two possible cases: $\forall v \in$ compute-list of u,

1. $level(v) < level(u)$, or
2. $level(v) > level(u)$.

(Note that the case when $level(v) < level(u)$ for some nodes v in the compute-list of u, while $level(v) > level(u)$ for other nodes, leads to cyclic dependencies, and cannot be respected. Also, the case when the $level(v) = level(u)$, leads to cyclic dependencies.) Both the cases, (1) and (2), are a form of tree accumulations [13] for the simple cases when the nodes in the compute-list of all nodes are either their parent, or all their children, resulting in downward accumulation and upward accumulation, respectively. Parallel algorithms for upward and downward tree accumulations as an adaptation of tree contraction algorithms were first given by Gibbons *et al.* [14] for the EREW PRAM. Sevilgen *et al.* [23] gave efficient parallel tree accumulation algorithms, using parallel prefix, for coarse-grained distributed memory parallel computers. These algorithms can be used to implement the computations for the above cases in our framework.

The general cases when the nodes in the compute-list of a node are either an ancestor, or descendants, (and not parent, or children, respectively), a generalization of the tree accumulations can be formed. A set of new trees is created using the dependency information. Let us consider the case (2), when $level(v) > level(u)$ for all nodes v in the compute-list of u. The other case can be dealt with similarly. Construct a set of new trees as follows: Each node v in the compute-list is made a child of the node u (and u is made the parent of v). After applying this operation on all nodes in the original tree, a forest of one or more trees are obtained. The disjoint trees in the resulting forest do not have any data dependencies among them, and therefore, can be treated as separate trees. Once the new trees are obtained, the parallel tree accumulation algorithms can be directly applied on them to obtain the final result.

The parallel tree accumulation algorithms require the tree to be present in post order form. If the tree is not available in post-ordering, it can be easily obtained using simple operations. Obtaining the Euler tour for a graph is described in [10] for a coarse-grained multicomputer. A post-ordering of a tree can be easily constructed using its Euler tour [19, 23]: A node may exist multiple times in the Euler tour, where the first occurrence is always preceded by its parent, and the last occurrence is after all the nodes in its subtree. Therefore, by removing all the occurrences of each node except its last one, the post ordering of the tree is obtained.

5 Sample Applications

To demonstrate the use of our framework, in this section we give a few sample applications. We give examples of tree based scientific computational applications, the All-Nearest-Neighbor computations and N-body simulations.

5.1 All Nearest Neighbor Computations

Consider the application where, given a set of data points, the nearest neighbor of each point needs to be computed. Suppose that the given data points are in two-dimensional space and a quadtree is built over them. Each leaf u of the tree contains a single data point in the region represented by it. This data point is stored in its value field $X_u.point$. The results to be computed, the nearest neighbor and distance to it, are stored as $X_u.nn$ and $X_u.dist$, respectively. By the property of the quadtree, the nearest neighbor of a point is always within the region defined by the immediate neighbor leaf nodes of the leaf node it lies in. Therefore, to compute the nearest neighbor, each leaf node needs to define a list of neighbor leaf nodes around it. To do so, we can use the same `generate`(u) function from Section 3.6 by setting d_1 as 0 and d_2 as the size of the node u, and adding another condition for the list to be generated only if u is a leaf node. Once the compute-list containing the neighbor leaf nodes is obtained, the `combine` function can be defined as given below to compute the nearest neighbor point among the nodes in the compute-list. The `distance` function used below computes the distance between two points.

```
combine(u, v) :
1: $temp \leftarrow \texttt{distance}(X_u.point, X_v.point)$
2: if $temp < X_u.dist$ then
3:     $X_u.nn \leftarrow X_v.point$
4:     $X_u.dist \leftarrow temp$
5: end if
6: return $u$
```

Note that for nearest neighbor search of a query point, the `treeSearch` function can be used, since it does not apply to every point in the data set.

5.2 N-body Simulations

Fast algorithms for the N-body simulations use a hierarchical spatial tree to approximate the particle interactions which are significantly far from each other, instead of performing the computationally prohibitive $O(N^2)$ computations. Greengard *et al.* presented the Fast Multipole Method (FMM) for the N-body simulations in [15]. Parallel algorithms for the FMM based particle simulations have since been developed for various applications (e.g. see [17]). We present here how the basic FMM based N-body simulation algorithm, which uses octrees, can be implemented using our framework. We only give a high level view of the original algorithm here; for a detailed description see [15] and [17]. Given the octree constructed on the particles in space, a single iteration of the simulation algorithm computes the force on each particle due to all other particles. The leaf nodes of the octree contain a number of particles, depending on the precision required. The algorithm performs the computations through the following five basic steps: (1) compute *multipole expansions* for each node in the tree using bottom-up traversal, (2) compute partial *far-fields* for each node using its *interaction list*, (3) compute total far-fields for each node using a top-down traversal, (4) compute *near-fields* for each node using its *neighbor-list*, (5) compute the final forces at the particles in the leaf nodes by summing the near-fields and far-fields. The multipole expansion of a node represents the force field generated by the particles within it. Far-field on a node represents the field due to all particles outside the node which lie in the non-neighboring nodes, while near-field on a leaf node represents the force field due to all particles contained in the neighboring leaf nodes. Let the multipole expansion for node u be denoted by $X_u.\phi$, partial far-field by $X_u.\psi'$, and the total far-field by $X_u.\psi$. Below we describe the implementation of these computations using multiple `treeCompute` functions for the different steps.

5.2.1 Computing Multipole Expansions

This step requires two sub-steps, where, first the leaf nodes compute their multipole expansions, followed by an upward accumulation on the tree to compute the multipole expansions for the other nodes. For local computations on the leaves, the `generate` function returns the node itself as described in Section 3.3. The `combine` function below defines the multipole expansion computations.

```
combine(u, v) :
1: if $u$ is a leaf node then
2:     compute $X_u.\phi$
3: end if
4: return $u$
```

The `treeCompute` function is then called, which results in the values being computed at the leaf nodes. Once this is done, the upward tree accumulation can be performed. Therefore, the `generate` function is defined as in Section 3.4, and the `combine` function is defined to compute the multipole expansions as given below, followed by a call to the `treeCompute` function.

```
combine(u, v) :
1: temp ← shift the $X_v.\phi$ to the center of $u$
2: $X_u.\phi \leftarrow X_u.\phi + temp$
3: return $u$
```

5.2.2 Computing Partial Far-fields

For each node, the partial far-fields due to nodes in its *interaction-list* are computed. The interaction-list for a node u is defined as the set of nodes which are children of the immediate neighbors of u's parent node. This list can be constructed through the `generate` function in a way similar to the one described previously in Section 3.6. The following `combine` function can be used to compute the partial far-fields.

```
combine(u, v) :
1: temp ← convert $X_v.\phi$ to local expansion about the center of $u$
2: $X_u.\psi' \leftarrow X_u.\psi' + temp$
3: return $u$
```

5.2.3 Computing Total Far-fields

For computing the total far-fields at all the nodes, a downward accumulation needs to be performed on the tree. The `generate` function is same as in Section 3.5, and the `combine` can be defined as follows.

```
combine(u, v) :
1: temp ← expand $X_v.\psi'$ to the center of $u$
2: $X_u.\psi \leftarrow X_u.\psi' + temp$
3: return $u$
```

5.2.4 Computing Near-fields

To compute the near-fields, the neighbors of the leaf nodes are required. Therefore, the compute-list can be constructed using the `generate` function from Section 3.6, similar to nearest neighbor computations as described before. Once these compute-lists are constructed for the leaf nodes, the `combine` function can be defined to compute the near fields as follows.

```
combine(u, v) :
1: for all particles $p$ in $v$ do
2:     compute interactions due to $p$ on every particle in $u$
3: end for
4: return $u$
```

5.2.5 Computing the Total Fields at the Particles

Once all the near-fields and far-fields are available, the total fields are computed for each particle at the leaf nodes locally. The `generate` function returns the node itself, and the `compute` function performs the final computations as given below.

```
combine(u, v) :
1: for all particles $p$ in $u$ do
2:     compute the interactions on $p$ due to all other particles in $u$
3:     add the near and far fields to obtain the final force field for $p$
4: end for
5: return $u$
```

6 Discussion and Future Directions

In our framework we assume that the tree under consideration is already constructed in a distributed fashion across the cluster. Providing a general tree construction framework is not practical since every kind of tree has a different efficient construction algorithm. In some trees, the data entities themselves are the nodes of the tree, for example, k-dimensional binary search tree (kd-tree) and structured documents. While in other kinds of trees, the nodes of the tree are inferred from the data points and the data may reside only in the leaf nodes, for example, the octrees. Libraries for parallel construction of various kinds of trees exist, and can be used for building the required tree. Our framework provides a third function, `treeConstruct`(list($data$), `TREETYPE`), which takes as arguments the raw data entities, list($data$), and a flag, `TREETYPE`, specifying which kind of tree to be built on them. Depending on the flag, this function can choose the required tree construction library and build the tree on the input data.

Our immediate future direction is to efficiently adapt the parallel algorithms mentioned in Section 4 to provide a scalable implementation of the proposed framework for a large cluster. A number of implementations of this framework are possible depending on the machine to be used. This can be either a small shared-memory multiprocessor, or a tightly coupled supercomputer, or a very loosely coupled cluster of machines. To scale the framework to very large clusters of commodity machines, it needs to provide fault tolerance. This can be done by using a master-slave technique, similar to the one in MapReduce [9]. The master can keep track of all the slave processors and reschedule a task in the event of a slave failure. The slaves participate in the operations on the distributed tree structure. Master failures can be answered by keeping multiple copies and checkpointing. Nowadays clusters are increasingly built using different kinds of processors and accelerator boards. A single large cluster may contain different processors, ranging from commodity CPUs, to multicores and specialized hardware like GPUs. We also plan to adapt our framework to such heterogeneous environments in the future.

7 Conclusions

In this paper we proposed a simple and generalized framework for search and computational applications on a parallel tree structure. This framework provides an easy to use programming paradigm, in a style similar to Google's MapReduce programming model. The user implements the logic behind the application through two primitives, `generate` and `combine`, for computations, and one primivite, `select`, for search operations on the tree. With this the implementation complexity is greatly reduced. Moreover, the parallelism is hidden from the user. Existing efficient parallel algorithms for multi-search and tree accumulations can be adapted to implement this framework. We cast some of the frequently used operations on the tree to our framework, and then demonstrated its the ease of use through some example tree-based applications: the binary search tree, range search on a spatial tree, all nearest neighbor computations, and a more complex, N-body simulations.

References

[1] Ibraheem Al-furaih, Srinivas Aluru, Sanjay Goil, and Sanjay Ranka. Parallel Construction of Multidimensional Binary Search Trees. *IEEE Transactions on Parallel and Distributed Systems*, 11(2):136–148, February 2000.

[2] M.J. Atallah, Frank Dehne, R. Miller, A. Rau-Chaplin, and J.J. Tsay. Multisearch Techniques: Parallel Data Structures on Mesh-Connected Computers. *Journal of Parallel and Distributed Computing*, 20(1):1–13, 1994.

[3] A. Bialecki, M. Cafarella, D. Cutting, and O. Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene.apache.org/hadoop*, 2005.

[4] John A. Board, Jeffrey W. Causey, James F. Leathrum, Andreas Windemuth, and Klaus Schulten. Accelerated molecular dynamics simulation with the parallel fast multipole method. *Chemical Physics Letters*, 198(1,2):89–94, 1992.

[5] Bryan Christopher Catanzaro, N. Sundaram, and Kurt Keutzer. A Map Reduce Framework for Programming Graphics Processors. In *Workshop on Software Tools for MultiCore Systems*, 2008.

[6] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2007.

[7] Mark de Berg, Marc van Kreveld, Mark Overmans, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, second edition, 2000.

[8] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell BE Architecture. Technical Report TR1625, Department of Computer Sciences, The University of Wisconsin-Madison, Madison, WI, 2007.

[9] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, Berkeley, CA, USA, 2004. USENIX Association.

[10] Frank Dehne, A. Ferreira, E. Caceres, SW Song, and A. Roncato. Efficient Parallel Graph Algorithms for Coarse-Grained Multicomputers and BSP. *Algorithmica*, 33(2):183–200, 2002.

[11] Frank Dehne and A. Rau-Chaplin. Implementing data structures on a hypercube multiprocessor, and applications in parallel computational geometry. *Journal of Parallel and Distributed Computing*, 8(4):367–375, 1990.

[12] John Dubinski. A Parallel Tree Code. *New Astronomy*, 1:133–147, 1996.

[13] Jeremy Gibbons. Upwards and Downwards Accumulations on Trees. In *Proceedings of the Second International Conference on Mathematics of Program Construction*, pages 122–138. Springer-Verlag, 1993.

[14] Jeremy Gibbons, Wentong Cai, and David B. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, 23:1–18, 1994.

[15] Leslie F. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, 1987.

[16] Bhanu Hariharan and Srinivas Aluru. Efficient parallel algorithms and software for compressed octrees with applications to hierarchical methods. *Parallel Computing*, 31(3+4):311–331, 2005.

[17] Bhanu Hariharan, Srinivas Aluru, and Balasubramaniam Shanker. A scalable parallel fast multipole method for analysis of scattering from perfect electrically conducting surfaces. In *SC'02: Proceedings of the 2002 Supercomputing Conference*, pages 1–17. IEEE Computer Society Press, 2002.

[18] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a MapReduce framework on graphics processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[19] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[20] Chao Jin and Rajkumar Buyya. MapReduce Programming Model for .NET-based Distributed Computing. Technical report, The University of Melbourne, Australia, October 2008.

[21] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE Computer Society, 2007.

[22] Hanan Samet. Spatial Data Structures. *Modern database systems: the object model, interoperability, and beyond*, pages 361–385, 1995.

[23] Fatih E. Sevilgen, Srinivas Aluru, and Natsuhiko Futamura. Parallel algorithms for tree accumulations. *Journal of Parallel and Distributed Computing*, 65(1):85–93, 2005.

[24] David B. Skillicorn. Structured parallel computation in structured documents. *Journal of Universal Computer Science*, 3:42–68, 1997.

[25] Alexander S. Szalay, Peter Z. Kunszt, A. R. Thakar, Jim Gray, and Don Slutz. Designing and mining multi-terabyte astronomy archives: The Sloan Digital Sky Survey. In *Proceedings of ACM SIGMOD*, volume 29, pages 451–462. ACM New York, 2000.

[26] Michael S. Warren and John K. Salmon. Astrophysical N-body simulations using hierarchical tree data structures. In *SC'92: Proceedings of Supercomputing*, pages 570–576, 1992.

[27] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040. ACM, 2007.