Global Trees: A Framework for Linked Data Structures on Distributed Memory Parallel Systems

D. Brian Larkins†, James Dinan†, Sriram Krishnamoorthy‡,
Srinivasan Parthasarathy†, Atanas Rountev†, P. Sadayappan†

† Dept. of Computer Science and Engineering
The Ohio State University
Columbus, OH 43221
{larkins, dinan, srini, rountev, saday}@cse.ohio-state.edu

‡ Pacific Northwest National Laboratory
Richland, WA 99352
sriram@pnl.gov

Abstract

This paper describes the Global Trees (GT) system that provides a multi-layered interface to a global address space view of distributed tree data structures, while providing scalable performance on distributed memory systems. The Global Trees system utilizes coarse-grained data movement to enhance locality and communication efficiency. We describe the design and implementation of GT, illustrate its use in the context of a gravitational simulation application, and provide experimental results that demonstrate the effectiveness of the approach. The key benefits of using this system include efficient shared-memory style programming of distributed trees, tree-specific optimizations for data access and computation, and the ability to customize many aspects of GT to optimize application performance.

I. Introduction

Developing high-performance parallel applications which use linked data structures on distributed-memory clusters is challenging. Many scientific applications utilize algorithms based on tree data structures. Trees are especially useful in representing hierarchical relationships between data which may not be known until runtime or may otherwise evolve during the course of a computation. Methods such as the Barnes-Hut n-body simulation algorithm [1], Fast Multipole Methods (FMM) [2], and Multiresolution Analysis [3] use trees to represent a fixed space populated by a dynamic distribution of elements. Other problem domains, such as data mining, use trees to summarize large input datasets into a hierarchy of relationships which capture the information in a form that lends itself to efficient mining [4].

Linked data structures can provide a compact and efficient representation when the problem is non-uniform and sparse. These computations are often amenable to recursive formulations. Irregular recursive structures present challenges in parallel distributed memory environments, both with the distribution of program data as well as balancing the computation over the tree. Distributed memory implementations of tree algorithms provide access to the shared data structure either with explicit message passing, via a shared memory model, or a combination of both. Existing parallel programming tools support these models at different levels.

Message passing systems such as MPI [5] provide a process-centric view of the computation which requires the programmer to explicitly manage data distribution and load balance, or to construct a system to do so on their behalf. Efficient implementations of irregular recursive algorithms can require significant modifications to avoid excessive synchronization and communication. Common approaches include statically partitioning the data among processors, which may lead to load imbalance – or replicating tree data across processors, which may limit problem size.

A shared memory programming model can be provided at different levels of abstraction. At the highest level, modern parallel programming languages such as Chapel [6], X10 [7], and Fortress [8] offer support for a global address view of distributed data structures and allow communication and load balance to be handled by the compiler

This research was supported in part by DOE grant #DE-FC02-06ER25755, by NSF grants #0403342, IIS-0347662 and CCF-0702587, and a State of Ohio Development Fund.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC2008 November 2008, Austin, Texas, USA 978-1-4244-2835-9/08 $25.00 ©2008 IEEE

and runtime. Similarly, distributed shared memory systems such as Intel's Cluster OpenMP provide support for a global address space directly and data movement is done transparently by the distributed shared memory runtime. At a lower level of abstraction, one-sided communication libraries such as ARMCI [9], GASNET [10], MPI-2 [11], and SHMEM [12] allow the program to efficiently access remote data asynchronously through the use of hardware supported rDMA operations. Languages such as UPC [13] utilize one-sided communication libraries to provide a shared view of global program data that is physically spread across the memories of nodes in a cluster.

In this paper, we describe a data-structure centric library based approach to efficient parallel global-address-space computing. The Global Trees (GT) library provides a global view of distributed linked tree structures and a set of routines which operate on these structures. GT can inter-operate with existing parallel programming models that either use message passing or global view approaches. GT provides support for parallel computations over irregular and dynamic tree structures. The approach is based on two key insights. First, tree-based algorithms are easily expressed in a fine-grained manner, but data movement must be done at a much coarser level of granularity for good performance. Second, since GT is focused on a single data abstraction, attributes unique to tree structures can be exploited to provide optimized routines for common operations. We make the following contributions:

Efficient Fine-Grained Data Access: The system combines the ease of programming in a shared memory environment with the efficiency of coarse-grained data movement during a dynamic computation. Tree data is grouped into *chunks*, from which subsequent data accesses can be serviced without communication. Each process involved in the parallel computation may independently and asynchronously access global tree data with no explicit cooperation from other processes. While GT provides fine-grained node-level data access similar to shared memory programming models, internally the system is aware of the data distribution and is able to exploit data structure specific knowledge to improve locality and yield good performance.

Tree Structure Optimizations: We exploit the linked nature of tree structures when resolving portable global references. Global pointers which reside inside the same chunk as the referenced node are modified to portable pointers which achieve nearly the same performance as normal pointer dereferencing.

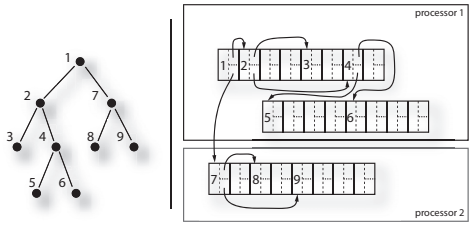
High Level Operations on Distributed Trees: The Global Trees framework provides a parallel implementation for common tree operations, optimized to take advantage of locality information known to the runtime.

Application-driven Customization: Our approach provides a set of data structures and operations which may be used to implement generalized tree algorithms. When high performance is critical, GT permits developers to directly influence runtime behavior and provides performance and profiling statistics to tune the application.

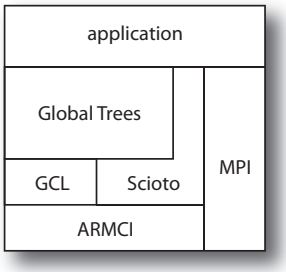
Empirical Evaluation: We present an empirical evaluation of our approach which demonstrates that our technique is effective. We evaluate the performance of Global Trees with the Barnes-Hut benchmark from the SPLASH-2 parallel benchmark suite [14]. We also provide a characterization of the relationship between chunk size, communication bandwidth, and locality due to chunk packing.

The remainder of this paper begins with an overview of the system in Section II, followed by a description of the programming model in Section III. The design and implementation of the runtime system is described in Section IV, followed by discussion of system support for performance tuning in Section V. Section VI presents an experimental evaluation of the system. Related work is described in Section VII and conclusions are presented in Section VIII.

II. Overview



(a) A global tree and corresponding chunk structure



(b) GT Library Framework

Fig. 1: Global Trees Programming Environment

Global Trees is a run-time library and programming interface which provides a global address view for tree-based data structures on distributed memory clusters. In contrast to traditional distributed shared memory systems, GT is designed specifically to support dynamic linked data

structures and as a result is able to focus on providing efficient access for applications which use these structures. Global Trees automates the allocation, distribution, and communication of shared data.

Applications which use tree data structures will usually access nodes within the tree in a fine-grained manner. Because it is crucial to accommodate element-wise access in an efficient manner, all global shared data is grouped into *chunks*. Chunks are of a sufficient size to offset the overhead of performing a communication. Figure 1a shows how a binary tree may exist in a collection of chunks. The chunk structure is transparent to the application, but is a key component of the Global Trees system.

As shown in Figure 1b, Global Trees is implemented on top of several lower-level runtime libraries with the ARMCI one-sided communication library at its base[9]. The design of GT was split into two components: the Global Chunks Layer (GCL) and a higher level Global Trees (GT) interface. The GCL provides a general framework for chunking collections of linked data elements, performing chunk-wise operations, and managing communication and caching. This layer provides much of the functionality of a distributed shared-memory system, customized for use exclusively with linked data structures. The GT layer provides higher level tree abstractions on top of the GCL and is the principal component that the user interacts with. However, the user is able to take advantage of lower-level functionality of each component as well as interoperability with MPI.

GT also takes advantage of the Scioto dynamic load balancing system to manage task parallelism exposed through parallel tree traversals[15]. Scioto provides a scalable, locality-aware runtime system for the managed execution of tasks that execute in global space. This system forms a key component in the GT programming model and allows the user to express irregular and nested parallel computations through global tasks that are automatically load balanced with respect to data locality.

III. Programming Model

GT programs are executed in an SPMD manner with MIMD task-parallel regions managed by Scioto where tasks operate on shared global trees. Typically, some collective calls occur during program initialization, but most communication happens via one-sided communication operations. Additional collectives are used for synchronization and termination. Each process can independently and asynchronously access any portion of the shared data, without requiring the invocation of application code on the remote end.

Internally, Global Trees represents tree nodes as collections called *chunks*. Similar to shared pages in distributed shared memory systems, chunks permit the application to access data in a fine-grained manner, but the runtime performs data movement at a coarser granularity to reduce communication overhead. The application accesses tree nodes without any explicit interaction with the chunking subsystem. For applications which require greater performance than the transparent mechanisms provide, the chunking layer can be exposed for greater control over data locality.

Much like Global Arrays [16], Global Trees uses a *get/compute/put* model for tree access. Prior to operating on a specific tree node, a GT global pointer must be dereferenced through the API. After the application has completed its computation on the element, it is either written back to the global address space with a *put* operation, or is discarded with a *finish* operation if it is a copy and no update is required. *Put* operations which update remote memory locations may return before the communication is complete. Remote updates are guaranteed to be complete only after the application calls one of the GT synchronization operations.

All global data is partitioned among the participating processes. Global data resident in a single process is defined to be local data for this process, and shared data is local only to a single process. Inter-process communication happens via the creation and use of global tree structures, however GT is interoperable with other message-passing and one-sided communication systems, such as MPI or ARMCI.

A. Using Global Trees

The basic structure of a GT application consists of a collective call to initialize the runtime and underlying communication layer, followed by the creation of one or more node groups. The program may then flow asynchronously, with each process allocating, accessing, or updating global tree data independently. GT provides built-in traversal routines which are collective, as well as operations to explicitly manage caching status.

Node Groups: (`gt_nodegrp_t`) A *node group* corresponds to a collection of nodes which have similar size and connectivity parameters. Node groups may consist of a single global tree, a forest of global trees, or simply a set of nodes. Distinct node allocations from the same node group will be made from the same pool of chunks.

Node groups are created to represent a collection of tree nodes which have similar linkage characteristics and are allocated from the same chunk pool. Node groups can be created or destroyed and are used when allocating new shared tree nodes. Caching tree data also happens at the node group level.

Global Node Pointers: (`gt_nodeptr_t`) These pointers are the basic node reference data type used by programmers. Global node pointers are portable references to any tree node. All operations on global node pointers are performed using the GT programming interface to ensure safety.

Global node pointers are returned by the allocation routines and are also used within a node to represent linkages to other nodes. Global pointers may be marked as either *active* or *inactive*. Inactive global pointers correspond to `NULL` pointers. GT provides primitives to copy global pointers, test for referential equality, as well as test for and mark pointers as inactive.

Tree Nodes: (`gt_node_t`) Tree nodes consist of two components: node *link structure* which is managed through the API and is visible to GT, and the *node body* which is a user-defined structure that is opaque to the runtime. The node link structure contains a number of global node pointers which may refer to other nodes. The application specifies the number of links when initializing the node group structure. Links may point to parent nodes, children, or be used to link related nodes together in an application specific way and the API provides a means to distinguish these links when performing tree traversals.

Tree nodes can be allocated, either collectively (typically only when creating the root of a tree), or from a single process. Allocations are done with respect to a specific node group, which may use the default chunk allocation scheme or a custom allocator to optimize node/chunk placement. One-sided allocation calls take an optional *hint* parameter which can provide context from the call site to inform the allocation choice. For example, it is often desirable to allocate a child node from the same chunk as the parent. A global pointer to the parent could be passed to the allocation routine to help the allocator place the new node "close" to the parent node. This will be discussed in greater detail below.

Nodes are accessed by the *get* operation. Get operations specify a node group and a global pointer and return a reference to node data. If the node is a remote node, the return value will be a reference to a copy of the remote node. When the node is local, different versions of the get operation can be used to work on the node directly, or on a copy which is safe to modify and discard.

Updates are affected by calling the *put* operation. Put operations take a global pointer and a pointer to a tree node and update the master copy of the node. If the data was local and direct access mode was used to get the node, the put operation becomes a no-op. The put may not happen immediately and is only guaranteed to complete at the remote end after a fence or barrier operation.

Connections are forged between two global tree nodes by using *link* operations. The actual link operation is communication-free, as it relies on prior get and subsequent put operations to handle all updates to the node structure. Linking creates a one-way reference between a source node and destination node. Routines are provided to explicitly set parent links and indexed child links, and also to create general links between any two nodes in the global space.

Global Trees also provides barrier and fence operations. Since remote updates via *put*, are one-sided and asynchronous communications, they may not complete before the updated values are needed to proceed with the computation. GT also provides mechanisms which ensure data consistency; they are designed to be used with control and data synchronization operations provided by other parallel runtimes to avoid redundant synchronization.

Example: The code given in Listing 1 demonstrates a basic application-defined traversal of a Global Tree which performs a recursive copy. For brevity, this routine is assumed to be called in parallel on independent subtrees. A global pointer to the root node of a subtree to copy is passed in as `src`, as is a previously allocated destination node pointer, `dst`. This routine copies the subtree rooted at `src` to a subtree rooted at `dst`. In lines 6-7, both node pointers are dereferenced by `gt_get_node()` which returns a pointer to the node structure. For each possible child of the source node, `gt_get_child()` is called in line 10, and the returned global node pointer is tested to see if this index points to a child in line 11. If so, a new tree node is allocated for the destination tree, with the destination node (parent of the new node) as the allocation hint in line 12. The newly allocated child node is then linked to the parent by a downward edge by the following call to `gt_link_child()`. A parent link could be handled similarly. The call proceeds recursively in line 14, followed by an update of the destination node and a finish (discard) of the source node.

B. Tree Traversals

Global Trees may be used to traverse shared trees using standard techniques (recursion, looping, etc.) in conjunction with the get/put operations. Because many applications use common traversal orders, GT provides optimized, parallel traversals which are capable of invoking application-specific code during the traversal. Global tree traversals are performed by collectively registering a visitor callback which operates on a single tree node. The visitor identifier is then passed as a parameter to the desired traversal pattern which automatically generates a valid parallel traversal and invokes the callback on each node in the traversal. Currently, GT provides general top-down, bottom-up, and level-wise traversals. Other traversals such

```
1  void tree_copy(gt_group_t ng, gt_nodeptr_t src,
2      gt_nodeptr_t dst) {
3    gt_nodeptr_t schild, dchild;
4    gt_node_t snode, dnode;
5    int i;
6
7    snode = gt_get_node(ng, src);
8    dnode = gt_get_node(ng, dst);
9
10   for (i=0;i<NUM_CHILDREN;i++) {
11     schild = gt_get_child(ng, snode, i);
12     if (gt_is_active(schild)) {
13       dchild = gt_node_alloc(ng, dst);
14       gt_link_child(ng, dst, dnode, i, dchild);
15       tree_copy(ng, schild, dchild);
16     }
17   }
18   gt_put_node(ng, dst, dnode);
19   gt_finish_node(ng, snode);
20 }
```

Listing 1: Example: Copying a Global Tree.

as parallel breadth or depth-first search could also be provided. These traversals are implemented in a task-parallel manner with runtime awareness of data locality in conjunction with the Scioto locality-aware dynamic load-balancing system.

Traversal Visitors:

A traversal visitor is a callback function which is invoked by a predefined traversal routine. The callback is invoked on every node in the tree or subtree being traversed in the order specified by the traversal. To use a traversal, the programmer creates a callback function and registers it with GT. Later, the programmer may invoke a traversal, passing the registered callback key to identify the desired visitor function to be invoked at each node. Visitor callbacks have the following form:

```
typedef void (*gt_visit_node_t)(gt_group_t nodegrp,
                                gt_nodeptr_t ptr);
```

Built-in Traversal Example:

The code fragment listed in Listing 2 demonstrates a typical usage of the built-in traversal routines. This example computes a "cost" value to measure the number of leaves contained in all subtrees within the tree. Presume leaves are initialized to a weight of 1.0. Upon calling `compute_cost()`, a bottom-up traversal will be performed, each node accumulating the weights of its children along the way.

Because GT will automatically parallelize the tree traversal, a node may be visited on any available processor. To preserve the portability of a visitor callback, all callbacks must be registered collectively. The callback function `cost_visitor()` performs this accumulation for a single node. To actually invoke the traversal, the node group, root of the tree or subtree, and the registered visitor key are passed to `gt_traversal_bottomup()`.

```
1  typedef struct {
2    double cost;
3  } mynode_t;
4
5  cost_visitor(gt_group_t nodegrp, gt_nodeptr_t ptr) {
6    mynode_t *thisbody, *childbody;
7    gt_nodeptr_t childptr;
8    int i;
9
10   thisbody = (mynode_t *)gt_get_node(nodegrp, ptr);
11   thisbody->weight = 0;
12
13   for (i=0;i<NUM_CHILDREN;i++) {
14     childptr = gt_get_child(nodegrp, thisnode, i);
15     if (gt_is_active(childptr)) {
16       childbody = (mynode_t *)gt_get_node(nodegrp,
             childptr);
17       thisbody->weight += childbody->weight;
18       gt_finish_node(nodegrp, childnode);
19     }
20   }
21   gt_put_node(nodegrp, ptr, thisnode);
22 }
23
24 void compute_cost(gt_nodegrp_t nodegrp,
25                   gt_nodeptr_t *root) {
26   gt_visitor_t costvisitor;
27
28   costvisitor = gt_register_visit(cost_visitor);
29
30   gt_traversal_bottomup(nodegrp, root, costvisitor);
31 }
```

Listing 2: Sample bottom-up tree traversal.

C. Data Consistency

Similar to UPC [13], GT provides both *strict* and *relaxed* consistency modes when accessing global data. The consistency mode is applied to a phase of computation and related to all data in a GT node group. Strict mode performs reads and writes immediately, without caching or buffering. This mode can be used to check algorithm correctness or when performance is not critical. Relaxed mode has weaker guarantees on write completion and synchronization, which permits optimizations that enable more efficient and scalable communication. Cache coherence is managed by the application. Data completion in relaxed mode is guaranteed by using either barrier or fence operations. While explicit consistency management may add some programmer burden, there is no coherence overhead for data-independent decompositions.

D. Customizing Node Allocation

GT allows the programmer to override the default allocation strategy with custom allocators which can take advantage of *a priori* knowledge of the data access pattern to improve locality. Custom allocators can keep private state in between calls to the allocator. Additionally the GT allocation routine, which wraps the custom allocator, takes a *hint* parameter which allows the programmer to provide

some context from the allocation site. Custom allocation is particular to a specific node group. Multiple node groups may each use distinct allocation strategies.

For example, the default node allocator uses a policy called *local open*. The local open allocator keeps references to the "current chunk" in its private internal state. When a new allocation is requested, the node is allocated from the current chunk. If the chunk is full, a new chunk is allocated and references updated in the private state. Hint information is not used with this strategy.

Depending on the importance of node placement for later computation, the allocator could be fairly sophisticated and do allocation from remote chunks, or allocate from the best candidate from locally kept pool of chunks. Similar to the built-in traversals, it is likely that a variety of different applications may benefit from similar allocation strategies. GT could be readily extended to support depth-first allocation, level-wise allocation, etc. to match placement with data access.

IV. Design and Implementation

We split the design of GT into two components: the Global Chunks Layer (GCL) which provides a DSM-like framework for managing chunks of nodes and the Global Trees (GT) interface which builds higher level abstractions and optimizations on top of the GCL. The two key data structures provided by the GCL used to implement Global Trees are chunks and global *chunk/node pointers*. As shown in Figure 2, a chunk/node pointer (or "global pointer") is a portable, global reference to an element within a chunk. In the basic form, a chunk/node pointer is a tuple consisting of a globally unique chunk index and the offset of a particular element within that chunk.

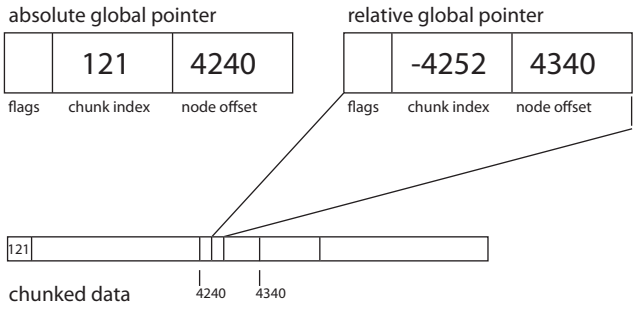


Fig. 2: Absolute and relative global pointers: An absolute pointer is a portable reference that consists of a chunk index and node offset. Relative pointers are lightweight intra-chunk pointers. A negative chunk index gives the displacement to the beginning of the chunk and an offset give the displacement from the beginning to the pointed-to node.

Each chunk in a Global Trees program is owned by a single process. Chunk locations and metadata are maintained by a distributed directory structure. In the present implementation, chunks do not migrate so directory communication is minimal and infrequent. However, we anticipate that in future implementations chunk migration will be an important feature to manage imbalance in the data distribution for evolving computations.

A. Caching and Buffering

Frequently, an application will not require strict consistency for correct operation of during its computation phases. To improve the performance when weaker consistency is sufficient, GT implements caching and buffering to reduce communication overhead and increase communication/computation overlap.

When an application is in the relaxed consistency mode, each process stores entire chunks in a local cache. This allows subsequent requests for remote nodes to be served out of cache without communication. Node updates issued during this phase may not be seen by all processes without invoking an explicit synchronization operation.

While the chunk cache on a process reduces communication when reading, Global Trees also performs write buffering as well. Multiple *put* operations will be buffered and the updates batched and sent using one-sided a vector put operation when the buffer is full, often resulting in a single bulk communication operation. Buffered writes update the chunk cache of the local process, so that local updates are seen immediately by the application, in program order.

B. Relative Global Pointers

Many applications will spend a great deal of time following links between nodes. Even when caching is used and chunks have good spatial locality, global pointer dereferencing is still considerably more expensive than a standard pointer dereference by a factor of 10-20. By adapting the chunk index field of a global pointer we can reduce the cost of a global dereference to approximately 1.5 times the cost of a C pointer dereference for references within the same chunk.

Consider the chunk in Figure 2, where the tree node at offset 4240 contains a relative global pointer to a child node located at offset 4340. The relative status of this pointer is signified by a negative value in the chunk index. The value that is contained in the index is the offset from the location of the relative pointer itself to the beginning of the chunk. The address of the referenced node is computed by adding the address of the pointer (4252), the negative chunk offset (-4252), and the node offset (4340). Since

relative pointers may exist when caching is disabled, they must be valid when the dereferencing processor does not have a copy of the entire chunk. The use of relative pointers is automatic and opaque to the programmer regardless of the consistency mode used (i.e. with both caching and non-caching reads).

The use of relative pointers improves the dereference operation by avoiding the directory lookup and allowing the target address to be computed directly by two integer additions. Relative pointers are only valid in the context of a link pointer in a chunked node. When relative pointers are copied, they are converted to the more portable absolute form. When either an absolute or relative pointer is used as the target address of a linking operation, GT will attempt to use a relative pointer, if possible.

C. Custom Allocation

The performance benefits of caching can be improved by reducing the miss rate. Because different applications may traverse the linked data structures with a variety of access patterns, no single chunking policy will be optimal for all applications. For example, some trees may be traversed in a depth-first manner, breadth-first, level-wise, etc. Global Trees provides a mechanism for application writers to customize node allocation to improve locality when operating on cached chunks. The default allocator designates a new node from an unfilled chunk on the local process. This is sufficient in the case that the data access pattern roughly follows the node creation pattern, but performance critical usage may require a custom allocator.

Custom allocation is handled by implementing a set of callbacks which perform initialization, finalization, and element allocation. Allocators are able to access private internal state kept between allocations, and are passed a global pointer in the allocation request to hint an optimal allocation. The internal state can maintain open chunks to allocate from, or affinity information of other unfilled chunks on other processes, etc. The hint provides some context from the allocation site. Together, this information can help improve data locality and increase performance.

For example, consider the computational kernel for an application which operates level-wise over a tree, but where creation occurs in a top-down manner. A custom allocator could keep several open chunks corresponding to various tree levels and would allocate a new node from the best match given the allocation hint provided.

D. Bulit-in Traversals

The built-in tree traversals provided by Global Trees are implemented by a decomposition into a task parallel form which is then run with the Scioto load balancing system. The initial task distribution is done in a locality-sensitive manner, with tasks seeded on processors which own the subtrees that are being traversed. At each node in the traversal order, GT invokes a callback into the application which then computes on the visited node and returns a flag to the runtime indicating that the possibly modified node should be either updated or discarded.

Global trees currently provides a pre-ordered top-down traversal, a post-ordered bottom-up traversal, as well as a level-wise traversal. Tree traversals may operate on either an entire tree or a subtree.

V. Tuning Application Performance

A. Performance Characterization Information

The GT runtime system can provide application programmers with insights into the communication and locality characteristics of their program. When configured with profiling extensions, Global Trees can provide detailed information about the quantity and time spent on communication. Data is gathered which tracks the number of get and put operations, how many of these were for local or remote data, etc. Cache performance information is also provided, including the number of chunk transfers that occurred as well the number of batched writes.

Global Trees can also be configured to keep statistics on the efficiency of the chunk packing strategy. Chunk utilization can be used to determine if the allocator is doing an adequate job, and can give valuable feedback to custom allocator authors. Chunk utilization data is presented in Figure 5a for the Barnes-Hut application.

B. Empirical Chunk Size Selection

The range of acceptable chunk sizes is determined by the spatial locality of the application, the effectiveness of mapping spatially related data to a single chunk, and communication efficiency. An extremely efficient communication subsystem could support very small chunk sizes. Likewise, an application with extremely high locality can exploit very large chunk sizes or an application with a very irregular access pattern would perform best with a small chunk size since it exhibits low locality.

As shown in our experimental results, we demonstrate that selecting a good chunk size can be done by combining application specific chunk access statistics with performance characterization data from the execution environment. Running the application once with extended profiling data enabled can provide information on the number of nodes accessed in each chunk, which can be used as a chunk packing efficiency metric. By sweeping over a range

of chunk sizes, this information can model application data access.

Using a small benchmark program, it is possible to model the bandwidth characteristics of the computing environment that will execute the application. As demonstrated by our experiments, composing the bandwidth data with the packing efficiency yields a value for chunksize which should strike a balance between increasing bandwidth and decreasing packing efficiency as chunk size increases. Since better packing efficiency allows larger chunksizes and presumably higher bandwidth, custom node allocators may significantly improve performance for some applications.

VI. Experimental Evaluation

We present an experimental evaluation of the GT system using the Barnes-Hut n-body gravitational simulation as an example. We selected the existing shared memory implementation of the Barnes-Hut benchmark from the Stanford SPLASH-2 benchmarking suite [14] as a basis for our comparison. For this application, we compare the performance of a GT implementation with the performance achieved using Cluster OpenMP [17], a commercial software distributed shared memory platform and explore GT's performance and design space through several microbenchmarks. The GT version required ten lines of code to adapt the force calculation kernel from the original SPLASH-2 codes, and the overall program modifications are limited to a few hundred lines of code, including instrumentation.

We conducted our experiments on a commodity cluster located at OSU. This cluster consists of quad-core 2.33GHz Intel Xeon nodes, each running 64-bit Linux and configured with 6GB of RAM. The cluster is connected via a 10GBps Infiniband interconnect network.

A. Barnes-Hut

The Barnes-Hut algorithm solves the *n-body problem*, which computes the gravitational force interactions between n bodies within a given region of space. A global tree structure represents a recursive spatial decomposition of a three-dimensional region containing the particles of interest. The algorithm has two principal components per iteration: tree construction and the force computation.

Initially, the position and mass of each body is determined by a known model and the bodies are distributed among processors. Each process iterates through its list of bodies and inserts them into the spatial decomposition tree. If there are too many bodies in a given cell, the cell is subdivided and the bodies inserted into the correct child at the next level down. Once all bodies have been inserted into the tree structure, a bottom-up traversal is performed to accumulate the centers of mass for each higher-level cell.

The particles are partitioned by a Morton-ordered tree traversal, which helps place spatially close bodies onto the same processor. Each process then iterates over its local bodies and performs a partial traversal of the tree to evaluate the force contributions from each body. If a distance threshold is met, the force contribution from many bodies in a cell may be approximated by the cell's center of mass. This phase consists of n independent tasks which may operate on cached tree nodes. Once the forces have been computed, updated position and mass values are stored into the global space.

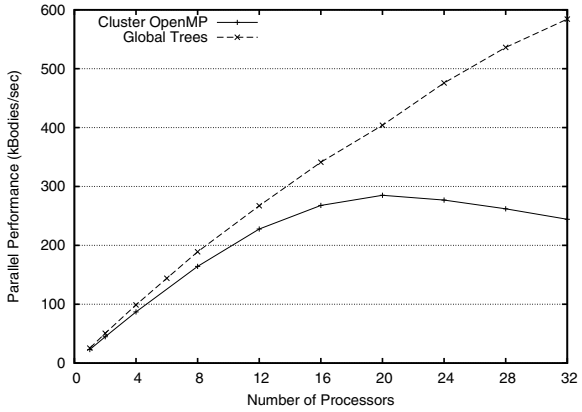
B. Cluster OpenMP

Cluster OpenMP is a commercial product distributed by Intel that provides support for OpenMP parallel programs on distributed memory systems [17]. OpenMP is a parallel annotation language that allows the programmer to annotate their source code with directives that enable parallel execution [18]. Intel's cluster OpenMP is built on top of the Intel C++ compiler and the TreadMarks software distributed shared memory (DSM) system [19].

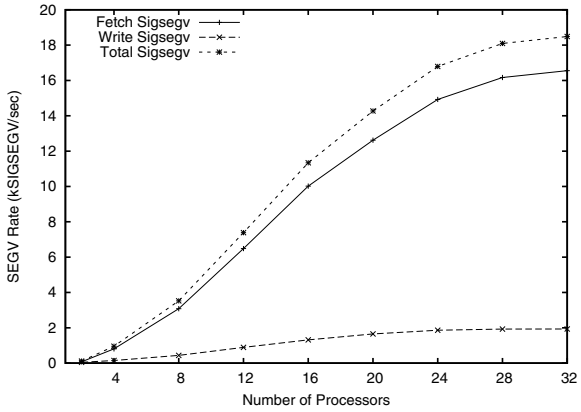
The OpenMP parallel model defines a flat, shared address space. This is supported under Cluster OpenMP by splitting the heap into private and shared portions. The private heap is only accessible by the local process and the shared heap is managed by TreadMarks which uses software DSM techniques to maintain consistency across all processes. Cluster OpenMP introduces additional markup that must be used to denote static variables as sharable and it also introduces API functions that can be used to perform dynamic allocation from the shared heap.

C. Performance Analysis

Figure 3a shows a parallel performance comparison between the Global Trees and Cluster OpenMP implementations of Barnes-Hut. For this experiment, we have focused on the performance of the force calculation kernel of Barnes-Hut that performs the n-body computation. From this data, we see that the Global Trees implementation achieves good performance up to 32 processors. The Cluster OpenMP implementation scales well to 16 processors, however it begins to slow down past 20 processors. This slowdown is due to coherence traffic generated by Cluster OpenMP's software DSM system, as shown in Figure 3b. In this figure, we show the rate at which coherence events (e.g. SIGSEGV signals) were triggered versus the number of processors. CLOMP's DSM takes advantage of existing memory protection mechanisms to mark newly fetched pages as read only and invalidated pages as inaccessible.



(a) Parallel performance of Barnes-Hut



(b) Cluster OpenMP coherence traffic vs. number of processors

Fig. 3: Left: Parallel performance of Barnes-Hut in thousands of bodies processed per second on 32 cluster nodes for Cluster OpenMP and Global Trees implementations on a 512k body problem with a chunk size of 256. Right: Rate in thousands of events per second at which coherence events occur for this experiment broken down into *fetch* and *write* events.

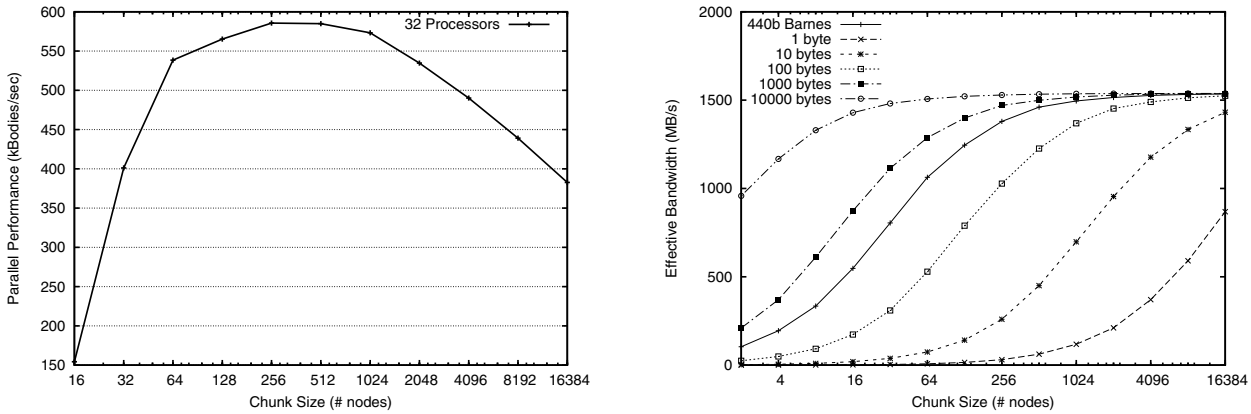
It then intercepts SIGSEGV signals generated by the user's code when these protections are violated to perform necessary coherence operations and satisfy the memory request.

From the data in Figure 3b we can see that the rate at which coherence events are triggered during the force calculation kernel increases rapidly with the processor count even for the fixed 512k body problem size. However, around 28 processors, the rate begins to flatten, matching the corresponding flattening of performance seen in Figure 3a. This coherence traffic is generated in response to concurrent reads and writes to the list of bodies. When processing a body during force computation, all other bodies must be examined to accumulate their gravitational force contributions into the current body's state in the next time step. There is no true data dependence between these reads and writes, however because both the t and $t+1$ states are stored in the same structure every update can potentially result in coherence traffic. In addition to this, under the SPLASH-2 implementation of Barnes-Hut all bodies are stored in a contiguous array causing multiple bodies to lie within the same shared page in memory and further exacerbating the false sharing problem. In comparison, Global Trees is able to achieve lower overhead through relaxed, user-controlled coherence and by exploiting high level knowledge of the tree data structure.

Compared with traditional DSM systems where the unit of sharing is an OS-level page, the unit of data transfer in GT is a chunk. While the size of a page is often fixed for a given system, under GT the chunk size parameter can be tuned to match the locality of a given application with the characteristics of a given machine. In Figure 4a we evaluate the performance of the GT Barnes-Hut code over a range of chunk sizes for a 32 processor execution of the 512k body data set. From this data, we see that a chunk size of roughly 256 yields peak performance and that chunk sizes in the range 64-2048 yield performance that is within 90% of peak. For chunk sizes smaller than this range performance falls off rapidly due excess communication overhead. For chunk sizes larger than this range, performance also decreases as locality within a chunk decreases and transfer time increases. Obtaining good performance therefore requires a judicious choice of chunk size that balances communication efficiency with locality.

In Figure 4b we present a performance characterization of the communication subsystem on the test cluster with respect to chunk transfers. Here we show the effective bandwidth achieved with respect to the volume of data transferred, or the product of the chunk size and the node size. Separate traces are shown for six different node sizes. For each node size we measured the communication bandwidth achieved over a range of chunk sizes. In general, the time required to send a message over a network is: $t = t_s + B * t_b$, where t_s refers to the startup overhead, t_b refers to the per-byte transfer cost, and B refers to the message length in bytes. For small transfers the startup cost dominates transfer time and for large transfers the bandwidth term dominates. This is apparent for the smallest node size, 1, where the bandwidth utilized is less that 40% even for the largest chunk size we evaluated. Likewise, for the largest node size in Figure 4b effective bandwidth starts high and reaches peak around a chunk size of 16. For the 440 Byte trace corresponding to Barnes-Hut, we see that the range of suitable chunk sizes from 64-2048 span the crest of the bandwidth curve.

Data locality is the second key factor in chunk size selection. In Figure 5a we quantify data locality for Barnes-



(a) Chunk Size vs. Performance

(b) Effective Communication Bandwidth

Fig. 4: Left: Parallel performance of force calculation in Global Trees Barnes-Hut over a range of chunk sizes. Right: Effective communication bandwidth as a function of chunk size and node size.

Hut as the percentage of nodes in a chunk that are accessed after a chunk has been fetched. A high percentage indicates a high degree of locality and a low percentage indicates that a smaller fraction of the data transferred in a chunk was used. This data shows that smaller chunk sizes have the highest percentage of nodes accessed while larger chunks have a lower percentage of chunks accessed.

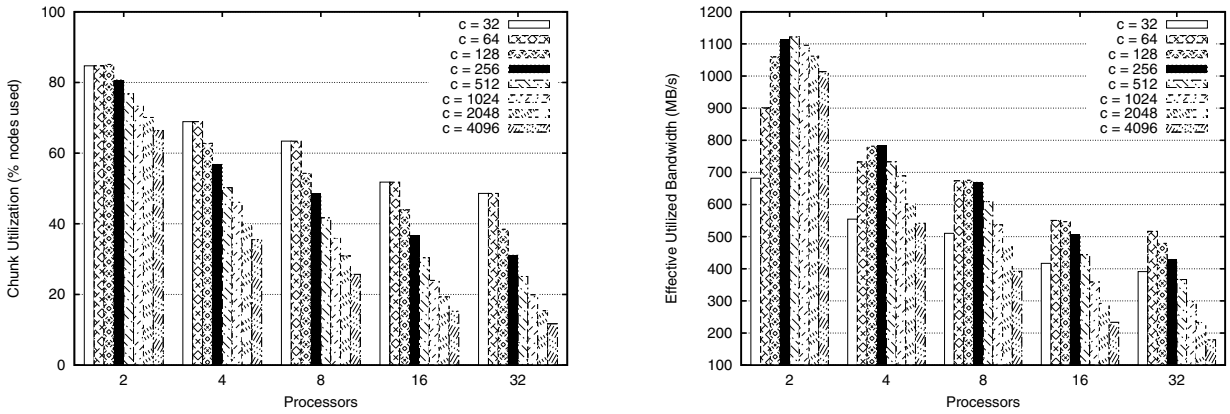
By finding the product of data in Figure 5a with the communication bandwidth data presented in Figure 4b we arrive at Figure 5b. Here we present the average fraction of communication bandwidth that was utilized by Barnes-Hut over a range of chunk sizes for different processor counts. This data summarizes the competing effects that bandwidth and locality have on chunk size selection. From this figure we see that chunk sizes in the range from 64-256 nodes yield good effective bandwidth utilization for processors counts of 4 and higher. This range of chunk sizes corresponds with the range that was found to yield peak performance in Figure 4a.

VII. Related Work

The topic of supporting linked, or pointer-based, data structures on distributed memory systems has been addressed in the literature through three mechanisms: Software Distributed Shared Memory (DSM), Distributed Shared Object (DSO) systems, and Partitioned Global Address Space (PGAS) models. In GT, we seek to combine strengths from all three systems: coarse granularity of data movement to exploit spatial locality and enhance communication efficiency; a global namespace for accessing shared data objects; and locality-aware, efficient one-sided remote access to shared data in the partitioned global address space.

Software DSM systems provide the illusion of shared memory in the presence of physically distributed data. These systems typically function by symmetrically mapping shared data into the address space of all processes and interacting with the memory management infrastructure to perform coherence operations and maintain data consistency. Examples of such systems include: TreadMarks [20], [21], [22], Cashmere [23], Beehive [24], Shasta [25] Blizzard [26], AURC [27], Cid [28], CRL [29], and Midway [30], [31], [32]. There are several key differences between DSM and the proposed GT system. Depending on the size of the coherence unit, typically a page, unrelated data elements may be grouped together in such systems, resulting in poor performance due to false sharing. In GT, the impact of false sharing is minimized through a tunable chunk size parameter and locality conscious chunk packing. Additionally, under GT's PGAS model, the address space grows linearly in the number of processors and is not constrained to the addressable space of a single processor. Finally, due to the nature of the shared memory interface that is provided by DSM systems, one is not able to leverage locality optimizations and locality-driven scheduling on such systems. GT also provides flexible consistency mechanisms allowing the user to exploit knowledge about data access patterns to further reduce communication overhead.

Distributed Shared Object (DSO) systems such as Linda [33], [34], Emerald [35], Charm++ [36], CHAOS++ [37], and Orca[38], [39] permit sharing of data objects through replication and object migration. They utilize a global object namespace to enable sharing of distributed objects and eliminate the need for implicit coherence through relaxed consistency semantics that may be managed by the user, compiler, or runtime system. GT also uses a global namespace to reference data objects that are dynamically created. However, GT differs fundamentally from these systems because it uses chunk based communication to enhance locality and improve



(a) Chunk Locality

(b) Effective Bandwidth Utilization

Fig. 5: Left: Chunk utilization reported as the number of nodes accessed in each chunk. Right: Effective bandwidth utilization reported as the product of effective bandwidth and chunk utilization.

communication efficiency.

PGAS parallel languages, including Co-Array Fortran [40], UPC [13], and Titanium [41] and one-sided communication systems including ARMCI [9], GASNET [10], SHMEM [12], and MPI one-sided operations[11] provide a partitioned global view of shared data. Under these models, a global pointer is the tuple $\langle p, a \rangle$ where p is a process ID and a is the address of a data element in that process' address space. These models provide efficient and locality-aware one-sided access to shared data in a global address space. The ideas that GT borrows from DSM and DSO, namely bulk data movement and global namespace, can greatly enhance the performance and communication efficiency of distributed linked structures under these models. GT is already compatible with CAF and other PGAS systems that utilize ARMCI. In future work we plan to investigate the impact of GT integration with other PGAS models.

Several systems have also been proposed to address the specific challenge of supporting linked data structures on distributed memory parallel computers: The Parallel Irregular Trees [42], [43] library provides an SPMD model for distributed tree computations and requires the use of stencils to define data dependencies. Olden [44] provides fine-grained access to pointer-based linked data structures on distributed memory clusters by migrating the computation (possibly many times) in response to accesses to non-local data. Olden provides fine-grained data access, but care must be taken with the data distribution to avoid excessive task migration.

VIII. Conclusion and Future Work

One of the major challenges of distributed memory parallel program development is that of reducing the programming effort required in achieving high performance. Several existing models allow programmers a global view of program data, but obtaining good performance by applying these general techniques is still elusive. We have developed the Global Trees system to provide a portable interface for shared-memory style programming on distributed tree data structures. GT provides a system for providing fine-grained data access with the efficiencies of coarse-grained data movement. Global Trees also takes advantage of properties specific to tree structures to provide extremely efficient global pointer dereferencing when traversing links. Additionally, Global Trees provides common tree operations that take into account the data distribution and irregular nature of the workload. Lastly, Global Trees gives a great deal of flexibility to programmers to both control performance-sensitive aspects of tree usage and use GT profiling data to increase understanding of program execution. We also present a model for tuning the Global Trees runtime based on both application and system characteristics.

We show that our approach is effective by modifying a shared-memory version of the Barnes-Hut algorithm to work with Global Trees. Our empirical results show that global address space programming can be made more efficient by employing these key features of GT.

We believe that this technique is applicable to generalized tree structures and in our future work, we plan to extend this system to other irregular linked data structures such as graphs. We also plan to further investigate strategies for enhancing locality within a chunk for a range of different access patterns. An ongoing direction of work is to examine the applicability of the GT system for XML data management, data space management, and data mining applications that heavily invest in the use of "global" tree based data structures [45].

References

[1] J. Barnes and P. Hut, "A Hierarchical O(N log N) Force Calculation Algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[2] J. Carrier, L. Greengard, and V. Rokhlin, "A Fast Adaptive Multipole Algorithm for Particle Simulations," *SIAM Journal of Scientific and Statistical Computing*, vol. 9, no. 4, 1988, yale University Technical Report, YALEU/DCS/RR-496 (1986).

[3] R. J. Harrison, G. I. Fann, T. Yanai, and G. Beylkin, "Multiresolution Quantum Chemistry in Multiwavelet Bases," in *International Conference on Computational Science*, 2003, pp. 103–110.

[4] G. Buehrer, S. Parthasarathy, and Y.-K. Chen, "Adaptive Parallel Graph Mining for CMP Architectures," *ICDM '06: Proceedings of the Sixth International Conference on Data Mining*, vol. 0, pp. 97–106, 2006.

[5] MPI Forum, "MPI: A Message-Passing Interface Standard," Tech. Rep. UT-CS-94-230, 1994.

[6] D. Callahan, B. Chamberlain, and H. Zima, "The Cascade High Productivity Language," *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on*, pp. 52–60, 26 April 2004.

[7] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing," in *OOPSLA*, R. Johnson and R. P. Gabriel, Eds. ACM, 2005, pp. 519–538.

[8] G. L. Steele Jr., "Parallel Programming and Parallel Abstractions in Fortress," in *IEEE PACT*. IEEE Computer Society, 2005, p. 157.

[9] J. Nieplocha and B. Carpenter, "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems," *Lecture Notes in Computer Science*, vol. 1586, 1999.

[10] D. Bonachea, "GASNet Specification, v1.1," U.C. Berkeley, Tech. Rep. UCB/CSD-02-1207, 2002.

[11] MPI Forum, "MPI-2: Extensions to the Message-Passing Interface," Technical Report, University of Tennessee, Knoxville, 1996.

[12] R. Bariuso and A. Knies, "SHMEM User's Guide," 1994.

[13] UPC Consortium, "UPC Language Specifications, v1.2," Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.

[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proceedings of the 22th International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, 1995, pp. 24–36.

[15] J. Dinan, S. Krishnamoorthy, B. Larkins, and J. Nieplocha, "Scioto: A Framework for Global-View Task Paralellism," *Proceedings of the 37th Intl. Conference on Parallel Processing*, 2008.

[16] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[17] Intel Corporation, "Cluster OpenMP User's Guide v9.1," no. 309096-002 US, 2005-2006.

[18] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," vol. 5, no. 1, pp. 46–55, Jan./Mar. 1998.

[19] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *IEEE Computer*, vol. 29, no. 2, pp. 18–28, 1996.

[20] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *Proc. of the 18th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, May 1992, pp. 13–21.

[21] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *Proc. of the Winter 1994 USENIX Conference*, Jan. 1994, pp. 115–131.

[22] P. Keleher, "Lazy Release Consistency for Distributed Shared Memory," Ph.D. dissertation, Department of Computer Science, Rice University, Jan. 1995.

[23] R. J. Stets, D. Chen, S. Dwarkadas, N. Hardavellas, G. C. Hunt, L. Kontothanassis, G. Magklis, S. Parthasarathy, U. Rencuzogullari, and M. L. Scott, "The Implementation of Cashmere," Tech. Rep. TR723, 1999.

[24] A. Singla and U. Ramachandran, "The Beehive Cluster System."

[25] D. J. Scales and K. Gharachorloo, "Design and performance of the Shasta distributed shared memory protocol," in *ICS '97: Proceedings of the 11th international conference on Supercomputing*. New York, NY, USA: ACM Press, 1997, pp. 245–252.

[26] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain access control for distributed shared memory," vol. 29, no. 11, pp. 297–306, Nov. 1994.

[27] L. Iftode, M. A. Blumrich, C. Dubnicki, D. L. Oppenheimer, J. P. Singh, and K. Li, "Shared virtual memory with automatic update support," in *International Conference on Supercomputing*, 1999, pp. 175–183.

[28] R. S. Nikhil, "Cid: A Parallel, "Shared-Memory" C for Distributed-Memory Machines," in *LCPC '94: Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*. London, UK: Springer-Verlag, 1995, pp. 376–390.

[29] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach, "CRL: High-Performance All-Software Distributed Shared Memory," in *Proc. of the Fifth Workshop on Scalable Shared Memory Multiprocessors*, Jun. 1995.

[30] B. N. Bershad and M. J. Zekauskas, "Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors," School of Computer Science, Carnegie-Mellon University, Tech. Rep. CMU-CS-91-170, Sep. 1991.

[31] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The Midway Distributed Shared Memory System," in *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, Feb. 1993, pp. 528–537.

[32] B. N. Bershad, "Practical Considerations for Non-Blocking Concurrent Objects," in *Proc. of the 13th Int'l Conf. on Distributed Computing Systems (ICDCS-13)*, May 1993, pp. 264–273.

[33] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *IEEE Computer*, vol. 19, no. 8, pp. 26–34, Aug. 1986.

[34] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *ACM Trans. on Computer Systems*, vol. 4, no. 2, pp. 110–129, May 1986.

[35] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System," *ACM Trans. on Computer Systems*, vol. 6, no. 1, pp. 109–133, Feb. 1988.

[36] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[37] C. Chang, A. Sussman, and J. Saltz, *Parallel Programming Using C++*, P. Lu and G. V. Wilson, Eds. Cambridge, MA, USA: MIT Press, 1996.

[38] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Experience with Distributed Programming in Orca," in *Proc. of the 1990 Int'l Conf. on Computer Languages*, Mar. 1990, pp. 79–89.

[39] H. E. Bal, A. S. Tanenbaum, and M. F. Kaashoek, "Orca: A Language for Distributed Programming," *ACM SIGPLAN Notices*, vol. 25, no. 5, pp. 17–24, May 1990.

[40] R. Numrich and J. Reid, "Co-Array Fortran for parallel programming," *ACM Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[41] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance Java dialect," in *ACM 1998 Workshop on Java for High-Performance Network Computing*, ACM, Ed. New York, NY 10036, USA: ACM Press, 1998.

[42] F. Baiardi, P. Mori, and L. Ricci, "PIT: A Library for the Parallelization of Irregular Problems," in *PARA 2002: Proceedings from the Applied Parallel Computing, Advanced Scientific Computing: 6th International Conference*. London, UK: Springer-Verlag, 2002, p. 185.

[43] ——, "Solving irregular problems through parallel irregular trees." in *Parallel and Distributed Computing and Networks*, 2005, pp. 246–251.

[44] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting dynamic data structures on distributed-memory machines," *ACM*

Transactions on Programming Languages and Systems, vol. 17, no. 2, pp. 233–263, March 1995.

[45] S. Parthasarathy, M. J. Zaki, and W. Li, "Memory placement techniques for parallel association mining," in *In 4th Intl. Conf. Knowledge Discovery and Data Mining*, 1998.

Appendix A: Global Trees Programming Interface

```
void gt_init(int *argc, char ***argv);
void gt_finalize(void);
void gt_abort(void);

gt_group_t gt_group_init(size_t chunksize, size_t nodesize,
     int nlinks);
void gt_group_destroy(gt_group_t nodegrp);

void gt_set_inactive(gt_nodeptr_t ptr);
int gt_is_active(gt_nodeptr_t ptr);
void gt_nodeptr_copy(gt_nodeptr_t from, gt_nodeptr_t to);
void gt_nodeptr_equals(gt_nodeptr_t p1, gt_nodeptr_t p2);

gt_node_t *gt_get_node(gt_group_t nodegrp, gt_nodeptr_t ptr);
gt_node_t *gt_get_node_direct(gt_group_t nodegrp, gt_nodeptr_t
     ptr);
void gt_put_node(gt_group_t nodegrp, gt_nodeptr_t ptr,
     gt_node_t *node);
void gt_finish_node(gt_group_t nodegrp, gt_node_t *node);

gt_nodeptr_t gt_node_alloc(gt_group_t nodegrp, gt_nodeptr_t hint
     );
gt_nodeptr_t gt_node_alloc_all(gt_group_t nodegrp);

gt_nodeptr_t gt_get_parent(gt_group_t nodegrp, gt_node_t *node)
     ;
gt_nodeptr_t gt_get_child(gt_group_t nodegrp, gt_node_t *node,
     int childidx);
gt_nodeptr_t gt_get_link(gt_group_t nodegrp, gt_node_t *node,
     int linkidx);

void gt_link_child(gt_group_t nodegrp, gt_nodeptr_t from,
                gt_node_t *fromnode, int index, gt_nodeptr_t
                     to);
void gt_clear_child(gt_group_t nodegrp, gt_node_t *node, int
      childidx);
void gt_clear_children(gt_group_t nodegrp, gt_node_t *node);

void gt_link_parent(gt_group_t nodegrp, gt_nodeptr_t from,
     gt_node_t *fromnode, gt_nodeptr_t to);
void gt_clear_parent(gt_group_t nodegrp, gt_node_t *node);

typedef void (*gt_visit_node_t)(gt_group_t nodegrp, gt_nodeptr_t
     ptr);

gt_visitor_t gt_register_visit(gt_visit_node_t visitor);
void gt_traversal_topdown(gt_group_t nodegrp, gt_nodeptr_t
     root, gt_visitor_t visit);
void gt_traversal_bottomup(gt_group_t nodegrp, gt_nodeptr_t
     root, gt_visitor_t visit);
void gt_traversal_levelwise(gt_group_t nodegrp, gt_nodeptr_t
     root, gt_visitor_t pre, gt_visitor_t post);

void gt_enable_strict(gt_group_t nodegrp);
void gt_disable_strict(gt_group_t nodegrp);
void gt_flush_cache(gt_group_t nodegrp);
void gt_flush_node(gt_group_t nodegrp, gt_nodeptr_t ptr);
void gt_write_fence(gt_group_t nodegrp, int proc);
void gt_write_fence_all(gt_group_t nodegrp);
void gt_barrier(void);
```