Efficient Run-time Support For Global View Programming of Linked Data Structures on Distributed Memory Parallel Systems

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Darrell Brian Larkins, M.S.

Department of Computer Science and Engineering

The Ohio State University

2010

Dissertation Committee:

P. Sadayappan, Advisor
Atanas Rountev
Paul A. G. Sivilotti
ABSTRACT

Developing high-performance parallel applications that use linked data structures on distributed-memory clusters is challenging. Many scientific applications use algorithms based on linked data structures like trees and graphs. These structures are especially useful in representing relationships between data which may not be known until runtime or may otherwise evolve during the course of a computation. Methods such as $n$-body simulation, Fast Multipole Methods (FMM), and multiresolution analysis all use trees to represent a fixed space populated by a dynamic distribution of elements. Other problem domains, such as data mining, use both trees and graphs to summarize large input datasets into a set of relationships that capture the information in a form that lends itself to efficient mining.

This dissertation first describes a runtime system that provides a programming interface to a global address space representation of generalized distributed linked data structures, while providing scalable performance on distributed memory computing systems. This system, the Global Chunk Layer (GCL), provides data access primitives at the element level, but takes advantage of coarse-grained data movement to enhance locality and improve communication efficiency. The key benefits of using the GCL system include efficient shared-memory style programming of distributed
dynamic, linked data structures, the abstraction and optimization of structural elements common to linked data, and the ability to customize many aspects of the runtime to tune application performance.

Additionally, this dissertation presents the design and implementation of a tree-specific system for 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 that operate on these structures. GT is built on top of the generalized data structure support provided by the GCL runtime and can inter-operate with other parallel programming models such as MPI, or along with existing global view approaches such as Global Arrays. This 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.

This dissertation also describes techniques for improving program performance and program understanding using these frameworks. Data locality has a significant impact on the communication properties of parallel algorithms. Techniques are presented that use profile-driven data reference traces to perform two types of data layout in Global Trees programs. Lastly, approaches for understanding and analyzing program performance data are presented along with tools for visualizing GT structures. These tools enable GT developers to better understand and optimize program performance.
To my wonderful wife, Lynette, who has fulfilled all my dreams.

To Rachel and Emily, two beautiful dreams that are still there when I wake.
ACKNOWLEDGMENTS

I am deeply grateful for the opportunity to study at The Ohio State University. Without the support and care of many people that I’ve met throughout my life, this endeavor wouldn’t have been possible.

First and foremost, I thank my advisor, P. Sadayappan. He has supported my growth and development throughout this entire process. As a mentor and role-model, I have learned from his example as an excellent researcher and teacher.

Other faculty at Ohio State have been influential in the way that I have come to think and reason about challenges that I face. Nasko Rountev, who has helped me to understand the discipline and process of research. Paul Sivilotti, who has forged many neural pathways regarding formal reasoning. Srini Parthasarathy, for your valuable insights about global view distributed programming. I especially thank Gerald Baumgartner, who helped repair the damage done by my undergraduate time at Ohio State.

I also wish to thank my parents, Darrell and Shirley, for encouraging my inquisitiveness and putting up with all my shenanigans. The encouragement of my sister, Cara, and my rabid Buckeye fan and brother-in-law, Joe, has also meant a great deal to me.

Having spent a decade in industry before returning to academia, there are many people whose nudging and listening have helped with this accomplishment. Thanks
to Joe Judge and Gary Ellison at AT&T Bell Laboratories, for seeing a diamond in the rough. Jim Hoburg for many years of intense conversations and showing me that critical thinking can be a lifestyle and not just a skill. I am grateful for all of the good friends and colleagues that I have met along the way: Chad Maue, Matt Curtin, Larry Ogrodnek, Kyri Sarantakos, Seth Robertson, James Tanis, Alex Dupuy, Brian Lindauer, Matt Miller and Oliver Stockhammer. Special thanks to Jen Yates, Scott Alexander and Martin Jansche for sharing their insights and experience with the doctoral process.

It’s difficult to imagine surviving this experience without the friends that I have made while in school. Matt Lang, your thoughts and humor have helped me preserve some sanity. Josh Levine and Jim Dinan, much thanks to each of you for both contributing to and distracting me from my progress with research. I would also like to thank my friends on the outside, Rick Martin, Melanie Fuller, and the many, many others who have aided me in keeping it all together for the last six years.

Lastly, to my dearest wife, Lynette, who has sacrificed much to permit me to pursue my dream. Without you, I would be incomplete, I love you. To my children, Rachel and Emily, who can’t read yet: I love you too and hope that one day you will understand why daddy spent so much time pushing buttons and looking at a glass screen with letters on it.

D. Brian Larkins
Columbus, Ohio
June 11, 2010
VITA

September 13, 1971 ................. Born – Cincinnati, OH

1996 .......................................... B.S. Computer Science, The Ohio State University

2008 .......................................... M.S. Computer Science, The Ohio State University

2005 — Present ......................... Graduate Research Associate, The Ohio State University

Research Publications


Instructional Publications

**FIELDS OF STUDY**

Major Field: Computer Science and Engineering

Studies in:

<table>
<thead>
<tr>
<th>Field</th>
<th>Professor</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Performance Parallel Systems</td>
<td>Prof. P. Sadayappan</td>
</tr>
<tr>
<td>Programming Languages and Software Engineering</td>
<td>Prof. Atanas Rountev</td>
</tr>
<tr>
<td>Distributed Systems</td>
<td>Prof. Paolo A.G. Sivilotti</td>
</tr>
<tr>
<td>Artificial Intelligence</td>
<td>Prof. Eric Fosler-Lussier</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter/Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
<tr>
<td>List of Program Listings</td>
<td>xv</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xvii</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Background</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview of Tree and Graph Structures</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Parallel Computation with Linked Structures</td>
<td>4</td>
</tr>
<tr>
<td>1.2.1 Data Models</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2 Control Models</td>
<td>10</td>
</tr>
<tr>
<td>1.3 Application survey of tree and graph algorithms used in HPC</td>
<td>11</td>
</tr>
<tr>
<td>2. Global Chunks: A Framework for General Linked Data Structures on Distributed Memory Parallel Systems</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Overview</td>
<td>14</td>
</tr>
<tr>
<td>2.2 Programming Model</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1 Data Models</td>
<td>16</td>
</tr>
<tr>
<td>2.2.2 Data Access Views</td>
<td>19</td>
</tr>
</tbody>
</table>

ix
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.3 Data Consistency Models</td>
<td>22</td>
</tr>
<tr>
<td>2.3 Programming Interface</td>
<td>27</td>
</tr>
<tr>
<td>2.3.1 Core Programming Constructs</td>
<td>27</td>
</tr>
<tr>
<td>2.3.2 Core Operations</td>
<td>29</td>
</tr>
<tr>
<td>2.3.3 Data Consistency and Coherence Management</td>
<td>34</td>
</tr>
<tr>
<td>2.3.4 Using Global Chunks</td>
<td>35</td>
</tr>
<tr>
<td>2.3.5 Customizing Node Allocation</td>
<td>38</td>
</tr>
<tr>
<td>2.4 Design and Implementation</td>
<td>39</td>
</tr>
<tr>
<td>2.4.1 Global Pointers</td>
<td>41</td>
</tr>
<tr>
<td>2.4.2 Caching and Buffering</td>
<td>44</td>
</tr>
<tr>
<td>2.4.3 Custom Allocation</td>
<td>48</td>
</tr>
<tr>
<td>2.4.4 Interoperability</td>
<td>49</td>
</tr>
<tr>
<td>2.5 Experimental Evaluation</td>
<td>50</td>
</tr>
<tr>
<td>2.5.1 Coarse-Grained Data Movement</td>
<td>50</td>
</tr>
<tr>
<td>2.5.2 Global Pointer Overhead</td>
<td>51</td>
</tr>
<tr>
<td>2.5.3 Chunk Global View</td>
<td>52</td>
</tr>
</tbody>
</table>

3. Global Trees: Specialized Support for Global View Tree Structures     | 54   |
| 3.1 Overview                                                           | 54   |
| 3.2 Programming Model and Interface                                    | 56   |
| 3.2.1 Core Programming Constructs                                      | 57   |
| 3.2.2 Programming Interface                                            | 58   |
| 3.2.3 Using Global Trees                                               | 63   |
| 3.2.4 Custom Node Allocation                                           | 65   |
| 3.2.5 Tree Traversals                                                 | 66   |
| 3.3 Experimental Evaluation                                            | 68   |
| 3.3.1 Cluster OpenMP                                                   | 69   |
| 3.3.2 Unified Parallel C                                               | 69   |
| 3.3.3 Global Trees Scalability                                         | 70   |
| 3.3.4 Barnes-Hut Performance Analysis                                  | 71   |

4. Data Locality in Global Trees                                        | 75   |
| 4.1 Overview                                                           | 75   |
| 4.1.1 Approaches                                                      | 76   |
| 4.1.2 Contributions                                                   | 78   |
| 4.2 Analysis                                                           | 79   |
| 4.2.1 Reducing Communication Time                                     | 80   |
| 4.2.2 Reducing Global Trees Overhead                                   | 82   |
| 4.2.3 Techniques                                                      | 83   |
| 4.3 Profile-guided Data Layout                                        | 85   |
4.3.1 Data Structures ........................................... 86
4.3.2 Profile Gathering ........................................ 87
4.3.3 Element Mapping Algorithm ............................. 88
4.3.4 Chunk Mapping .......................................... 89
4.3.5 Profile-driven Allocation in Iterative Applications .... 93
4.3.6 Collective Data Relayout ................................ 94
4.4 Data Relayout Implementation ............................. 95
  4.4.1 Path Identifiers ........................................ 95
  4.4.2 Chunk Reservation ..................................... 97
  4.4.3 Supporting Remote Allocation ......................... 99
  4.4.4 Collective and Iterative Data Relayout ................ 103
4.5 Empirical Evaluation ..................................... 103
  4.5.1 Remote Node Allocation Overhead .................... 104

5. Understanding Global Trees Program Performance and Behavior .... 113
  5.1 Overview .................................................. 113
  5.2 Collected Runtime Performance Data ..................... 114
    5.2.1 GCL/GT Memory Usage .............................. 114
    5.2.2 Data Access Statistics ............................ 115
    5.2.3 Chunk Packing Efficiency .......................... 117
  5.3 Interpreting Statistics .................................. 118
    5.3.1 Element-to-Chunk Mapping ......................... 118
    5.3.2 Data Distribution ................................ 121
    5.3.3 System Performance ............................... 122
  5.4 Data Visualization ..................................... 123
    5.4.1 Owner-Labeled Global Tree Visualization .......... 124
    5.4.2 Profile-Labeled Global Tree Visualization .......... 124

6. Related Work .............................................. 128
  6.1 Programming Models for Global-View Distributed Data Structures 128
  6.2 Support for Parallel Linked Data Structures ............ 131
  6.3 Data Locality Improvement ................................ 132
  6.4 Performance Analysis and Improvement ................... 134

7. Future Work ............................................... 136
  7.1 Global Chunk Layer ...................................... 136
  7.2 Locality-Aware Load Balancing .......................... 137
  7.3 Hierarchical Run-time System Support ................... 138
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>2-D Spatial Decomposition for Barnes-Hut</td>
<td>6</td>
</tr>
<tr>
<td>1.2</td>
<td>Quadtree Representation of a Non-Uniform Space</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>Global Chunks Programming Environment</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Relationships between the core GCL programming constructs</td>
<td>27</td>
</tr>
<tr>
<td>2.3</td>
<td>Memory layout of GCL related structures on a single process</td>
<td>39</td>
</tr>
<tr>
<td>2.4</td>
<td>Data Layout of Linked Elements</td>
<td>42</td>
</tr>
<tr>
<td>2.5</td>
<td>Absolute and relative global pointers</td>
<td>42</td>
</tr>
<tr>
<td>2.6</td>
<td>Impact of chunking on element data access</td>
<td>51</td>
</tr>
<tr>
<td>2.7</td>
<td>Global pointer overhead relative to computational workload</td>
<td>52</td>
</tr>
<tr>
<td>2.8</td>
<td>Performance of Vector Scaling with Uniform Global and Chunk Global Views</td>
<td>53</td>
</tr>
<tr>
<td>3.1</td>
<td>Global Tree and corresponding chunk structure</td>
<td>55</td>
</tr>
<tr>
<td>3.2</td>
<td>Scalability of GT with a vector scaling operation</td>
<td>70</td>
</tr>
<tr>
<td>3.3</td>
<td>Parallel performance of Barnes-Hut in thousands of bodies processed per second on 32 cluster nodes for Cluster OpenMP, UPC, and Global Trees implementations on a 512k body problem with a chunk size of 256</td>
<td>72</td>
</tr>
<tr>
<td>3.4</td>
<td>Cluster OpenMP coherence traffic vs. number of processors</td>
<td>73</td>
</tr>
</tbody>
</table>
3.5 Parallel performance of Global Trees Barnes-Hut over a range of chunk sizes. .............................................. 74
4.1 Allocation Costs by Allocation Method and Distance .............. 105
4.2 Depth-First Access Relative Performance .......................... 106
4.3 Depth-First Access Relative Pointer Usage .......................... 107
4.4 Depth-First Access Cache Misses .................................. 108
4.5 Similarity in Tree Structure over Simulation Iterations .......... 110
4.6 Barnes-Hut Relative Performance ................................. 111
4.7 Barnes-Hut Relative Pointer Usage ................................. 112
5.1 Effective Communication Bandwidth ............................... 119
5.2 Chunk Packing Efficiency in Barnes-Hut ............................ 120
5.3 Effective Bandwidth Utilization ................................. 121
5.4 Large Octree Visualization ....................................... 126
5.5 Owner-Labeled Global Tree ....................................... 127
5.6 Profile-Labeled Global Tree ....................................... 127
LIST OF PROGRAM LISTINGS

2.1 GCL Operations on Global State. ........................................... 29
2.2 GCL Operations on Element Groups. ................................. 29
2.3 GCL Operations on Global Pointers. ................................. 30
2.4 GCL Operations on Chunks. .............................................. 30
2.5 GCL Operations on Elements ............................................. 31
2.6 GCL Operations on Chunked Elements ......................... 33
2.7 GCL Consistency and Coherence Operations ...................... 35
2.8 Example: Searching a linked list structure. .................... 36
2.9 Example: Scaling all elements in a data structure. .......... 37
2.10 Typical Recursive Access Structure in GCL .................... 47
3.1 GT Operations on Global State ........................................ 58
3.2 GT Operations on Node Groups ....................................... 59
3.3 GT Operations on Global Pointers ................................... 60
3.4 GT Operations on Tree Nodes ......................................... 60
3.5 GT Operations to Allocate Nodes ..................................... 62
3.6 Example: Copying a Global Tree. .................................. 65
3.7 Example: Bottom-up tree traversal. ............................... 67
4.1 Example: Conventional Child Allocation ....................... 99
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Global Pointer Dereferencing Algorithm</td>
<td>45</td>
</tr>
<tr>
<td>2 Profile-driven Element Mapping</td>
<td>90</td>
</tr>
</tbody>
</table>
CHAPTER 1

Background

1.1 Overview of Tree and Graph Structures

Advances in technology have enabled us to collect, record and manage data rather efficiently. Large data stores abound and are growing. For example, scientific data from observations, experiments or via simulation are easily exceeding peta-byte scales. As of 2006, Google’s BigTable file system was storing over 800 terabytes of data just for Internet web crawling alone [26]. The need to process, visualize and analyze such data efficiently is an important consideration in a wide range of application domains.

The information represented by such data can often be represented in the form of trees or graphs in a very concise and meaningful fashion. Examples abound, ranging from gene expression networks to social networks, from XML datasets to access patterns on the world wide web. Similarly a large number of data intensive applications often rely on complex, dynamic, tree- or graph-based data structures to house important meta-data information. Examples here include data mining, visualization, numerical and scientific algorithms.

The compute and data-intensive nature of such applications and the scale of the data involved necessitates the use of large-scale modern parallel computers. Although
there have been dramatic strides in hardware performance of modern high-end systems, these improvements have been accompanied by a corresponding increase in the complexity of these systems - modern parallel computers have increasingly large numbers of processors, multiple levels of parallelism (multi-core, SMP, cluster), and deeper memory hierarchies. Therefore, efficient deployment of such applications on modern and emerging clusters is challenging.

Achieving high performance on such systems requires both a balanced workload within the parallel structure of the computation to best utilize the available processors, and the exploitation of data locality to minimize data access times. The distribution of the data amongst the processors needs to take into account the cost of data movement between the processors. In addition, movement of data between the different levels of the memory hierarchy needs to be minimized in order to reduce the overall data movement cost. This is usually achieved by adapting applications and runtime systems to be aware of the different levels of the memory hierarchy that incur different data movement costs.

From a programmer’s viewpoint, the complexity of the code required to implement a given algorithm or simulation is a function of the level of detail that the programming model exposes to the programmer, the number of decisions and choices to be made, together with the level of detail required to manage performance-related aspects of the underlying hardware. Currently, the dominant parallel programming model is message passing using MPI. MPI programming is tedious and difficult to debug because of the explicitly partitioned view of the program’s data among the
processes [45]. In addition to the fundamental data structures of an algorithm, additional meta-data structures must be maintained by the programmer to hold information about where the different parts of the data are located, and explicitly orchestrate the movement of the data between processes. With out-of-core algorithms, another level of complexity is added, in maintaining which data is in memory versus on disk. An added complexity is the identification of the most efficient realization in a multi-level cluster, where individual nodes are multi-core processors and interconnection bandwidth access to remote memory is faster than access to local disk.

Parallel programming models provide a combination of data and control abstractions. The data abstraction divides the addressable memory in the system into units of locality. All elements in the same unit of locality are assumed to incur the same data movement cost. A simplified data abstraction that improves productivity ideally presents a uniform view of the memory in the system. On the other hand, achieving good scalability requires a data abstraction that can be efficiently mapped to the non-uniform deep memory hierarchy in modern parallel systems. This is typically achieved by creating one unit of locality for each processor memory, and distinguishing between access to the data in local memory and to that in non-local memory.

The control abstraction provides mechanisms to identify and express the parallelism in the computation. It also optionally provides mechanisms for automatic load balancing. The abstraction is either computation-centric, with parallelism specified in terms of data or functions used in the computation, or architecture-centric. In an architecture-centric control abstraction a fixed number of control flow units are defined, typically one per processor, and these are allowed to execute in parallel with user managing the synchronization between them. While an architecture-centric control
abstraction can be easily mapped to the processors in a parallel system, computation-centric abstractions facilitate ease of programming. In addition, architecture-centric specification of parallelism encourages the programmer to partition the computation with implicit dependences between the tasks in a part. The absence of this information to the runtime makes automatic support for load-balancing a challenge.

The middleware frameworks described herein have been developed to reconcile the seemingly conflicting requirements of performance and productivity for computations involving linked structures such as trees and graphs. These frameworks take a data-structure centric view of shared data where graph-based dynamic data structures such as trees, for example, drive the rest of the system. A key feature of these systems is to allow the programmer to have multiple views of the shared data as well as multiple views of the control and tasking model.

This flexibility of data and control abstractions can be leveraged to varying degrees depending on whether the goal is to develop a quick prototype for validating ideas on small scale problems or the objective is an efficient realization of large scale problems or somewhere in between these extremes.

1.2 Parallel Computation with Linked Structures

Applications using shared linked data structures such as trees and graphs are parallelized with respect to a data model and also a control model. A data model defines the means for data distribution as well as the mechanisms used to access non-local data. The control model similarly describes the process by which work is mapped to physical processors and how this mapping changes over the entire execution of the program.
1.2.1 Data Models

Data Distribution

When developing an application that relies on a shared linked data structure, the designer must consider where this shared data should be placed and how non-local access happens. The placement of the shared data is governed by the data distribution. Data distribution is done with either static or dynamic techniques.

A static data distribution is defined by a set of a priori conditions that map some unique feature of the data to a particular location. Static distribution can be as simple as having a master process that stores the entire shared structure and all other processes replicate the data. Alternatively, each process could “own” a specific portion of the problem space and would be responsible for storing all shared data corresponding to this part of the problem space. Because the criteria for locating data is statically defined, it must be known a priori and the resolution mechanism must be universal and deterministic. One primary advantage of static techniques is that the distribution is relatively simple and requires low overhead to implement in actual applications. On the other hand, because the distribution is defined in advance, the application may not be able to effectively adapt to changes in the data structure as the computation evolves.

Consider the Barnes-Hut technique for solving the \( n \)-body problem. Given a three-dimensional space and a collection of \( n \) particles, Barnes-Hut builds an octree in which the root node represents the entire space and its children each represent a single octant of the same space. The two-dimensional problem is simpler and demonstrates the spatial decomposition of a planar space into a quadtree, as shown in Figure 1.1a and the tree in Figure ??.
The entire space is subdivided into quadrants, with each quadrant containing a number of bodies corresponding to the physical system under simulation. Each region is continually subdivided until only a fixed number particles reside in a given quadrant (in this example, 2 particles per quadrant). Each quadrant containing particles becomes a leaf in the tree, with interior nodes representing larger volumes of space. Thus, sparsely populated regions will be represented by a shallow branch of the tree and dense regions will be deeper. The computation performs an interaction computation of each particle which then alters the position and direction of each particle for the next iteration of the simulation.

For small values of \( n \), it is possible to statically partition the octree by creating a single data structure and replicating it on all processes. Larger values for \( n \), however, will result in a structure that exceeds the memory of a single machine and must be partitioned over multiple processors. Static partitioning could be done by having each process claim a specific region of space and manage the storage of all tree data.
representing the spatial location and contained particles of that region. Since the initial distribution of particles may be highly non-uniform, like the tree in Figure 1.2, static partitioning based on spatial location could result in a significant imbalance of data among processors. Alternatively, the particles could be partitioned into $p$ groups and assigned to processes by linearizing the octree using a space-filling curve. This approach would result in a twofold advantage over the basic spatial decomposition: first, processes would each contain a similarly sized part of the tree; second, the set of particles on a given process would be spatially close to one another (due to the linearization), which improves data locality.

A dynamic data distribution is one that may adaptively change the placement of data based on information known at runtime. Because linked structures are often used to represent dynamic and irregular data, it can be beneficial to re-evaluate data location with respect to the evolving state of the computation. When shared tree or graph structures are being created, data placement can depend on the placement of related elements, rather than a predetermined static map. Dynamic data distribution also allows data to migrate between processors over time and permits the data layout to be restructured during program execution.
The space-filling curve linearization used in the Barnes-Hut algorithm described above can create a tension between load balancing and optimizing communication when the distribution of particles is highly irregular [44]. Dynamic data distribution allows the data placement decision to depend on information available at runtime, such as current workload estimates or other metrics to heuristically quantify tree irregularity. Since these heuristics may be computationally intensive, the overhead of dynamic distribution techniques must be balanced with the benefit they provide to application performance.

**Data Communication**

Given a distributed data structure, a computation will typically require non-local information to complete. At a high level, an application may approach this by communicating the non-local data to the local process or alternatively, by moving the computation to the process where the data is located. Data communication is traditionally performed in either a shared memory style or via message passing. Applications may use either style, both, or none depending on program design and implementation.

Parallel applications written in the shared memory style rely on some form of a global namespace to uniquely identify data elements across a cluster. This global namespace represents a virtual shared memory in a compute cluster which is built on top of a message passing communication fabric. Data can be accessed with a globally unique identifier and a dereferencing operator. Some distributed shared memory systems create a single global address space and replicate it among all processes, whereas other approaches partition and distribute the shared memory. Replicating the shared heap space is very convenient for application developers but may result in poor performance due to problems such as false sharing. Additionally, scalability
is a problem with heap replication because the shared heap size cannot exceed the physical memory limit of the smallest participating process.

In the distributed case, globally addressed data is accessed by mapping the global identifier to a tuple which contains the owning process identifier as well as a pointer to the region of memory on that process. This approach is known as a Partitioned Global Address Space (PGAS) [90]. In some systems, dereferencing a global pointer is done by using a language-level operator, but most often is performed by function call. Global pointers may refer to arbitrary memory structures. In contrast to the majority of message passing techniques, dereferencing a shared pointer is usually an asynchronous (or one-sided) operation. This is beneficial because it reduces the need for synchronization and can take advantage of hardware supported remote DMA operations.

Asynchronous operations are defined with a specific consistency model which permits the system to balance programmer convenience with communication efficiency. Using global pointers to refer to specific tree or graph nodes makes parallelization of applications a much simpler task but performance will likely be poor if the system issues a communication to access each element. Explicitly increasing the granularity of data access can improve communication efficiency, but also tends to complicate the data access procedure for a program.

Message passing systems such as LAPI and MPI are often used to take more control over the communication and computational structure of a parallel program. With this finer level of control comes the burden of managing it. Tree and graph algorithms are typically expressed in terms of nodes and edges, often with a recursive
structure. Each algorithm must be altered to an efficient message passing form, which can be a daunting task.

1.2.2 Control Models

Program control can be expressed either in terms of the available compute resources or as the work remaining to be completed. Viewing the computation from a resource perspective, a fixed number of control units are provided with a program and allowed to execute in parallel. This is typically realized by mapping each control unit to a physical processor core. This direct resource/core mapping makes it straightforward to map a given computation to a particular parallel computing system, but is more difficult to program. Task decomposition, scheduling, synchronization, and balancing is all left as a burden to the programmer.

Alternatively, control can be modeled as a set of tasks which need to be completed before the program can terminate. Tasks can represent the computation required for some data or as a function defined by the program. Program decomposition into tasks makes it much easier for the developer to reason about control flow, but requires that either the program or a run-time system map tasks to available physical resources.

With a task-centric approach, computation can be mapped to processor cores either statically or dynamically. With a static task mapping, tasks are assigned to processors by a static, deterministic process. For example, a matrix multiplication may be broken into $p$ tasks, with one task assigned to each of $p$ processors. As with static data distribution, this makes the control model very simple, but may result in load imbalance with tasks that have a variable amount of computation. Similarly, dynamic task mapping uses runtime information to map available work to compute
resources. This can result in greater load balance, but results in greater overhead in spawning and executing tasks.

Dynamic mapping systems can perform load-balancing by executing tasks on underutilized resources. Because the physical resource that may execute a task is not necessarily the same resource on which the task was created or scheduled to execute, tasks must be migratable structures. Tasks may migrate at the behest of a busy node, at which point it may push tasks to a less busy node. Alternatively, idle processors may steal tasks from busy ones, using a technique called work stealing.

Balancing a program workload creates a tension with the goal to minimize data communication. By definition, irregular structures such as trees and graphs will often be unbalanced and the computational dependencies not known until runtime. This sets up a conflict between trying to keep program data collocated with the corresponding computation (minimizing data communication) with a desire to use all available program resources effectively (load balancing). Depending on the specific problem that is being solved, it may be more efficient to tolerate some load imbalance in order to avoid the communication overhead of migrating the task and/or corresponding data or vice versa.

1.3 Application survey of tree and graph algorithms used in HPC

Scientific Data Analysis:

Many scientific and engineering applications rely on tree- and graph- based data structures. Approximate solutions to \( n \)-Body problems rely on tree-like data structures to dynamically refine the granularity of computation to ensure the desired accuracy. Examples of applications from this class of algorithms include solutions to the
gravity problem such as the Barnes-Hut method [10] and the Fast Multipole Method (FMM) [21] used in the calculation of electrostatic forces. Similar approaches are employed in the numerical calculus domain [2]. Engineering applications also draw on similar spatial structures for computing material properties [94, 19]. Examples can also be drawn from biology where genetics applications employ graph-based data structures for performing genome analysis [34, 87].

Other scientific frameworks such as MADNESS [48], use spatial decompositions to represent the domain of highly irregular multi-dimensional functions which correspond to quantum chemistry models. These functions are represented by trees which use hierarchical structure to allow for efficient multiresolution analysis. The computations involved on these trees are immense and require the use of very large-scale systems in terms of both computation and memory.

**Clustering and Classification Algorithms:** Clustering is unsupervised learning often used by marketing departments to group similar types of customers. The goal of clustering is to partition the data points into groups or clusters such that the intra-cluster similarity is maximized while the inter-cluster similarity is minimized. In the basic algorithm (kMeans), the clusters are refined iteratively by assigning each data point to the cluster with closest center [51]. More sophisticated algorithms [73, 93] make use of complex trees such as $k$-dimensional (kd-) trees and cluster feature (cf-) trees. These trees and can potentially exceed the main memory and the disk capacity of a single node. From an algorithmic standpoint, parallelization of algorithms with such large meta structures is non-trivial in a non-shared environment. Other structures, such as R-trees, are often used to efficiently represent geo-spatial data and can
be used to answer queries relative to location [46].

**Frequent Pattern Mining Algorithms:** The goal in frequent pattern mining is to find groups of items, sequences, trees or graphs that co-occur frequently in the given data set. We will consider three algorithms for our purpose here. Consider the most popular frequent itemset mining algorithm FPGrowth [47]. The algorithm concisely represents the data set in the form of a prefix tree. Furthermore, it employs tree projections to prune the number of candidates at runtime. It has been shown that the pointer-based nature of the tree coupled with poor access locality limits the performance of this algorithm on modern architectures [42, 17, 18]. Previous research has demonstrated that the meta data structure (FP-tree) can easily exceed the capacity of a single node’s memory and disk [18]. Furthermore, efficiently parallelizing this algorithm is often hard due to its memory intensive and data dependent nature.
CHAPTER 2

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

2.1 Overview

This chapter describes a run-time framework which supports global view programming of linked data structures on distributed memory parallel systems. The run-time system has several components which work together to provide this support. This system is designed to support parallel applications which use pointer-based data structures, such as tree- and graph-based structures, that are globally shared and available to all processes active in a computation.

Since there is a large amount of commonality between these types of data structures, a lower-level substrate is used to abstract out functionality unique to working with irregular, linked data in general. This system is called Global Chunks and can be used as the basis for higher-level user libraries which support specific tree (e.g. Global Trees, described in Chapter 3) or graph structures.

The Global Chunk Layer (GCL) provides a base global namespace for sharing linked data structures and provides a rich set of operations to access and modify the shared data. This approach is based on the key insight that algorithms which rely on linked data structures are easily expressed in a fine-grained manner, but
data movement should occur at a coarser level of granularity for good performance. To permit fine-grained access with coarse-grained data movement, the GCL groups related data elements into *chunks* which can improve locality and communication efficiency.

As shown in Figure 2.1, the Global Chunk Layer is implemented on top of the ARMCI one-sided communication library[67]. ARMCI provides low-level support for the partitioned global address space and hardware-supported remote memory access. The Global Trees layer is an example of a higher-level library which 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.

The GCL also works in conjunction with the Scioto dynamic load balancing system to manage task parallelism exposed through parallel data structure traversals[35].
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 programming model and allows the user to express irregular and nested parallel computations through global tasks that are automatically load balanced across the cluster.

This section details the Global Chunk Layer and the contributions that it makes to providing efficient, generalized support for global view programming of data-structure specific applications. First, the programming model is described with respect to the traditional approaches described earlier. Second, the interface for this library is described, followed by the implementation. We then provide some experimental analysis of the runtime performance.

2.2 Programming Model

2.2.1 Data Models

In the context of a distributed computing cluster, fine-grained element-wise global access will yield poor performance. To mitigate this effect, all globally-addressed data structures are grouped into *chunks* which provide greater locality and therefore higher performance. The view of the data presented to the tasks in the computation need not necessarily match the data layout.

Programs and libraries which utilize the Global Chunk Layer are executed in an SPMD manner with MIMD task-parallel regions. The application may be structured to concurrently execute asynchronously, or these task-parallel regions may be managed by Scioto. When using Scioto, the application is structured to use portable thread-like tasks to operate on globally shared data. 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, the GCL represents individual data elements as collections called *chunks*. Similar to shared pages in distributed shared memory systems, chunks permit element-level fine-grained access, but enable the runtime to perform coarse-grained data movement in order to reduce communication overhead. The application can access data elements without any explicit interaction with the chunking subsystem. For applications which require greater performance than the transparent fine-grained mechanisms provide, the chunking layer can be exposed to the application, permitting greater control over data locality.

Chunks have the following properties:

**Contiguous:** A chunk consists of a single contiguous memory segment, all elements within a chunk reside within this segment.

**Global:** A chunk is globally accessible. A reference to a chunk on one process is portable and refers to the same chunk when used within another process.

**Ownership:** A chunk is physically local to only one process. Each chunk has a master location, which exists in the shared memory segment of one and only one process.

**Collections:** A chunk is a collection of user-defined elements. The GCL imposes some minor restrictions on data structure definition (specifically with regards
to connectivity information), but is otherwise defined by the application and is
treated as opaque by the runtime system.

**Granularity:** A chunk is the unit of data transfer. All elemental data accesses cor-
respond to an element located within a chunk. When operating in the default,
relaxed mode, access to a specific element which is not local will result in the
communication of an entire chunk.

Much like Global Arrays [69], the Global Chunk Layer uses a *get/compute/put*
model for accessing elements of the global data structure. Prior to operating on a
specific data element, a 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 updates are 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 GCL
synchronization operations.

Global Chunks works under the partitioned global address space model (PGAS),
in which all global data is partitioned among the participating processes. Global
data resident in the shared memory segment of 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, access, and updating of global data
structures, however, the GCL is interoperable with other message-passing and one-
sided communication systems, including MPI and ARMCI. In addition to the GCL
global namespace, these other systems may be used to communicate shared program
data.
2.2.2 Data Access Views

The GCL provides two data access views for using elements in a global linked data structure. These views differ in access modes and the granularity of access to the elements in the data structure:

The uniform global view provides the programmer with element-level access to individual nodes in the linked structure. This provides a simplified data abstraction to prototype applications, where chunking is transparent to the programmer.

The chunked global view requires the programmer to be aware of the organization of the data structure, encouraging explicit locality-aware organization of the computation in order to exploit the data locality inherent in the chunked data layout.

These different data views are inter-operable and the mixing of uniform global and chunked-global programming styles is permitted in order to provide a continuum between rapid development and optimized performance. For operations which are performance-sensitive, the programmer may use the lower-level methods at the expense of higher code-complexity. Conversely, when application development resources are at a premium, some performance may be traded for coding effort by using the uniform global view of the data.

Chunking happens in concert with the local virtual memory subsystem on each local processing element. This permits native support for out-of-core datasets. When the number of local chunks exceeds the size of core memory, chunks are automatically paged out by the virtual memory system. Likewise, when an out-of-core chunk is referenced, it will be paged in and made available for reading.
Uniform Global View

Referring to data that is shared amongst processors must be done in a portable manner. The GCL is targeted to pointer-based data structures, so any data element may contain references to other elements. Since these structures may be communicated to different processes, the use of traditional pointers is problematic. Uniform global access is supported through the use of special global pointers which may refer to nodes in a global address space. Global pointers consist of a reference to a chunk identifier and an offset within the chunk for the node which is referred to by the pointer. Uniform global access to data occurs explicitly using the API provided by Global Chunks.

The data elements used with the Global Chunk Layer are manipulated using the get, compute, and put model. The programmer explicitly creates a local copy of the nodes of interest using the get operation. The runtime system copies these nodes to the local processor’s memory, where direct pointer-based node access can be used for efficiency. The computation, including any updates, are performed on the local copy of the nodes. Any changes to the nodes are then updated in the global space using the put operation.

This model of computation is similar in spirit to the Global Arrays [68, 69] library. The get/compute/put model is more burdensome than direct pointer access, but is more similar in style to sequential or threaded programming than the traditional SPMD model used on distributed memory systems. The get/compute/put schema can be used in straightforward fashion around each global data reference, or can be used with greater regard for locality when optimizing data access in an application. For efficiency, chunks in the local process address space can also be directly accessed.
without a copy. Avoiding these unnecessary copies allows the GCL to yield performance that is closer to applications implemented using efficient distributed memory programming models over traditional distributed shared memory systems.

The elements in the global structure are chunked and the runtime system automatically copies an entire chunk from the global address space when a node in the chunk is requested. Multiple copies of chunks necessitate managing coherence and consistency. Rather than providing a general distributed shared memory solution to the problem, we leverage the linked nature of these data structures to support a restricted but efficient solution.

Using GCL directives, the user can advise the runtime system of the access characteristics within a phase of the program. For example, when a data structure is known to be accessed in a read-only fashion, runtime overhead can be reduced by removing extra checking to ensure mutual exclusion and coherence. Similarly, relaxed consistency guarantees allow the runtime to keep locally consistent views of the global data without the need for expensive coherence mechanisms. This is often sufficient for a large class of applications of interest. Data consistency is discussed in more detail below.

**Chunked Global View**

The chunked global view exposes the explicit chunking structure of the global linked data structure. This allows application designers to optimize algorithms for intra-chunk locality. A chunk-aware algorithm is able to directly compute on all nodes contained in a given chunk, and may also choose to explicitly overlap communication and computation when accessing entire chunks located on remote processes. The
chunked global view defines a set of operations which are useful in realizing this functionality.

In addition to node access via global pointers, the chunked view allows get access to return references to chunks and collections of chunks. This allows a program to request the communication of one or more chunks, then compute on them locally. Asynchronous operation provides the ability to overlap these communication and computation phases in order to provide good performance. Updating chunked data is supported at the chunk level as well as at the node level.

The chunked global view can provide general traversal primitives that operate on chunks. The application may use different traversal techniques depending on the desired behavior at chunk boundaries. Custom traversal patterns can be implemented by extending the general framework.

Since better spatial and temporal locality will yield higher overall performance, it is important to provide flexibility in chunking and distributing shared data. For many applications, algorithm-specific routines can be written which better match the chunk allocation with the data access pattern, giving better performance. The chunked global view provides default chunking as well as hook extensions which permit algorithm-specific chunking and allow forced re-chunking on demand. Similar to node layout within chunks, the default data distribution performed by the system may also be improved by taking into account application-specific details.

2.2.3 Data Consistency Models

The Global Chunks system provides two consistency models in order to efficiently support a wide range of linked-structure applications. The explicit consistency model
leaves issues of data consistency largely to the programmer. The location consistency (LC) model provides a scalable model for weak consistency. Both of these modes are augmented by additional synchronization primitives when consistency provided by the system is insufficient for the algorithm used.

Software distributed shared memory systems (DSM) typically use home-based lazy release consistency (HLRC) [57] which is a stronger memory consistency model than either mechanism described below. Both HLRC and LC use acquire/release semantics for exclusive access and as a synchronization mechanism. Maintaining cache coherency with HLRC requires that updates are always propagated to the owner of the modified page, then page invalidations are done when a process performs an acquire operation. Partitioned global address space languages use a variety of consistency models. UPC provides its own strict and relaxed consistency modes, with strict corresponding to sequential consistency and relaxed allowing arbitrary reordering of reads and writes relative to overall program execution order [86]. UPC provides fence, barrier, and wait/notify primitives. Titanium is another PGAS language which is a superset of Java and provides sequential consistency [91].

**Programmer-Explicit Data Consistency**

In a programmer explicit mode, the Global Chunks runtime relies on the developer to manage synchronization and data consistency. Read and write semantics can be operated in either a strict or relaxed form.

The strict mode performs reads immediately, with program control returning when the requested data is available. Data is always fetched from the master location, and no caching of chunks is performed. *Put* operations similarly do not return until the write is complete at the remote end. The strict operating form is used mostly for
checking algorithm correctness or where performance is not important. If two updates
to the same location are issued concurrently, the runtime behavior is undefined.

While the relaxed mode has weaker guarantees on data completion and synchro-
nization, this permits optimizations allowing efficient and scalable communication.
In this mode, when a program issues a `get` request, the entire chunk which contains
the requested node is transferred and stored in a local cache. Subsequent requests for
other remote nodes in this chunk are served directly from cache without communi-
cation. The relaxed mode follows location consistency semantics without supporting
acquire/release operations.

When the program performs a `put` operation, the version in cache will be up-
dated directly ensuring local sequential consistency. Global Chunks performs write
buffering in the relaxed mode. Multiple `put` operations will be buffered and the up-
dates batched and coalesced. When the buffer is full, the updates are sent using a
one-sided a vector put operation, often resulting in a single bulk communication op-
eration. Buffered writes update the chunk cache of the local process, so that updates
are seen immediately by the application.

Versions stored in other processor chunk caches are not invalidated or automatic-
ically updated with the new values. If the program requires greater data synchro-
nization at certain program points, atomic operations, fences, barriers, and flush
operations may be used to explicitly control synchronization.

Relaxed mode programmer-explicit consistency requires very little runtime over-
head and is very efficient. Since the parallel decomposition of many algorithms often
naturally yields a set of data-independent computations, they can be directly mapped
into a program which uses Global Chunks and pay no penalty for unneeded synchronization and data consistency management. For example, many tree and graph computations enter a read-only phase after they have been constructed. Frequently, creation is a small portion of the runtime relative to the read-only phase of the program. In this case, explicit consistency will provide maximum performance for the bulk of the computation.

**Location Consistency**

*a Location consistency* is a weak consistency model proposed and detailed by Gao and Sarkar in [41, 40]. Location consistency models a memory location as a partially ordered multiset of write and synchronization operations. The basic model supports read and write operations (which map to GCL get/put operations) and acquire/release operations for exclusive access to and synchronization of a shared memory location.

Formally, the LC model defines the state of a memory location, $L$, as the partially ordered multiset: $\text{state}(L) = (S, \prec)$, with $S$ being a multiset and $\prec$ being a partial order on $S$. Each element $e$ of $S$ denotes both the type of write, acquire, or release event involving $L$ and the processor performing the operation.

Upon performing a new memory operation, $e$, the state set is updated by adding $e$ to $S$ and updating the order, $\prec$, such that $e$ succeeds all other operations which involve the same processor as $e$. If $e$ is an acquire operation, the update rule for the partial order must also ensure that most recent release operation $e'$, if well defined, precedes the acquire operation $e$. 

25
Read semantics are defined by extending the set $S$ to include read events, and by modifying the ordering relation such that any read on processor $P_i$ is preceded by all existing operations in $S$ also involving $P_i$.

The values that may returned for a read operation, $e$, of location $L$ from processor $P_i$ are defined to be the set of values $V$:

$$V(e) = \{v | \exists w = write(P_j, v, L) \in S \text{ and satisfies the two conditions below.}\}$$

The first condition is that if $w$ precedes the read $e$ in $\prec$, $v$ can only be included in $V(e)$ if it is the most recent predecessor write. The second is that if $w$ does not precede the read, it is automatically included in $V(e)$. A system is defined to be location consistent if for each read operation $R$ on location $L$, the result of $R$ belongs to the value set $V$. Further, if the value set of $R$ contains more than one value, there is a data race in the program. This is presented in more formal detail in [41].

Informally, a get operation of a shared data element may return any value that has been written to it from a different processor, or the value last written by the local process. Using acquire/release pairs both provides exclusive access to the element, as well as ensuring that the local copy has been synchronized with the master.

Location consistency is a useful model when access to distributed shared data is not independent. Concurrent access is well defined both with mutually exclusive access and without. Even though the consistency provided is very weak, implementation is tractable on distributed memory clusters, whereas many stronger models are not. Again, when applications require stronger consistency than the GCL operations under LC provide, traditional synchronization techniques such as explicit locking, barrier, and atomic operations can still be used to strengthen data consistency.
2.3 Programming Interface

The Global Chunk Layer uses several structures to manage the globally shared namespace and related data. This section first describes these constructs and then gives an overview of the programming interface. Next, the mechanisms used to keep the global data in a coherent state are described, followed by an overview of the custom element allocation interface.

2.3.1 Core Programming Constructs

Figure 2.2: Relationships between the core GCL programming constructs.
• **Global Context**: The GCL keeps some global state replicated in the private address space of each participating process. This includes state for the GCL interface, callback registrations, as well as process size and rank information.

• **Element Groups**: (gcl_elemgrp_t) An *element group* corresponds to a collection of elements which have similar size and connectivity parameters. Element groups may consist of a single global structure, a collection of global structures, or simply a set of elements. Distinct data element allocations from the same group will be made from the same pool of chunks. Element groups also maintain some semantic information about node linkage to avoid looping during traversal and managing links between elements. The data consistency mode is also associated with element groups.

• **Chunks**: (gcl_chunk_t) A *chunk* consists of a collection of data elements grouped together for moving in a single communication transaction. Chunks are only visible to programmers using the chunked global view. The API exposes routines to both create and update chunks directly, as well as accessing the data elements contained within the node. The chunk data structure is logically defined as a header containing chunk metadata and an arbitrarily long sequence of data elements. The sequence length is determined during the chunk creation process.

• **Global Element Pointers**: (gcl_elemptr_t) These pointers are the basic element reference data type used by programmers. Global element pointers are portable references to any shared data element. All operations on global element pointers must be performed using the API to ensure safety.
• **Data Elements**: (gcl_elem_t) Data elements consist of two components: element *link structure* which is managed through the GCL API and is visible to the runtime, and the *body* which is a user-defined structure that is opaque to the runtime.

### 2.3.2 Core Operations

**Operations on the Global State**

```c
void gcl_init(int *argc, char ***argv);
void gcl_finalize(void);
void gcl_abort(void);
```

Listing 2.1: GCL Operations on Global State.

The *gcl_init()* and *gcl_finalize()* routines are both called only once during execution, and perform initialization and teardown of the runtime state, one-sided communication system, etc. Calling *gcl_abort()* halts program execution and gracefully releases network resources. These operations are all called collectively.

**Operations on Element Groups**

```c
gcl_group_t gcl_group_init(size_t chunksize, size_t elemsize);
void gcl_group_destroy(gcl_group_t elemgrp);
```

Listing 2.2: GCL Operations on Element Groups.

The *gcl_group_init()* routine sets up an element group which can be used for element creation, usage, etc. The programmer must specify the *chunksize* parameter which determines the number of elements per chunk, the *elemsize* which typically would be the *sizeof()* of the user-defined element structure.

These routines are all collective routines.
Operations on Global Pointers

```c
void gcl_set_inactive(gcl_elemptr_t ptr);
void gcl_is_active(gcl_elemptr_t ptr);
void gcl_elemptr_copy(gcl_elemptr_t from, gcl_elemptr_t to);
int gcl_elemptr_equals(gcl_elemptr_t pl, gcl_elemptr_t p2);
```

Listing 2.3: GCL Operations on Global Pointers.

Since references to elements in the global space must be portable, they are implemented as a structured datatype which may be embedded within a data element. Traditionally, applications which use pointer-based data structures check pointer values against the NULL value to test for the presence of children, parents, edges, etc. With an embedded structure type, testing against NULL is meaningless, so these checks must be handled differently. The GCL provides `gcl_set_inactive()` and `gcl_is_active()` to perform the equivalent of NULL assignment and comparison (i.e. `p = NULL` and `p == NULL`). Both `gcl_elemptr_copy()` and `gcl_elemptr_equals()` provide support for safe global pointer duplication and testing for referential equality.

Operations on Chunks

```c
long gcl_chunk_create(gcl_group_t *egrp);
long gcl_chunk_create_remote(gcl_group_t *egrp, int proc);
void gcl_chunk_destroy(long chunkid);
long gcl_chunk_nextfree(long chunkid);
int gcl_chunk_active_elem(long chunkid, int off);
void gcl_chunk_free_elem(long chunkid, int off);
int gcl_chunk_size(long chunkid);
int gcl_chunk_allocated(long chunkid);
```

Listing 2.4: GCL Operations on Chunks.

These operations provide the basic operations for dealing with chunks. These routines allow for the creation and destruction of new local or remote chunks and primitives for directly allocating from a specific chunk. `gcl_chunk_nextfree()` returns the offset of the next unused element in a chunk. Both `gcl_chunk_active_elem()` and `gcl_chunk_free_elem()` test and set (respectively) the chunk metadata corresponding
to the allocation status of a specific element within a chunk. Both `gcl_chunk_size()` and `gcl_chunk_allocated()` are used to interrogate a given chunk about its size and the number of allocated elements.

**Operations on Elements**

```c
/* get/compute/put */
gcl_elem_t *gcl_get_elem(gcl_group_t elemgrp, gcl_elemptr_t ptr);
gcl_elem_t *gcl_get_elem_direct(gcl_group_t elemgrp, gcl_elemptr_t ptr);
void gcl_put_elem(gcl_group_t elemgrp, gcl_elemptr_t ptr, 
gcl_elem_t *elem);
void gcl_finish_elem(gcl_group_t elemgrp, gcl_elem_t *elem);

/* location consistent synchronization */
gcl_elem_t *gcl_acquire_elem(gcl_group_t elemgrp, gcl_elemptr_t ptr);
void gcl_release_elem(gcl_group_t elemgrp, gcl_elemptr_t ptr);

/* elem allocation */
gcl_elemptr_t gcl_elem_alloc(gcl_group_t elemgrp, gcl_elemptr_t hint);
gcl_elemptr_t gcl_elem_alloc_all(gcl_group_t elemgrp);
void gcl_elem_free_elem(gcl_group_t elemgrp, gcl_elemptr_t elem);

/* follow/modify link structure */
gcl_elemptr_t *gcl_get_link(gcl_group_t elemgrp, gcl_elem_t *elem, int linkidx);
void gcl_link_elem(gcl_group_t elemgrp, gcl_elemptr_t from, gcl_elem_t *fromelem, 
int index, gcl_elemptr_t to);
void gcl_clear_link(gcl_group_t elemgrp, gcl_elem_t *elem, int linkidx);
void gcl_clear_links(gcl_group_t elemgrp, gcl_elem_t *elem);
```

Listing 2.5: GCL Operations on Elements

These operations are the core routines for accessing, modifying, and updating global data in the uniform global view. None of these routines are collective except for `gcl_elem_alloc_all()`.

- **gcl_get_elem()**: Dereferences a global pointer. Ensures that returned value is always a local copy of the element. A get operation must always be paired with either a call to `gcl_put_elem()` for globally updating an element or `gcl_finish_elem()` for declaring that access to the element is complete and it is safe to discard. The `get` routine may involve communication if the requested data is non-local and the containing chunk is not in the local cache.
• **gcl.getElemDirect()**: Dereferences a global pointer. May return either a direct reference to the element data or a pointer to a local copy. This routine is typically used when in-place element modification is legal and the shared data is known to be local. Non-local data is copied into a buffer similar to `gcl.getElem()`, above. A direct get operation must always be paired with either a call to `gcl.putElem()` for globally updating an element or `gcl.finishElem()` for declaring that access to the element is complete and it is safe to discard. This routine may involve communication if the requested data is non-local and the containing chunk is not in the local cache.

• **gcl.putElem()**: Stores the updated element into the global location specified. If the element was modified directly, this may be a no-op. The shared element data becomes invalid after this call and may subsequently be removed from the local process address space. This routine may involve communication if the master location for the element resides on another process.

• **gcl.finishElem()**: Declares that data access to the specified element is complete and it is safe to discard. The element buffer may be reclaimed by the system or replaced with data for another element. No communication is performed.

• **gcl.acquireElem()**: Used under location consistency to acquire mutually exclusive access to an element and also to guarantee that the value used is up-to-date as specified by the rules of the consistency model. This routine may involve communication.
• **gcl_release_elem():** Used under location consistency to release mutually exclusive access to an element and possibly update the master location of the element. This routine may involve communication.

• **gcl_elem_alloc():** Allocates a new data element. This allocation follows the policy associated with the specified element group, with the default policy being *local-open*. The optional hint parameter may be used by the allocation mechanism to allocate an element from the same chunk. May involve communication depending on the allocation policy.

• **gcl_elem_alloc_all():** Collectively allocates a single new data element and returns a reference to it to all calling processes. Typically this is used to allocate globally visible entry-point to the shared data structure, for example, the root node for a tree or subtree. This routine is collective and involves communication.

• **gcl_link_elem():** Creates a uni-directional link between an element and a specified reference to another element. If possible (i.e. the linked nodes are within the same chunk), an optimized relative reference will be used (described in the next section). The updated link is not guaranteed to be globally visible until after a subsequent put operation and fence. No communication is performed.

### Operations on Chunked Elements

```c
#include "gcl_common.h"

// Get number of chunks in a group
gcl_chunk_t **gcl_chunk_get_local(gcl_group_t *egrp, int *numchunks);

// Get chunk given a chunk identifier
gcl_chunk_t *gcl_chunk_get(gcl_cid_t chunkid);

// Put a chunk in the chunk list
void gcl_chunk_put(gcl_chunk_t *chunk);

// Initialize an iterator for a chunk
void gcl_iter_init(gcl_iter_t *chunkiter, gcl_chunk_t *chunk);

// Get next element in a chunk
gcl_elem_t *gcl_iter_next(gcl_iter_t *chunkiter);

// Check if there is a next element in a chunk
int gcl_iter_has_next(gcl_iter_t *chunkiter);

// Get element at a given offset in a chunk
gcl_elem_t *gcl_chunk_get_elem(gcl_chunk_t *chunk, off_t off);

// Put an element at a given offset in a chunk
void *gcl_chunk_put_elem(gcl_chunk_t *chunk, off_t off,
                         gcl_elem_t *elem);
```

Listing 2.6: GCL Operations on Chunked Elements
These operations support programming in the chunked global view.

The `gcl_chunk_get()`, and `gcl_chunk_get_local()` routines are for accessing chunk structures. The `get` operation will perform communication, if necessary, to retrieve a copy of the chunk corresponding to the identifier. The `get local` operation returns an array of references to all chunks corresponding to a specific chunk group located on the current process. This facilitates owner-compute operations at the chunk level. `gcl_chunk_put()` is used to update chunks with modified versions. None of these routines are collective.

The remaining routines are used to move through the elements of a chunk and compute on them. `gcl_iter_init()` initializes an iterator structure to iterate over the elements of a given chunk. The traditional iterator operations are implemented by `gcl_iter_next()` and `gcl_iter_has_next()` to iterate over the elements of a chunk in a loop. The `gcl_chunk_get_elem()` and `gcl_chunk_put_elem()` routines provide random access to elements in a chunk. The chunk `get` and `put` routines may involve communication if the chunks reside on remote processes, otherwise these routines are neither collective nor perform communication.

### 2.3.3 Data Consistency and Coherence Management

The GCL provides support for relaxed mode programming which caches reads and buffers writes. Often, a program will contain parts which have different consistency requirements for the global data structure. For example, an application may enter a phase where a tree is used primarily as a read-only structure. In this case, relaxed mode access would take advantage of caching and buffering to reduce communication and improve performance when accessing global tree data. The relaxed mode is
the default consistency model. The GCL provides routines to change the consistency mode as well as operations to explicitly manage data consistency in the relaxed mode.

```c
void gcl_enable_strict(gcl_group_t elemgrp);
void gcl_disable_strict(gcl_group_t elemgrp);
void gcl_flush_cache(gcl_group_t elemgrp);
void gcl_flush_elem(gcl_group_t elemgrp, gcl_nodeptr_t ptr);
void gcl_write_flush(gcl_group_t elemgrp, int proc);
void gcl_write_flush_all(gcl_group_t elemgrp);
void gcl_write_fence(gcl_group_t elemgrp, int proc);
void gcl_write_fence_all(gcl_group_t elemgrp);
```

Listing 2.7: GCL Consistency and Coherence Operations

Enabling strict mode disables caching and buffering during a phase of computation. The remaining routines are used to explicitly manage data consistency and synchronization in the relaxed mode.

The `gcl_flush_cache()` operation marks all lines of the cache as invalid, forcing subsequent reads to be updated from the master chunk. The `gcl_flush_node()` routine explicitly flushes the cached chunk containing the specified node from cache. No communication is performed.

The `gcl_write_flush()` and `gcl_write_flush_all()` routines are responsible for draining the write buffers for all pending `gcl_put_elem()` operations either to a specific processor or to all processes, respectively. The `gcl_write_fence()` and `gcl_write_fence_all()` routines similarly flush the write buffers and also guarantee that the writes have completed on the remote processor.

### 2.3.4 Using Global Chunks

This section will demonstrate two simple use cases for the Global Chunks library for working with distributed linked data.
int search_list(gcl_group_t grp, gcl_elemptr_t head, int key) {
    gcl_elemptr_t cur, *next;
    gcl_elem_t *curelem;

gcl_elemptr_copy(head, cur);

    while (gcl_is_active(cur)) {
        curelem = gcl_get_elem(grp, cur);
        if (curelem->value == key) {
            gcl_finish_elem(grp, curelem);
            return 1;
        }
        next = gcl_get_link(grp, curelem, 0);
        gcl_elemptr_copy(*next, cur);
    }

    return 0;
}

Listing 2.8: Example: Searching a linked list structure.

Uniform Global View

The first example, shown in Listing 2.8, involves a linked-list structure implemented with the GCL. This code uses the uniform global view, in which the chunking of the actual data as well as the location in the partitioned global address space is transparent to the application programmer. The caller specifies an element group, a global pointer to the head of the list, and a key value to search the list for.

This function begins by copying the global pointer to the head of the list into a cursor pointer. Next, the function loops through the list until the cursor points to an inactive (i.e. NULL) global address (line 7). Each element is fetched from the partitioned global address space, and the data value compared against the key. If the key matches, then the search completes and returns a success value.

If the search does not complete, the function calls `gcl_get_link()` on line 15, which gets the value of the “next” global pointer in the list. Since this pointer is
stored in the current element, it must be copied (line 16) to the cursor pointer. Lastly, the current element is discarded with `gcl_finish_elem()` and the search continues. If the end of the list is reached without finding a match to the key, the search terminates and returns a failure value.

**Chunked Global View**

This example, shown in Listing 2.9, demonstrates an application of the chunked global view. The chunked global view can be extremely efficient for any operations which are completely independent of one another, i.e. “do-all” parallelism. Assume that there is a distributed shared data structure and the application needs to scale all of the data values in it by some constant factor.

First, the function gets the list of all chunks which are local to the process by calling `gcl_chunk_get_local()`. The scaling function then loops over each of the local chunks, so that all computation operates only on local data. To iterate over the

```c
void scale_elems(gcl_group_t grp, double factor) {
  gcl_chunk_t **localchunks;
  gcl_iter_t iter;
  int nchunks, i;
  my_elem_t *element;

  localchunks = gcl_chunk_get_local(grp, &nchunks);
  for (i=0; i<nchunks; i++) {
    gcl_iter_init(&iter, localchunks[i]);
    while (gcl_iter_has_next(&iter)) {
      element = (my_elem_t *)gcl_iter_next(&iter);
      element->data *= factor;
    }
  }
}
```

Listing 2.9: Example: Scaling all elements in a data structure.
elements in each chunk, GCL provides an iterator construct, which is initialized for each local chunk (line 10). Using the iterator, each element in the chunk is accessed with \texttt{gcl_iter_next()}, and the data value scaled by the scaling factor.

This operation is much more efficient than the equivalent uniform-global view implementation for two main reasons. The first is that scaling happens without the need for any communication. All computation is constrained to local data, avoiding the possibility of remote data access. Second, this approach avoids much of the overhead supporting the distributed global address space. The scaling function avoids the use of global pointers altogether, including the use of \texttt{get/put} operations, and the iterator construct uses local native memory pointers to quickly access the locally chunked data.

### 2.3.5 Customizing Node Allocation

The GCL allows the programmer to override the default allocation strategy with custom allocators which can take advantage of \textit{a priori} knowledge of the data access pattern to improve locality. Custom allocators can keep private state in between calls to the allocator. Additionally \texttt{gcl_elem_alloc()}, which wraps the custom allocator, takes a \textit{hint} parameter which allows the programmer to provide some context from the allocation site.

For example, the default node allocator uses a policy called \textit{local open}. The local open allocator uses the notion of a “current chunk”, and stores a reference to it 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 reference
is updated to reflect the new current chunk 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.

### 2.4 Design and Implementation

![Memory layout of GCL related structures on a single process.](image)

Figure 2.3: Memory layout of GCL related structures on a single process.
Upon program initialization, the library allocates a large, contiguous memory segment through ARMCI, which is used for all globally visible data, as shown in Figure 2.3. ARMCI shared memory allocation is a collective process, so the memory is allocated at program initialization, then non-collective chunk allocations from it are managed by the GCL during program execution.

Since global data is implemented as distributed sets of chunks, indexing these chunks globally can be done with either a directory approach or by assigning an “owner” to each chunk. Directories must be updated when chunk migration occurs, whereas single-node ownership may result in inefficiencies when updates are all remote and never local. The approach used in the Global Chunks library is to partition these global chunk identifiers among processors, then have each processor maintain a directory for its mapped chunks.

Each process keeps a portion of the distributed chunk directory in its own shared memory segment. In order to specify a one-sided communication operation used to transfer a chunk, the communication system needs to know both the processor owning the data, as well as the location of the data on that process. Since the chunk identifiers are partitioned over the number of processors, chunk ownership (at least relative to the distributed chunk directory) can be recovered from the identifier directly.

The directory maps a set of chunk identifiers to their locations in the partitioned global address space. Because this directory information may be referenced frequently, it is cached by each process. Inside the global context structure, each process keeps a chunk directory cache, which stores local copies of the chunk directory information. This structure is array based, indexed by chunk identifier and owning process. The
directory cache is shared by all globally chunked data, and is not specific to a given
global data structure.

When a remote chunk is required for a local computation, the system will have
to perform a communication for the initial lookup in the distributed chunk cache.
Subsequent lookups can be served from the directory cache, avoiding additional com-
munication. In the case of frequent chunk migration, the cached information will
need to be invalidated frequently, resulting in more communication requests.

Each instance of a global data structure corresponds to an element group within
the GCL. In addition to group-specific parameters (data element size, chunk size,
etc.), the element group maintains a data cache for remote chunks, as well as a
collection of buffers used to batch write communications for efficiency.

2.4.1 Global Pointers

A key feature of the Global Chunks system is to provide applications the ability to
use pointer-based data structures on a distributed memory system. Global pointers
are the fundamental data structure used to provide portable, globally visible, elemental
data access. There are two representations for global pointers. In the basic form, a
global pointer is a tuple consisting of a globally unique chunk index and the offset
of a particular element within that chunk. This structure is called an absolute global
pointer, since the components are specified in globally unique, absolute terms.

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. Reducing the dereferencing cost of global pointers by an order of magnitude is a significant contribution of this work.
Consider the data layout shown in Figure 2.4. There are two nodes with a parent-child relationship. Each node consists of 12-bytes of content data, and at least two global pointer values located at offsets 12 and 28. As shown in Figure 2.5, an absolute global pointer can be used to refer to the parent node, located in chunk 121 at an offset of 4240 bytes from the beginning of the chunk.

If the child element is located in the same chunk as the parent, the global pointer can be adapted to refer to the distance between the nodes within the same chunk. A negative value in the chunk index is used to signify that the contained pointer is a relative global pointer, rather than absolute.

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. In this case, the location of the pointer to the child node is 4252 (4240 + 12-byte data). This is stored as a negative value in the chunk index.

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 caching status.

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
Since Global Chunks provides a relaxed consistency model, the runtime is permitted to cache remote data and defer updates for greater communication efficiency.

**Caching**

By default, Global Chunks operates in the relaxed explicit consistency mode. When performing a \textit{get} operation, the runtime must determine the pointer type, data location, and whether or not there is a local copy of the data in order to provide the application access to the data.

The procedure for performing a \textit{get} operation is shown in Algorithm 1. This procedure performs the following steps to provide the application access to the global data:

1. The global pointer is checked to ensure that it is active (i.e. non-null)

2. If the pointer is a relative pointer, then the pointer must reside inside either a cached chunk or a local chunk. The address of the pointed-to data is computed directly as described above and returned to the application.

3. If the pointer is not a relative pointer, then the chunk identifier is used to determine ownership of the chunk.

4. If the referenced chunk is local to this processor, the address of the pointed-to element is computed and returned.
Algorithm 1 Global Pointer Dereferencing Algorithm

input: $ptr$ — a global pointer to an element
output: a local pointer to the referenced data element

gcl-get-elem(ptr)

if is-active(ptr) then
    if ptr.chunkid < 0 then
        // relative global pointer
        return compute-relative-addr(ptr)
    else
        // absolute global pointer
        owner ← get-owner(ptr.chunkid)
        if owner = self then
            return compute-local-addr(ptr)
        else
            (line, found, full) ← search-cache(ptr.chunkid)
            if found then
                return compute-cache-addr(ptr, line)
            else
                if full then
                    line ← evict-from-cache(line)
                end if
                raddress ← get-directory(ptr.chunkid)
                fetch-chunk(owner, raddress, line)
                return compute-cache-addr(ptr, line)
            end if
        end if
    end if
else
    error: null pointer dereference
end if
5. Otherwise, the referenced chunk is remote, compute the hash value for the chunk in the local cache, and check the cache for the chunk.

6. If the chunk is in the cache, return the address of the referenced data element.

7. If the chunk is not in the cache and the cache is not full, issue a communication to fetch the chunk into an empty location in the cache. Return the address of the referenced data element.

8. Otherwise, find an entry in the cache which is safe to evict, and remove it from the cache. Then issue a communication to fetch the chunk into the new empty location in the cache. Return the address of the referenced data element.

The data cache is implemented as a hashed array of chunk-sized cache lines, allocated when the GCL element group is created. If possible, the cache memory is allocated from registered RDMA memory, to permit high-performance communications from the remote memory segment directly to the cache via the network interface card. The GCL maintains a valid/invalid bit flag for each cache line, to reduce the cost of flushing the cache and avoid redundant memory allocation and release operations.

**Cache Eviction and Chunk Replacement**

Caching within Global Chunks happens at the chunk level, but data access occurs with respect to specific data elements. This can be problematic when the data cache is at capacity on a processor and chunks must be replaced from it to make room for more recently accessed data.

Since data access occurs within the `get/compute/put` model, it is unsafe to eject chunks from in between a get and put operation. To complicate matters, programs
which use pointer-based shared data structures frequently access data in the following format:

```c
void recursive_function(ptr) {
    node_t *node = get_node(ptr);
    for (int i=0; i < NUM_CHILDREN; i++) {
        recursive_function(node->child[i]);
    }
    put_node(ptr, node);
}
```

Listing 2.10: Typical Recursive Access Structure in GCL

In this example, the first call to `recursive_function()` would make the chunk containing the element pointed to by `ptr` ineligible for eviction until the recursion fully terminates. Caching occurs at the chunk level, so we must track which chunks have elements that are in use. To prevent illegal chunk replacements, Global Chunks increments a reference counter in the chunk header during a `get` operation, and decrements the reference count during a `put` or `finish` operation. This has the drawback of incurring an additional reference and increment operation to each global pointer dereference operation but allows the system to continue to operate when the cache is at capacity.

**Write Buffering**

In addition to caching chunk data for efficient reads, Global Chunks stores updates into write buffers. The system maintains a write buffer for every other process participating in a computation. When data is written via a `put` operation, the copy in the local data cache is immediately updated to reflect the changes. Additionally, the updated element data is stored in the write buffer corresponding to the process owning the updated element.
When the buffer is full, or when an explicit flush or fence operation is invoked, the buffer is drained. Redundant updates are eliminated from the buffer, leaving the update from the most recent put operation. The updates are coalesced into a vector of I/O updates and written to the shared memory segment on the remote process with an ARMCI update operation.

### 2.4.3 Custom Allocation

The performance benefits of caching reads can be greatly improved by reducing the cache miss rate. Because different applications may traverse linked data structures with a variety of access patterns, no single chunking policy will be optimal for all applications. For example, some algorithms may traverse a tree structure in a depth-first manner, breadth-first, level-wise, etc. Global Chunks 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 a pool of open chunks to allocate from or affinity information of 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 allocate new nodes from the best match given the allocation hint provided.

2.4.4 Interoperability

The Global Chunk Layer is implemented in C and currently uses the ARMCI one-sided communication library for all data movement. Programs which use Global Chunks directly, or any derivative libraries are interoperable with a variety of programming languages including C, Fortran, C++, and Co-Array Fortran (CAF). Global Chunks can also operate in an MPI environment as well, allowing computation phases that may be more efficiently implemented with message passing.

When programming under multiple programming models concurrently, details such as synchronization must be composed carefully to ensure correct behavior. Global Chunks provides API access to the routines used to manage the communication consistency to allow the construction of composed synchronization routines. For example, when composing MPI and Global Arrays with a Global Chunks program, barrier operations will have slightly different semantics with regards to communication completeness. Instead of requiring program developers to correctly order multiple barrier, fence, or flush operations, Global Chunks provides API access to the synchronization internals. This allows the construction of composed synchronization routines which reduce programmer burden and reduce complexity.
2.5 Experimental Evaluation

The Global Chunks Layer is a runtime system which provides an abstraction layer between the communication layer of the partitioned global address space and a data-structure specific library, targeted to a specific application domain. As such, the experimental analysis in this section focuses on features unique to the GCL runtime, with application-specific results left to later sections.

Most of these experiments were conducted on the Pentium 4 cluster at the Ohio Supercomputing Center. This cluster utilized 2x2.4 GHz dual-core nodes each running 64-bit Linux and configured with 4GB of RAM. The interconnect used is 10Gbps Infiniband. Other experiments were performed on the Glenn cluster at the Ohio Supercomputing Center. This cluster consists of 2.6 GHz quad-core and 2.5GHz eight-core AMD Opteron processors each running 64-bit Linux and configured with 8GB and 24GB of RAM, respectively. The cluster is connected via a 10Gbps and 20Gbps Infiniband interconnect.

2.5.1 Coarse-Grained Data Movement

One of the key advantages of the Global Chunks Layer over other partitioned global address space systems is the use of chunks for bulk data transfer. Using chunks involves a larger initial communication, but subsequent requests for remote data are satisfied without the need for additional transfers. As shown in Figure 2.6, two traversals of a large linked data structure are performed with a system implemented using GCL and another version implemented using Berkeley UPC. When data access is completely local, the two systems perform similarly. If the data access is completely remote, the Global Chunks Layer can provide nearly the same level of performance,
whereas the large number of communication events in the UPC version add substantial overhead to performance.

2.5.2 Global Pointer Overhead

Dereferencing global pointers in a partitioned global address space is more costly than traditional pointer dereferencing in a sequential program. How this impacts performance depends greatly on the workload characteristics of the target application. Figure 2.7 shows the relative overhead of pointer dereferencing relative to the amount of computation performed at each data element. With little or no workload, data access is dominated by following links between data elements. The more that the application is dominated by link traversal, the greater the relative impact of the global dereference operation overhead. As can be seen in the figure, the C pointer

Figure 2.6: Impact of chunking on element data access.
dereference is much faster than a GCL pointer dereference, which is considerably faster than global pointer evaluation in either of two UPC implementations. The vertical lines at 6 and 39 FLOPS represent the computational workload of Barnes-Hut, a common tree-based benchmark used for evaluation of these systems. Given the relatively small amount of computation performed at each node in Barnes-Hut, global pointer overhead is a significant factor in application performance.

2.5.3 Chunk Global View

When a computation can be decomposed into an operation that permits out-of-order processing of the global data structure, the chunk global view can dramatically provide an increase in performance. Using the chunk global view eliminates the overhead due to global pointer dereferencing for both read and write operations.
The computation used in this benchmark is a vector scaling operation, common in applications such as MADNESS [48]. Each node in the tree contains a vector of coefficients which are to be scaled by a constant factor. As shown in Figure 2.8, the impact of using the chunk global view varies depending on the per-node workload of the computation. With a unit-length vector, the chunk global view provides a speedup between 4 and 11 times the same computation following explicit link structure. As the amount of computation increases to scaling a vector of length 1000, the computation is dominated by the scaling operation and any inefficiencies arising from the use of the link structure become insignificant.
CHAPTER 3

Global Trees: Specialized Support for Global View Tree Structures

3.1 Overview

Global Trees (GT) 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 3.1 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 earlier, in Figure 2.1, Global Trees is implemented on top of several lower-level runtime libraries with the ARMCI one-sided communication library at its
Global Trees is built upon the Global Chunks Layer (GCL) runtime system. 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 either component and also maintain interoperability with MPI code.

GT also takes advantage of the Scioto dynamic load balancing system to manage task parallelism exposed through parallel tree traversals[35]. 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 across all processors.
3.2 Programming Model and Interface

Because GT is built upon the Global Chunks Layer, it inherits the majority of its programming model. Like Global Chunks, GT programs are executed in an SPMD manner with MIMD task-parallel regions managed by Scioto. Most communication in Global Trees happens via one-sided communication operations. Additional collectives are used for synchronization, termination, and tree traversal. Each process can independently and asynchronously access any portion of the shared tree without requiring the cooperation of the process storing the remote data.

Global Trees uses a get/compute/put model for tree access. Computation on global tree nodes happens by dereferencing a GT global pointer through the programming interface. After the application has completed its computation on the node, 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 updates are 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. Global Trees currently uses the programmer managed consistency model provided by Global Chunks.

Shared tree data is distributed 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. Communication happens via the creation and use of global tree structures, however GT is interoperable with any other communication system compatible with the Global Chunks system, such as MPI, Global Arrays, ARMCI, etc.
3.2.1 Core Programming Constructs

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 also 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 opaque portable references to any tree node. All operations on global node pointers must be 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, and also to 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 a parent node, child nodes, or to group nodes together in an application specific manner. The API provides a means to distinguish between these links when updating them or performing tree traversals.

### 3.2.2 Programming Interface

#### Operations on the Global State

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

Listing 3.1: GT Operations on Global State

The *gt_init()* and *gt_finalize()* routines are both called only once during execution, and perform initialization and teardown of state kept internal to Global Trees, as well as the GCL and ARMCI system layers. The *gt_abort()* function terminates program execution while providing a graceful exit of system and network resources. These operations are all called collectively.
Operations on Node Groups

```
typedef struct {
    size_t chunksize;
    size_t nodesize;
    int nlinks;
} gt_nodegrp_t;

void gt_nodegrp_init(gt_nodegrp_t *grp, size_t chunksize, size_t nodesize, int nlinks);
void gt_nodegrp_destroy(gt_nodegrp_t grp);
void gt_nodegrp_free(gt_nodegrp_t grp);

gt_nodegrp_t gt_tree_create(size_t chunksize, size_t nodesize, int nlinks);
```

Listing 3.2: GT Operations on Node Groups

The `gt_nodegrp_init()` function initializes a node group (and related GCL element group), which is to be used with a set of one or more trees which have similar node size characteristics. The programmer must specify the number of elements to be grouped together in a chunk as `chunksize`, as well as the size of the node in the tree. Multiple node types in a single tree are typically specified as union types. The `gt_nodegrp_destroy()` is used to free all resources associated with a node group and `gt_nodegrp_free()` keeps the group, but releases all allocated tree node data associated with it.

The `gt_tree_create()` routine is provided for convenience. This routine creates a node group, and allocates a root node for a new tree. The global pointer for this root node is broadcast to all processes. All of these routines are collective routines.

Operations on Global Pointers

These functions encapsulate the similarly named functions from the GCL library which perform the same tasks.
Operations on Tree Nodes

The functions `gt_get_node()`, `gt_get_node_direct()`, `gt_put_node()`, and `gt_finish_node()` are similar to the GCL equivalents described in Section 2.3.2.

Global Trees supports a more structured notion of linking nodes than those provided by the GCL. Specifically, Global Trees distinguishes between three different categories of links within tree data. The `gt_link_child()`, `gt_link_parent()`, and `gt_link_node_to_link()` routines manage these different link types described as follows:
• **Child Links:** Child links are used to specify downward, directed edges in a tree from a parent node to a child. The function `gt_link_child()` updates a global pointer in the parent node, corresponding to the child located at index. This operation can be specified only in terms of the `from` and `to` global pointers, but if the actual node data is passed in, it will be used directly for efficiency. If possible, GT will use a relative pointer if the parent and child nodes are located within the same chunk.

Child links are the links which are traversed with the built-in traversal patterns. Applications are responsible for preventing the introduction of cycles into the tree structure, which may cause traversals to loop indefinitely.

• **Parent Links:** Parent links are used to specify upward, directed edges in a tree from a child to a parent. As with setting child links, the `from` and `to` global pointers are required, but the child element may be passed as `fromnode` to avoid redundant processing and communication. Relative pointers will be used when possible.

• **Other Links:** Global Trees permits applications to keep additional links between nodes in a directed, acyclic manner. These may be used to traverse the leaves of a tree, or otherwise group together logically related, but structurally disjoint sections of the tree. GT provides mechanisms for updating and retrieving these application-specific pointers because the runtime manages all connectivity information for efficiency. Relative global pointers are used when possible.
Operations to Allocate Nodes

In addition to the allocation routines provided by the underlying GCL framework, Global Trees also provides allocation primitives which are both convenient for tree operations as well as provide node allocators with extra context to improve node placement within chunks. Basic node allocation is done with `gt_node_alloc()`, which takes a `hint` parameter that is passed to the specific allocation strategy in use and may be used specify a desired chunk to allocate the new node from. Similarly, `gt_node_linkalloc()` performs the same allocation procedure, but also creates a link between the two nodes.

Both the `gt_alloc_parent()` and `gt_alloc_child()` functions are used for extending a tree, growing either bottom-up, or top-down respectively. In addition to allocating a new tree node and linking it into the existing tree structure, these routines provide valuable context to the underlying allocation strategies. While any allocation placement will be correct, using specific parent and child allocation primitives allows node placement to be made with respect to the overall tree structure, rather than a sequence of disjoint memory allocations.
Other Operations

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 provides a fence operation to wait for all pending writes to complete and also a fence operation particular to writes destined for a specific process.

3.2.3 Using Global Trees

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 may provide context from the call site to inform the allocation choice. For example, it is usually desirable to allocate a child node from the same chunk as the parent. The global pointer of the parent could be passed to the allocation routine to help the allocator place the new node “close” to the parent node. This is 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 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 general links between any two nodes in the global space.

**Example:** The code given in Listing 3.6 demonstrates a basic use of Global Trees to perform a recursive copy operation of a global tree structure. 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 of the source node.
void tree_copy (gt_group_t ng, gt_nodeptr_t src, gt_nodeptr_t dst) {
    gt_nodeptr_t schild, dchild;
    gt_node_t snode, dnode;
    int i;

    snode = gt_get_node (ng, src);
    dnode = gt_get_node (ng, dst);

    for (i = 0; i < NUM_CHILDREN; i++) {
        schild = gt_get_child (ng, snode, i);
        if (gt_is_active (schild)) {
            dchild = gt_node_alloc (ng, dst);
            gt_link_child (ng, dst, dnode, i, dchild);
            tree_copy (ng, schild, dchild);
        }
    }
    gt_put_node (ng, dst, dnode);
    gt_finish_node (ng, snode);
}

Listing 3.6: Example: Copying a Global Tree.

3.2.4 Custom Node Allocation

GT provides a facility for the programmer to override the default allocation strategy with custom allocators. This allows the allocation process to take advantage of \textit{a priori} knowledge of the data access pattern to improve locality. Custom allocators are permitted to keep private information in between calls to the allocator. The hint value passed into the allocation routine 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.

The default node allocator uses a policy called \textit{local open}. The local open allocator keeps a reference to the chunk currently being allocated from. When a new allocation is requested, the node is allocated from the current chunk. A new chunk is allocated when the current chunk is full, and the private state is updated. Hint information is not used.
If the placement of nodes in chunks is critical 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.

3.2.5 Tree Traversals

Global Trees may be used to traverse shared trees by any application-defined method which follows the get/compute/put model of node access. Because many applications use common traversal orders, GT provides optimized, parallel traversals which are capable of invoking application code on each node during the traversal. Global tree traversals are performed by collectively registering a visitor callback, which performs a computation on a single tree node. A reference to the visitor callback routine 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 pre- and post-order top-down traversals, as well as bottom-up and level-wise traversals.

Traversal Visitors:

A traversal visitor is a callback function which is invoked by one of the predefined traversal routines. 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 invokes a traversal operation, passing the registered callback key to identify the desired visitor function to be invoked at each node. Indirection is required because the parallel traversals may be dynamically load-balanced, preventing the use of
traditional function pointers to refer to callback routines. Visitor callbacks have the following form:

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

**Built-in Traversal Example**

The code fragment listed in Listing 3.7 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.

```c
typedef struct {
  double cost;
} mynode_t;

cost_visitor(gt_group_t nodegrp, gt_nodeptr_t ptr) {
  mynode_t *thisbody, *childbody;
  gt_nodeptr_t childptr;
  int i;
  thisbody = (mynode_t *)gt_get_node(nodegrp, ptr);
  thisbody->weight = 0;
  for (i=0;i<NUM_CHILDREN;i++) {
    childptr = gt_get_child(nodegrp, thisnode, i);
    if (gt_is_active(childptr)) {
      childbody = (mynode_t *)gt_get_node(nodegrp, childptr);
      thisbody->weight += childbody->weight;
      gt_finish_node(nodegrp, childnode);
    }
  }
  gt_put_node(nodegrp, ptr, thisnode);
}

void compute_cost(gt_nodegrp_t nodegrp, gt_nodeptr_t *root) {
  gt_visitor_t costvisitor;
  costvisitor = gt_register_visit(cost_visitor);
  gt_traversal_bottomup(nodegrp, root, costvisitor);
}
```

Listing 3.7: Example: Bottom-up tree traversal.
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()`.

### 3.3 Experimental Evaluation

We present a scalability study of Global Trees using a benchmark which models the compression and reconstruction operators from the MADNESS multi-resolution quantum chemistry package. This experiment was performed on the Glenn cluster at the Ohio Supercomputing Center. This cluster consists of 2.6 GHz quad-core and 2.5GHz eight-core AMD Opteron processors each running 64-bit Linux and configured with 8GB and 24GB of RAM, respectively. The cluster is connected via a 10Gbps and 20Gbps Infiniband interconnect.

We also 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 [88] as a basis for our comparison. For this application, we compare the performance of a GT implementation with the performance achieved using Cluster OpenMP [50], a commercial software distributed shared memory platform and explore the GT performance and design space through several microbenchmarks.
The Barnes-Hut experiments were conducted 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.

3.3.1 Cluster OpenMP

Cluster OpenMP is a commercial product distributed by Intel that provides support for OpenMP parallel programs on distributed memory systems [50]. OpenMP is a parallel annotation language that allows the programmer to annotate their source code with directives that enable parallel execution [32]. Intel’s cluster OpenMP is built on top of the Intel C++ compiler and the TreadMarks software distributed shared memory (DSM) system [3].

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.

3.3.2 Unified Parallel C

Unified Parallel C (UPC) is programming language which allows for fine-grained data access and global, pointer-based data structures [86]. UPC is built on top of the GASNet partitioned-global address space communication layer, which is similar in spirit to ARMCI [15]. UPC is a full language, which is a superset of C. The UPC
compiler is a source-to-source compiler which emits C code annotated with calls to the one-sided communication system. Fine-grained data access in UPC is performed with fine-grained communication, unless the programmer explicitly writes the program to the contrary.

### 3.3.3 Global Trees Scalability

![Graph showing scalability of GT with a vector scaling operation](image)

Figure 3.2: Scalability of GT with a vector scaling operation

Figure 3.2 demonstrates the scalability of the Global Trees system. This example uses a connected-tree version of the vector scaling benchmark used in Section 2.5. This benchmark was configured to use a tree with roughly 90 million nodes and a workload comparable to the coefficient scaling operation from the MADNESS framework.
3.3.4 Barnes-Hut Performance Analysis

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.

Figure 3.3 shows a parallel performance comparison between the Global Trees, UPC, 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 both the Global
Figure 3.3: Parallel performance of Barnes-Hut in thousands of bodies processed per second on 32 cluster nodes for Cluster OpenMP, UPC, and Global Trees implementations on a 512k body problem with a chunk size of 256.

Trees implementation and UPC implementation achieve good performance on up to 32 processors. Global Trees can outperform UPC due to its use of chunking to gain additional communication efficiency. 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 in response to updates of shared data.

From the data in Figure 3.4 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 3.3. 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 3.5
Figure 3.5: Parallel performance of Global Trees Barnes-Hut over a range of chunk sizes.

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. Chunk size selection is explored in more detail in Section 5.3.1.
CHAPTER 4

Data Locality in Global Trees

4.1 Overview

Achieving good performance from a parallel application which is based on a large
distributed irregular data structure is difficult. Extracting the most performance
from modern large-scale distributed memory clusters places a great burden on both
application software and run-time systems. Irregular problems in particular, have
unique challenges with respect to load balance and data locality. The nature of many
of these applications and the systems on which they run makes statically anticipating
and improving program performance challenging. Additionally, systems which are
able to adapt and balance one component of the dynamic execution of a program,
like workload, often end up being at odds with another, such as data locality.

Global Trees provides a programming model which allows developers to access a
distributed tree structure using global pointers. It is possible to treat shared data
accesses uniformly, however, doing so will generally result in poorer performance than
a locality-aware system when run on a compute cluster. A program which references
data in local memory will clearly be faster than one which needs data resident in
remote memory. Intuitively, reducing the communication volume and communication overhead to support a computation will result in higher performance.

4.1.1 Approaches

Improving the data reference locality of a program can occur by addressing the inherent data reference locality as well as the realized data locality [27]. The inherent data locality is specific to a particular algorithm, whereas the realized data locality is concerned with the system-level mapping of program data to actual storage locations within the cluster. Common approaches which aim to improve data reference locality are algorithmic modification, static data mapping techniques, the use of heuristic allocation strategies, and using dynamic program behavior to optimize data placement. Since Global Trees is a general framework for tree-based computations, we focus on improving the realized data reference locality as provided by the Global Trees system.

All communication in GT occurs at the chunk level, hence, element allocations must be mapped to chunks. Both mapping a given data element to a particular chunk, as well as the assignment of a specific chunk to a particular process will impact the amount of communication performed during program execution.

Static data mapping approaches, such as the local-open technique used by default in GT can perform very well when the data access pattern is able to be characterized by a static rule. With the local-open allocator, nodes are placed in chunks in allocation-order. If the data access matches the allocation pattern, then intuitively, this is an optimal method for placing data. If the data access pattern differs from the static mapping, the program will likely have a larger communication volume and lower overall performance.
Heuristic allocation strategies employ a set of static rules along with the dynamic state of the program to make an educated guess about the best processor and chunk to allocate a node from. When deciding where to place a new node, these strategies only have access to information from the call site as well as information kept between allocations. Since many tree-based applications have a recursive structure during tree creation, allocation call site information may not provide enough context to make a good allocation choice.

A major drawback of improving data locality through heuristic allocation is the lack of global information from within the allocation mechanism. For example, consider an application where the dominant access pattern is depth-first. If the tree is allocated in a depth-first fashion, then a static mapping is sufficient. If the tree is allocated in a different order, it is difficult to know whether the current allocation is for a leaf, or is the root of a long branch. This is important because chunks should be full in order to get the benefits of coarse-grained communication, but they should also contain related data, so as to take advantage of spatial locality within the tree. Without additional global knowledge of future allocations, it is very difficult to develop heuristics which result in improvements to realized data reference locality.

Profile-driven techniques measure various data access and communication patterns during a representative execution of a program and then use this information to guide future allocations. Since the profile is unique to each specific tree application, these approaches can be very accurate. Gathering profile data and performing analysis on it, however, can be very expensive and offset the benefits of better data placement.
4.1.2 Contributions

The remainder of this chapter will focus on a profile-driven technique which improves data reference locality in tree-based applications. This technique improves the realized data locality of an application by grouping related nodes in the same chunks and placing those chunks on processors in order to reduce the communication volume.

This work makes the following contributions:

- **Mapping Algorithm**: A profile-driven technique is described which maps nodes to chunks and processors in a GT application. The profile collects the number of data references for each edge in the tree structure. This information is used to guide the placement of nodes corresponding to the most frequently traversed edges within the same chunk.

- **Iterative Data Layout**: A technique for applying the node mapping layout to tree-based algorithms which have an iterative structure, such as the Barnes-Hut algorithm. One or a small number of iterations are used to generate the data profile, then future iterations will create the global tree according to the mapping.

- **Collective Data Layout**: A technique for applying the node mapping layout to tree-based algorithms with a collective data relayout operation. The program is allowed to operate for a period of time in the profile gathering phase. When a good profile has been obtained, the program can collectively call a routine which computes a new mapping and moves the global tree data according to the mapping.
- **Remote Data Allocation**: An efficient technique for allocating nodes on remote processes under the get/compute/put model. This approach avoids both unnecessary communication and blocking for communications to complete.

- **Empirical Evaluation**: An empirical evaluation of these techniques is presented which demonstrates the effectiveness of this approach. Performance is evaluated using a breadth-first allocation / depth-first access benchmark, as well as the Barnes-Hut application from the SPLASH-2 benchmark suite [88].

4.2 Analysis

The goal of any locality optimization is to reduce overall runtime of the program. By understanding the components of the overall runtime, we can focus on specific optimization opportunities that will result in a lower execution time.

Consider the sequential execution of a Global Trees program with a running time of $t_{seq}$. Then the ideal running time for a parallel execution on $p$ processors is:

$$t_{ideal} = \frac{t_{seq}}{p}$$

A more realistic model for parallel execution time accounts for communication ($t_{comm}$) and GT parallel runtime system overhead ($t_{gt}$):

$$t_{real} = \frac{t_{seq}}{p} + t_{comm} + t_{gt}$$
Decomposing the communication overhead into the traditional startup time \( (t_s) \) and per-byte transfer time \( (t_b) \) yields the following, with \( i \) communication events each of size \( B_i \):

\[
t_{\text{real}} = \frac{t_{\text{seq}}}{p} + \sum_i (t_s + B_it_b) + t_{gt} \quad (4.1)
\]

Therefore, improving performance at the system-level involves the minimization of time spent communicating (specifically, time spent waiting for communication), as well as time lost inside of Global Trees and lower libraries managing the parallel execution and communication.

### 4.2.1 Reducing Communication Time

Extracting the communication cost from Equation 4.1 gives the following expression:

\[
t_{\text{comm}} = \sum_i (t_s + B_it_b)
\]

This definition of \( t_{\text{comm}} \) accounts for the time spent in non-overlapping communication, and is the sum of startup cost and transfer time for \( i \) communication events in the execution.

In the PGAS programming model, global data is distributed amongst the participating processors. The computation on a given processor will need some portion of this global data to compute, which can be broken into two groups, local and non-local data. Because local data access time is accounted by the \( t_{seq}/p \) and \( t_{gt} \) terms, we can
restrict our focus to the access of non-local (or remote) data elements for the purpose of reducing non-overlapping communication time.

For a given computation, presume that \( \lambda \) data references are local and that \( \rho \) data references are non-local. The total number of data references (\( \tau \)) can be expressed as:

\[
\tau = \lambda + \rho
\]

Global Trees uses the concept of chunks to perform bulk data transfers and improve locality. Non-local data may be cached, which may be subsequently accessed with only local access cost. If we define \( \lambda' \) as the number of non-local data accesses which are served from cache and \( \rho' \) as the non-local accesses requiring communication, we get the following relation:

\[
\tau = \lambda + \lambda' + \rho'
\]

Under the chunking model, unnecessary data may communicated in a chunk communication due to the element-to-chunk mapping. The number of data elements which are communicated, but never referenced is defined as \( \sigma \).

Now, if we let \(|\rho|\) and \(|\sigma|\) represent the respective data volume of these elements in bytes, we get the following expression for \( t_{\text{comm}} \):

\[
t_{\text{comm}} = \rho't_s + (|\rho| + |\sigma|)t_b
\]  

(4.2)

Thus, to improve communication time, the following factors must be considered, assuming \( t_s \) and \( t_b \) are inherent properties of the system:
• $\rho'$ - **Non-local data accesses requiring communication.** Reducing the number of communication events will reduce the contribution of $t_s$ to overall runtime.

• $\rho$ - **All non-local data accesses.** Reducing the amount of remote data referenced will reduce the overall communication volume.

• $\sigma$ - **Unnecessary data transferred.** Any unreferenced data which is communicated without computational overlap is wasted time spent in transfer.

### 4.2.2 Reducing Global Trees Overhead

Using Global Trees to provide a global namespace for distributed data access incurs a certain amount of overhead to program execution time. Node allocation, reading and writing global data, and other components of the GT framework add complexity and time as compared to native language-supported data creation, access, and deletion operations. In particular, operations on the shared data structure which deal with the link structure of the tree are all handled by GT routines.

While startup and teardown costs may be larger than a sequential code, they are typically far outweighed by overhead from the data access routines in the computational kernel. A large amount of the complexity that must be handled by Global Trees is the conversion from the global pointer representation to the $\langle \text{processor}, \text{address} \rangle$ representation used by the PGAS access subsystem.

One of the key techniques to reduce overhead when handling global pointers is to take advantages of relative pointers whenever possible. Recall from Algorithm 1, that dereferencing an absolute global pointer involves function call overhead, several conditionals, a directory lookup, and some pointer arithmetic, whereas a relative
pointer is limited to a single conditional (“is this a relative pointer?”) and two
pointer additions.

When an application has a heavy workload at each node, then $t_{gt}$ will be small
relative to the other terms in Equation 4.1. On the other hand, when the computa-
tional workload is low, and the communication small, global pointer overhead can be a
substantial impediment to good performance. If the computation can be decomposed
into a form amenable to the chunked global view, then global pointer overhead can
be completely eliminated. In reality, many computations which are based on linked
data structures do not have a structure which allows for the unordered access of the
tree data. Hence, any system-level support for improving the use of relative pointers
will be effective in reducing the impact of GT overhead, $t_{gt}$.

4.2.3 Techniques

Reducing communication and overhead in GT programs can be distilled into ap-
proaches which reduce the number of non-local data references ($\rho$ and $\rho'$) as well as
improving the element-to-chunk mapping which can additionally reduce the transfer
of unnecessary data ($\sigma$) from Equation 4.2.

Non-local data access occurs when a) initial data allocation occurred elsewhere, b)
the computation has been migrated but the data has not, or c) the data is shared and
needed by many processes. Data layout is controlled by the parallel decomposition of
the tree allocation process in conjunction with the chunk mapping strategy used by
the allocator. Computation migration is a reactive process which is used when static
task partitioning is inadequate, the destination of the migration will greatly impact
the number of non-local data elements accessed. Lastly, shared data will need to be communicated irrespective of the originating location.

The following factors are useful when considering means for reducing communication and overhead in Global Trees applications:

- **Chunksize**: The number of elements in each chunk determines the granularity of communication in a GT application. Larger chunksizes will incur less overhead due communication costs, lowering $\rho'$. Smaller chunksizes will often have less unnecessary data in them, lowering $\sigma$.

- **Element-Chunk Mapping**: Because of the dynamic, irregular nature of tree computations, mapping data elements to chunks (via node allocation) is imperfect. Improving the effectiveness of this mapping will lower $\sigma$ and may also reduce $\rho'$. Additionally, this factor is significant when determining the legality of using a relative pointer. Relative pointers can only be between nodes in the same chunk, so the mapping of correlated nodes to the same chunk has a significant impact on global pointer overhead.

- **Communication Implementation**: The total runtime is only increased when we must *wait* for communication needed by the computation. Therefore, any improvement in communication overlap or speculative communication may reduce $\rho'$, the number of blocking communications that we must wait for.

Of these factors, the locality optimization techniques presented in this chapter focus on improving the element-chunk mapping. Determining the best chunksize for a given application is discussed in detail in Chapter 5 and the communication implementation is not considered.
Influencing the element-to-chunk mapping can be done initially at allocation time, or later through a data re-layout operation. Element-to-chunk mapping is done initially at allocation time according to an allocation strategy, with some additional semantic information provided by the application.

In general, the problem of finding an optimal algorithm for data placement and code arrangement that minimizes the number of cache misses (i.e. $\rho'$), is NP-Hard. It has also been shown that efficient approximate algorithms produce sub-optimal mappings relative to an optimal solution [74].

Instead of attempting to make good placement choices at allocation time, we opt for a profile-driven approach to data layout. This approach takes a data reference trace for a Global Trees parallel program and uses this information to determine an element/chunk mapping for a subsequent phase of computation. The element mapping can be used to move nodes as a collective operation, or used to inform future element allocations. This approach extends Global Trees to support dynamic data distributions, as described in Section 1.2.1.

4.3 Profile-guided Data Layout

Many tree-based applications have a phase where the tree structure is created and then used for computation in another phase. It is frequently the case that the data access pattern used in the computation differs from the specific order used to create the tree.

Global Trees must make a decision to place an element in a chunk at allocation time, with imperfect knowledge about subsequent allocations or the dominant
data access pattern. As a result, profile data may improve data placement after the application has run in a steady state for awhile.

The profile-guided layout process works in two steps: First, the program runs in a profile gathering mode, where data is collected about data references during an application specified phase of the program. Next, the runtime performs a partitioned layout which assigns elements to chunks, and chunks to processes based on the information gathered from the profile data.

4.3.1 Data Structures

The partitioning step relies on two key data structures. The first data structure, edgerefs, tracks the number of times a specific tree edge has been traversed throughout a phase of computation. This data is maintained during the profiling step and is provided as input to the partitioning step. The edge is defined to be a two-tuple containing the unique identifiers of each vertex on the edge, and the edgerefs structure is a mapping from edges to reference counts:

\[
edgerefs = \text{global}_\text{id} \times \text{global}_\text{id} \rightarrow \text{edge}_\text{traversal}_\text{count}\]

The ptrs structure, is a collection keyed by unique node identifiers and contains the global pointer mapping from the profile run to the new node location in the partitioned layout. This structure also keeps a boolean flag assigned, which is initially false, and set once the node has been placed into a chunk, within the layout algorithm.

\[
\text{ptrs} = \text{global}_\text{id} \rightarrow \text{old}_\text{ptr} \times \text{new}_\text{ptr} \times \text{assigned}
\]
4.3.2 Profile Gathering

The edgerefs structure is updated throughout the profile phase of the computation. When the application accesses the child or parent of a node, a counter corresponding to the edge in edgerefs is incremented.

A key premise of this approach is the ability to identify a specific location in the tree which is stable between runs and over different phases of computation. We desire an approach which is stable in the presence of minor differences in the tree structures, which eliminates using global pointers for this purpose, a technique used by Chilimbi in his work [28].

For spatial applications, each tree node represents a specific domain in space, at least in a relative sense. (In Barnes-Hut, for example, the bounding box represented by the octree may change each iteration). Still, because there is a fixed out-degree of node, each node can be identified by either a level and index pair, or by enumerating all possible locations in the tree and using this index to identify a node.

A more general approach is to uniquely identify a node by its path from the root of the tree to itself. The path consists of the indices of each child pointer that would need to be followed to find the given node. This has the advantage that it can be used to determine the unique identifiers of parents and children. This identifier may be very long, however. The minimum storage needed for unique path identifier is defined by:

\[
\text{bits} = \log_2(\text{outdegree}) \times \text{max_depth} \tag{4.3}
\]
In practice, this size may be increased by a factor of 2 or more in order to provide efficient access at runtime. Since the identifier needs to be stored within the node, this may have a negative impact on applications with very small node sizes.

Additionally, applications may also have specific techniques for uniquely identifying nodes, which could also be used as a key into the `edgerefs` mapping.

### 4.3.3 Element Mapping Algorithm

During a profile run, presume that edges are traversed via a GT API function, and that the profile data is stored in the `edgerefs` structure.

The element partitioning algorithm has the following steps:

1. Sort `edgerefs` in descending order as `edgerefs'`.
2. Initialize the `ptrs` structure from `edgerefs`.
3. Partition the nodes into logical chunks.
4. Assign the chunks to specific processors.
5. Merge small chunks together.
6. Assign all nodes physical `(chunk, offset)` addresses.
7. Emit the allocation map to all processes.

First, the list of edges, `edgerefs`, is sorted in descending order of the reference counts for each edge access and stored in `edgerefs'`. Then, each of the vertices of an edge are stored in the `ptrs` structure, initially unallocated. Next, the vertices of the edges in `edgerefs'` are co-located in the same chunk, in descending reference count.
order. This algorithm is detailed in Algorithm 2. The chunks that are assigned are *logical* chunks, which may be partially filled and have not been assigned to a specific processor.

Once nodes have been placed in chunks, partially filled chunks are assigned to specific processors and merged together. Finally, the mapping of unique identifiers to partitioned chunk/offsets is output and is used for the data relayout operation or for allocations in the next iteration.

This algorithm has good complexity characteristics. The first step is $O(n \log n)$ in the number of nodes in the tree, which dominates performance of the algorithm. The other steps are either $O(n)$ in the number of tree nodes, or $O(n \log n)$ in the number of chunks if sorting is used in step 5.

The partitioning algorithm takes a greedy approach, working through the list of edges with the highest reference counts first. Each vertex of the edge is checked to see if it has already been assigned to a chunk. If neither node has been assigned, a new chunk is created and both nodes are placed within it.

If only one node has been placed in a new chunk, then we check to see if there is enough room in the same chunk, and the nodes are co-allocated. If there is not, the unassigned node is placed into a new chunk. If both nodes have been assigned, we attempt to merge the two chunks together if they are less than the max chunksize.

### 4.3.4 Chunk Mapping

The procedure listed in Algorithm 2 results in a set of chunks which have logical identifiers, but have not been assigned to a specific process. The element mapping
Algorithm 2 Profile-driven Element Mapping

for edge($n_1,n_2) \in \text{edgerefs'}$ do
  if $\neg \text{ptrs}[n_1].\text{assigned} \land \text{ptrs}[n_2].\text{assigned}$ then
    // neither vertex has been assigned
    $c = \text{new-chunk}()$
    $np_1 = \text{add-node}(c,n_1)$
    $np_2 = \text{add-node}(c,n_2)$
    $\text{ptrs}[n_1].\text{new_ptr} = np_1$
    $\text{ptrs}[n_2].\text{new_ptr} = np_2$
  else if $\neg \text{ptrs}[n_1].\text{assigned}$ then
    // only $n_2$ has been assigned
    if $\text{size(chunk-of}(n_2)) < \text{chunksize}$ then
      $np_1 = \text{add-node(chunk-of}(n_2),n_1)$
      $\text{ptrs}[n_1].\text{new_ptr} = np_1$
    else
      $np_1 = \text{add-node(new-chunk}(),n_1)$
      $\text{ptrs}[n_1].\text{new_ptr} = np_1$
    end if
  else if $\neg \text{ptrs}[n_2].\text{assigned}$ then
    // only $n_1$ has been assigned
    if $\text{size(chunk-of}(n_1)) < \text{chunksize}$ then
      $np_2 = \text{add-node(chunk-of}(n_1),n_2)$
      $\text{ptrs}[n_2].\text{new_ptr} = np_2$
    else
      $np_2 = \text{add-node(new-chunk}(),n_2)$
      $\text{ptrs}[n_2].\text{new_ptr} = np_2$
    end if
  else if $\text{size(chunk-of}(n_1)) + \text{size(chunk-of}(n_2)) < \text{chunksize}$ then
    merge-chunks(chunk-of(n_1), chunk-of(n_2))
  end if
end for
algorithm can improve performance by placing related nodes together, reducing \( \sigma \) from Eq. 4.2 and also by increasing the opportunity for using relative global pointers.

Ideally, each chunk will be assigned to an owner which will minimize \( \rho' \), the number of non-local data accesses requiring communication. In practice, imperfect processor mappings can be tolerated as long as the benefits from reducing unnecessary data communication, using more relative pointers, and dynamic load balancing outweigh inaccuracies in the chunk-to-processor mapping.

**Round-Robin Chunk Distribution**

The simplest means for mapping chunks to processors is to just assign them in a round-robin manner. Because there is usually some affinity between one or more processes and a given chunk, this technique is not able to exploit any of that data locality, and is less effective than the other methods listed below. Round-robin distribution is very fast and results in a random data distribution which may be useful in some highly dynamic or load-balanced computations.

**Edge-Weight Chunk Distribution**

The edge-weight distribution uses the inter-chunk edges between nodes to guide which chunks should be placed together to create a balanced data distribution. Once all nodes have been placed within chunks, we construct a set of inter-chunk edges. For a given chunk \( c \), each node within \( c \) may have a set of edges which reach to other chunks. The union of all such sets is then weighted by the edge reference weight given in \textit{edgerefs}.

The set of global inter-chunk edges is then sorted in descending order by the reference count between each chunk. The pair of chunks with the highest inter-chunk
reference counts are placed together. This process is repeated until all chunks have been placed together. This process is similar to the element mapping algorithm and results in a balanced data distribution.

A drawback of this algorithm is that no information about chunk ownership is used to assign a group of related chunks to a specific processor. In particular, this algorithm is a good candidate for use with a locality-aware dynamic load balancing system, but may suffer with static task mapping.

**Owner-Weight Chunk Distribution**

The owner-weight distribution considers the processor making each data reference from the profile gathering phase and uses this information to assign chunks to specific processors. The algorithm which performs the chunk mapping relies the existence of a data structure, `chunkowners`. The `chunkowners` data structure is a mapping from chunk identifier, `c`, to an integer vector from `1...P`. Each vector element `p`, contains a count of the number references made for nodes in chunk `c` by processor `p`.

This distribution can be implemented to provide a uniform data distribution, which may require limiting the number of chunks assigned to processor. Often, better performance is observed with a non-uniform data distribution due to reduced communication. If this is acceptable, the partitioner iterates over each chunk and maps the chunk to the index corresponding to the maximum value of `chunkowners`, ignoring any process-chunk limits.
Previous Owner Chunk Distribution

The previous owner distribution assigns a chunk to a processor according to the owner of the mapped data during the profile run. Similar to the chunkowners structure, each chunk maintains a vector of counters, one for each process. As a node is assigned to a chunk, the counter corresponding to the process which owned the node is incremented. When assigning chunks to processors, the max counter value determines where the chunk should be placed.

This approach is useful because it can be highly effective in reducing remote node allocations when performing iterative profile-driven data layout. If the application has a good inherent data distribution and subsequent iterations create the tree in a similar manner, using previous owner information will minimize the number of remote allocations performed. This distribution will have a minimal impact on cache misses and will be most useful in improving relative pointer usage.

4.3.5 Profile-driven Allocation in Iterative Applications

Once the element mapping process is completed, we are left with a ptrs structure which maps global node identifier to a new global pointer which specifies the location that the node should be moved to. Many scientific applications which use tree structures have an iterative nature, where the tree is used to model a physical system over time. In the case of $n$-body simulation, for example, a tree is constructed every timestep representing the physical space with leaf nodes corresponding to particles (planets, stars, electrons, etc.). Since the simulation timestep is usually small, the tree structure remains similar from step to step.
We can use the profile gathered from one timestep to build a mapping which is used to specify locations to allocate specific nodes from during the next timestep. In this case, the mapping only is used to specify future allocations according to the data layout, so the *ptrs* structure does not need to maintain the *old_ptr* information containing the previous location in the mapping.

This information is used with a special custom allocator in GT which gathers the profile using the *local-open* allocation strategy. Once a sufficient amount of profile data has been collected, the layout engine develops the new element mapping which is loaded into a special profile-driven allocator. When new nodes are allocated in the tree, the allocator determines the global path identifier of the current allocation and returns the location specified by the mapping to the application program.

### 4.3.6 Collective Data Relayout

To handle applications which do not follow this iterative structure, Global Trees provides a collective data layout operation which moves the data in a global tree in-place. As with the iterative approach, the program operates in a profile gathering phase for a period of time, with allocations being made using the *local-open* allocator. The profiling phase is specified explicitly by the application programmer. When a sufficient data reference profile has been generated, the layout engine develops a new element mapping, resulting in a *ptrs* structure which specifies both old and new locations.

The collective data layout step is effectively a tree copy, with data being copied from old locations to the new ones in the element mapping. To avoid synchronization and dependency problems, this step currently requires enough free memory in the
partitioned global address space to make a full copy of the tree. Since the layout engine knows the full size of the global data structure, a sufficient number of free chunks are reserved by mapping engine and used as targets to copy into during the data movement phase.

The actual data movement is done by iterating through each chunk using the chunk global view. Each node is updated by rewriting all global pointers contained within it to reflect the new addresses of its parent and children. Lastly, the node data is copied to its new location, using a buffered write if the new node location resides on another processor.

Once this step has completed, the old chunks may be returned to the GT/GCL freelist for re-use. Any global pointers maintained by the application prior to the relayout step in either stack, heap, or global variables must be invalidated and regenerated by the application.

4.4 Data Relayout Implementation

4.4.1 Path Identifiers

One of the key premises of the element-mapping algorithm is the existence of globally unique node identifiers which represent the same location within a tree structure. These identifiers are required for supporting profile-driven allocation for iterative applications. In Barnes-Hut, for example, the tree is created anew for each timestep, and it may be subtly different than the tree used in the previous iteration.

To satisfy the need to identify nodes from iteration to iteration, the path from the root of the tree can be used to uniquely identify a node’s location within the tree. A path identifier consist of the indices of each child pointer that would need
to be followed from the root to the target node. The path identifier is stored in the GT managed link structure of the user-defined tree node data structure. The path identifier is stored as a string of bytes, one for byte for each level of the tree. To save space and reduce communication transfer times, this can be compressed to the size specified in Eq. 4.3, but at the cost of more computation when manipulating the path identifier during profile gathering and node allocation.

The element mapping algorithm works on edges between nodes, but the path identifier is stored within a node itself. The convention is that data reference count corresponding to a path identifier, $p$ from node $n$, represents the edge between $n$ and the parent of $n$. Having the path identifier of the lowest vertex of the edge allows the other vertex (the parent) to be quickly determined by simply truncating the path identifier.

Path identifiers are set on node creation, if possible. Many tree-based applications construct trees in a top-down fashion, allowing the path identifier of a newly allocated child to be determined by the identifier of the parent and the child index of the new allocation. GT supports this mechanism by providing a child allocation function ($gt\_alloc\_child()$) which is used by the profile-driven custom allocator to set the path identifiers of new allocations automatically.

Because the $gt\_alloc\_child()$ routine relies on the path identifier of the parent to already exist, care must be taken to initialize the path identifier of the root node before any child allocations take place. Since root node allocation and pointer distribution happens through the $gt\_tree\_create()$ routine, this is handled automatically by the custom profile-driven node allocator.
In the case of a program which must construct the tree in a bottom-up manner or some other fashion, path identifiers can be set with a collective call after the tree has been created. This does require modification to the application code, but only requires calling a single collective call after the tree has been constructed.

Some unstructured tree algorithms, such as some data-mining techniques may not be able to use path-identifiers to represent unique nodes within a global tree. In this case, the application must provide some mechanism to maintain edge reference count information or a unique numbering of tree nodes between iterations.

4.4.2 Chunk Reservation

Allocating data from the element mapping is done by extending the existing mechanisms to support the notion of reserved chunks. This is important for both iterative profile-driven allocation as well as collective data relayout.

In the iterative mode, the tree structure may differ slightly from the profiled iteration, requiring nodes that need to be allocated in the current iteration to be allocated according to an alternate allocation method (typically local-open). The alternate allocator must not allocate from locations that may be used to service future allocations from the profile map. To prevent conflict, the profile-driven allocator marks the chunks specified by the layout engine as reserved. Reserved chunks are unavailable for servicing allocations by other node allocation mechanisms. With the collective data relayout operation, a number of chunks are reserved so that the entire global tree structure can be migrated in one operation, without having to deal with conflicting allocations.
To support chunk reservation, the runtime system had to be modified to skip over
reserved chunks when finding a new chunk to allocate from. Chunks are marked as
reserved by the profile-driven allocator when changing state from the profile-gathering
phase to the optimized allocation phase. The layout engine provides the ranges of
chunk identifiers needed for the layout mapping to the allocator, which in turn marks
the new chunks as reserved.

When nodes are allocated from reserved chunks, some overhead normally associ-
ated with allocation can be eliminated. Normally, chunk meta-data is updated with
the new allocation information and additional checks have to be made to ensure that
it is safe to allocate from a specified chunk. Under chunk reservation, it is presumed
that the allocation is safe. When releasing all of the chunks that correspond to a
global tree, all reserved chunks are moved to the global chunk freelist, ready to be
used for a subsequent iteration.

The element mapping algorithm may result in having some reserved chunks which
are partially full. This may occur if chunks are unable to be merged together, or
that the last mapped chunks do not fill an entire chunk. Normally, the GCL does
not check the master directory for the actual size of the chunk to avoid an additional
communication. Each chunk is fetched at the full possible size in case local chunk
directory information is out of date. In the case of chunk reservation, the number of
nodes reserved for a chunk is known in advance, ensuring that cached chunk directory
is always up-to-date. If a reserved chunk only has a fraction of the possible number
of elements in it, then a smaller communication operation can be issued, reducing the
transfer time of the chunk.
4.4.3 Supporting Remote Allocation

Given the layout engine and element map, it is now possible for a process to allocate a node which is best placed in the shared memory segment of another process. It is critical to avoid fine-grained communication associated with a remote node allocation in order to get good performance during the creation phase of a tree application.

Efficient Node Allocation

One of the first problems encountered when implementing support for remote node allocation is the \textit{get/compute/put} semantics used by GT, GCL, and ARMCI. The problem can be demonstrated by Listing 4.1, which demonstrates a typical node allocation.

```c
1 alloc_child(gt_nodeptr_t parent, mynode_t *parentnode, int index) {
2     gt_nodeptr_t child;
3     mynode_t *cnode;
4
5     ...       
6     gt_alloc_child(nodegrp, parent, parentnode, child, index);
7     cnode = gt_get_node(nodegrp, child);
8     ... initialize cnode data ...
9     gt_put_node(nodegrp, child, cnode);
10 }
```

Listing 4.1: Example: Conventional Child Allocation

When \texttt{gt_alloc_child()} returns a global pointer, \texttt{child}, the subsequent call to \texttt{gt_get_node()} on line 7 may cause a chunk communication to happen if the chunk containing \texttt{child} is not in the local chunk cache. Worse, this routine doesn’t even need to fetch the node because it is simply initializing the node structure and then overwriting the node on the remote end. This can cause a dramatic increase in
allocation time, which must be paid for by an increase in performance by the layout optimization.

To avoid the unnecessary communication, the semantics for get/put are relaxed for the special case of initialization. This is done by creating API support for two different operations. The first replaces the get operation with an allocate+initialize operation. The second replaces the put operation with a weaker write operation.

The allocate+initialize operation is handled by modifying the semantics of `gt_alloc_child()`. We add an additional argument, `childnode`, which is a buffer that may be stack or heap allocated and is to be used as a local node for initialization purposes:

```c
void gt_alloc_child ( gt_nodegrp_t grp, gt_nodeptr_t parent,
                     gt_node_t parentnode, gt_nodeptr_t *child,
                     int index, gt_node_t *childnode);
```

By permitting the use of locally allocated storage for initialization, there is no need to call `gt_get_node()` and incur the cost of unneeded chunk transfers.

The put node operation also needs to be updated because we have eliminated the call to `gt_get_node()`. This assumption is important because `gt_put_node()` updates metadata which is operating under the assumption that there has been a preceding get (for example, reference counting chunk cache entries). To handle this, a new operation is introduced:

```c
void gt_init_node ( gt_nodegrp_t grp, gt_nodeptr_t ptr,
                    gt_node_t *node);
```
This operation doesn’t assume that there has been a previous fetch. The local cache is updated only if a copy of the chunk is present in the cache, otherwise the cache is left alone. All calls to `gt.init_node()` which are to local addresses are written directly, otherwise the node data is copied into the write buffer and will be sent to the remote node when the buffer is flushed.

The child allocation example shown above would be rewritten as:

```c
alloc_child(gt_nodeptr_t parent, mynode_t *parentnode, int index) {
    gt_nodeptr_t child;
    mynode_t cnode;
    ... 
    gt_alloc_child(nodegrp, parent, parentnode, child, index, &cnode);
    ... initialize cnode data ...
    gt_init_node(nodegrp, child, &cnode);
}
```

In this example, the call to `gt.get_node()` has been replaced with the updated `gt.alloc_child()` call, which also initializes the GT portion of the stack allocated buffer passed in by the application. Most importantly, the only place which may introduce communication is the call to `gt.init_node()`, which will only happen when the write buffer to the destined processor is full.

**Maintaining Location Consistency**

Adapting the API to support these routines also requires some changes to the runtime behavior to preserve location consistency. Consider a new node allocation using the improved alloc/init method, as shown in Listing 4.2. In this example, presume that the element map specified that a child of the root node which is initialized inside of `tcreate()` is to be allocated in remote memory. When the call to `gt.init_node()`
Listing 4.2: Example: Improved Child Allocation

is called, the initialized tree node is enqueued in the write buffer corresponding to the owner process. A potential problem arises when the call to \texttt{gt\_get\_node()} on line 18 is issued. The runtime will check the local chunk cache and not find the chunk containing the child node. If the call to \texttt{gt\_get\_node()} is issued before the write buffer has been flushed, then the runtime will get a copy of the node data from the owning process, while the locally initialized version sits in the write buffer. This causes a race condition which violates location consistency and requires some special handling to correct.

The rules for handling cache misses must be modified to include a search through the write buffer of the owning process. When a cache miss occurs, the chunk is fetched using a one-sided memory operation. Then, the write buffer is examined and any pending writes to the destination chunk are replayed on the local copy of the chunk. By replaying pending updates, the view of those global memory locations are kept locally consistent, preserving the consistency guarantees of the GT/GCL system.
4.4.4 Collective and Iterative Data Relayout

The data relayout process is largely handled by a special custom allocator in Global Trees. The profile gathering step uses an array-backed hash table to store the reference counts corresponding to a given unique path identifier. Allocations made during the gathering phase are done according to the *local-open* strategy.

The actual data layout engine is written in Python, for flexibility and ease of prototyping. As a result, the existing implementation can be inefficient on larger profiles and datasets. Each process communicates the edges and reference counts through the filesystem using a memory-mapped file, which is then read into the layout engine. This can be made much more efficient using native C support and optimized parallel communication routines. Simulation of a C version of the layout algorithm shows good performance on medium and large-scale profile data.

At the largest scale, it may be necessary to parallelize the layout algorithm, which could be done using parallel sort algorithms and straightforward divide-and-conquer techniques [43]. A drawback of this approach to improving data locality is that the element mapping itself may become very large when profiling extremely large data structures.

4.5 Empirical Evaluation

The experimental evaluation of the profile-driven layout technique described above comprises three experimental setups. The first experiment uses a benchmark which allocates nodes from different locations in the partitioned global address space and illustrates the performance characteristics of different node allocation operations. The second experiment focuses on a benchmark which does breadth-first tree construction,
but with a depth-first dominant access pattern. Lastly, this work is applied to a GT-adapted version of the Stanford SPLASH-2 benchmark suite [88] and compared with the existing local-open based solution described in [61].

These experiments were conducted on the Glenn cluster at the Ohio Supercomputing Center. This cluster consists of 2.6 GHz quad-core and 2.5GHz eight-core AMD Opteron processors each running 64-bit Linux and configured with 8GB and 24GB of RAM, respectively. The cluster is connected via a 10Gbps and 20Gbps Infiniband interconnect.

4.5.1 Remote Node Allocation Overhead

This benchmark uses two different approaches for allocating global tree nodes. The first is the traditional alloc/get/put method and the second uses the alloc/init method described in Section 4.4.3. This benchmark allocates a large number of linked nodes from a shared memory segment that is local, one that is co-located on the same cluster node (intra-node), and one that is on a process located on another cluster node (inter-node).

As shown in Figure 4.1, this benchmark was run on different node sizes to highlight the differences between the amortized transfer speeds versus startup costs. For the most expensive remote node allocations, inter-node, the modified allocation method reduces the overhead roughly by half. The performance improvement is substantial for intra-node allocations as well, with an improvement of just under a doubling of performance. Even for local allocations, using the local buffer and modified allocation routines results in a speedup. This indicates that this technique should be used for all node allocation, when possible, regardless of the potential for remote allocations.
Breadth-first Creation / Depth-first Access

The benchmark used for these experiments simulates an application which creates a tree incrementally, in a breadth-first style. As nodes are added to the tree, the top of the tree fills out first, expanding level-by-level. The dominant access pattern uses depth-first traversal, which is repeated many times, in a phase distinct from the tree creation. While this benchmark is designed to exhibit an extreme difference between creation and access patterns, large differences between creation and access patterns are not that uncommon in parallel tree-based algorithms such as R-trees or certain operators in MADNESS [46, 48]. For the profile run, the first traversal runs in the profile-gathering mode, then the program performs the collective layout operation, and runs the remaining traversals with the new data layout.
These experiments are based on a tree with approximately 1.4 million nodes and a local chunk cache size of about 100MB. While the actual layout was done by the Python-based layout engine, simulation of the layout algorithm on a dataset of this size takes less than a second. The dominant overhead is the sorting algorithm, which takes about 500ms on the system under test.

Figure 4.2 shows that the performance of the profile-driven approach is substantially better than local-open due to the poor data locality realized by the default allocation strategy. This application is dominated by link access time, and the profile-driven layout is able to amend the locality problems by re-arranging the nodes into a depth-first ordering within the chunks, and also by locating these chunks on the processors which access them the most.
The amount of reads which are performed using relative pointers is a major contributor to the performance improvement of the profile-driven implementation. High relative pointer usage is a key indicator that related nodes are being allocated from the same chunk. As shown in Figure 4.3, the profile-driven method is able to adapt the breadth-first chunking into a depth-first chunking. With the local-open allocator, as the tree gets deeper, the chunk is filled entirely with nodes at the current level, which shows very poor spatial locality for the default allocator.

Since there is no difference in the total number of reads, we can compare the cache misses between the runs directly. By examining Figure 4.4, we can see a contributing factor for the initial performance drop of the profile-driven benchmark on a small number of processors. With a only a few processors, the profile-driven layout shows a higher number of cache misses than the default method. As more processors are
added, the profile-driven run stabilizes and the number of cache misses increases, but at a rate much slower than the local-open run. These results show that the layout mapping has been successful in placing tree data on the correct processor, especially at higher process counts.

**Barnes-Hut**

The Barnes-Hut algorithm solves the *n-body problem* to compute force interactions between *n* bodies in a *k*-dimensional body of space. For the SPLASH-2 benchmark, this is 3-dimensional space which is decomposed and represented using a shared octree structure. Initially, bodies in the system have position and mass determined by a known model, and the bodies are distributed among the processors. Each process
owns several regions of space and is responsible for inserting bodies into the subtrees that correspond to these regions.

When computing the interaction, the particles are partitioned by a Morton-ordered tree traversal, which places spatially close bodies on the same processor for computation. Each process iterates over these bodies and performs a partial traversal of the octree, evaluating the force contributions from each other body in the system. The experiments below were run on a Barnes-Hut simulation with 512k bodies, simulated for 10 timesteps.

Barnes-Hut is suitable for the iterative layout approach because it computes the gravitational simulation over many timesteps. Since the purpose of the simulation is to understand the behavior of the particles in the system, rapid changes to the bodies in space are undesirable. This implies that the tree does not change drastically between each timestep. As shown in Figure 4.5, the tree retains over 80% of its structure over several timesteps. From experience, tree similarity does drop off with each subsequent timestep, but the decline is stable, keeping 75% tree similarity at 20 timesteps.

In this case, the tree represents a spatial domain, so tree similarity can be computed easily. Two trees are similar when they have the same nodes in the same location. Tree similarity is computed by the counting the nodes that are the same in two trees, divided by the total number of nodes in both trees. Alternate techniques exist for computing tree similarity for trees which do not have a fixed spatial representation [85].

When running the Barnes-Hut benchmark with both local-open and profile-driven allocation methods, the relative improvement in performance is shown in Figure 4.6.
The profile-driven allocator shows between a 10-17% improvement in overall run-time performance. These results do not account for the computation of the new data layout. As discussed above, an efficiently implemented layout engine should not reduce the performance improvement significantly. The greater the number of timesteps which are simulated using the profile-mapping will also reduce the amortized layout cost across iterations.

Analysis of the program performance data shows that the number of cache misses between the local-open and profile-driven runs is roughly the same. This indicates that the distribution of data to processes done within Barnes between tree-creation and force calculation is quite good.
Since the allocation pattern does differ from the access pattern, the performance improvement is realized through the more efficient use of relative global pointer accesses. This is demonstrated in Figure 4.7. As the processor count increases, relative pointer usage improves for both cases. Since the total tree data is fixed, each processor requires a smaller portion of the overall tree as the number of processors increases. Because these portions are smaller, there are fewer chunks (and inter-chunk links) to traverse and the effective impact of relative pointers increases.

Figure 4.6: Barnes-Hut Relative Performance
Figure 4.7: Barnes-Hut Relative Pointer Usage
CHAPTER 5

Understanding Global Trees Program Performance and Behavior

5.1 Overview

One of the major advantages of using both the Global Chunks and Global Trees systems is in the ease of programming with globally shared data structures on distributed memory machines. While they remove much of the burden of explicit data management, these systems may also operate less efficiently than task-specific, hand-tuned monolithic applications. To achieve high-performance, it is important to understand the data locality and communication properties of each unique application implemented with the GT and GCL frameworks. Each system keeps a variety of information useful in understanding GT and GCL program performance.

This chapter details the information that is available to application programmers and how it may be used to analyze program performance. The first section discusses the data collected by the runtime, followed by a brief overview of analyzing common performance issues and choosing runtime parameters. Lastly, some tools for visualizing the tree data are presented, along with a few examples visually correlating the global tree structure with collected profile and performance data.
5.2 Collected Runtime Performance Data

Global Trees and Global Chunks can be compiled to gather a large amount of information about the runtime characteristics of program performance. Many of the statistics gathered below can only be counted with additional processing, sometimes along the critical path for global data access. To prevent unnecessary reductions in performance when this information is not needed, GT/GCL uses conditional compilation to include support for performance counting. To enable performance counting, the runtime libraries are re-compiled with the `GCL_PERF_COUNTING` symbol defined.

5.2.1 GCL/GT Memory Usage

When a program which uses GCL or GT calls `gcl_init()` or `gt_init()`, the following message is printed on process 0:

```
GCL/GT footprint: 512.0 MB shared heap 992.3 MB local cache (8192 lines)
chunk allocator: elensize 488 chunksize 256 (threads=4)
context size: 273.7 MB allocator size: 65.8 MB directory: 1.1 MB
```

This displays a general overview of the memory usage of a Global Trees program. The first line describes the size of the local portion of the shared memory segment, in this case 512 megabytes. Next the local chunk cache size, in both bytes and in number of lines. Since the caching system operates like a fully-associative cache with keys generated by a hash function, it will contain the largest number lines which is a power of 2, limited by either a run-time or compile-time limit set by the application, in this case set to 1GB.

Next, the system displays information specific to a given instance of a globally shared data structure. In this case, there is only a single global tree structure, so
most of the remaining information pertains only to it. This information includes the size of a single element, in this case 488 bytes including both node link structure and node content. The chunking factor for this run is set to be 256 nodes per chunk, and the program is executing on 4 cores.

The system also prints the size of the context structure, which is a global data structure that is responsible for managing the shared heap, chunk directory cache, and other information: the allocator structure itself, which maintains the per-process write buffers, the chunk allocation and free lists, counters, and the current allocation placement strategy. Lastly, the system displays the size of the distributed chunk directory portion located in the shared memory segment on each process.

5.2.2 Data Access Statistics

Global Trees also provides two routines, `gt_alloc_print_stats()` and a collective call, `gt_alloc_print_global()`, which display per-process and global data access statistics, respectively. The performance counters may be cleared at any time by calling the `gt_alloc_clear_counters()` function. When the print functions are called, the system prints output similar to the following:

```
GT Global Data Access Statistics:
total reads: 18920628 total writes: 3065637 relreads: 6363831 (33.634%)
direct reads:  1 local  0 remote - chunks allocated: 4489
cache reads: hits: 9775286 misses: 2982 local: 9142359 remote: 9778268
   lrel: 2423790 rrel: 3940041
cache writes:  20027 local writes: 3045510 flushes: 3047
   cache stats: evictions:  0 collisions: 7140 (1.00)
total allocations: 1148494 chunks allocated: 4489 : chunking efficiency: 0.999
  alloc:  1 link:  0 parent:  0 child: 1148493
```
This printout contains a dense amount of information and can be broken down into different groups:

**Data Access:**

The first line gives a breakdown of the total number of reads and writes since the counters were initialized or cleared. Of particular interest is the number of reads which were performed using relative pointers, so the relreads counter prints this value as well as a percentage of total reads. The next line displays how many node accesses were made in the strict mode, rather than under a relaxed consistency mode with caching and buffering. Since this application uses relaxed mode throughout, the only count here is for the root node allocation, which occurred collectively in the strict mode. Lastly, the overall number of chunks allocated during the program phase is displayed.

**Caching System Statistics:**

The next four lines refer to performance of the caching and buffering systems used in the relaxed access mode. The first line shows the total number of local and remote reads during the application. Remote reads are broken down further and categorized as either cache hits or misses. The system also tracks how many local and remote reads were done with relative pointers (lrel and rrel), respectively.

The total number of reads can be expressed by the following relation:

\[ r_{total} \equiv (r_{hit} + r_{miss} + r_{direct} + r_{local}) \equiv (r_{local} + r_{remote}) \]

The next line tracks the number of buffered writes to cached nodes, local writes, and the number of times that the write buffer was flushed. The last line counts the
number of chunk replacements that occurred during the run as a result of the cache being full, and the collisions counter tracks the efficiency of the hash table used to manage the cache.

**Allocation Statistics:**

The last component of the data access statistics is focused on the way that data was allocated since the counters were last reset. Since Global Trees supports different allocation mechanisms, each allocation strategy provides a callback which displays performance data specific to each allocator. For example, the profile-driven allocator provides a count of how many nodes were allocated from the element map, and how many nodes were allocated from the default strategy. This gives a simple measure to determine tree similarity between iterations.

The chunking efficiency is a ratio of how many chunks were filled relative to the actual data volume needed by the program. It is computed by the following formula:

\[
\text{eff} = \frac{\text{# total allocations}}{\text{# chunks allocated} \times \text{chunksize}}
\]

Lastly, the printout gives a distribution of the four different node allocation routines used. In this example, the program had a single call to `gt_alloc_node()`, to allocate the root node, but the remainder of the tree structure was created with calls to `gt_alloc_child()`.

### 5.2.3 Chunk Packing Efficiency

Global Trees also provides an enhanced profiling mode which tracks the specific nodes which have been referenced from every chunk transferred. This gives a direct measure of the \(\sigma\) parameter from Equation 4.2 and is useful in quantifying the amount
of wasted communication. Understanding the utilization of chunk contents can aid in selecting the best chunk size parameter, as discussed in the next section. This statistic is captured using a large array structure and can significantly increase the memory footprint of an execution. As such, this feature is disabled by default, even when performance gathering is enabled.

5.3 Interpreting Statistics

5.3.1 Element-to-Chunk Mapping

The two biggest tools in understanding the quality of the element-to-chunk mapping are relative pointer usage and the static chunk packing efficiency counts available with extended profiling. Each statistic tells a different part of the performance story; understanding their relationship is important for assessing the mapping quality.

Since relative pointers are only legal when the pointer refers to another node within the same chunk, high relative pointer use is indicative of a good grouping for the most referenced edges in the data structure. Relative pointer usage helps quantify both temporal and spatial locality within a chunk. Counters are incremented for every relative access. Relative pointer counts are not specific to any given chunk, rather they are aggregated over all data accesses and may be skewed by a few exceptionally good or bad chunks.

Static chunk packing efficiency captures the spatial locality of elements within individual chunks. Each chunk is tracked as a bitfield, with a single bit for each node. When a node is referenced, the bit is enabled. The collection of chunk access bits can be analyzed at any point during program execution, displaying the individual nodes referenced in each chunk, or with aggregate statistics such as min, max, and
average nodes referenced per chunk. These statistics can be used to quantify wasted communication and can also aid in the selection of chunk size.

**Empirical Chunk Size Selection**

The range of acceptable chunk sizes is determined by application locality and communication efficiency. An extremely efficient communication subsystem could support very small chunk sizes with low transfer startup costs. Likewise, an application with extremely high locality can take advantage of very large chunk sizes, maximizing the interconnect bandwidth. Another application with a very irregular access pattern may perform best with a small chunk size given the low inherent data locality.

![Figure 5.1: Effective Communication Bandwidth](image)

Using a small benchmark program, it is possible to model the bandwidth characteristics of the computing environment that will execute the application. As shown in Figure 5.1, communication efficiency improves as chunk size increases. Larger chunk
communications incur less communication startup cost and are better able to take advantage of the available interconnect capacity.

While communication efficiency encourages larger chunk sizes, chunk packing efficiency is often higher with smaller chunk sizes. Figure 5.2 shows the chunk packing efficiency over a range of chunk sizes and on a variety of processors with the Barnes-Hut application. As both processor count and chunk size increase, the chunk packing efficiency decreases.

Selecting a good chunk size can be done by combining the static chunk packing efficiency statistics with performance characterization data from the execution environment. As shown in Figure 5.3, 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. When these analytical results are compared with the actual results from Chapter 3 shown in Figure 3.5, we can see that the chunksizes with the highest performance are similar to those shown in Figure 5.3.

![Figure 5.3: Effective Bandwidth Utilization](image)

5.3.2 Data Distribution

Understanding the efficacy of the data distribution can be informed by both examining the number of remote accesses and also the number of misses in the chunk cache. The number of remote accesses relative to the total is especially useful if the parallel tree computations work largely on independent data. If processes should be
working on local portions of the tree structure, a large number of remote accesses would indicate a poor data distribution.

More importantly than the total number of references made to nodes which reside on remote processes is the number of times that remote chunks must be fetched into the local chunk cache. Each cache miss causes a communication event which must block progress until the cache miss is serviced. Looking at the global cache miss statistics is useful in understanding how much runtime is spent in blocking communication. The per-process cache miss counts can help identify specific imbalances in the data distribution.

### 5.3.3 System Performance

One of the key factors to keep in mind when running a GT or GCL based application is the memory usage. The basic memory footprint is given by the size of the shared heap, chunk cache, global context, and chunk directory. In addition, at least one chunk allocator/node group will be needed, possibly more. This footprint does not include any application memory requirements for stack or heap data. Care must be taken to set the parameters with respect to the execution environment. Since much of the memory used is allocated at initialization time, setting memory limits too large may cause the application to begin paging and reducing performance. The shared heap size and cache size can be set either at compile-time or runtime. The chunk directory size is determined by compile-time limits on the maximum number of processors and chunks allocated per-process.

Typically, runs on smaller numbers of processors will require larger shared heap segments for data, but can be configured with a smaller chunk directory structure.
Larger runs will need less room for shared data, but may need larger chunk directory structures and larger chunk caches. Selecting values too small for shared heap size or the chunk directory will result in runtime exceptions with relevant error messages. Selecting too small of a value for the chunk cache will result in poor performance, albeit correct execution.

The chunk cache may be the source of performance problems if it is sized too small. The number of cache replacements made is printed along with the other data access statistics. In general, the cache should be as large as possible while still providing enough memory for the program to run. Large numbers of evictions indicate a loss in performance due to cache size.

5.4 Data Visualization

Global Trees provides some basic tools for visualizing the global tree data structure. At any point during the program, a GT program can call a special routine, `gt_write_tree()`, which will write the tree connectivity structure to a file. This function takes a GT node group and a global pointer which can point to the root node, or any other node at the top of a subtree of interest.

Visualizing these data structures can be a difficult task, given their size and shape. Consider the octree formats used by Barnes-Hut and MADNESS, at a depth of four hops away from the root node, the tree potentially has over four thousand nodes and grows by a factor of eight at each subsequent level. Traditional, hierarchal views of the tree data quickly become unwieldy and difficult to understand.

To address this, the tree representation is stored in the popular GraphViz format, commonly used for representing tree and graph formats [39]. This representation can
be navigated dynamically, with tools such as the Zoomable Visual Transformation Machine [75] and paired with GT specific data for better understanding program behavior. GraphViz supports multiple layout algorithms for displaying the tree data, which can be especially useful with large datasets [38]. The octree shown in Figure 5.4 is of a spatial octree from Barnes-Hut displayed in a circular arrangement. This layout, with the root node in the center, gives a quick, intuitive view of the irregularity in the data and can be zoomed and navigated for more structural detail.

While understanding tree structure is useful in and of itself, Global Trees provides the ability to process runtime characteristics of the tree structure and overlay these as annotations on the visual data structure.

5.4.1 Owner-Labeled Global Tree Visualization

Figure 5.5 shows a small tree with nodes colored to reflect ownership of the actual node data. To collect this information, a simple program performs analysis of the global pointer data captured with the tree structure and uses it to color the tree data for each unique process. This information may be useful in understanding the cause of excess communication from chunk cache misses if the data access pattern is well understood, but the runtime data layout is not.

5.4.2 Profile-Labeled Global Tree Visualization

In addition to ownership information, GT also provides a means for coloring tree nodes based on a data reference profile captured in conjunction with the tree structure data. Capturing data reference information happens using specialized API routines to mark the beginning and end of a data reference profiling phase. The data reference trace records every data access performed during the profile phase, both for reads
and writes. This information is used to determine the “hottest” nodes within the tree structure.

An example of this display is shown in Figure 5.6. The blue nodes are “cool” nodes, which do not constitute a substantial number of accesses. As the number of times a tree node was referenced increases, the node changes color from blue to red. The red nodes are the “hottest” nodes, which are referenced most frequently. The profile coloring is done by sorting the nodes in the tree relative to their access count. The most heavily accessed node is colored red, then all other nodes are colored by their access count relative to the maximum value.

It is important to note that writing the tree structure happens at a specific snapshot during the program execution, whereas the profile is gathered over a period of time. The tree structure may change over time, in which case some data references may not correspond to nodes contained in the snapshot. The visualization software eliminates these from the display, but prints a message if it finds references for unseen tree nodes.
Figure 5.4: Large Octree Visualization
Figure 5.5: Owner-Labeled Global Tree

Figure 5.6: Profile-Labeled Global Tree
CHAPTER 6

Related Work

There is a great deal of literature regarding the implementation of parallel applications which are based on irregular data structures. The related work can be broken down into four primary areas: parallel programming models, particularly the partitioned global address space (PGAS) model; support for parallel linked data structures; techniques to model and improve data locality; and parallel program performance analysis and visualization.

6.1 Programming Models for Global-View Distributed Data Structures

The topic of supporting linked, or pointer-based, data structures on distributed memory systems has been addressed in the literature through, roughly speaking, three mechanisms: Software Distributed Shared Memory (DSM), Distributed Shared Object (DSO) systems, and Partitioned Global Address Space (PGAS) models. With the Global Chunks and Global Trees frameworks, 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 [57, 58, 56], Cashmere [83], Beehive [82], Shasta [78], Blizzard [79], AURC [49], Cid [70], CRL [53], and Midway [13, 14, 12].

There are several key differences between DSM and Global Chunks/Global Trees. 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 the GCL, the impact of false sharing is minimized through a tunable chunk size parameter, locality conscious chunk packing, and a relaxed consistency model. Additionally, under the 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 such systems, one is not able to leverage locality optimizations and locality-driven scheduling on such systems. Our system 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 [1, 22], Emerald [54], Charm++ [55], CHAOS++ [25], and Orca[7, 8] are systems that permit sharing of data objects through replication and object migration. These systems 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. However, a fundamental limitation of many of these systems is that they typically cannot handle the large scale data-structures (exceeding main memory of a single node). More importantly, a majority of them do not expose a data structure view wherein locality and load balance can be improved via application-specific adaptation. Lastly, while our runtime also uses a global namespace to reference data objects that are dynamically created it differs fundamentally from these systems because it uses coarse-grained data movement to enhance locality and improve communication efficiency.

PGAS parallel languages including Co-Array Fortran [71], UPC [86], and Titanium [91] all face the fundamental problem of providing efficient access to global shared data structures. In addition to the cost of moving data between processors there is the overhead of detecting whether each reference is local or global. With Co-Array Fortran, the problem of compiling for efficient communication is somewhat simpler than for UPC or Titanium because the programming model involves explicit partitioning of arrays by process, so that each global data reference explicitly specifies the process involved. The partitioned view of the program however limits the generality of the Co-Array Fortran model. UPC and Titanium have a very general model of shared data, but determining the processor and location is required for each global dereference. In order to achieve high efficiency, there is a significant burden on the compiler to transform global references to local references at compile time, if possible. Thus current work on these global address space languages faces several challenges, such as: generalized access to shared data via pointers limits the ability
to bundle communications; references must be checked at runtime to determine if
pointers are local or global (an issue with dynamic distributed data structures built
from pointers as the only primitive), reliance upon programmer specified algorithms
for tree traversal or only randomization for load balance; lack of detailed performance
modeling into either compile-time or run-time analysis. The Global Chunks approach
alleviates several of the challenges currently faced in compiling global address space
programming models for efficient parallel execution.

Other PGAS one-sided communication systems including ARMCI [67], GASNET
[15], SHMEM [9], and MPI-2 [66] provide a partitioned global view of shared data.
Under these models, a global pointer is the tuple \(p, a\) 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 our system borrows from DSM and DSO, namely bulk data movement
and global namespace, can greatly enhance the performance and communication ef-
ficiency of distributed linked structures under these models.

6.2 Support for Parallel Linked Data Structures

Many existing approaches support shared, distributed linked data structures by
mapping them into more regular structures, such as sparse arrays. Efficiently sup-
porting block-sparse arrays on distributed memory computers shares many of the
challenges of linked data structures, and has been studied by previous members of
our research group especially with respect to data movement and load balance [60, 59].
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 [5, 6] library supports distributed chunked trees and exploits the characteristics of spatial decomposition trees to achieve good performance. Olden [76] supports pointer-based linked data structures on distributed memory clusters by migrating the computation when an access is made to non-local data. Olden was also adapted to perform data caching similar to Global Chunks, but heavily relied on computation migration [20].

N-body simulation on large-scale parallel systems was implemented with the ChaNGa framework[52]. This system focuses on n-body simulation exclusively and is able to achieve very high scalability by relying on application-specific coarse grained data movement similar to the general technique provided by Global Chunks.

6.3 Data Locality Improvement

There is a great deal of literature devoted to improving data locality and program performance. Much of the previous work has focused on data references from sequential programs, specific applications, or other runtime systems. Our contributions to improving locality within the Global Trees framework address novel data movement issues that arise when operating under the PGAS programming model.

Application-specific techniques have been written about extensively. Custom allocation strategies have been shown to improve performance in shared memory data mining applications as described in [72]. The inspector/executor model has been shown to be effective with irregular SPMD scientific applications including common
molecular dynamics codes [33]. Space-filling curves have also been applied to parallel irregular applications in order to improve memory performance using data and computation reorderings [64]. Some work has been done to automate data locality improvement at the system-level, but has mostly focused on adapting coherence mechanisms and cache management options for distributed shared memory systems [11].

Work by Chilimbi has shown that data reference profiles are often stable over multiple program executions for a variety of applications [28]. This is a key result which motivates the generality of the locality optimizations described in Chapter 4. Other work such as tree orientation analysis [4] and predicting locality phases for memory optimizations [80] could also be used to extend and enhance the described technique.

Much of the prior work has focused on modeling and characterizing data locality optimizations using data reference traces on sequential programs. Chilimbi has published extensively on the use of hot stream representations of program data locality [27]. The algorithm used to compute these patterns of frequently occurring data references is $O(n)$ in the size of the data reference trace, whereas the technique outlined in Chapter 4 is dominated by an $O(n \log n)$ sorting step in the size of the data structure. Ding has used the concept of a data reuse distance to characterize program data locality [37] and has described (along with Chilimbi) a composable model for analyzing locality in multi-threaded programs [36]. In general, the problem of finding an optimal algorithm for data placement and code arrangement that minimizes the number of cache misses has been shown to be NP-Hard. It has also been shown that efficient approximate algorithms produce sub-optimal mappings relative to an optimal solution [74].
These data locality models and characterizations have been used to create online systems which simulate layouts from profile data [77] and instrument the program dynamically to prefetch data based on the profile [29]. Since garbage collection is often relocating memory, it has been the subject of some work to include locality-aware techniques that improve data access [30] as well as other systems which relayout data during collection according to the dominant access pattern during a phase of computation [92].

Other approaches involve making locality improvements at allocation time, rather than through a layout process. Allocating related data together using enhanced pool allocations was described first by Lattner [62] and later extended to utilize hot stream information [31].

6.4 Performance Analysis and Improvement

There are a wide variety of approaches in understanding parallel program performance. While many of the existing tools below are useful in conjunction with the GT performance tools, a unique property of most GT applications is the irregularity of the data structure and the difficulty of understanding the tradeoffs between load imbalance and data distribution.

Capturing profile information for large-scale parallel applications can be challenging due to the volume of data. The scalable log format, SLOG, provides a drawable log format, which uses a hierarchy of bounding boxes to allow for efficient time-based access to large-scale profile data [24]. The GASP framework provides a set of callbacks which are inserted by a compiler system to track events of interest in PGAS-based applications, such as remote variable accesses, get and put operations, barriers, fences,
etc [84]. Since GT/GCL supports these operations through explicit function calls, much of the same data can be gathered automatically by the GT runtime.

There are a number tools and frameworks for visualizing program performance in PGAS programming environments. Jumpshot is a zoomable viewer which reads the SLOG format that enables browsing of program events (such as communication) over the course of a program run [89]. The Parallel Performance Wizard (PPW) can be used to analyze programs written in UPC and is built on top of the GASP framework [63]. It can display information using Jumpshot, as well as many UPC-specific calls and communication events. Global Arrays provides tools to visualize the access contention of data elements in GA programs [69]. Mohr, et. al. have also studied performance analysis of Co-Array Fortran (CAF) programs [65].

Other established and powerful tools exist for performance analysis of other programming models. VAMPIR is a robust and well-established toolkit for the performance analysis of MPI programs [16]. Likewise, the TAU framework can help developers perform extensive performance analysis of multi-threaded programs [81].
CHAPTER 7

Future Work

There are many opportunities to extend and enhance the work presented in this dissertation. In particular, there are several avenues of promising research available which adapt and improve on existing techniques to improve the element-to-chunk mappings. The layout computation is a modular step which could be improved by incorporating elements from other successful modeling techniques [27, 37] or advanced partitioning mechanisms [23]. In addition to incremental enhancements of existing contributions, there are three areas of interest which form a basis for future work.

7.1 Global Chunk Layer

To date, the Global Chunk Layer has been used exclusively with Global Trees. This system is low-level enough to be of use in a wide range of applications, but provides a higher-level of functionality than typical PGAS one-sided communication layers. Much of the functionality present in the GCL is often re-implemented to achieve good performance in PGAS applications. For example, the UPC version of the Barnes-Hut application was modified to add caching and bulk data movement from the original SPLASH-2 code, whereas the GT version was only adapted to use GT get/put operations.
The flexibility provided by Global Chunks makes it a good candidate for supporting a wide range of irregular applications, such as block-sparse arrays, graphs, and other structured irregular data. Earlier work using a similar approach for handling block-sparse data has been shown to be effective with the bulk data movement style provided by the GCL [59].

7.2 Locality-Aware Load Balancing

Both Global Trees and Global Chunks can take advantage of techniques which balance the irregular workload and improve data locality, but it is a challenge to do both at the same time. Currently, support for true locality-aware load balancing in Global Trees is limited to the built-in tree traversals. In general, tasks will have affinities to certain processes depending on the data that is needed for the computation. Using this affinity information to assign costs to tasks may be useful in balancing workload and minimizing communication.

Within the co-developed Scioto framework [35], work-stealing happens using a locality oblivious approach. The stealing operation in Scioto may be enhanced from a simple one-sided communication to an active message which can evaluate the candidate tasks to steal and select those which have greater data affinity to the process making the steal.

Similar to Olden [20], GCL/GT could be extended with robust support for explicit task migration using active messages. In particular, the chunk global view is useful for iterating over local data, but problems arise when data dependencies occur involving remote elements. These could be addressed by migrating a portion of the computation to the target process, then resuming the dependent computation upon completion.
7.3 Hierarchical Run-time System Support

Lastly, the current implementation makes no distinction for remote data between different cores, processors or nodes. As shown in Figure 4.1, operations such as remote node allocation have very different costs depending on where the allocation actually needs to be performed. Extending GCL and GT to be aware of this hierarchy and the actual topology and structure of the cluster system can lead to higher performance and economies of scale.

For example, processes which are able to share memory on the same cluster node could also share chunk directory information, directory caches, and chunk caches. While some additional synchronization overhead may be needed to support the sharing of these resources, there are opportunities for greater efficiencies in data access, memory footprint, and other runtime characteristics.
CHAPTER 8

Conclusion

This dissertation has presented a framework which efficiently supports global view programming with linked data structures on distributed memory systems. This system takes advantage of the partitioned global address space model to provide the convenience of fine-grained global data access, while internally using coarse-grained data movement for communication efficiency. Experimental results have validated this approach, providing a system that scales well and provides flexibility as well as good overall performance.

Providing support for fine-grained access with coarse-grained data movement has exposed additional opportunities for improving data locality and understanding program performance. These opportunities have been leveraged to provide algorithms which reduce communication costs and lower the runtime overhead by improving the element-to-chunk mapping of data.

The Global Chunks and Global Trees systems provide a basis for ongoing research into supporting global view programming of irregular data structures. These include support for unstructured trees and graphs, deeper understanding of program performance, and improving data locality and performance through the use of hierarchical systems which are aware of the different levels of the system architecture and memory hierarchy.
BIBLIOGRAPHY


