

Implementation of a Binary Tree Driver (OAKc) in Cactus

Jeff DeReus
Computer Science
Center for Computation
& Technology,
Louisiana State University
302 Johnston Hall
Baton Rouge, LA 70803

Faculty Advisor: Dr. Gabrielle Allen & Yaakoub Y El Khamra

Abstract

Cactus Code is an open source framework designed primarily for scientists and engineers, in which the core or “flesh” acts as a central unit to which external modules or “thorns” can interface. Cactus is used on many different architectures and can be used to implement different codes from various disciplines. While initially developed for solving problems in numerical relativity, it has since expanded to include thorns for computational fluid dynamics, climate modeling and bioinformatics.

Binary trees are fast insert and lookup recursive data structures with at most two children at each node. Many different applications use binary tree structures for the efficiency they provide; including high performance databases, visualization hierarchies, discrete mathematics, Monte Carlo simulations, logic programming and computational econometrics. This paper discusses the design and implementation of binary trees in Cactus, which will then provide capabilities for new application domains. The core module discussed here is special in that it can allocate binary tree nodes for other thorns to use, in effect realizing the role of a Cactus driver thorn. Due to the high computational loads involved in most real world problems, one basic requirement of this binary tree driver is parallelism to make use of high performance computing environments.

Keywords: Cactus, Binary Search Tree, Binary Tree Driver, Parallel Trees, Computational Fluid Dynamics, Unstructured Mesh, Galaxy Formation Modeling, N-body Problems

1.0 Introduction

By implementing a parallel binary tree driver in Cactus, we wish to accomplish several goals, primarily to solve very complex problems by utilizing the speed and efficiency of binary search trees. We also implement a BST driver to develop a driver that is extensible for many different kinds of applications to plug into. These will be Cactus thorns that are used for several different areas of high-level computation. Among these will be thorns for applications involving: N-body problems, k-d trees, causal sets, discrete quantum gravity problems, particle methods, fast multipole methods (CFD), genetics, data mining, and statistics.

One of the more difficult areas to address is not the fairly straightforward implementation of a binary search tree, but rather, the use of parallel processors to further increase the amount of data that can be properly analyzed in a reasonable amount of time. As a case in point, for N-body problems (example: galaxy formation modeling), there are N^2 forces acting in the area in question. The amount of forces grows exponentially and quickly beyond the capabilities of serial programs to process in a reasonable amount of time, if at all.

“A galaxy might have, say, 10^{11} stars. This suggests that 10^{22} calculations have to be repeated. It would require significant time on a single-processor system. Even if each calculation could be done in $1 \mu\text{s}$ (10^{-6} seconds, an extremely optimistic figure, since it involves several multiplications and divisions), it would take 10^9 years for one iteration using the N^2 algorithm and almost a year for one iteration using the $N \log_2 N$ algorithm. The N-body problem also appears in modeling chemical and biological systems at the molecular level and takes enormous computational power.”⁴

2.0 Introduction to Binary Trees

Most programmers are at least passingly familiar with the data structure known as a binary tree. Binary trees are defined as data structures in which each node may have at most two children. These are typically labeled *left* and *right* children. Each node of a binary tree also contains pointers to its children, to be used in the traversal and searching operations. The population of a binary tree imposes no order on the values being assigned to the nodes. Data is simply read in and assigned from left to right as the tree is traversed. Without a key value to act as a parameter for population, there is not an ordered build of the tree. The node that begins the tree is known as the root; the nodes that sit at the ends of the tree's branches are usually called the leaves. With a simple binary tree structure with no ordering of the nodes by value the time to search the tree becomes $O(1)$ because in the worst-case scenario, each node would have to be searched to examine if it contains the key value.

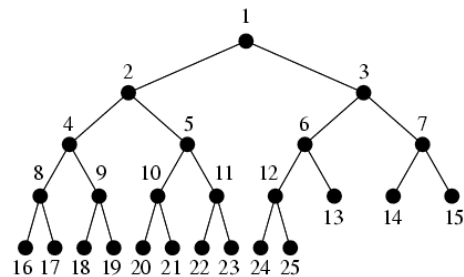


Figure 2. A complete binary tree

Binary search trees (BST), while structured in the same way, utilizes key values to order the data upon population of the tree with larger data values being assigned to the right child and lesser to the left. To find a particular node, the binary tree is traversed from the root. At each node, the desired key is compared with the node's key: if they don't match, one branch of the node's sub tree or another is selected based on whether the desired key is less than or greater than the node's key. This process continues until a match is found or an empty sub tree is encountered. Due to this structure, search times for any particular node are dramatically decreased compared to linear data structures. This binary search structure automatically lends itself to very fast and efficient searching due to the ability to search only one half of the tree with each recursion. This becomes increasingly important with the ever-growing sizes of data sets to be analyzed. Simple binary trees, although easy to understand and implement, have disadvantages in practice. If keys are not well distributed, the tree can become quite asymmetric, leading to wide variations in tree traversal times. Parallelism will help to remedy this situation by dynamic tree division based on the overall build structure, while still keeping initial order of data intact, if desired.

The gain in performance is not directly attributed to dividing trees onto several processors. Instead, by dividing trees onto separate processors (P) we obtain several trees that are of a lesser depth ($\log_2 N / P$) than one tree with all nodes combined ($\log_2 N$). Another benefit of parallel trees is the RAM (random access memory) requirements to store the data on the processor itself. For instance, if one tree contains data equivalent to dozens of gigabytes of RAM, other than shared memory architectures, there are no machines out there that can hold a structure larger than that in RAM. Dividing the memory requirements over many processors solves this.

As we shall see, the binary tree also lends itself to fairly simple extensibility to more complicated structures. This may include, but is not limited to heaps, graphs and n -node trees. With the extension of the initial structure comes an even further increased ability to analyze more complex problems. We will discuss some of these examples later in the paper.

3.0 Cactus

The Cactus Code Framework (Cactus), <http://www.cactuscode.org>, is an open source environment designed primarily for scientists and engineers. Cactus is based on a framework design in which the core or "flesh" acts as a central unit to which external modules or "thorns" can interface. These thorns can be written in Fortran 77/90 and C/C++. Integration of these thorns in the framework is achieved by specifying a schedule, an interface, and a parameter file.

The flesh technically has no functionality to speak of, but instead provides the main program. It parses parameters and controls specific thorns, which are passed control as necessary. Utility functions are included in the flesh to determine information about variables and determine which thorns have been passed into the framework.

The thorns comprise most of the actual computation of the Cactus framework. Calls from different thorns are passed through the flesh and onto each other. Configuration files are used to inform the flesh which thorns are active. Upon running the configuration files through a Perl based interpreter, code is generated to bind the different thorns to the flesh.

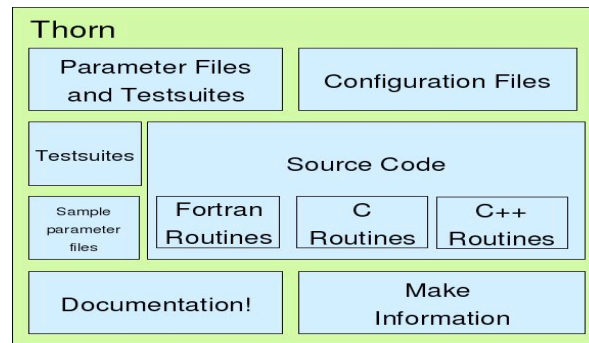


Figure 1. Typical thorn structure

Due to its design as a modular framework, Cactus has become a system in which computational scientists can tailor their personal choice of thorns in order to customize their runs to environments most fitting to solve the problem. With this organization, the flesh acts as the coordinator of the runs and the thorns plug in as extensions. Because of the structure of Cactus, thorns are able to communicate with each other, bypassing the language barrier inherent in some computational systems.

A standard list of available thorn toolkits is available for download from the Cactus web site (<http://www.cactuscode.org>) and are briefly described in Table 1.

Table 1. Standard thorns in the Computational Toolkit

Computational Toolkit	Thorns for standard capabilities such a parallelization, memory, utilities for I/O in various formats and computational steering
HDF5 Toolkit	Thorns needed for I/O using the HDF5 data format
PETSc Toolkit	Thorns for interacting with the PETSc library for solving elliptic equations
Web Browser Toolkit	Thorn that act as web servers. Users are able to steer computations via http requests. Output streams from running applications can be displayed as jpegs
CFD Toolkit	Thorns for solving problems in Computational Fluid Dynamics

By including such a standard set of thorns, it is not necessary for computational scientists to have an in depth knowledge of the individual technologies behind each thorn. Instead, they can concentrate on applying their problem to the solvers, decreasing the overhead of learning the technology needed to build their own applications.

“Note that Cactus is not an application in itself, but a development environment in which an application can be developed and run. A frequent misinterpretation is that one “runs Cactus”, this is broadly equivalent to stating that one “runs perl”; it is more correct to say that an application is run within the Cactus framework.”¹

4.0 Design Methodology

Initially, when building a stand-alone working code, the primary obstacle was the input of data from a file with unknown amounts and types of data. Using some rather atypical loops, we were able to allow for the user to explicitly assign the number of values and their types in each node of the binary tree. Once passed into the input function, these values were used to tokenize entire lines of data, with a two fold goal: to parse entire lines of data at a time and to zero out any excess values if they were missing either because of user error or there were simply not enough values in a specified data stream to fill all values of the node. This allows for some values to have had more relevant data associated with them. Situations such as this typically arise in the statistical fields where missing data is commonplace. A more effective method for parsing streams with missing data will be discussed later in the paper.

Typical tried and true recursive algorithms were used for all binary search tree data structures. These include the usual search, including searching for ranges of values, for deletion and insertion. The user is able to specify the ranges and types of the data to be searched. Situations like this can arise for computational econometrics applications for instance, where certain stock trends are of interest to the analyst. The only real difficulty was the modification of those algorithms to allow for custom data types to be stored in each node.

The item that differentiates this binary search tree driver (OAKc) from others is the fact that we allow for all nodes to

contain their own entire data structures. With this implementation we allow for most manners of data sets with as little future modification as necessary. This, however, adds complexity, as there are now an unknown number of data points that potentially need to be searched. And with that complexity goes the fact that over several processors different types and locations might need to be searched for analysis.

5.0 Prospective Applications

A driver of this sort can be used directly by computational fluid dynamics (CFD). Fast multipole methods require a binary tree data structure. Another prime candidate for OAKc is adaptive mesh refinement applications (AMR). By utilizing the nodes as distinct and individual 3-dimensional meshes, the tree structure is utilized for continued refinement as we traverse through the nodes.

Along the same line of thought, we can also utilize OAKc for the analysis of discrete quantum gravity problems and causal sets problems. Causal sets would require the additional functionality of posets, which simply put, allows for each node to know all of its descendants and ancestors. Posets also increase the difficulty of dividing the processes among processors. The difficulties of applying posets will be discussed in later sections.

Initially, OAKc would seem to be a simple implementation of a BST. This is because, in parallel, the efficiency of the search could still not achieve better results than $O(\log_2 N)$. But with parallel processing, speedup can be achieved. Even in galaxy modeling or adaptive mesh refinement, all nodes must be traversed. This is because the algorithm to solve the scientific problem requires us to calculate results from every node to find that particular answer. Galaxy modeling and adaptive mesh refinement require a complete search and then data extraction. Upon building new separate trees, these trees will be ported to other processors. In some cases, multiple traversals of the same trees would be required. These new trees will be specified by the user as a “real” tree residing on a new processor, or a “virtual tree” that will simply pass the addresses of the information to the new processor. The advantage of allowing for a choice is that for smaller problem sets, communication overhead between processors will not become prohibitively large. With larger “real” trees, it quickly becomes more efficient to physically pass the information to the processor. Then only the results of the analysis need be passed back to the parent process.

There are currently no adequately developed platforms in Cactus for adaptive mesh refinement. OAKc’s structure, however, can be used to implement adaptive mesh refinement methods without having to write a stand-alone body of code. Once OAKc extracts the necessary data (or data of interest) smaller trees can be analyzed independently on separate processors using MPI calls to pass the new tree to that processor. Indeed, with a simple equation (or function) parameter, the BST can simply crawl recursively over the extracted trees and perform the required calculations. Parameters can be set for returning values in a desired range. Because of OAKc’s dynamic memory allocation capabilities, the final result of the computation can be added to a node. Additionally, as the results are returned in an array, with proper output formatting parameters, all relevant data can be returned in a formatted text file. With a general enough output function, this could be applied to many similarly structured output formats.

With the web interface capabilities of the toolkit included in a standard Cactus configuration, dynamic steering can be achieved. Dynamic steering is the ability to control the application via a web browser. Utilizing HTTPD thorn allows the user to modify the parameters in real time in the case of erroneous output while the job runs. Additionally, checkpoints can be set in case the application will not be able to complete before its time in the supercomputer allocated queue expires. If checkpoints are applied, the user can then reenter the queue and restart from the checkpoint location. Dual output formats, text and web page, will be returned to the remote user.

Similarly, using the HTTPD thorn makes formatting the output more uniform for different applications. Output parameters are passed to HTTPD and formatted accordingly for the problem being examined. Optionally, output functions could be written and passed to HTTPD in such a manner that the thorn will output accordingly. If implemented correctly, custom output can be generated for each different type of problem analysis.

6.0 Future Directions

OAKc at this time is a fully parallelized thorn residing in Cactus. As noted earlier, OAKc maintains the ability for the user to explicitly define the type of content residing in each node. The current structure of the driver allows for compilation with other thorns in the Computational Toolkit as well as application specific codes. This allows for differing types of applications to utilize OAKc. One of these types will include the functionality of posets. Posets further increase the difficulty of applying this structure to a parallel implementation. Simply put, as the number of ancestors increases, the amount of information stored in the nodes further down the tree hierarchy increases rapidly. This also requires much more information sharing between processors in order to access the information contained in the preceding nodes.

There are many ways in which we will increase the functionality of OAKc in the future one of the most important being the reorganization of the tree division techniques. It is desired to implement a dynamic tree division based on the initial tree organization and hierarchy. By dividing based on the initial structure, we will no longer need to parse the data into arrays

then populate the trees. While processing times are not of primary concern because of the speed of modern processors, with the larger data sets for N-body problems, a more efficient input function will be written. Currently, as mentioned earlier, all data streams are tokenized from data strings and initially placed in arrays. Only then are the arrays manipulated to create the tree data structure. With dynamic tree division, while still tokenizing the streams, the individual data structures are populated directly into a tree structure.

In later implementations, OAKc will be extended to implement graph structures. Graphs are a superset of binary trees without a single root node to “anchor” the tree. The difficulty of using graphs is the fact that there is no explicitly defined entry point at the root. This leads to the use of OAKc for solving problems on unstructured meshes, mortars for unstructured meshes, moving unstructured meshes and adaptive mesh refinement for unstructured meshes. If several separate sub-trees are created, when a vertex changes location, nodes can be added or deleted from the relevant sub-tree. This will result in a dynamic data structure, able to model real time changes. As a result, the HTTPD thorn will become more effective as a way to monitor the individual runs and load balancing of the unstructured mesh.

After a stable next stage driver has been completed detailed benchmarking will be performed. These benchmarks will reflect the performance differences of serial processing as compared to parallel processing. We will also compare benchmarking runs illustrating differences in performance numbers of “virtual” trees compared to “real” trees. In addition, we will analyze the overhead of processor communication based on the size and type of the problem being solved.

Also in future modifications of this driver will be the optimization of the overall code structure. Streamlining functions can further increase the performance numbers in an attempt to be able to solve applied problems more quickly. In addition, because there are many similarities in the search functions, modification to allow for a single function to perform any of these searches will be quite beneficial. It is also desired to design a better organized function to display the initial physical structure of the tree.

The parsing of streams with missing values will be redesigned in the future. Currently, any missing values are required to reside at the end of the data line. Allowing for missing values or similarly, streams with a lesser amount of values, while still being able to traverse over those locations will be added. This will greatly simplify the analysis of statistical applications.

7.0 Acknowledgements

The author would like to thank the following for their help and support:

First and foremost, Yaakoub Y El Khamra, supervisor and mentor, Gabrielle Allen, Assistant Director for Computing Applications, Tom Goodale, Chief Architect of the Cactus Code Framework, The Center for Computation and Technology, for helping with funds, and to my wife Lisa, for understanding the long hours.

8.0 References

1. T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel and J. Shalf, “The Cactus Framework and Toolkit: Design and Applications”, Vector and Parallel Processing - VECPAR '2002, 5th International Conference, Springer, (2003).
2. W. Gropp, E. Lusk, A. Skjellum, *Using MPI – Portable Parallel Programming with the Message-Passing Interface*, 2nd ed. (MIT Press, 1999).
3. W. Gropp, S. Huss-Lederman, A. Lumsdaine, et al., *MPI – The Complete Reference. Volume 2, The MPI Extensions* (MIT Press, 1998).
4. B. Wilkinson and M. Allen, *Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computers*, 2nd ed. (Pearson Education, Inc., 2005), 4.
5. R. Sedgewick, *Algorithms in C – Parts 1-4 Fundamentals, Data Structures, Sorting, Searching*, 3rd ed. (Addison Wesley Professional, 1997).
6. R. Sedgewick, *Algorithms in C – Part 5 Graph Algorithms*, 3rd ed. (Addison Wesley Professional, 2002).