Parallelization: Binary Tree Traversal

By Aaron Weeden and Patrick Royal
Shodor Education Foundation, Inc.

August 2012

Introduction:

According to Moore's law, the number of transistors on a computer chip doubles roughly every two years. First formulated in 1965, this law remains amazingly accurate as a rule of thumb for predicting how chips will change in the future. The implications of this have led to a major conceptual shift in the limitations of computing. Whereas originally the primary limiting factor in a computer was storage, today enormous amounts of data storage are available at relatively low cost. As a result, storing all of that data is becoming less of a problem than analyzing and using it.

The most basic tool of data analysis, and the subject of this module, is to search a set of data for a particular value. On the face of things, searching might seem very simple – just look through bits of data one by one until the correct one is found. However, as computers store more and more information this sort of linear search becomes more and more unwieldy. If you wanted to search 1 billion items, for example, on average you would have to examine 500,000 of them. Instead, we can use something called a **Binary Search Tree** to organize the data more effectively. Once we write the data into a binary tree, we can search through, for example, 1 billions items and pick out the one we want in just 30 actions, almost 17 million times less work.

A binary tree is essentially an ordering of data into interconnected nodes. The "parent" root node of the tree contains a single data entry and references to two "child" nodes. Each of these nodes is then effectively a root of its own subtree, with a data entry and references to two more child nodes, and so on. Due to the nature of the tree, a tree with *n* nodes has a **height** (the number of **levels** in the tree) of $\log_2 n$, assuming the tree is fully populated, which we will assume throughout this module. The main advantage of this structure is that, in a properly sorted tree, searching it only requires looking at one node from each generation, so it is possible to search through 2^n-1 nodes in just *n* actions. Figure 1 shows a picture of a binary search tree.

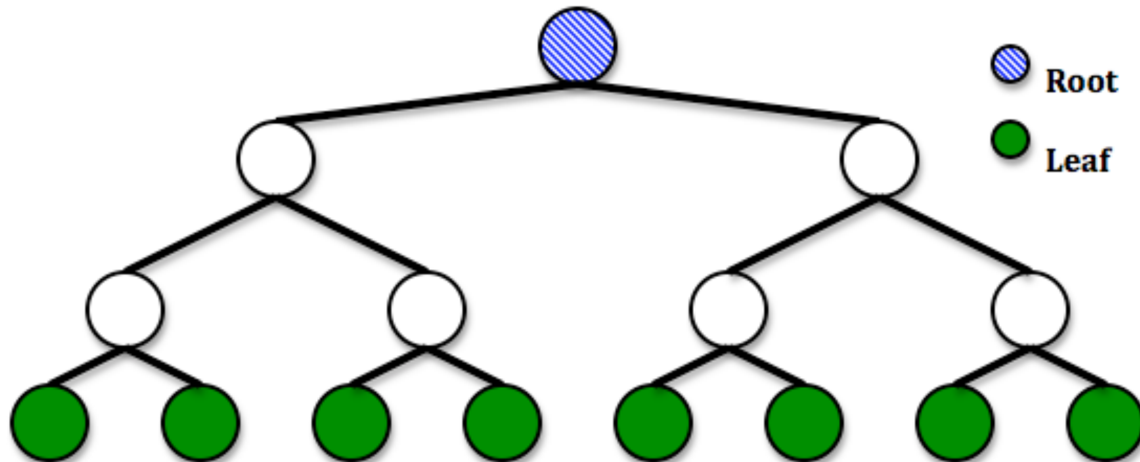


Figure 1: A binary search tree of height 4 with $2^4 - 1 = 15$ nodes.

Types of Traversals:
There are a number of orders in which a processor can **traverse**, or visit tree nodes, exactly once, in a specific order, in a binary tree. Some of the most common ones are listed and described in the table below.

Traversal Type	Description	Real-World Use
Pre-order (depth-first)	Examine the root first, then the left child, then the right child	Creating (writing) a new tree
In-order (depth-first)	Examine the left child, then the root, then the right child	Searching a sorted binary tree
Post-order (depth-first)	Examine the left child, then the right child, then the root	Deleting an existing tree
Breadth-first	Examine each level of the binary tree before moving to the next	Finding the shortest path between two nodes

Pre-Order Traversal:

A pre-order traversal of a binary tree is a recursive algorithm where the program first prints the label of the root of the tree. It then sends the same command to the left and right subtrees. An implementation of this algorithm in C is as follows:

```
void preordertraversal(struct node * root) {
	if(root != NULL) {
		printf("%d", root->label);
		preordertraversal(root->left);
		preordertraversal(root->right);
	}
	return;
}
```

Post-Order Traversal:

A post-order traversal of a binary tree is a recursive algorithm where the program first invokes itself on the left and right subtrees, traversing them completely, and then prints the label of the root. When the traversal of the left or right subtree reaches a node with no children, it prints the label of that node. An implementation of this algorithm in C is as follows:

```
void postordertraversal(struct node * root) {
     if(root != NULL) {
          postordertraversal(root->left);
          postordertraversal(root->right);
          printf("%d", root->label);
     }
     return;
}
```

In-Order Traversal:

An in-order traversal of a binary tree is a recursive algorithm where the program first invokes itself on its left subtree, then when that finishes prints the root, and finally invokes itself on its right subtree. An implementation of this algorithm in C is as follows:

```
void inordertraversal(struct node * root) {
    if(root != NULL) {
        inordertraversal(root->left);
        printf("%d", root->label);
        inordertraversal(root->right);
    }
    return;
}
```

Breadth-First Traversal:

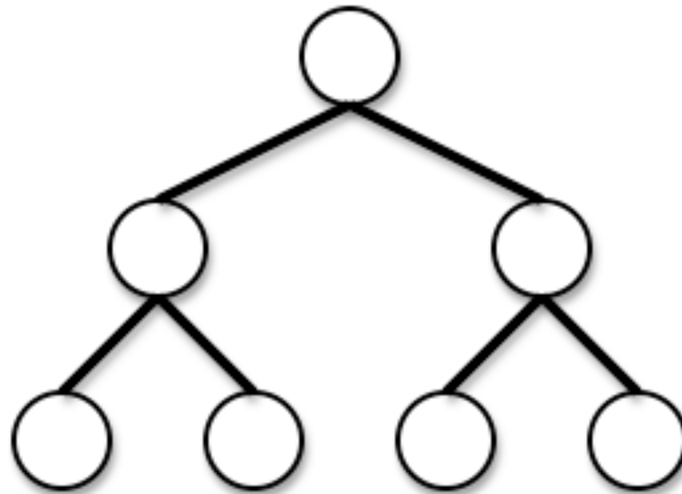
A breadth-first traversal works differently from a depth-first traversal. The program loops downward through each level of the tree, pushing all of the nodes at each row in the order in which they appear in the row. It does this by using two stacks: one to hold the nodes to be printed, and one to hold their children. An implementation of this algorithm in C is as follows:

```
void breadthfirsttraversal(node * root) {
     stack activenodes;
     stack nextnodes;
     activenodes.push(root);
     while(activenodes.peek() != NULL) {
          while(activenodes.peek() != NULL) {
               node * current = activenodes.pop();
               printf("%d", current->label);
               nextnodes.push(current->left);
               nextnodes.push(current->right);
          }
          while(nextnodes.peek() != NULL) {
               activenodes.push(nextnodes.pop());
          }
     }
     return;
}
```

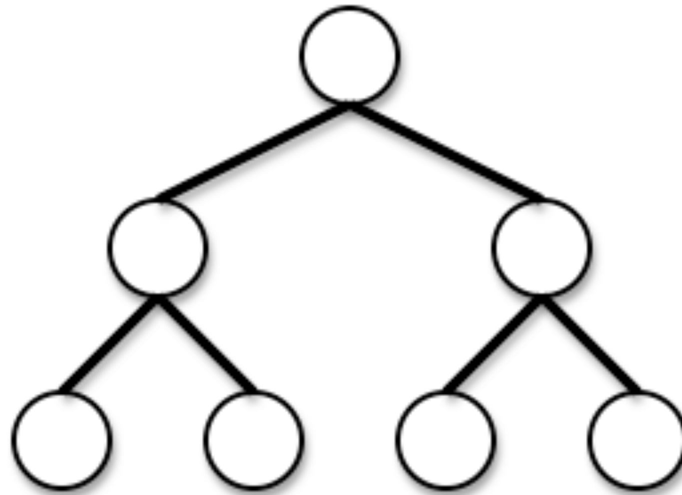
Exercise 1

Complete questions 1 through 4 below.

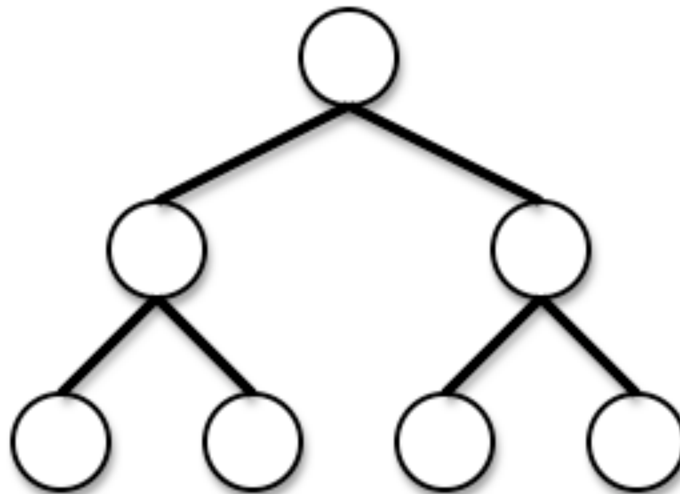
1. Label the nodes of the tree below 1 through 7 in the order they are processed by a **pre-order traversal**.



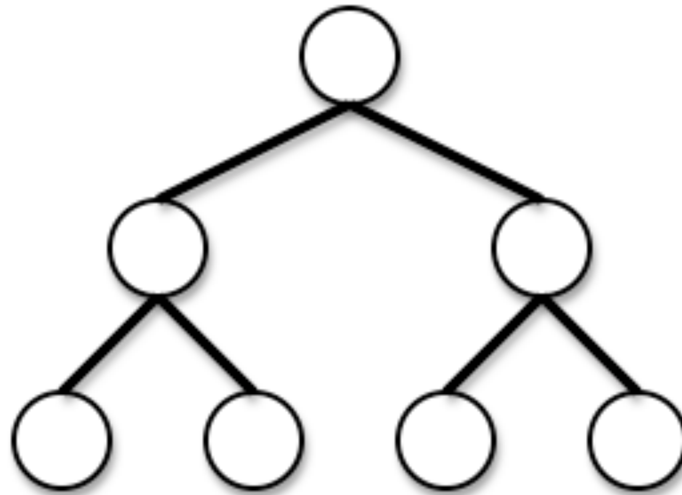
2. Label the nodes of the tree below 1 through 7 in the order they are processed by a **post-order traversal**.



3. Label the nodes of the tree below 1 through 7 in the order they are processed by an **in-order traversal**.



4. Label the nodes of the tree below 1 through 7 in the order they are processed by a **breadth-first traversal**.



Exercise 2
Follow the steps below on a computer, from the command line. Example commands are shown below each instruction.

1. Change directories to the “code” directory (an attachment to this module).

   ```
   cd BinaryTreeTraversal/code
   ```

2. Compile the code.

   ```
   make traverse
   ```

3. Run the pre-order program and compare it to your result from Exercise 1 Question 1.

   ```
   ./traverse --order pre
   ```

4. Run the in-order program and compare it to your result from Exercise 1 Question 3.

   ```
   ./traverse --order in
   ```

5. Run the bread-first program and compare it to your result from Exercise 1 Question 4.

   ```
   ./traverse --order breadth
   ```

Parallelism and Scaling

The current setup works well enough on small amounts of data, but at some point data sets can grow sufficiently large that a single computer cannot hold all of the data at once, or the tree becomes so large that it takes an unreasonable amount of time to complete a traversal. To overcome these issues, **parallel computing**, or **parallelism**, can be employed. In parallel computing, a problem is divided up and each of multiple processors performs a part of the problem. The processors work at the same time, in parallel.

There are three primary reasons to parallelize a problem. The first is to achieve **speedup**, or to solve the problem in less time. As one adds more processors to solve the problem, one may find that the work is finished faster[1]. This is known as **strong scaling**; the problem size stays constant, but the number of processors increases.

The second reason to parallelize is to solve a bigger problem. More processors may be able to solve a bigger problem in the same amount of time that fewer processors are able to solve a smaller problem. If the problem size increases as the number of processors increases, we call it **weak scaling**.

The third reason to parallelize is that it allows a problem that is too big to fit in the memory of one processor to be broken up such that it is able to fit in the memories of multiple processors.

In many ways, a tree is the perfect candidate for parallelism. In a tree, each node/subtree is independent. As a result, we can split up a large tree into 2, 4, 8, or more subtrees and hold subtree on each processor. Then, the only duplicated data that must be kept on all processors is the tiny tip of the tree that is the parent of all of the individual subtrees. Mathematically speaking, for a tree divided among n processors (where n is a power of two), the processors only need to hold $n - 1$ nodes in common – no matter how big the tree itself is. A picture of this is shown below in Figure 2. In this picture and throughout the module, a processor's **rank** is an identifier to keep it distinct from the other processors.

---

[1] An observation known as **Amdahl's Law** says that there is a theoretical limit to the amount of speedup that can be achieved from strong scaling. At some point, the number of processors exceeds the amount of work available to do in parallel, and adding processors results in no additional speedup. Worse, the time spent **communicating** between processors overwhelms the total time spent solving the problem, and it actually becomes slower to solve a problem with more processors than with fewer.

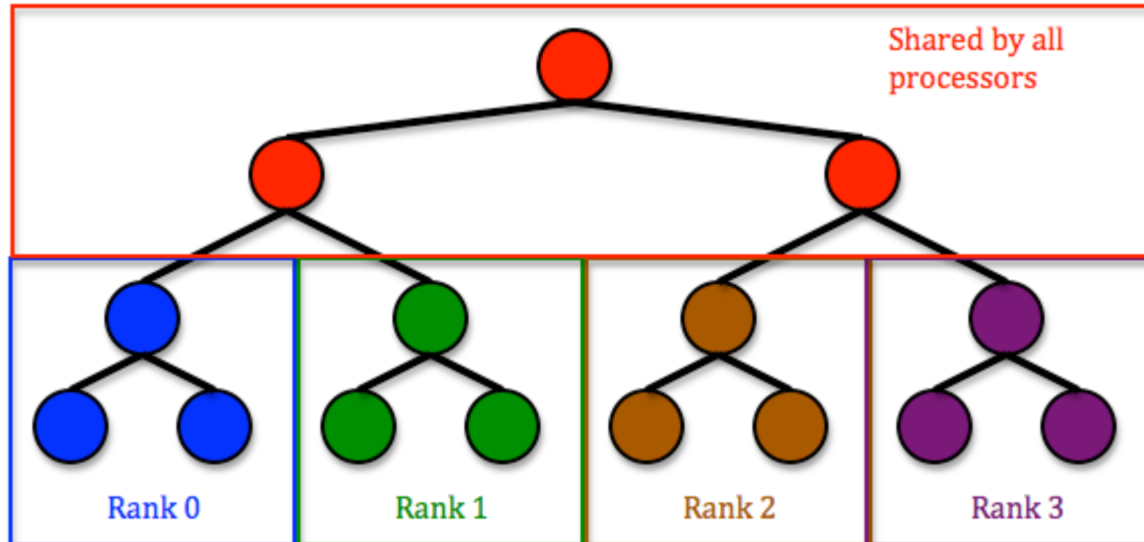


Figure 2: A tree of height 4 split among 4 processors

Parallel Traversals:
The fact that trees are comprised of independent sub-trees makes parallelizing them very easy. Properly done, the portion of these traversals that is parallelizable grows at 2^n for an *n*-generation tree, while the processors only need to synchronize once, at the end, so it approaches 100% for large trees (but keep in mind Amdahl's Law, footnote 1). The basic steps for parallelizing these traversals are as follows:

1. Perform the traversal on the parent part of the tree.
2. Whenever you get to a node that is only present on one processor, ask that processor to execute the appropriate C algorithm detailed above.
3. The processor will return its result that can be used exactly as if it was a serial processor.

A Breadth-First traversal is somewhat more complicated to implement as a parallel system because at each level, it must access nodes from all of the parallel processors. Theoretically, a Breadth-First traversal can achieve the same 100% speedup of Pre-, In-, and Post-Order traversals. However, the amount of processor-processor data transmission adds in a greater potential for delays, thus slowing down the algorithm. Nevertheless, as the size of the tree increases the size of the generations grows at the rate of 2^n while the number of synchronizations grows at a rate of *n* for an *n*-generation tree, so the parallelizable portion of these traversals also approaches 100%. The basic steps for parallelizing this traversal are as follows:

1. Perform the traversal on the parent part of the tree.
2. Whenever you get to a node that is only present on one processor, ask that processor to execute the Breadth-First C algorithm detailed above, but **wait** after it finishes one generation.
3. Combine all the one-generation results from the different processors in the correct order.
4. Allow each processor to execute the next generation of the Breadth-First C algorithm detailed above, and then wait again.
5. Repeat Steps 3 and 4 until there are no nodes remaining.

Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a standard for performing the type of distributed memory parallelism described above, in which each processor has its own memory. With MPI, parallel processes must communicate with each other by sending messages, as in the case of the breadth-first search algorithm above, in which processes must synchronize with each other in step 4 before continuing. MPI provides bindings in C, Fortran, and a host of other languages. When MPI processes are invoked, they each run the same copy of source code, in which MPI functions are written. Some of the common MPI C functions are shown in the table below.

Function	Description	Description of Arguments
`MPI_Init(&argc, &argv);`	Initialize the MPI environment and strip out the relevant MPI command line arguments. No other MPI functions may be called before this function.	`argc` – the number of arguments passed to the program on the command line. `argv` – the arguments passed to the program on the command line.
`MPI_Comm_rank( MPI_COMM_WORLD, &rank);`	Obtain the rank of the current process.	`MPI_COMM_WORLD` – the communicator (group) of MPI processes working together. `rank` – the rank (identifier) of the process currently running the program.
`MPI_Comm_size( MPI_COMM_WORLD, &size);`	Obtain the total number of processes.	`MPI_COMM_WORLD` – the communicator (group) of MPI processes working together. `size` – the total number of MPI processes.

`MPI_Send(buf, count, datatype, dest, tag, comm);`	Send a message to another process.	`buf` – a pointer to the memory to send. `count` – the number of values to send. `datatype` – the type of the values, e.g. MPI_INT, MPI_CHAR, MPI_FLOAT. `dest` – the rank of the receiving process. `tag` – an identifier for the message (must match the receive's tag). `comm` – the communicator, e.g. MPI_COMM_WORLD.
`MPI_Recv(buf, count, datatype, source, tag, comm, status);`	Receive a message from a process.	`buf` – a pointer to the memory to receive. `count` – the number of values to receive. `datatype` – the type of the values, e.g. MPI_INT, MPI_CHAR, MPI_FLOAT. `source` – the rank of the sending process. `tag` – an identifier for the message (must match the send's tag). `comm` – the communicator, e.g. MPI_COMM_WORLD. `status` - shows the status of the message. Can be ignored with MPI_STATUS_IGNORE
`MPI_Finalize();`	Finalize the MPI environment. No other MPI functions may be called after this function.	

Using MPI requires including the MPI header file, mpi.h, using an MPI compiler, such as mpicc, and running with an MPI execution command such as mpirun or mpiexec.

Exercise 3
In this exercise you will write and run a small MPI program. This exercise requires the use of a computer with MPI installed, preferably also a multi-processor machine (MPI can run on a serial processor, but it is meant for parallelism). Example commands are shown below each instruction.

1. Open a new C source file in a text editor such as vi.

```
vi hello.c
```

2. Create a small, serial "Hello, World!" program.

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

3. Save the file and compile the program.

```
gcc -o hello hello.c
```

4. Run the program.

```
./hello
```

5. Open the source file again.

```
vi hello.c
```

6. Add in the MPI code to initialize the environment, determine the process rank and size, and finalize the environment.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv) {
     int rank = 0;
     int size = 0;

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);

     printf("Hello, World!\n");

     MPI_Finalize();

     return 0;
}
```

7. Compile the code with the MPI compiler[2].

```
mpicc -o hello-parallel hello.c
```

8. Run the code on two processes with the MPI execution command.

```
mpirun -np 2 hello-parallel
```

9. Did the result change?

10. Open the source file again.

```
vi hello.c
```

[2] If you attempt to compile the code with a non-MPI compiler, it will be unable to find the mpi.h header file and will exit with an error.

11. This time add code to print the process rank and size.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv) {
     int rank = 0;
     int size = 0;

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);

     printf("Hello, World from rank %d of
               %d!\n", rank, size);

     MPI_Finalize();

     return 0;
}
```

12. Save and compile.

```
mpicc -o hello-mpi hello.c
```

13. Run the program.

```
mpirun -np 2 hello-mpi
```

Exercise 4

Include a post-order parallel traversal in `traversal.c` in the `code` directory (an attachment to this module). Make changes to the `depthfirstparalleltraversal` function and use the pre-order and in-order codes as reference. Ensure the code compiles and runs correctly, producing the same result as the serial post-order traversal for different sizes of trees.

Scaling
The goal of scaling an algorithm is either to obtain a performance increase for a problem (strong scaling) or to solve a bigger problem with more processors with comparable performance to solving a smaller problem with fewer processors (weak scaling). An analogy is painting a house. If a single painter can paint only one room in an hour, the goal of adding more painters would be to either paint the same room in less than an hour or paint multiple rooms in the same hour. The task may require communication between the painters, and this communication may become appreciable, just as it does in the parallel breadth-first traversal. Thus, scaling is not always a be-all end-all way of obtaining speedup. However, there is usually an advantage to adding processers so that one can solve a bigger problem.

Exercise 5
This exercise makes use of the Unix `time` command, which provides a simple, if not very accurate, measure of the amount of time it takes an algorithm to execute. By comparing the "real" times of the serial and parallel programs, we can get an idea of the speedup achieved by parallelism. Similarly, by comparing the "real" time of the parallel program running with different sizes of trees, we can get an idea of how the program scales. This exercise requires the use of a multi-processor computer with MPI installed.

1. Change directories to the `code` directory (an attachment to this module).

   ```
   cd BinaryTreeTraversal/code
   ```

2. Compile the code.

   ```
   make all
   ```

3. Time how long it takes to run the serial versions of each program.

   ```
   time ./traverse --order pre
   time ./traverse --order post
   time ./traverse --order in
   time ./traverse --order breadth
   ```

4. Time how long it takes to run the parallel versions of each program. Note that the `-np` argument to `mpirun` specifies how many parallel processes to use, and the file `~/machines` must exist and contain the hostname of each parallel processor, as well as how many processes MPI is allowed to spawn based on the number of processors and **cores** (if it is a multi-core system).

5. When you run the parallel traversal programs, choose a value for `-np` that matches the total number of cores available.

```
time mpirun -np 5 -hostfile ~/machines
    ./traverse-parallel --order pre
time mpirun -np 5 -hostfile ~/machines
    ./traverse-parallel --order post
time mpirun -np 5 -hostfile ~/machines
    ./traverse-parallel --order in
time mpirun -np 5 -hostfile ~/machines
    ./traverse-parallel --order breadth
```

6. Focusing only on the "real" time results, describe how the run times differ between each run of the program.

7. Run each of the parallel versions of the program for bigger trees.

```
time mpirun -np 5 -hostfile ~/machines
   ./traverse-parallel --order pre --num-rows 20
time mpirun -np 5 -hostfile ~/machines
   ./traverse-parallel --order post --num-rows 20
time mpirun -np 5 -hostfile ~/machines
   ./traverse-parallel --order in --num-rows 20
time mpirun -np 5 -hostfile ~/machines
   ./traverse-parallel --order breadth --num-rows
   20
```

8. Describe how these run times differ from the parallel run times obtained above.

Possible Extensions

This module has covered the traversing of binary trees. Binary search trees are just one way of organizing data, however. In fact, they are only one way of organizing data into trees; data can also be organized into trees whose nodes each have 3 children, or 4 children, or n children, where n may be fixed or may vary. Trees are also a special form of a **graph**, which consists of nodes and edges between them. Trees happen to be ordered hierarchically, as the search occurs top-to-bottom. This restriction is not placed on graphs in general, in which traversals can occur in multiple directions. Some graphs may not have roots, and some may not have leaves.

Different kinds of data can be placed into different kinds of structures based on which traversal is more likely to quickly find specific elements of the data. Traversals through these various structures have different algorithms that may be more-easily or less-easily parallelized. These algorithms may also scale in different ways from the algorithms explored in this module.

This module has considered the **message passing** paradigm, but the algorithm would also work under other parallel paradigms such as **shared memory**, in which threads are able to synchronize without having to pass messages to each other.

Student Project Ideas

1. Generalize the algorithms in the module to include search trees whose nodes have more than 2 children. How does this change affect the algorithms involved? Is the parallelism affected? Include working code samples of the new serial and parallel algorithms. How do these algorithms scale compared to the binary search tree traversals?
2. Explore scaling the parallel algorithms over more processors and with bigger tree sizes. Create a graph of the scaling data, plotting number of processors against run time. Research the concepts of **strong** and **weak** scaling, and collect data for each type of scaling for each algorithm.

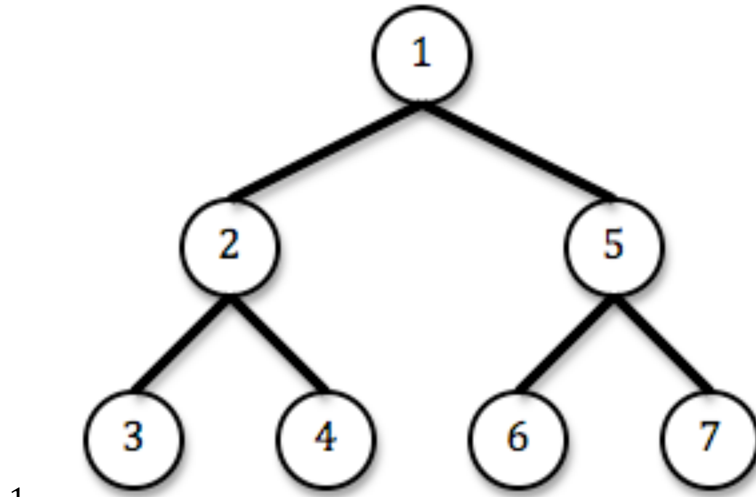
Rubric for Student Assessment

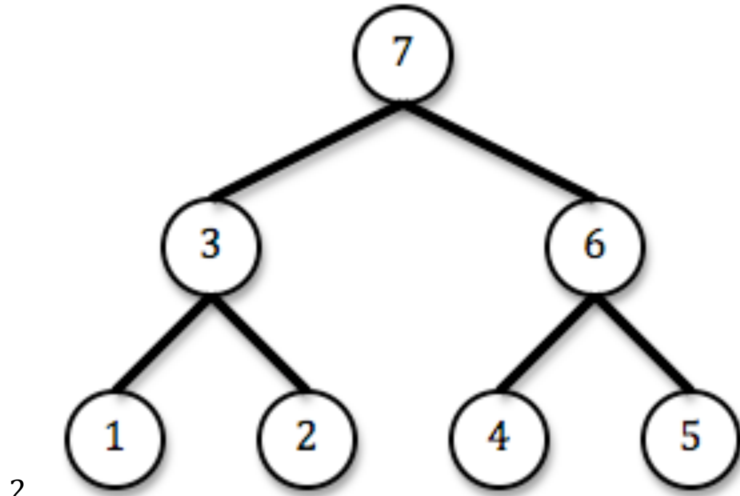
Students can be graded based on the following rubric, whose point values can be adjusted to fit a different scale or to assign a percent-based grade.

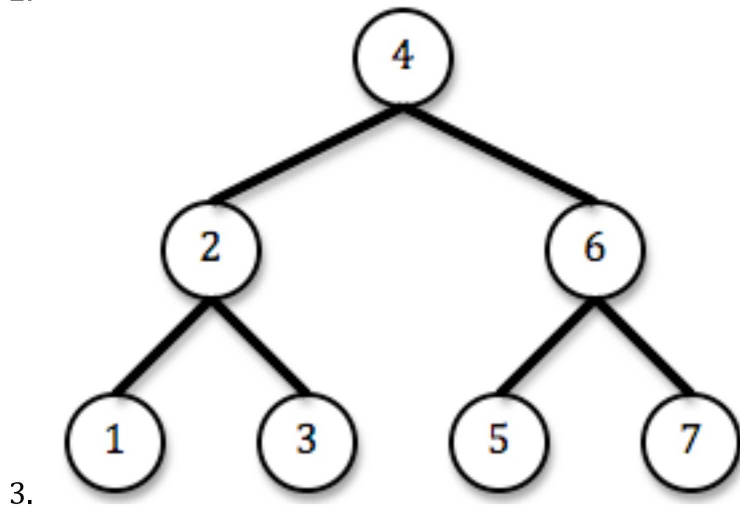
Assignment	**Number of Points**
Exercise 1	15
Exercise 2	10
Exercise 3	10
Exercise 4	15
Exercise 5	10
Student Project	40
Total	**100**

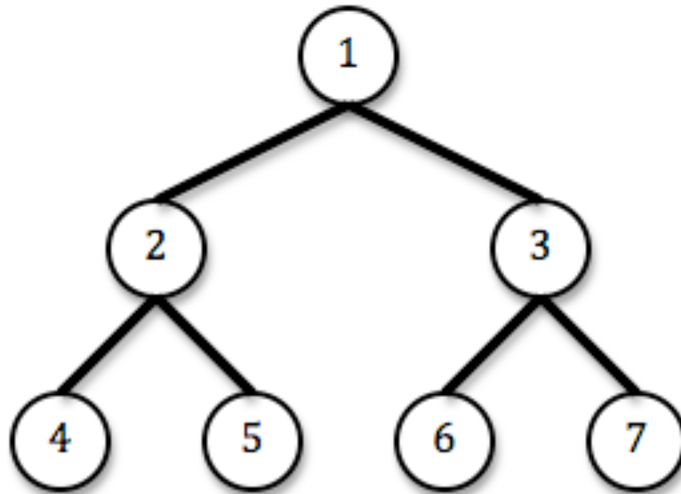
Solutions to Exercises

Exercise 1

1. 

2. 

3. 

4. 

Exercise 2

1. 4 2 1 3 6 5 7
2. 1 2 3 4 5 6 7
3. 4 2 6 1 3 5 7

Exercise 3

Step 4 should print, `Hello, World!`

Step 8 should print, `Hello, World!` twice (thus, the answer to Step 9 is yes).

Step 13 should print,

```
Hello, World from rank 0 of 2!
Hello, World from rank 1 of 2!
```

However, it is a race condition as to which processor finishes first, so it may instead print,

```
Hello, World from rank 1 of 2!
Hello, World from rank 0 of 2!
```

Exercise 4

(Available as an attachment)