Concurrent Algorithm to Globally Balance a Binary Search Tree

Mohammed-Ali Khan

Department of Computer Science and Engineering, York University,
4700 Keele Street, Toronto, M3J 1P3
Ontario, Canada

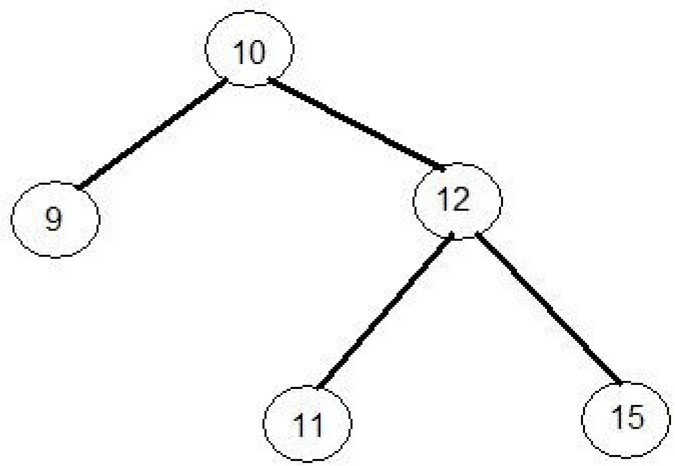
m23khan@cse.yorku.ca

Presented on: March 14, 2011

Agenda

- Brief review of Balancing Algorithms (S and P1)
- Thread synchronization - synchronized method
- Performance Analysis (Average Run Time)
- Conclusion

A Binary Search Tree



Node	
ROOT	10
KEY	9
LSON	null
RSON	null

Node	
ROOT	10
KEY	10
LSON	9
RSON	12

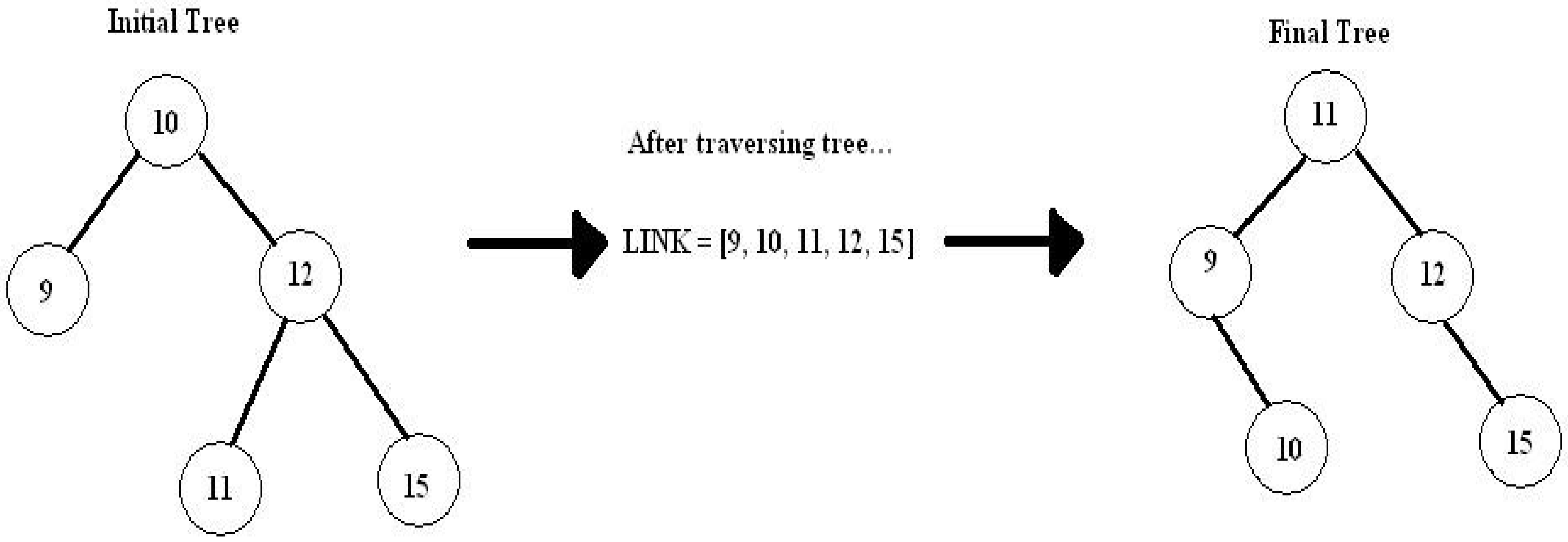
Node	
ROOT	10
KEY	11
LSON	null
RSON	null

Node	
ROOT	10
KEY	12
LSON	11
RSON	15

Node	
ROOT	10
KEY	15
LSON	null
RSON	null

Balancing Binary Tree

- Using a sequential algorithm (algorithm S) or a parallel algorithm (algorithm P1), balance a given binary search tree.



Sequential Algorithm S

- Utilizes the concept of 'folding' to simultaneously balance the left and right sub tree.

- In case of tree with odd # of nodes, folding value is the *median* of ordered set of key values (after tree traversal).

- In case of tree with even # of nodes, the folding value becomes (# of nodes / 2) + 1.

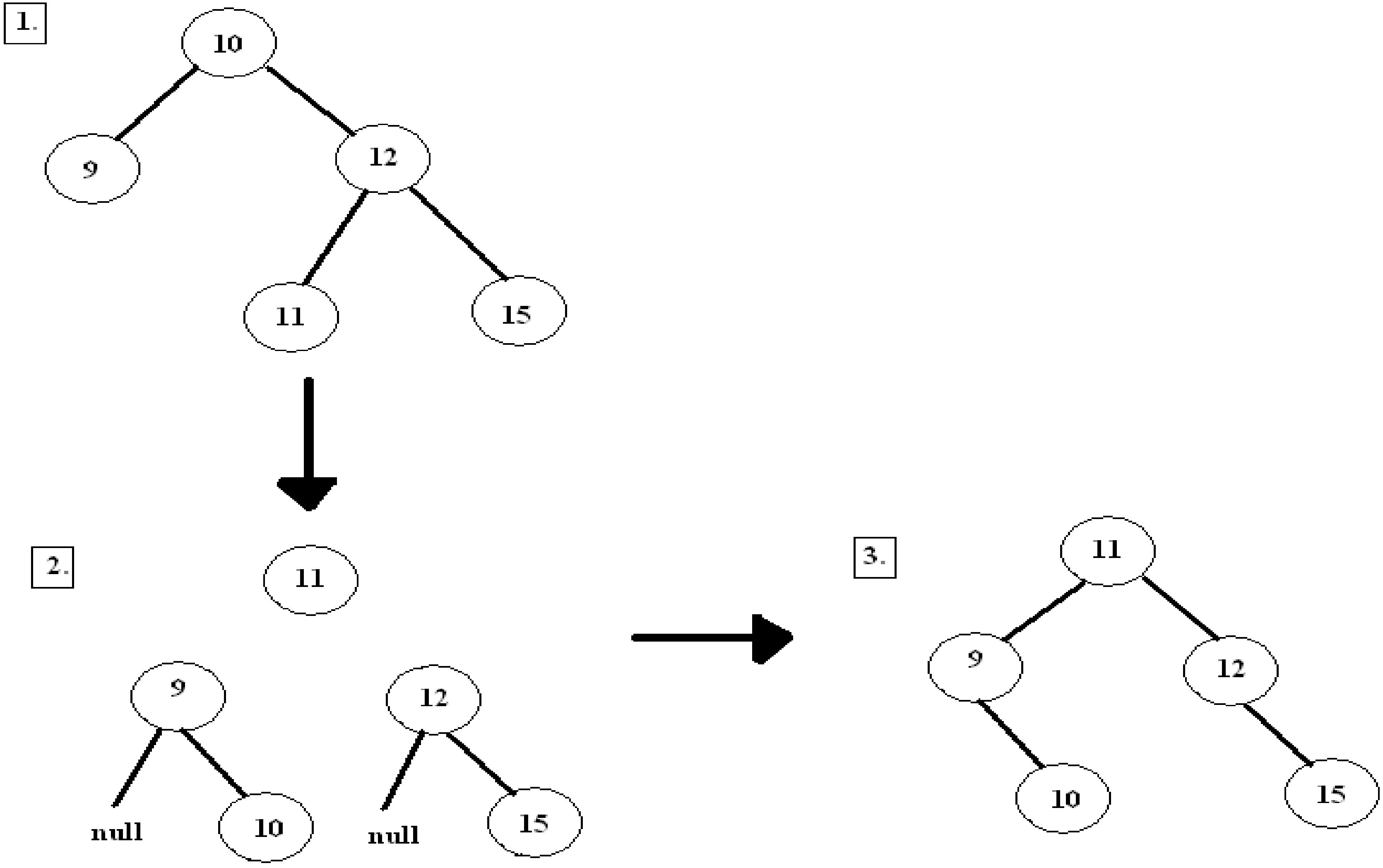
- Key idea is that if we know the position of element K in left subtree, we can determine position of counterpart element K+M in right subtree.

Parallel Algorithm P1

• Just as in sequential algorithm S, the balancing procedure first defines the root of the balanced tree equal to the median element of the set of nodes.

• The set of nodes to be balanced are split into two subsets and each forms a balanced subtree (left and right) concurrently. The process is recursive in nature in that the process continues (splitting/balancing concurrently) until there are no more elements left to split.

Visualizing Concurrency for P1



Thread Synchronization for P1

• Simply tag all methods containing thread sensitive code with the keyword *synchronized*.

• Since every object in Java has one built-in lock and built-in condition variable, for every method tagged 'synchronized', whenever it is called, the calling object owns the lock of the 'synchronized' method until it returns from the method and therefore unlocks the method.

•If a synchronized method is called but another object has already acquired a lock on it, the caller is automatically deactivated and it needs to wait until the object which owns the lock on synchronized method has released it.

•Synchronized keyword ensures that the lock/try/finally/unlock processes are automatically implemented for the built-in lock.

Thread Synchronization for P1

Operational Flow...

- There are 3 parts for using synchronized methods for concurrency. As always, we use a class object which implements Runnable interface to provide us with the run() method and to execute thread-sensitive code.

- However, since I needed to retrieve data from the Runnable object, Java has another interface called 'Callable' which implements a call() method which can return values.

Thread Synchronization technique

Part 1:
Tag the method containing thread sensitive code with the keyword 'synchronized' and add keyword 'volatile' to any instance variable in that class which would be shared among threads.
i.e...

```
public class AlgorithmP1{

  public AlgorithmP1{counter = 0;}

  public synchronized int GROW() {counter++;}

  private volatile int counter; }
```

Thread Synchronization technique

Part 2:
Create an object which implements Callable interface (implement the call() method which will in turn call the synchronized method in AlgorithmP1.java)
i.e...

```
public class RunnableP1{

  public RunnableP1(AlgorithmP1 aObj)
  { anObj = aObj;}

  public Integer call() {
    int theCount = anObj.GROW();
    return theCount;}}
```

Thread Synchronization technique

Part 3:

Create object(s) of type Callable and assign them object(s) of class which implements Callable.

Since we can't pass Callable into a Thread for execution, we use ExecutorService object instead (in my case, a thread pool).

This in turns gives you a Future object on which you can call the get() method - get() method will return the value from call() AND it acts like join() for threads.

Thread Synchronization technique

Part 3 (continued)...

```
public class BalancingAlgorithms throws Exception {

  public void P1() {
    ExecutorService threadPool =
ExecutorService.newFixedThreadPool(3);

    Callable<Integer> call1 = new RunnableP1(aP1Obj);
   Future<Integer> future1 = threadPool.submit(call1);

  int firstVal = call1.get();}

  private AlgorithmP1 aP1Obj;}
```

Performance Analysis

Machines used for performance analysis:

- MTL's 32 core machine with 12 GB memory specified in job card.
  - Each test executed 20 times (first 5 trials discarded) and the parallel algorithm was executed with 20, 32, and 96 threads.

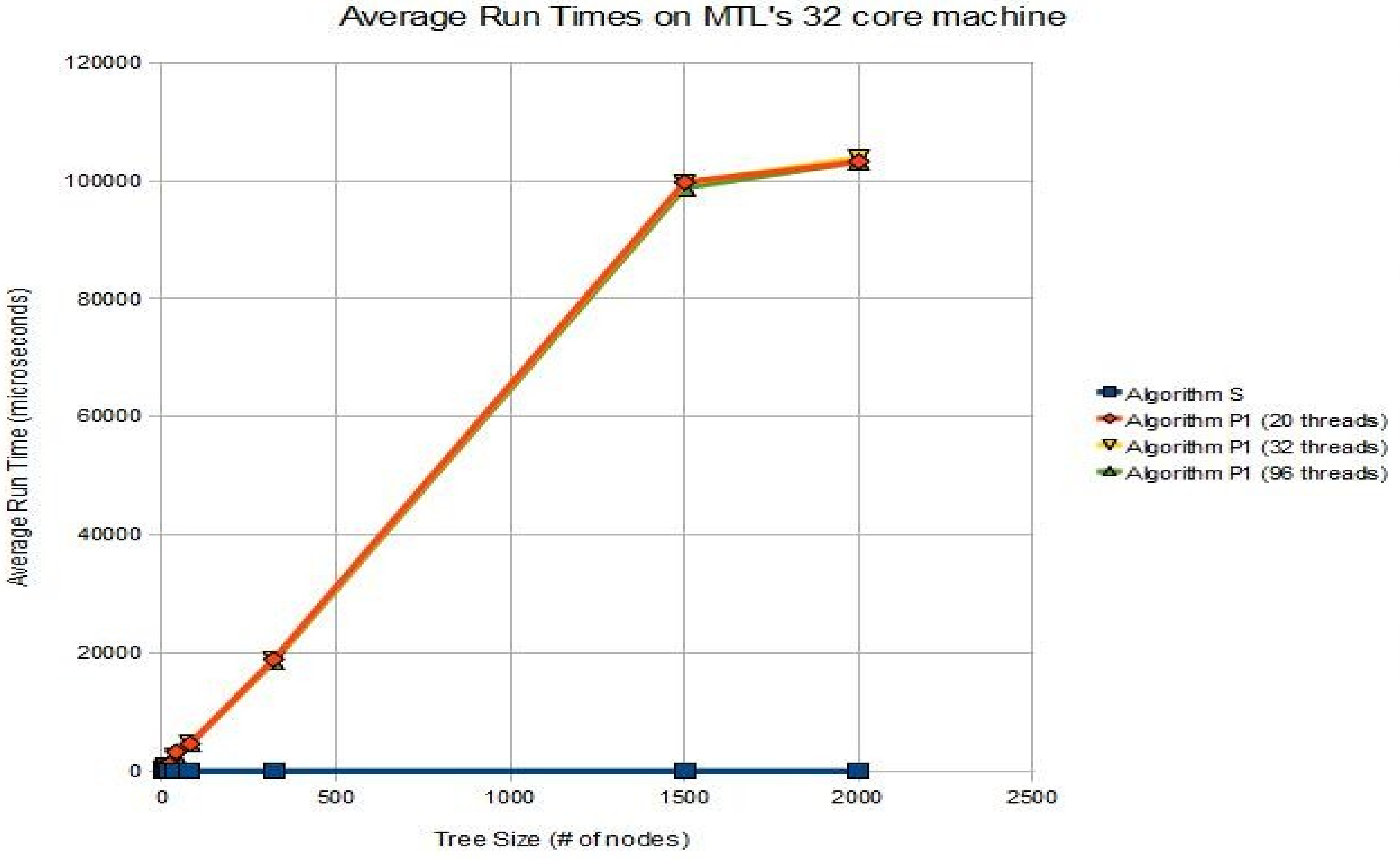
- Intel(R) Core(TM)2 Duo CPU T8100 @ 2.10 GHz with 2.00 GB RAM.
  - Each test executed 20 times (first 5 trials discarded) and the parallel algorithm was executed with 40 threads.

Performance Analysis (MTL)

Average Run Times on MTL's 32 core machine

	Average Run Time (microseconds)			
Tree Size (nodes)	Algorithm S	Algorithm P1 (20 threads)	Algorithm P1 (32 threads)	Algorithm P1 (96 threads)
5	0.01	313	329	318
6	0.01	312	325	340
10	0.01	612	597	600
11	0.01	713	755	720
20	0.02	1177	803	1168
40	0.02	3221	2384	2374
80	0.04	4599	4676	4661
320	0.1	18940	18802	18524
1500	0.1	99809	99696	98849
2000	0.1	103312	103917	103272

Performance Analysis (MTL)



Performance Analysis (laptop)

Average Run Times on my Lenovo T61 Laptop (2-core machine)		
	Average Run Time (microseconds)	
Tree Size (nodes)	Algorithm S	Algorithm P1 (40 threads)
1000	0.04	50878
5000	0.1	242353

Conclusion

• Java has various methods for thread synchronization and perhaps using methods involving explicit locking/unlocking might result in better run time.

• Instead of tagging the entire method as 'synchronized' further effort can be made to tag a subset of statements inside the concurrent method with 'synchronized' for finer grained locking.

• Based on analysis, the sequential algorithm (S) seems to be much more efficient.

References

- Cay Horstmann. BiG JAVA. 2nd Edition. John Wiley & Sons, Inc., 2006.

- S. Sitharama Iyengar and Hsi Chang. Efficient algorithms to globally balance a binary search tree. *Communications of the ACM*, 27(7):695–702, July 1984.

- John Zukowski. "Using Callable to Return Results From Runnables" ORACLE. 03 December 2007. Web. 05 Mar. 2011. <http://blogs.sun.com/CoreJavaTechTips/entry/get_netbeans_6>.

QUESTIONS...