Assignment 3 ECSE 420

Elvric Trombert
260673394
Simon Zheng
260744353

December 4, 2018

---

1. (a) t_0 shows that time it takes for reading an element in the array that is in the cache. L' is the number of elements that can be contained in cache simultaneously: if $s = 0$ then $L' = 4$. If $s = 1, 2$ then $L' = 2$. If $s > 2$ then $L' = 1$.

   (b) t_1 indicates the time it takes to read an element from main memory as the next index cannot be access through the cache.

   (c) 1 means that all the element in the array can be cached at once and accessed in constant time, thus the time does not change. It is the lowest time possible since accessing something from the cache is faster then accessing it from main memory.

   2 increases exponentially as the more cache miss we get the more time it takes to access an index on our array thus the ratio of cache miss and cache hit increases. The more cache miss we get the greater our average access time.

   3, is the case where only one element can be cached at a time thus when accessing the next element we always have to go to main memory which hence causes every access to have the same average access time. It is highest average access time as accessing something from main memory takes more time than accessing something from the cache.

   (d) Padding technique in ALock degrades the overall performance of the lock as in order get the lock, each thread must go to main memory to retrieve the value in the array index associated with it, since that information cannot be kept in the cache. Thus this slows down the lock as in order to get the lock each thread will make a call to main memory which takes more time than accessing its cache.

2. The class implementing the contains is called BoundedLockBasedQueue.java.

   The TestContains class is the one that will test whether contains work appropriately. After talking to the TA, he mentioned that since the only concurrent part that we had to achieve was in the contains method. Thus in order to test the contains method we added an add method that itself however is not concurrent.

   Our test method performs the following: we randomly add integers from 0 to 14 where the chances of adding a new number after adding a new number decreases (at first they are

1/2 then after adding the first element 1/3 and so on). This ensure that upon every try we get different integers on different list sizes (ensuring that the results cannot be hard coded and work on different lists with different sizes and item values). When a number is added to the list we mention it in a print statement.

Then we concurrently launch all the contains call on the integers from 0 to 14 and print out the result of the contains method. To make the results more easily readable we compare the contains output with an array that is set to false when the number i was not added to the list and true when the number was added to the list. We then print at "Test passed" or "Test fails" based on these results at the end.

Our implementation is correct since our test passes which means that the lock are correctly acquired and released by the method and that the implementation does not cause any false reads or deadlocks. This is due to the fact that we acquire the lock of the previous and current nodes which ensure that the current node we are reading is part of the list, as to remove or add a node in the list these two same locks must also be acquired by the add or remove method for them to remove the current node we are accessing or add a new node for the add method.

3. The class is called BoundedLockBasedQueue.java

   (b) The first difficulty comes when enqueuing. Here we must make sure before enqueuing that there is enough space in the queue for that, so first we must retrieve the *tail* and *head* value and ensure that $head - tail! = queueMaxSize$ if that passes then the issue is when adding the object to the queue we must make sure that the another thread did not add an object before us which implies testing that again and so on. Thus the only way is to use a test and set approach after wards which makes it more complex than with the lock. Further more each thread has no way of knowing without accessing the tail and head whether it can perform its tasks, this implies that the CPU will be used by thread that just idle getting the tail and head values while not making any progress compare to the lock where the thread will just wait to be awaken by the condition.

   The second difficulty is the exact same idea for dequeuing in this case if the $head == tail$ then we must wait for en enqueue to happen but even if it did when dequeuing we also have to check that since our last verification the queue was not dequeud by another dequeuer thus we must test and set before being able to dequeue an object. Again that involves a lot of CPU idle time just getting and testing variables instead of doing useful work.

4. The test class is called MatrixVectorMultiplication.java

   (c) Given that we run into the following error message "Exception in thread main java.lang.OutOfMemoryError: Java heap space" when running the code with 2000 side splitting the matrix up to 1x1 or 2x1 or 1x2. We run the code on the server with 2000 matrix splitting the matrix until it reaches 250. With this the parallel running time is 1181563085 nano seconds and the sequential time is 31571979 nano seconds. Here the parallel runtime is greater than the sequential we believe that this is due to the fact that even with the split to 250 the 8 cores provided by the server take too much time to divide the work than just actually solving the multiplication thus being slower than the sequential part. This we get a "speed up" of $\frac{31571979}{1181563085} = 0.0267$ which is a slow down again this is due to the fact that our operation technically requires a lot more thread than just 8 to perform faster than the sequential application.

(d) i. Calculate the work of the add function for vectors let $n = 2^m$:

$$\begin{aligned}
A_1(2^m) &= 2^1 A_v(2^{m-1}) + \Theta(1)\\
A_1(2^{m-1}) &= 2^1 A_v(2^{m-2}) + \Theta(1)\\
A_1(2^{m-2}) &= 2^1 A_v(2^{m-3}) + \Theta(1)\\
A_1(2^m) &= 2^2 A_v(2^{m-2}) + 2\Theta(1) + \Theta(1)\\
A_1(2^m) &= 2^3 A_v(2^{m-3}) + 4\Theta(1) + 2\Theta(1) + \Theta(1)\\
2^0 + 2^1 + 2^2 + 2^3 + .. + 2^m &= 2^{m+1} - 1\\
A_1(2^m) &= 2^m * 1 + (2^{m+1} - 1)\Theta(1)\\
A_1(n) &\approx n + 2n\Theta(1)\\
\Theta(n) &= 3n
\end{aligned}$$

Calculating the critical path length:

$$\begin{aligned}
A_\infty(2^m) &= A_v(2^{m-1}) + \Theta(1)\\
A_\infty(2^{m-1}) &= A_v(2^{m-2}) + \Theta(1)\\
A_\infty(2^{m-2}) &= A_v(2^{m-3}) + \Theta(1)\\
A_\infty(2^m) &= A_v(2^{m-2}) + 1\Theta(1) + \Theta(1)\\
A_\infty(2^m) &= A_v(2^{m-3}) + 1\Theta(1) + 1\Theta(1) + 1\Theta(1)\\
A_\infty(2^m) &= 1 + m\Theta(1)\\
A_\infty(n) &\approx m\Theta(1)\\
\Theta(\log(n)) &= m
\end{aligned}$$

ii. Calculate the work of the multiply method for matrix vector multiplication $n = 2^m$:

$$\begin{aligned}
M_1(2^m) &= 2^2 M_v(2^{m-1}) + \Theta(2^m) \\
M_1(2^{m-1}) &= 2^2 M_v(2^{m-2}) + \Theta(2^{m-1}) \\
M_1(2^{m-2}) &= 2^2 M_v(2^{m-3}) + \Theta(2^{m-2}) \\
M_1(2^m) &= 2^4 M_v(2^{m-2}) + 2^2\Theta(2^{m-1}) + \Theta(2^m) \\
M_1(2^m) &= 2^6 M_v(2^{m-3}) + 2^4\Theta(2^{m-2}) + 2^2\Theta(2^{m-1}) + \Theta(2^m) \\
M_1(2^m) &\approx 2^6 M_v(2^{m-3}) + 2^2 * 2^m + 2 * 2^m + 2^m \\
M_1(2^m) &\approx 2^6 M_v(2^{m-3}) + (2^2 + 2 + 2^0)2^m \\
2^0 + 2^1 + 2^2 + 2^3 + .. + 2^m &= 2^{m+1} - 1 \\
M_1(2^m) &\approx (2^m)^2 + (2^{m+1} - 1)2^m \\
M_1(2^m) &\approx (2^m)^2 + 2^{2m+1} - 2^m \\
M_1(n) &\approx n^2 + n^2 * 2 - n \\
\Theta(n^2) &= 3n^2 - n
\end{aligned}$$

Calculate the critical path the multiply method for matrix vector multiplication $n = 2^m$:

$$\begin{aligned}
M_\infty(2^m) &= M_v(2^{m-1}) + \Theta(m) \\
M_\infty(2^{m-1}) &= M_v(2^{m-2}) + \Theta(m-1) \\
M_\infty(2^{m-2}) &= M_v(2^{m-3}) + \Theta(m-2) \\
M_\infty(2^m) &= M_v(2^{m-3}) + \Theta(m-2) + \Theta(m-1) + \Theta(m) \\
M_\infty(2^m) &\approx M_v(2^{m-3}) + m - 2 + m - 1 + m \\
1+2+3+4+..+n &= \frac{(n+1)*n}{2} \\
M_\infty(2^m) &\approx \frac{(m+1)*m}{2} \\
\Theta(\log(n)^2) &= \frac{m^2+m}{2}
\end{aligned}$$

iii. The parallelism achieved is therefore:

$$\frac{M_1}{M_\infty} = \Theta(\frac{n^2}{\log(n)^2})$$

This is less then the Θ running time required for matrix multiplication which makes sense since, every time we multiply, we "create" 4 new threads compared to 8 with matrix multiplication, thus, in our case, for a 2000×2000 matrix, we would need about 364770 cores to run the multiplication at its fastest speed. More cores would not improve the speed up.

Discussing now the implementation for the sequential time, it is two basic for-loops that just iterate through the matrix and the vector and perform the multiplications.
For the parallel part, we followed the approach given to us in class for the matrix multiplication.

We created a Vector and Matrix class and added *split* methods to be able to "virtually" split a vector in half and a matrix in 4 without having to copy the matrix entries into a smaller matrix over and over again (the idea of doing this with a *split* method in its own class as opposed to inline came from the book). Then, using these methods, for the *add* Task we cut each vector into 2 pieces recursively passing it to the executor as submissions, until we reach a 1 by 1 vector. Then, we perform the addition.

For the matrix vector multiplication, we created two new vectors that will hold the result of the multiplication and addition to the left and right side of the matrix, independently. Then, we split the matrix in 4 and split the two new vectors in 2 (along with the vector the matrix is being multiplied with). We then recursively pass each quadrant, with its corresponding answer vector-half along with the multiplication vector-half. Once done, we pass the two answer vectors with the final answer vector to the add vector task.