**Student:** Hong Quan Doan

**Student ID:** 986956

**Day 08 – Mar 05**

**Environment**

MacBook, Core i5, 4 CPU, 8GB

**Merge Two Sorted Lists**

Result

Generating arrays with 40,000,000 items

Sorting arrays ...

Merging arrays ...

Sequential merging with Binary: 10323 ms

Sequential merging with While: 124 ms

Parallel merging with Binary Search: 4215 ms

Speedup between Sequential (While algorithm) and Parallel: 0.029419

Processor Utilization: 0.007355

Speedup between Sequential (BinarySearch algorithm) and Parallel: 2.449110

Processor Utilization: 0.612278

Based on the above results, we can see:

* Sequential with while loop is the fastest.
* The parallel version of the Binary Search algo outperforms the sequential one, because it divides the arrays into smaller blocks, which cost less time to merge.
* There’s no issue with data race due to:
  + Each source array has distinct values
  + Both arrays do not include any item from each other.

**Source code:**

Sequential merge with Binary Search algorithm

public static void mergeWithBinarySearch(int[] arr1, int[] arr2, int[] arr3) {

int pos = 0;

for (int i = 0; i < arr1.length; i++) {

pos = findPos(arr1[i], arr2);

arr3[i + pos] = arr1[i];

}

for (int i = 0; i < arr2.length; i++) {

pos = findPos(arr2[i], arr1);

arr3[i + pos] = arr2[i];

}

}

private static int findPos(int key, int arr[]) {

int start = 0;

int end = arr.length - 1;

int index = (int) Math.floor((end - start) / 2) + start;

if (key > arr[arr.length - 1]) { // The target is beyond the end of this array.

index = arr.length;

} else {

// Start in middle, divide and conquer.

while (start < end) {

// Get value at current index.

var value = arr[index];

if (value == key) {

// Found our target.

break;

} else if (key < value) {

// Target is lower in array, move the index halfway down.

end = index;

} else {

// Target is higher in array, move the index halfway up.

start = index + 1;

}

// Get next mid-point.

index = (int) Math.floor((end - start) / 2) + start;

}

}

return index;

}

Sequential merge with While loop

public static void mergeWithWhile(int[] arr1, int[] arr2, int[] arr3) {

int cur1 = 0;

int cur2 = 0;

int cur3 = 0;

while(cur1 < arr1.length && cur2 < arr2.length) {

if(arr1[cur1] <= arr2[cur2]) {

arr3[cur3++] = arr1[cur1++];

} else {

arr3[cur3++] = arr2[cur2++];

}

}

while (cur1 < arr1.length) {

arr3[cur3++] = arr1[cur1++];

}

while (cur2 < arr2.length) {

arr3[cur3++] = arr2[cur2++];

}

}

Parallel merge with Binary Search (using the same findPos() method)

public ParallelMergeTask(int[] arr1, int[] arr2, int[] arr3, int tid, int startIndex, int endIndex) {

this.arr1 = arr1;

this.arr2 = arr2;

this.arr3 = arr3;

this.tid = tid;

this.startIndex = startIndex;

this.endIndex = endIndex;

this.arr1Len = this.arr2Len = endIndex - startIndex;

}

@Override

public void run() {

int idx = 0;

for (int i = startIndex; i < endIndex; i++) {

idx = findPos(arr1[i], arr2);

arr3[idx + i] = arr1[i];

}

for (int i = startIndex; i < endIndex; i++) {

idx = findPos(arr2[i], arr1);

arr3[idx + i] = arr2[i];

}

}

**Exercises**

**J2**

Question: How does a fair Read-Write lock prevent many active readers from starving a writer? Can more than one writer acquire the lock at the same time?

Answer: If new threads were constantly granted read access the thread waiting for write access would remain blocked indefinitely, resulting in starvation. Therefore, a thread can only be granted read access if no thread has currently locked the ReadWriteLock for writing, or requested it locked for writing.

Hence, the read reentrance is only granted if no threads are currently writing to the resource. Additionally, if the calling thread already has read access this takes precedence over any writeRequests.

Write reentrance is granted only if the thread has already write access.

**J3**

Question: In the slide “Potential Deadlock,” describe a series of event that could lead to a Deadlock.

Answer: The deadlock can occur the below scenario:

* Thread t1 starts and calls test1 method by taking the object lock of s1.
* Thread t2 starts and calls test2 method by taking the object lock of s2.
* t1 calls method1 and t2 calls method2 and both waits for 1 second, so that both threads can be started if any of them is not.
* t1 tries to take object lock of s2 and call method test2 but as it is already acquired by t2 so it waits till it become free. It will not release lock of s1 until it gets lock of s2.
* Same happens with t2. It tries to take object lock of s1 and call method test1 but it is already acquired by t1, so it has to wait till t1 release the lock. t2 will also not release lock of s2 until it gets lock of s1.
* Now, both threads are in wait state, waiting for each other to release locks. Now there is a race around condition that who will release the lock first.
* As none of them is ready to release lock, so this is the Dead Lock condition.

**J4**

Question: In the slide “Potential Deadlock,” why do we need two locks? Would the code still work if we only used one lock?

Answer:

To avoid deadlock, we should consider some possible methods.

* Avoid nested locks: This is the main reason for dead lock. Dead Lock mainly happens when we give locks to multiple threads. Avoid giving lock to multiple threads if we already have given to one.
* Avoid unnecessary locks: We should have lock only those members which are required. Having lock on unnecessarily can lead to dead lock.
* Using thread join(): Dead lock condition appears when one thread is waiting other to finish.

The slide “Potential Deadlock actually falls to the Nested Lock condition, so, one lock could possibly help to avoid the deadlock. Furthermore, we could review the algorithm while we create the multithreading environment.

**J5**

Threads: 4

Number of items to push and pop to the stack: 10,000,000

Sequential time: 1494 ms

Number of threads: 4

Thread 0 with 2500000

Thread 1 with 2500000

Thread 2 with 2500000

Thread 3 with 2500000

Parallel time: 292 ms

Speedup: 5.116438

Processor Utilization: 1.279110