CS 3310 Assignment 3

Due Date: November 3rd, 2016

Ken Rivard

1. **PROBLEM:** In class we discussed how to convert the recursive version of merge-sort by unfolding the recursive steps to an iterative version when n, the number of elements to sort, is a power of two (i.e., n = 2d, for some d ≥ 0). Using similar approach, extend the design and analyze an efficient iterative solution to merge sort for any value of n ≥ 0. You need to give pseudo-code and document the code so a layman can understand what your code is trying to do.

In class, we discussed the mergesort algorithm when the number of elements was a power of two. This works and is very easy because when dividing the elements into 2 equal lists, there is the same number of elements on each side. For example, if n = 16, we split that up into 2 sets of 8 elements, then 2 sets of 4, and so on until we can no longer divide the list anymore. The question raised is what if n != a power of 2 and Java cannot divide the list equally? Below is pseudocode for the mergesort algorithm when n != 2d.

\*\*assume the array contains an odd number of elements/is not a power of 2\*\*

if array.length > 1

divide the array by 2

if array is size 2d//each side will have equal number of elements

if array is size != 2d //for example 7, one side will have 4, and the other will have 3 //since Java rounds down, 7/2 will be 3

sizeSubArray1 = sizeSubArray2 //initialize each sub array to be of equal lenth

if array.length % 2 == 1

add 1 extra element to the second array //so each subarray adds up to 7

mergeArr1 = 3 elements //the lower number mergeArr2 = 4 elements //the higher number

//copy the first parts of the original array into subArray1 until full for(int i=0; i < sizeSubArray1; i++) sizeSubArray1[i] = array[i] //the loop above is iterating through the first subarray until it is filled

//copy the second parts of the original array into subArray2 until full for(int i=sizeSubArray1; i < sizeSubArray1 + sizeSubArray2; i++) sizeSubArray2[i – sizeSubArray1] = array[i] //the loop above is iterating through the second subarray until it is filled

//now we recursively call mergeSort until both mergeArr1 and mergeArr2 are sorted

mergeArr1 = mergeSort(mergeArr1)//recursively call mergeSort to sort mergeArr1 mergeArr2 = mergeSort(mergeArr2)//recursively call mergeSort to sort mergeArr2

//the following code will merge and sort mergeArr1 and mergeArr2

//run this loop until one sub array is empty

while mergeArr1.length != arr1Index and mergeArr2.length != arr2Index

if mergeArr1 < mergeArr2

copy the current element of mergeArr1 into the original array

increase the index of final array to avoid replacing the previous element

increase the index of mergeArr1 to avoid a comparing the same element

endif

else //do the same thing except with mergeArr2

copy the current element of mergeArr2 into the original array

increase the index of final array to avoid replacing the previous element

increase the index of mergeArr2 to avoid a comparing the same element

endelse

endwhile

//the following code executes when there are no more comparisons to be done

while mergeArr1.length != arr1Index //this loop adds SORTED mergeArr1 to the beginning of the final array

add mergeArr1 to the original array

increment the index of the final array

increment the index of mergeArr1

endwhile

while mergeArr2.length != arr2Index //this loop adds SORTED mergeArr2 after mergeArr1 in the final array

add mergeArr2 to the original array

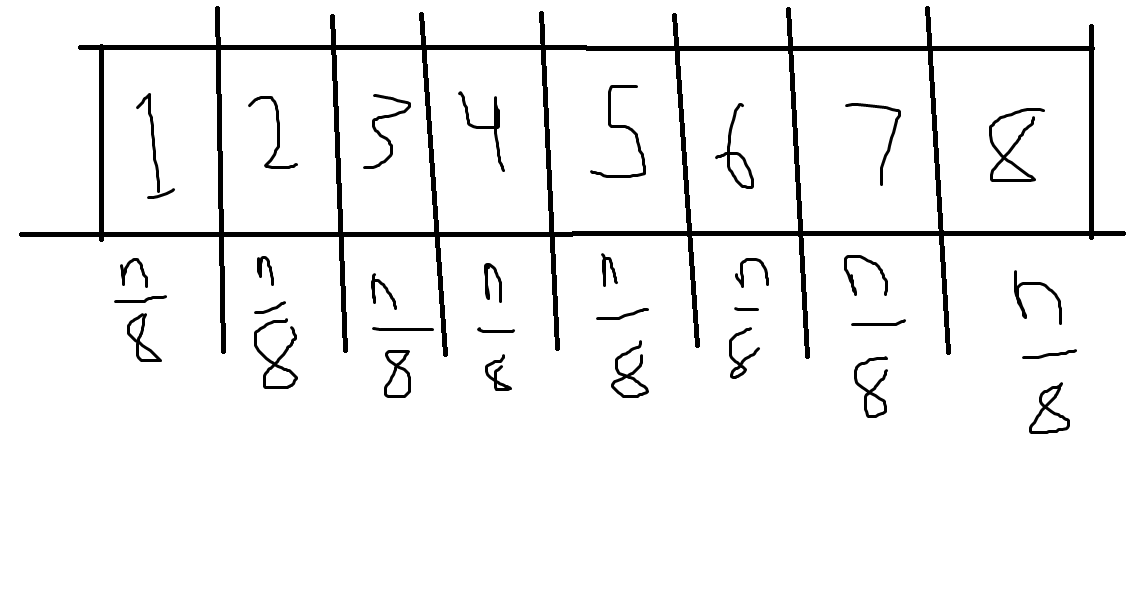
increment the index of the final array

increment the index of mergeArr2

endwhile

1. **PROBLEM:** Let us get an intuitive start on designing multi-core HPC solutions when single-core solutions may not be sufficient. Assume that you are given an array A[0..n-1] of n elements in shared-memory and that you have eight PEs (aka processing elements / cores / CPUs) in your machine. Each PE can access a shared memory location in Θ(1)-time. (In other words, each PE can read or write an element from the shared memory in Θ(1)-time; of course the type of elements has to be same as A or one of the primitive types, e.g., int, float, Boolean, double, char.) At most one PEs is allowed to access a memory location in the shared memory at any given time (i.e., two PEs can’t access the same memory location x at any given time t, so if they need to access the memory location x, they will have to do that at different times). Each PE has a small amount of fast local memory (such as registers) to execute a basic instruction in Θ(1)-time on the data that is in its local memory. Design and analyze an efficient solution to sort the list A using the eight PEs efficiently using merge-sort. Again pseudo-code is sufficient but explain your design so anyone (with a reasonable programming expertise) can understand your solution easily. You may want to give a high-level description of your idea first and then give pseudo-code.

**OVERVIEW**



The above picture represents an array being split up into 8 separate parts for each PE. Since each PE can only access one shared memory location at a time, we will make a copy of each section of the array (1 through 8) so instead of only being able to access one shared memory location at one time, each PE is now able to access copies of those memory locations at the same exact time. Therefore, this whole process will be sped up by at least 8x. To clarify, PE1 has access to part 1, PE2 has access to part 2, PE3 has access to part 3 and so on up until PE8 has access to part 8. A tradeoff that needs to be made in order to achieve this is that making copies of each of the 8 parts will take 4 units of time; fortunately, this is a constant and only needs to happen once and will benefit the rest of the program as it runs.

Here is a visual representation of why making copies of 8 parts of the array will take 4 units of time

1 PE: 1

2 PE: 1 2

4 PE: 1 2 3 4

8 PE: 1 2 3 4 5 6 7 8

\*\*where the number before PE is the total number of processing elements

**PSEUDOCODE**

Array with n elements in shared memory is initialized

//split the array into 8 parts for each PE and make copies

while array != 8 parts

divide array into 8 parts //for each PE [PE1, PE2, … , PE8]

endwhile

for(int i=0; i<numPE; i++)

make a copy to each part of the array //for easy PE access

endfor

//sort the array using all 8 PEs at the same time

if array > 1

PE1 cuts its part in half for mergeSort

. . .

PE8 cuts its part in half for mergeSort

initialize PE1 through PE8 once size is determined

//copy PE1 part into arr1

for(int i=0;i<sizePE1;i++) arr1[i] = array[i]

. . .

//repeat this process for until PE8 and arr8

//recursively call mergeSort on each part of the array (PE1 to PE8)

arr1 = mergeSort(arr1) arr2 = mergeSort(arr2) arr3 = mergeSort(arr3) arr4 = mergeSort(arr4) arr5 = mergeSort(arr5) arr6 = mergeSort(arr6) arr7 = mergeSort(arr7) arr8 = mergeSort(arr8)

//the following will merge each sorted part back to the final SORTED array

while arr1.length through arr8.length != their own indexes

if current arr1 element < current arr2 element copy the current element to final array increment index of final array //to avoid replacing wrong elements increment index of arr1 //to avoid double comparisons

endif

. . .

//repeat this process for all PEs until PE8

endwhile

//the following code places each sorted array (arr1 to arr8) to the final array

while arr1.length != index

finalArray[finalArrIndex] = arr1[arr1Index]

increment finalArrIndex

increment arr1Index

endwhile

. . .

//repeat this while loop for arr2 to arr8

//after this, each part of the array should be sorted and put back into the final array in a sorted order using mergeSort and 8 PEs at the same time

**EXTRA CREDIT:** For 20 points extra credit, generalize your solution so it can work for p-PEs where p = 2d, for some d ≥ 0.

The above solution will work for any number of PEs where p = 2d. The only things that need changing is how many parts get divided up. For example, if there are 2 PEs, the array needs to be divided into 2 parts, if there are 4 PEs, the array needs to be divided into 4 parts, if there are 8 PEs (this example), the array needs to be divided into 8 parts, and this process occurs all the way to 2d parts.

Also, this will make the process of dividing the array easier because if we know how many PEs we have, we can pre-determine how many parts the array needs to be divided into (i.e. 2 PEs = 2 divided parts). Then, the only thing that will change in the above code is how many PEs are being utilized, whether it needs more or less. But since it will always be a power of 2, the division is easy.

Finally, the above code works for any p = 2d PEs while the only things changing are the number of PEs needed as well as the number of arrays and copied of arrays needed for comparison.