There is a file in the handouts directory for 603 that we'll add in as well... MergeSort.java

ReactReply

- copy it to your computer and add it in to your project

-Up until now we’ve looked at setting up tasks and functions to operate on separate threads but that’s not the same as actual parallel programming, which is what we’ll look at today

- We are going to look at implementing a mergesort in parallel. That will mean primarily the dividing part of the algorithm will separate into separate parallel threads which will then, once they’re finished, be put back together to form a solution

This will also be a recursive solution

Java provides a framework for doing this which is shown in figure 32. 28 and figure 32. 29

The two main parts are forkjointask and forkjoinpool

- Will be setting up a recursivetask which is a sub class of Forkjointask. we will then use a forkjoinpool to actually envoke it

- Now recall how merge sort works. We repeatedly divide the problem in half until we are left with a collection of single item lists. We then merge the lists back together to produce a sorted result.

When we create a runnable we implement the run method. When we create a recursivetask we implement a compute method. this is what we’ll do for this algorithm and our sort will be implemented there

Let's get started… create three things inside your program class :

1- Main()

2- parallelMergeSort as a public static void method

3- A private static class called SortTask which extends RecursiveAction

-ParallelMergeSort will take an int array as a parameterGraphical user interface, text, application

Description automatically generated



Timeline

Description automatically generated

Graphical user interface, text, application, chat or text message

Description automatically generated

-note that i've added the mergesort file in to my project

Will the define our algorithm inside of the reccursive action

So refresher on the original merge sort algorithm. first thing we do in the algorithm is check the list length and as long as it’s greater than one will continue splitting if it’s not greater than one then we can stop

We then call merge sort recursively on each half and then merge after that to put them back together

Our recursiveaction class will have a couple of member variables, one constructor which will take an int array as a parameter and then we’ll override the compute method

The variables are a constant called threshold which will be equal to 500 and an int array called list

Then have a constructor that simply takes a list array as a parameter and sets the list variable

Then override compute

Text

Description automatically generated

- Threshold is going to act as our stop condition in a sense but it’ll be 500 instead of one but you can change it to whatever you like to see what the effect is

Text

Description automatically generated

-This might seem odd but we will be creating a very large array to sort

- Next we create our two halves as we did before

In fact, you can use the exact code to create both of these from the merge sort program

The only difference is we won’t call the merge sort quite yet

A screenshot of a computer

Description automatically generated with medium confidence

- Next key point, when it comes time to run one of these things we invoke the task using an invoke() or invokeAll() method

This can be done in two places:

An executor can do it which is how we will start the process. Also it can be done from within a recursive action class. That’s how we’ll implement our recursion

-An executor can do it which is how we will start the process. Also it can be done from within a recursive action class. That’s how we’ll implement our recursive calls

-So we’ll call invokeall and pass, as parameters, two instances of a sort task with each half list as a parameterA screenshot of a computer

Description automatically generated with medium confidence

This implicitly forks these calls to separate processors for execution.

- Now all that’s left is merging and we can use the original merge method from the merge sort file to do this. we don’t need different code

MergeSort.merge(firstHalf, secondHalf, list);

- Since the main part of this is forking stuff to begin with., And invokeAll takes care of that, once that’s done we only need to complete. completing will take care of joining after all the instances

- Since the main part of this is forking stuff to begin with., And invokeAll takes care of that, once that’s done we only need to complete. completing will take care of joining after all the instances are finished execution. that’s why we can just call merge at the end

Let’s take a quick break and come back in a few minutes

-You might wonder why were using a recursive action rather than a recursive task. recursive actions are essentially like void methods, recursive tasks are essentially like returning methods. If you look carefully at the compute method in each one, in one case it’s void, in the other case it returns a value

Now that's done, next step…

- We need to start the process to begin with. This is where parallelmergesort comes in. So we’ll create a task here, we’ll create a pool of Forkjoinpool and then will call invoke from the pool using the task to start the process

First instantiate a recursive action using list as a parameter

- This is the same as the call inside of the task except it’s invoke() not invokeAll()

And there’s only one task to pass to it not two

Then instantiate a forkjoinpool simply called pool as just new forkjoinpool with no parameters

- This constructor can be called with or without a parameter. If you include a parameter it’s the number of processors you want used for your problem

-Finally call the invoke method from pool, passing your reccursive action variable as a parameter

Text

Description automatically generated

- That’s it for setting up our task, now we can go ahead and run both the parallel version and the original version and will do some timing with both so we can see what the efficiency of each one is. We’ll also check out how many processors are being utilized for our parallel task

-So into the main method, will start by creating two separate int arrays as lists. Set up a constant size variable set to 7 million … this will be the size of each array

I called my arrays list1 and list2

- Next set up a loop to populate the arrays. I’m using a random number generator, generating numbers between zero and 10 million

Then call each algorithm, one with the list1 and one with list2

But… We’ll add a couple of other things so we can check efficiency

- You can call System.currentTimeMillis before and after to get a current timepoint and then subtract them to find how many milliseconds elapsed for your execution. I'm creating two variables, startTime and endTime to do this for both algorithm calls and I'll do a System.out to output the results after each algorithm call

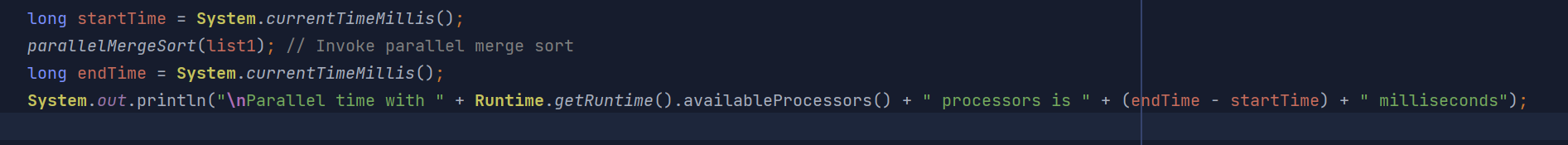
- You can also call a method Runtime.getRuntime().availableProcessors to find how many logical processors are available for your processing

So the setup stuff:

Text

Description automatically generated

My first algorithm call looks like:



- Now do the exact same thing except reuse the the two time variables, call merge sort with list2 and dont make the call for the number of processors as this execution will be regular sequential.

Graphical user interface, text, website

Description automatically generated

-Now you can go ahead and run your program

The parallel should be 2-3 times faster

Parallel time with 12 processors is 362 milliseconds

Sequential time is 1165 milliseconds

This is my output

I want to see some others post theirs as well...