In class on [Thursday Feb 15th, 2024](https://coastal.yuja.com/V/Video?v=9704145&node=42801875&a=1964620&autoplay=1), we discussed multithreading and showed my example program (two java files) running. (*the code is given at the end of this document*). (It can also be found [here](https://moodle23-24.coastal.edu/pluginfile.php/871809/mod_page/content/17/src.zip)).

After grabbing that code, here’s how you can compile and run it:

A screenshot of a computer code

Description automatically generated



Notice when I double the number of threads from 1 to 2, I get a shorter runtime and thus a speedup.

A black text on a white background

Description automatically generated



These speedup and efficiency metrics are useful to describe how well the parallelization went to speedup the activity using this multithreaded technique:

A black text on a white background

Description automatically generated

A black text on a white background

Description automatically generated

(here the number of processors can he viewed as the number of threads)

**PART I:**

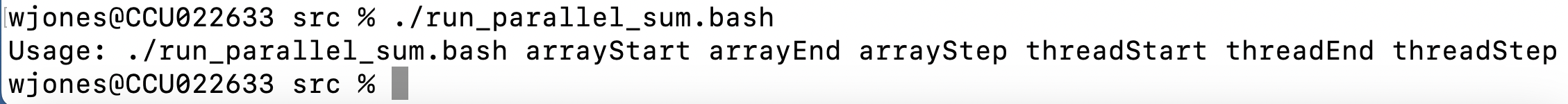
Get this code on your computer, and make sure that it will run. Try increasing the number of threads up to the number of cores you have, and take screenshots each time of the values you get back (especially the speedup and efficiency metrics). Everyone in here as machines that have at least 4 cores – so at least try up to that point.

Document this in your report.

**PART II:**

(replicate / document all these steps on your system, and make sure that I can tell that it is your system by having a unique prompt name or something, or add your name to the output of the programs so that I can tell you ran it)

You will notice in the zip files there are scripts to run the program repeatedly, so that one could gather the data more quickly than running it by hand:



(there are some windows PS and BATCH scripts provided, but you need to make sure these work / fix them)

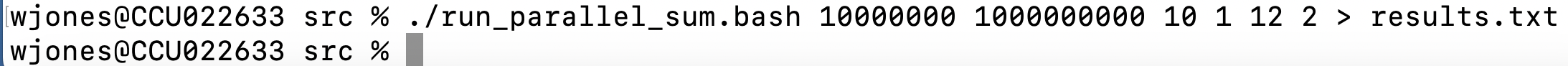
As you can see from this script:

A close-up of text

Description automatically generated

The outer loop loops over the different array sizes you want to test, going up by a multiplicative factor, while the inner loop, loops over different thread counts, going by an additive step (l

For example, if I want to sweep sizes from 1E7 to 1E9 (going by factors of 10) and then thread counts from 1 to 12 (going 2 at a time), then I could run it like this:



Notice the results that would be coming to the screen have been redirected to text file, results.txt.

If we look at the results file:

A screenshot of a computer code

Description automatically generated

We see we have some data in there we can scroll through it, for example, let’s look at the largest size, for every tested thread count:

A screenshot of a computer program

Description automatically generated



Look at that trend, and see how the speedup and efficiency are changing as the number of threads increases.

Now, modify your program to output a datafile that has these numbers in it

N, P, T\_1, T\_P, S\_P, and E\_P

N problem size

P number of threads

T\_1 time on one thread

T\_P parallel time using P threads

S\_P speedup on P threads

E\_P efficiency on P threads

For example, the last time the program was invoked by the script above,

A black text on a white background

Description automatically generated

The data from this would have been

N P T\_1 T\_P S\_P E\_P

1000000000 11 0.56 0.08 6.67 0.61

**TODO:**

Update the Java program so that it will take an additional parameter at the commandline (after the prior last one) that is the file name that the numerical results from each program execution are stored – (like created if never been used, but appended to if already exists) – this tab delimited file will keep the data that is generated by the program execution, so that you can then go and PLOT this data after collecting it.

Note, if the user does not specify this file name, then assuming that no data is to be written directly to the file, but if they do specify the filename, then it uses it (in other words, it is an optional parameter).

So, if we ran the above script with the same parameters it would produce that results file:

N P T\_1 T\_P S\_P E\_P

1000000000 11 0.56 0.08 6.67 0.61

But with a lot more rows, specifically, it would have 18 numeric rows, since the script (as called above) did N = 1E7, 1E8, 1E9, (three combinations) and for each of those, it did P = 1, 3, 5, 7, 9, 11(six combinations) 🡺 3 x 6 = 18 experiments.

Then, go use Python (matplotlib or seaborn) to plot the following:

**Time VS P** -- would have 3 curves on it, corresponding to the 3 different values of N tested above (Y axis is time in seconds, X axis is thread count, P)

**Speedup VS P** -- would have 3 curves on it, corresponding to the 3 different values of N tested above (Y axis is speedup, X axis is thread count, P) – this would also have one additional (dotted line) called “ideal” whose formula is Speedup = P (so the line like Y = X)

**Efficiency VS P** -- would have 3 curves on it, corresponding to the 3 different values of N tested above (Y axis is efficiency, X axis is thread count, P) this would also have one additional (dotted line) called “ideal” whose formula is efficiency = 1 (so the line like Y = 1.0)

**Format the plots to look great – labeled axes and good font sizes.**

Make sure the phots are well-formatted, and have labeled axes and also that there is a key that identifies what each line is for (maybe have the lines have different colors and shapes (dotted, dashed, etc --- for accessibility reasons)

* Write and document the code.
* Write a report that indicates the thought process how it all works, and that shows it in operation for gathering all the data used above
* Include all data, graphs, and plots, tables in the report

**You ZIP file should have this folder structure, with a top level, and three sub folders:**

/hw05-your-name

./code // put the code here – java source and your bash / BATCH /PS scripts --- make sure you update the BASH one to be compatible with your program, because I will run it on my computer --- even if you are Windows user, I expect it to work – you can also use ci.coastal.edu to test it out, since that is a Linux environment

./report // put the report here, Word DOC and PDF

./data // raw data and plots and plot-generating items, Python

**Submission:**

ZIP and compress and submit.

PROGRAMS FROM CLASS

**public** **class** ParallelSum {

**public** **static** **void** main(String[] args) **throws** InterruptedException {

**if** (args.length < 2) {

System.***out***.println("Usage: java ParallelSum <number of threads> <array size>");

**return**;

}

**int** numberOfThreads = Integer.*parseInt*(args[0]);

**int** arraySize = Integer.*parseInt*(args[1]);

**int**[] numbers = **new** **int**[arraySize];

**for** (**int** i = 0; i < arraySize; i++) {

// numbers[i] = (int) (Math.random() \* 100); // Fill the array with random numbers

numbers[i] = 1; // why did I do this ?

}

// Sequential execution

**long** startTime = System.*nanoTime*();

**long** seqSum = *sequentialSum*(numbers);

**long** endTime = System.*nanoTime*();

**long** sequentialTime = endTime - startTime;

System.***out***.println("Sequential sum: " + seqSum + ", Time: " + (sequentialTime / 1e9) + " seconds");

// Parallel execution

startTime = System.*nanoTime*();

**long** parallelSum = *parallelSum*(numbers, numberOfThreads);

endTime = System.*nanoTime*();

**long** parallelTime = endTime - startTime;

System.***out***.println("Parallel sum: " + parallelSum + ", Time: " + (parallelTime / 1e9) + " seconds");

// Calculate speedup and efficiency

**double** speedup = (**double**) sequentialTime / parallelTime;

**double** efficiency = speedup / numberOfThreads;

System.***out***.println("Speedup: " + speedup);

System.***out***.println("Efficiency: " + efficiency);

}

**public** **static** **long** parallelSum(**int**[] numbers, **int** numberOfThreads) **throws** InterruptedException {

**int** partSize = numbers.length / numberOfThreads;

SumTask[] tasks = **new** SumTask[numberOfThreads];

Thread[] threads = **new** Thread[numberOfThreads];

**long** totalSum = 0;

**for** (**int** i = 0; i < numberOfThreads; i++) {

**int** start = i \* partSize;

**int** end = (i + 1) \* partSize;

**if** (i == numberOfThreads - 1) end = numbers.length;

tasks[i] = **new** SumTask(numbers, start, end);

threads[i] = **new** Thread(tasks[i]);

threads[i].start();

}

**for** (**int** i = 0; i < numberOfThreads; i++) {

threads[i].join();

totalSum += tasks[i].getPartialSum();

}

**return** totalSum;

}

**public** **static** **long** sequentialSum(**int**[] array) {

**long** sum = 0;

**for** (**int** value : array) {

sum += value;

}

**return** sum;

}

}

**public** **class** SumTask **implements** Runnable {

**private** **int**[] array;

**private** **int** start, end;

**private** **long** partialSum = 0;

**public** SumTask(**int**[] array, **int** start, **int** end) {

**this**.array = array;

**this**.start = start;

**this**.end = end;

}

@Override

**public** **void** run() {

**for** (**int** i = start; i < end; i++) {

partialSum += array[i];

}

}

**public** **long** getPartialSum() {

**return** partialSum;

}

}