PARALLEL ALGORITHMS HOMEWORK 3

BRAE TROUTMAN

(1) Define k to be the number of levels of the mergesort recursion tree (assume that $\frac{n}{b}$ is a power of two).
   (a) How many mergesort base cases are there as a function of k?
   if the recursion tree of mergesort has k levels, then the number of leaves in the recurrence tree (the number of base cases) will be 2^k leaves. This is because at each level of the recurrence tree each node can have two children, so as the number of levels grows from 0 onwards the number of leaves grows to 2^k
   (b) How many merge base cases are there as a funtion of k?
   such a function $f(k)$ would be $f(k) = k \times 2^k$. This is because for every base case of mergesort, we then have to progressively merge those solutions back together once for each level of the tree to percolate back up to our full list of sorted values.
(2) Parallelize the code correctly:
   (a) Parallelize the mergesort and merge functions using OpenMP's task and taskwait constructs.
   Parallelized code in the mergesort and recmerge functions in the submitted C code.
   (b) Write a recursive parallel function to replace the call to STL's copy function, and use the same base case size as mergesort and merge
   Implemented in the parcopy function.
(3) Explore the performance of your parallel code.
   (a) What effect does *bc* have on the performance of your parallel code? What is the best strategy for choosing *bc* given n and p? Justify your choice with empirical evidence.

Date: Tuesday 13th September, 2022.

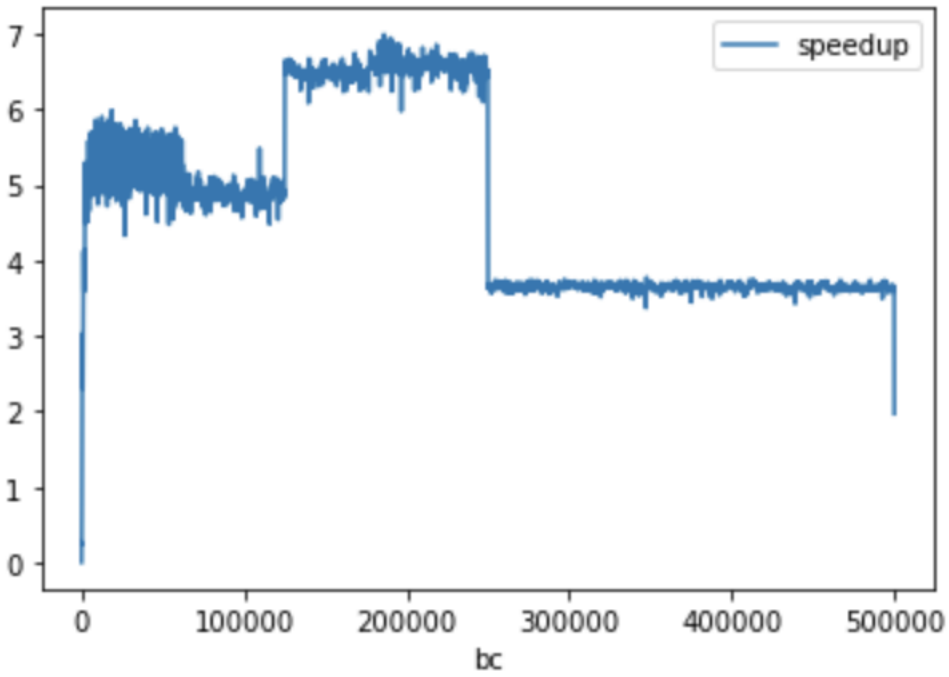

FIGURE 1. Parallel Speedup as *bc* increases. p is fixed at 8 threads.

In figure 1 we see the speedup of parallel execution as the size of the base case increases from 1 to 50000. Here, the size of the list is fixed at 1,000,000 and the number of threads is fixed at 8. As we can see here, the speedup increases quickly to about 5x and then levels out, until our base case size reaches $\frac{n}{p} = \frac{1{,}000{,}000}{8} = 125,000$. Here the speedup increases sharply and then levels out until it reaches about $\frac{n}{0.5p} = 250,000$.

(b) Explain the performance behavior you observe as you vary *bc*.
The fact that the ideal base case size being equal to $\frac{n}{p}$ is intuitive when we think about the how the recursion tree is built: when we reach the leaf nodes of the tree (aka the base cases of the mergesort) each thread will have a roughly equal portion of the list to sort all in parallel using the STL sort algorithm. Before this point, the overhead of spawning and scheduling threads outweighs the speedup we gain from executing in parallel, and after this point the closer our base case gets to the length of our list the less evenly work is distributed between threads, until in the worst case scenario our base case size is equal to the length of our list, at which point our execution is one sequential sort of the entire list.

(4) Analyze the task scheduling problem.

(a) What mergesort base cases can be executed in parallel? What merge base cases can be executed in parallel?
Because each mergesort base case is on an independent subsection of the list, they can all be executed in parallel. Merge base cases are a different story however: because mergesort modifies the full list of values in place, merge base cases cannot execute in parallel if they both depend on the mutations performed by the same

mergesort base cases below their mergesort node in the recursion tree. For this reason, two merge base cases must belong to two distinct mergesort parent nodes in the recursion tree to be executed in parallel.

(b) If you could decide on a static schedule of base cases to threads, what would it be? Under the assumption that $bc = \frac{n}{p}$, my static schedule would look like this: as we go down the recursion tree, each parent node will fork a new thread for one child and continue the same thread to execute the next child. In this way, at the leaves of the tree (base cases) the each thread will be responsible for one pth of the original input list. In other words, the number of base cases will be equal to the number of processors. After the mergesort base cases finish executing, we would traverse back up the recursion tree to merge each sorted sublist: these merges can't take full advantage of our p processors however, because at each level of the tree, our number of merges halves. Since the merges at each level depend on the results of the merges below them, we can only execute the merges one level of the tree at a time, meaning the highest number of cores we can use for parallel merging is $p/2$, with this number halving for every level we go up the tree until the final merge is executed sequentially at the root of the tree. With the specified static scheduling above however, this parallelization of merges is implicit in the structure of the tree.

(c) Instrument your code to determine how OpenMP schedules tasks to threads (it will likely vary from run to run). How does it compare to your preferred static schedule? What makes your static schedule better (or worse)?

```
Thread #0 is executing a mergeSORT base case on the list: [ 8 13 ]
Thread #7 is executing a mergeSORT base case on the list: [ 0 13 ]
Thread #6 is executing a mergeSORT base case on the list: [ 7 5 ]
Thread #5 is executing a mergeSORT base case on the list: [ 3 7 ]
Thread #4 is executing a mergeSORT base case on the list: [ 11 3 ]
Thread #2 is executing a mergeSORT base case on the list: [ 15 8 ]
Thread #6 is executing a MERGE base case on the lists:[ 11 ] [ 7 ]
Thread #3 is executing a mergeSORT base case on the list: [ 6 5 ]
Thread #1 is executing a mergeSORT base case on the list: [ 2 9 ]
Thread #2 is executing a MERGE base case on the lists:[ ] [ 5 6 ]
Thread #3 is executing a MERGE base case on the lists:[ 8 15 ] [ ]
Thread #7 is executing a MERGE base case on the lists:[ 3 ] [ 5 ]
Thread #5 is executing a MERGE base case on the lists:[ 9 ] [ 7 ]
Thread #3 is executing a MERGE base case on the lists:[ 2 ] [ 3 ]
Thread #5 is executing a MERGE base case on the lists:[ 15 ] [ 9 ]
Thread #4 is executing a MERGE base case on the lists:[ 5 6 ] [ ]
Thread #7 is executing a MERGE base case on the lists:[ 8 ] [ 7 ]
Thread #1 is executing a MERGE base case on the lists:[ ] [ 2 3 ]
Thread #0 is executing a MERGE base case on the lists:[ 13 ] [ 13 ]
Thread #2 is executing a MERGE base case on the lists:[ 0 ] [ 8 ]
Thread #0 is executing a MERGE base case on the lists:[ ] [ 13 13 ]
Thread #3 is executing a MERGE base case on the lists:[ 5 7 ] [ ]
Thread #5 is executing a MERGE base case on the lists:[ 3 ] [ 0 ]
Thread #4 is executing a MERGE base case on the lists:[ 11 ] [ 8 ]
Thread #6 is executing a MERGE base case on the lists:[ 6 ] [ 7 ]
Thread #1 is executing a MERGE base case on the lists:[ 2 ] [ 0 ]
```

```
Thread #4 is executing a MERGE base case on the lists:[ ] [ 11 13 ]
Thread #0 is executing a MERGE base case on the lists:[ 15 ] [ 13 ]
Thread #2 is executing a MERGE base case on the lists:[ 7 8 ] [ ]
Thread #7 is executing a MERGE base case on the lists:[ 3 ] [ 3 ]
Thread #5 is executing a MERGE base case on the lists:[ 5 ] [ 5 ]
Thread #3 is executing a MERGE base case on the lists:[ 9 ] [ 8 ]
```

In the output above we instrumented our code with a critical section at each base case in merge and mergesort to print the base case to the console (this had to be a critical section to prevent interleaving of output). Looking at this output, we see that in terms of the mergesort base cases OpenMP's dynamic scheduling executes exactly as our static schedule: each base case of the mergesort is assigned to its own thread. For the merge base cases, OpenMP does not achieve maximum parallel efficiency because some threads execute more merges than others: for example, thread #0 merges 3 base cases, whereas thread #1 does 2. All in all though, under the same assumption as our static schedule (that $bc = \frac{n}{p}$, this dynamic scheduling is just as performant, in that both share the issue of declining parallelism as we traverse back up the levels of the tree with merges.

(5) Evaluate your parallel performance.
   (a) Report your parallel performance (speedup and efficiency) for a significantly large problem from 1 to 44 threads. Always compare to the sequential performance of STL's sort function.

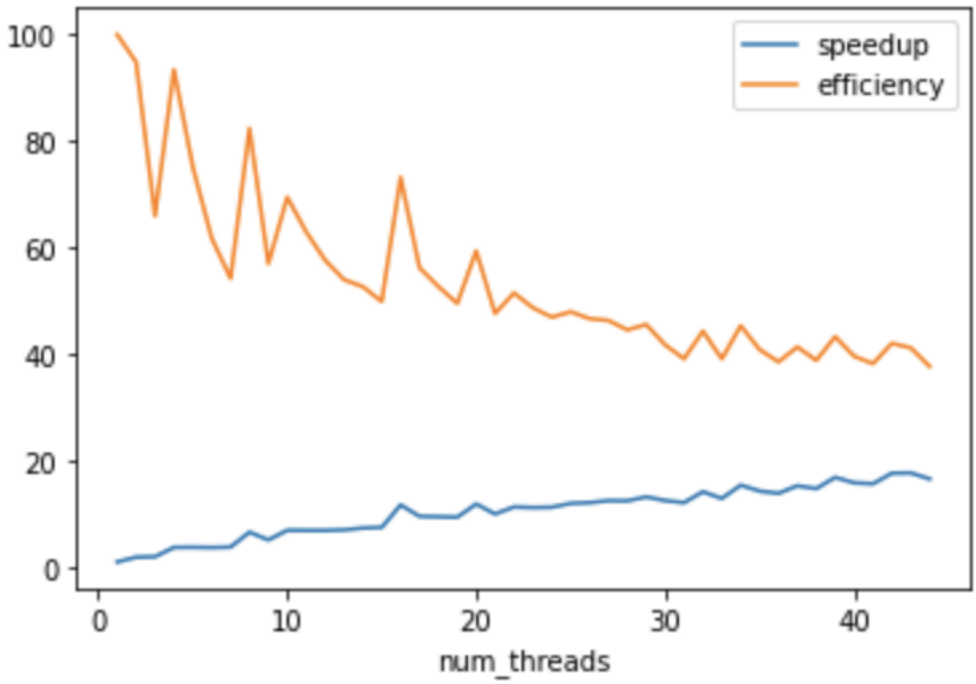

FIGURE 2. Parallel speedup and efficiency as number of threads p increases. n is fixed at 2^{24} and bc is always equal to $\frac{n}{p}$

(b) Why does efficiency differ between p being a power of two and not?
When p is a power of two, the division of work between threads is more evenly distributed at the base of the tree (the leaf nodes/base cases) because the mergesort recursion tree has 2^k leaf nodes.

(c) What are the barriers to better parallel scaling?
Lack of parallel efficiency in the merge operations is one barrier to better scaling. Another is the cache complexity of the algorithm: in a LPRAM model each processor has their own local copy of some subsection of the input array, varying in size depending on the size of the cache lines. When one mergesort or merge base case thread modifies it's subsection in-place, overlapping cache-lines loaded in local memory on other processors get marked as dirty and have to be reloaded into memory before they can be read again locally. This leads to high communication overhead that increases with the number of processors.