Problem 1: Define k to be the number of levels of the mergesort recursion tree (assume that n/b is a power of two).

1. How many mergesort base cases are there as a function of k?

   2^k

   We assume that there are k levels in the recursion tree of mergesort. For each recursion, the parent node will fork two children, so the number of each level of this tree is a power of 2, so the number of the leaf nodes (the number of the base cases) will be 2^k.

2. How many merge base cases are there as a function of k?

   $k * 2^k$

   For the base case level in the tree of merge, it needs to merge different base cases and return back to it until we have the sorted array. So, we have k levels that need to merge and each one has 2^k to be processed.

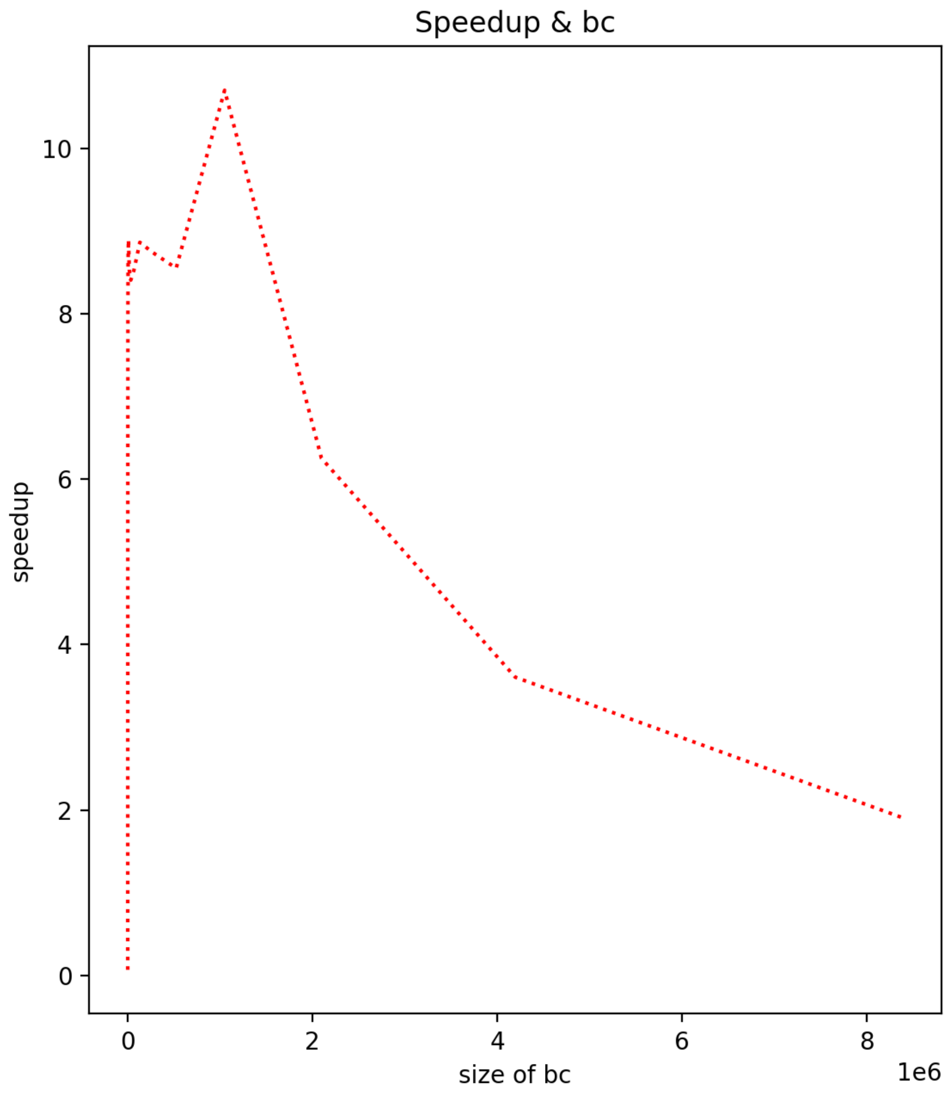
Problem 3: Explore the performance of your parallel code

1. 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.

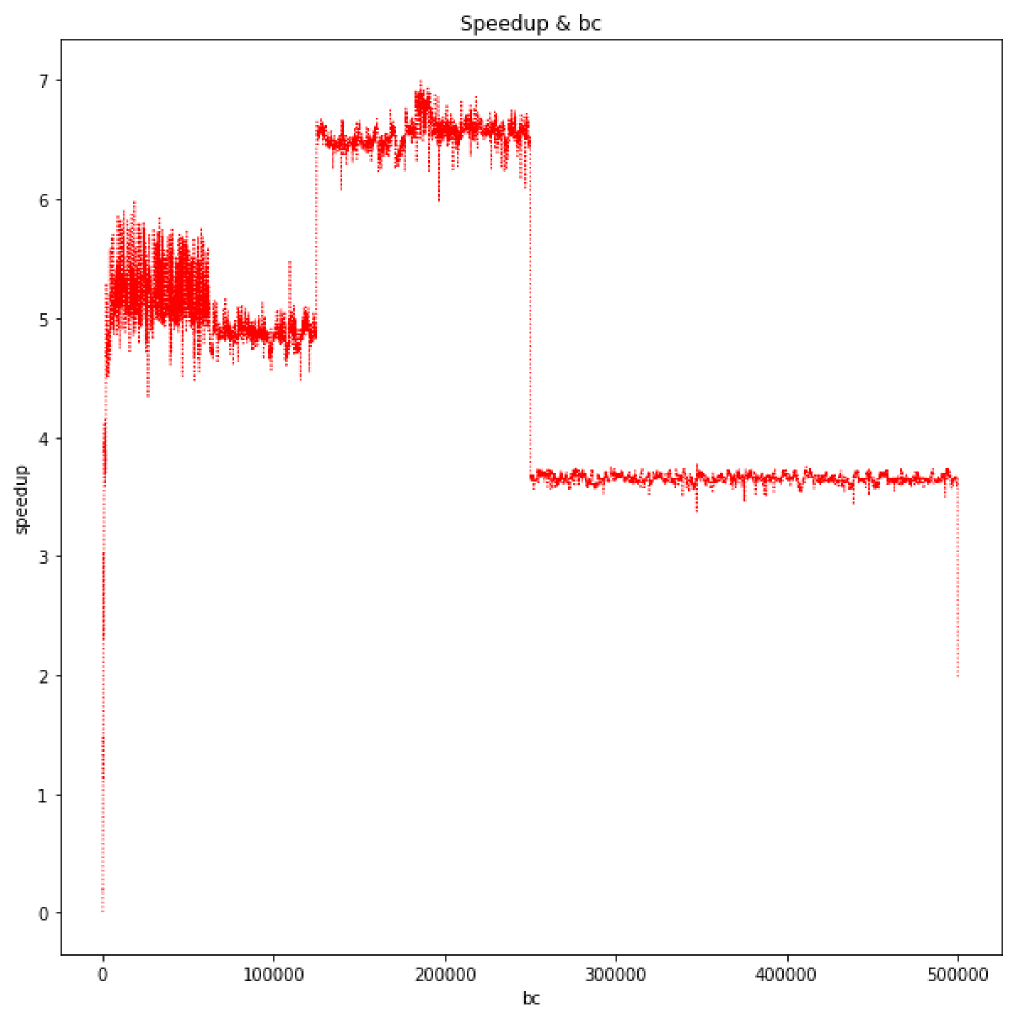
   I think the ideal strategy is $bc = \frac{n}{p}$, in which the number of the base cases are p. In this situation we can make full use of all the processors at the base case level. If $bc > \frac{n}{p}$, the number of the base cases is less than p, which means there are some processors not working (although I also think this situation is suitable for doing merge base cases because we need more processors than mergesort); If $bc < \frac{n}{p}$, the number of the base cases is greater than p, which means we need to calculate the scheduling cost and data locality cost (but I found $bc < \frac{n}{p}$ performs better in some situations, it means scheduling cost is less than the cost of the processing of larger base cases).

2. Explain the performance behavior you observe as you vary bc.

   From what I observed from the result of different, I find the base case size can influence the performance of the code if it is too small or over large, and the code will perform better when the size comes to $\frac{n}{p} * k \; and \; k \geq 0$. I let the size of the base cases from 2 to $\frac{n}{2}$, and the speedup is raised sharply at first and will go flat when the base case size gets larger and in the end as the size increases, there will be some fluctuations or even decrease. For example, when I test the case that $\{length = 50,000,000; p = 20;\}$, we assume that all base cases start from 20, and its growth is multiplied by two. Besides, when I tested the case that $\{length = 2^{24}; p = 2^{4};\}$, we assume all the bases are the power of 2. Here, I found when the $bc = 2^{16}$, it has a greater speedup as 10.706.



Here, I found when the $bc \approx 2^{22}$, it has a greater speedup as 11.237. The following plot is based on $\{length = 1,000,000; p = 8;\}$



Problem 4: Analyze the task scheduling problem

1. What mergesort base cases can be executed in parallel? What merge base cases can be executed in parallel?

   After the discussion, I think all the base cases of mergesort can be executed in parallel. But as for the base cases of merge, we need to require that only they belong to the different parent nodes in the recursion tree.

2. If you could decide on a static schedule of base cases to threads, what would it be?

   For the recursion tree, we will process, we let each parent node generate a new thread for one of its children and let another child use its own thread. Here, we require all parent nodes for their children even if there are no processors available. We are trying to make the number of the base cases and the number of threads equivalent to make full use of processors, under the assumption that $\#bc = \#p$. This way, we can evenly divide the leaf nodes and let their parent nodes continue to use their children's processors, which can help us reduce the cost of data communication when doing the merge. Also, there will be some other situations. If the number of threads is less than the number of base cases, we let the extra base cases wait until we have rested processors and process them. This way, we try to keep finishing the same level in the recursion tree first.

3. 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)?

For static scheduling, we are based on knowing its specific process and recursion tree, and we also assume that the Sort function of STL is cache-optimal. So after we deal with base cases, we can use the same processor for their parent nodes, because part of the elements is the same, so we don't need to load all the data we need, which saves the cost of data communication. Compared with OpenMP, it uses dynamic scheduling that can't make sure the parent nodes use the processors from the children's nodes.

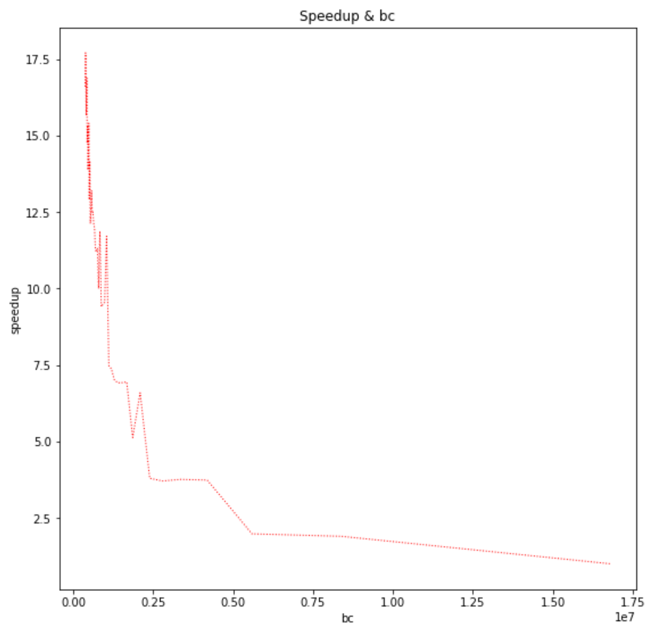
Problem 5: Evaluate your parallel performance.

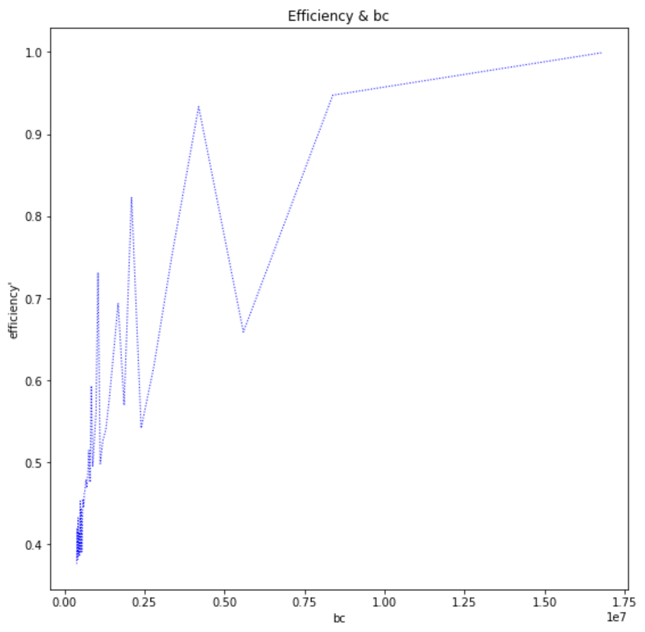
1. Report your parallel performance (speedup and efficiency) for a significantly large problem from 1 to 44 threads. Always compare to the sequential performance of STL's sort function.

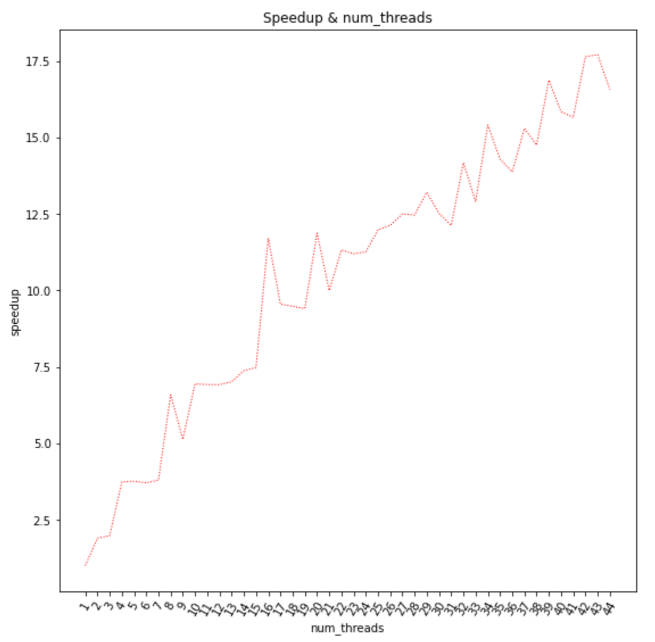
Here, we are testing based on $\{length = 2^{24}; bc = \frac{n}{p}\}$. When the num of threads is 1, the efficiency and the speedup are 1.0.

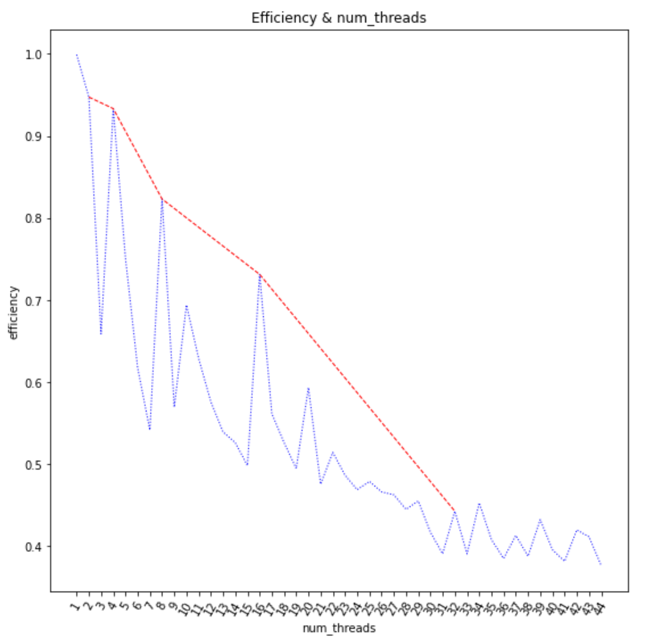
As the number of threads increases, the speedup also increases, with small fluctuations in the process. I think when the number of threads rises, the base case size becomes smaller so that the processors can do the task faster, besides we have more processors. In this situation, the scheduling cost for processors is likely to be less than the cost to process a big base case.

As for the efficiency, it's going down continuously, but unlike "speedup", there are big fluctuations under different conditions. From the graph, we can find that it performs better on efficiency when the number of threads is power of 2. For example, when the number of threads is equal to 4 and 16, its performance in efficiency is similar to that when the number of threads is equal to 2 and 6, but the speedup is significantly faster.









2. Why does efficiency differ between p being a power of two and not?

I think that p can be evenly divided for each base case when p is a power of 2. For the base cases, there are 2^k nodes in total and all the processors can work at the same time in ideal conditions. Therefore, there is no rested processor when p is a power of 2.

3. What are the barriers to better parallel scaling?

After discussing with Brae, he came up with the great assumption that one of the barriers is “Dirty Cache”. “Dirty Cache” means that when we want to load new data to a new assigned processor, we need to empty all the data in that processor and load.