ECE 572 Project2 report - Pthread

Hua Liu, Jianqin Gao, Siyuan Zhong

1. Question 1 - Sorting Algorithms Parallelisation

1.1 Quicksort:

- **Sequential quicksort**:
    - Pick an element, called a *pivot*, from the array.
    - Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it
    - Recursively apply the above steps to the subarray of elements with smaller values and separately to the subarray of elements with greater values.
- **Version 1 of parallelisation**: make the algorithm be parallel in the 'partitioning' step mentioned above. Specifically, for initial array *a*, in a current thread *t0* with target subarray from *a[left]* to *a[right]*, every time one *pivot* is picked and the array is rearranged, one additional branch-thread *t1* is created to handle subarray from *a[left]* to *a[pivot]*. While *t0* is in charge of sorting subarray from *a[pivot+1]* to *a[right]* only.
    - To prevent too many threads are created, before each thread's creation, we used 'depth' of current thread and the length of its target subarray as conditions to decide whether further derive another subthread. However, it didn't result in big performance improvements.
    - Problem: This parallel quicksort algorithm is likely to do a poor job of **load balancing**. In our implementation of this version, no performance improvement observed.
- **Version 2 of parallelisation**: We came up with another version of parallel mechanism and, in this version, performance improvement is achieved for multiple test cases. What's more, it can be easily extended to parallelize many other sorting algorithms. The method is described as the following:
    - Hash the original N-elements array into NUM_THREAD (i.e. The number of threads) chunks of subarrays with size.
    - Each thread working on sorting their corresponding chunk in parallel.
- **Theoretical complexity analysis**:
    - Serial execution, in average, is of time complexity nO(log n)
    - Parallel execution:
        - in theory, is of time complexity nO(log n)/K, where K is the number of threads.
        - Hashing step is of time complexity O(n) and space complexity O(n).
        - In total, the parallel version if of time complexity O(n) + nO(log n)/K = nO(log n)/K.
- **Experiment:** In the experiment, to explore the performance improvements in pre-hashed quicksort parallel execution. We tested it with multiple thread number(specifically,

{10,100,1000,10000}) for different data size N. And the log-log plot of execution time is shown in Figure 1, with corresponding speedup(i.e. Serial execution time divides parallel execution time) plot shown in Figure 2.

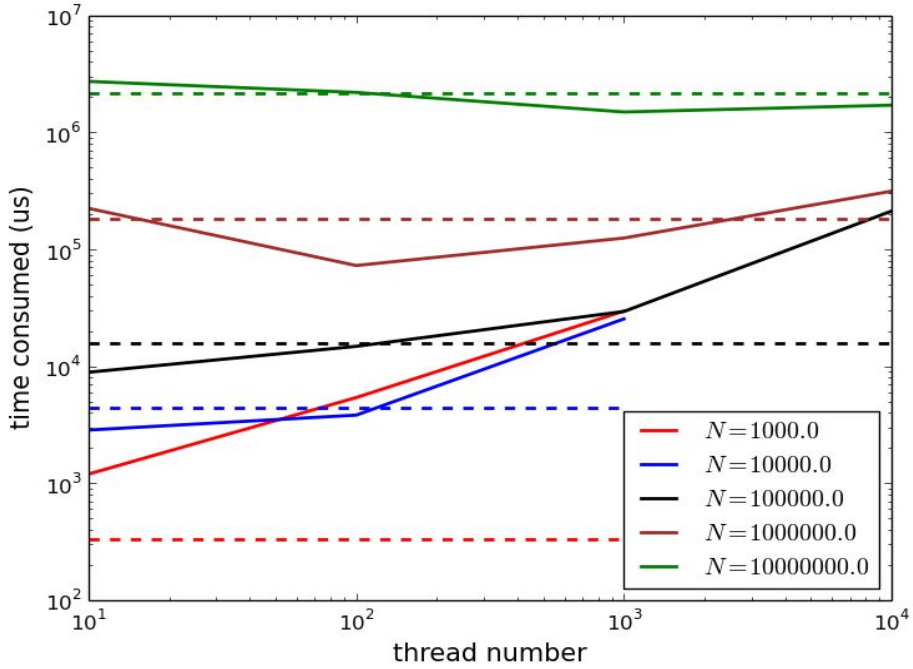


Figure 1. Log-log plot of time consumption vs thread number for different test data size N. The dash lines represent serial quicksort execution time. While the solid lines represent the pre-hashed parallel quicksort execution time with different thread number.

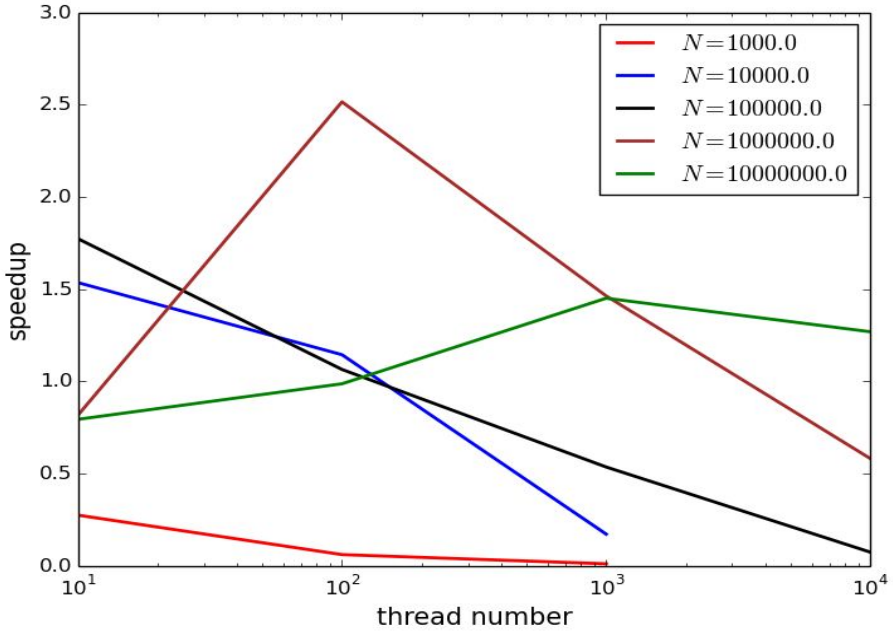


Figure 2. The plot of time consumption vs thread number for different test data size N in pre-hashed quicksort parallelization. The dash lines represent serial execution time. While the solid lines represent the parallel execution time with different thread number.

- **Result 1**. From figure 1, it is clear that, when N is small, specifically, 1K and 10K, there is no performance gain achieved by parallelisation. However, when N is 100K, with thread less than 100 there is performance improvement. What's more, as N grows to 1M and 10M, the range of threads that has a boost in performance gets widder: 20 to 2000 for N equals to 1 M, and 100 to 10000 for N equals to 10 M. This means that **both the time saved by multithreading and the extra time consumption due to thread creation overhead and thread communication does not grow in linear against thread number.** While speedup is 1/k in theory, in practice, the it displays a trend pattern: from low to high, then back to low again. We believe that, the main reason for low speedup when thread num are small is that **the time saved by hiding memory access latency using multithread is not big and is largely wasted by the creation overhead**. As more threads are created, **the speedup by multithreading has a grows quicker than the speed down due to growing overhead and communication**. While when thread is extremely big, **the management of threads(such as communication and context switching) exploded**, which results in the speedup lowing down again.
- **Result 2**. The thread range of improved performance in parallel execution compared with their serial version shifts to the right as N grows bigger. This means that **the "sweet spot" for different N is different. In addition, the bigger the N, the bigger, or say the righter. the "sweet spot"**.

1.2 Radix Sort:

A Least significant digit (LSD) Radix sort is a fast stable sorting algorithm which can be used to sort keys in integer representation order. Keys may be a string of characters, or numerical digits in a given 'radix'. The processing of the keys begins at the least significant digit (i.e., the rightmost digit), and proceeds to the most significant digit (i.e., the leftmost digit). The sequence in which digits are processed by an LSD radix sort is the opposite of the sequence in which digits are processed by a most significant digit (MSD) radix sort.

- **Sequential radix sort**:
  - Take the least significant digit (or group of bits, both being examples of radices) of each key.
  - Group the keys based on that digit, but otherwise keep the original order of keys.
  - Repeat the grouping process with each more significant digit.
- **Version 1 of Parallel implementation**:
  - Partitioned the array into several parts
  - Count the frequency of the digit for each number in the different parts of array in parallel
  - Merged the counting result

- **Theoretical complexity analysis**:
    - For the serialization implementation, we need to traverse the array and count the digit frequency from least significant digit to the most significant digit for each element of the array. Therefore, the time complexity should be O(mn) for m elements in the array and the longest one has n digits.
    - For the parallel implementation, we divided the array into N parts with identical size and count the frequency in parallel. After finished counting, we summarized the counting results of each digit and order it like serialization implementation. Ideally, the time complexity is O(mn/N).
- **Experiment:** In the experiment, to explore the performance improvements in pre-hashed quicksort parallel execution. We tested it with multiple thread number(specifically, {10,100,500,1000}) for different data size N. And the log-log plot of execution time is shown in Figure 3, with corresponding speedup(i.e. Serial execution time divides parallel execution time) plot shown in Figure 4.

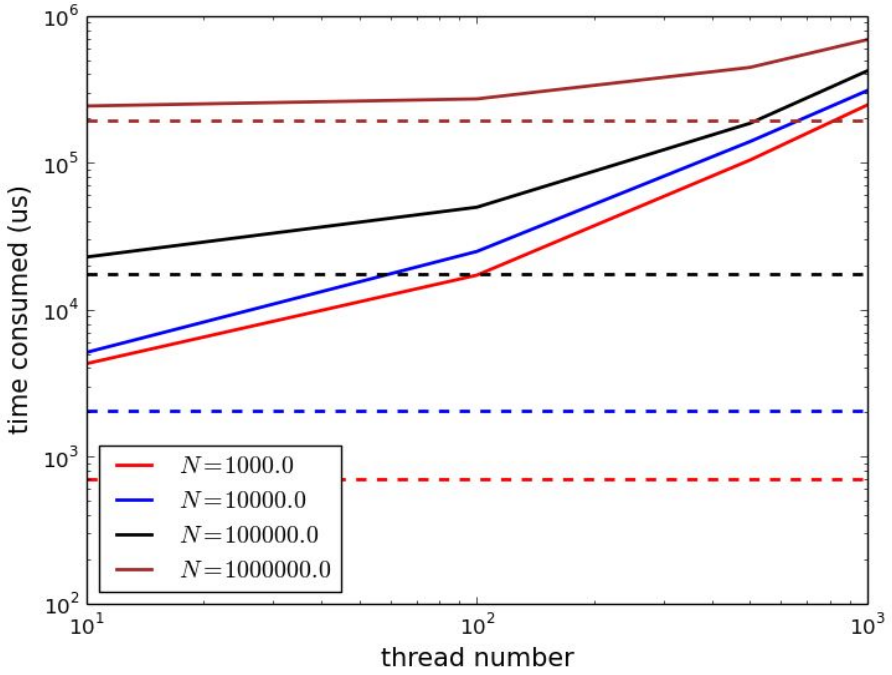


Figure 3. Log-log plot of time consumption vs thread number for different test data size N. The dash lines represent serial radix sort execution time. While the solid lines represent the version 1 parallel radix sort execution time with different thread number

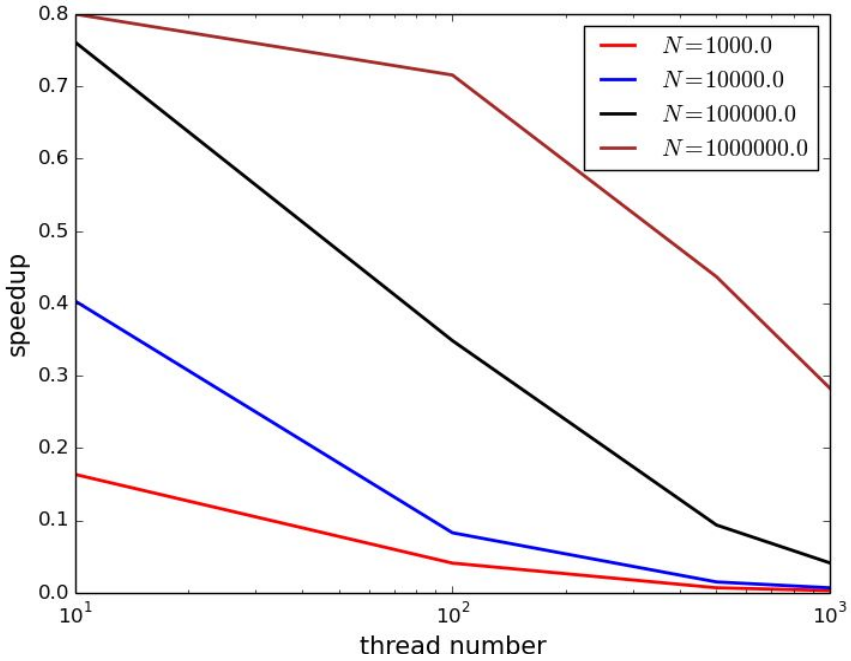


Figure 4. The plot of speedup vs thread number for different test data size N in version 1 radix sort parallelization. The dash lines represent serial execution time. While the solid lines represent the parallel execution time with different thread number.

  - **Result Analysis:** As we can see in the above graphs, the performance of the parallel implementation of the radix sort is not as good as we expected. What we can observe is that when the test array size getting larger, the performance gap between the serial and parallel implementations will be reduced and with more and more threads we created the performance keep decreasing. The possible reason we guess is that there will be too much overhead during the thread creating and synchronizing stage. So far the largest input array size we test is one million, which relatively is not a large number and the parallel part we implemented in the program accounts for a small proportion. Therefore, the performance of our parallel implementation did not meet our expectation.

- **Version 2 of Parallel implementation**: We applied the pre-hashing method described in the version 2 of quicksort parallelization above to radix sort. But it turned out the performance improvements wasn't as we expected.
- **Theoretical complexity analysis**:
  - Serial execution is of time complexity O(mn) as mentioned above.
  - Similar to pre-hashed quicksort parallelization, pre-hashed radix sort parallelization is supposed to be of time complexity O(mn)/K, where K is the number of threads used.
- **Experiment:** This experiment is similar to the experiment for pre-hashed quicksort experiment. So the details is not repeated here. And the result is illustrated in Figure 5 and 6.

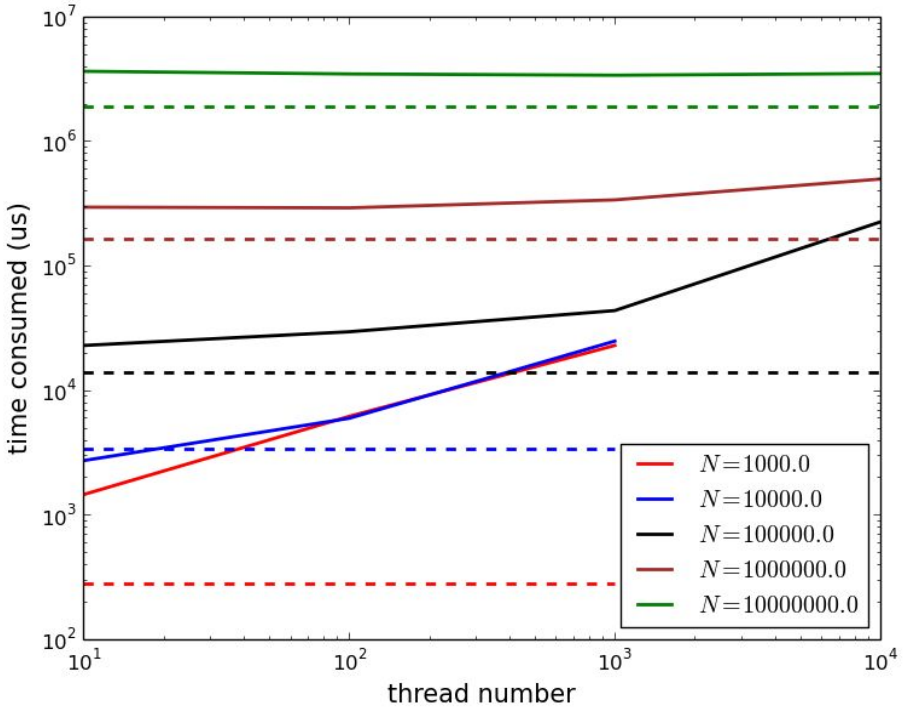


Figure 5. Log-log plot of time consumption vs thread number for different test data size N. The dash lines represent serial radix sort execution time. While the solid lines represent the pre-hashed parallel radix sort execution time with different thread number

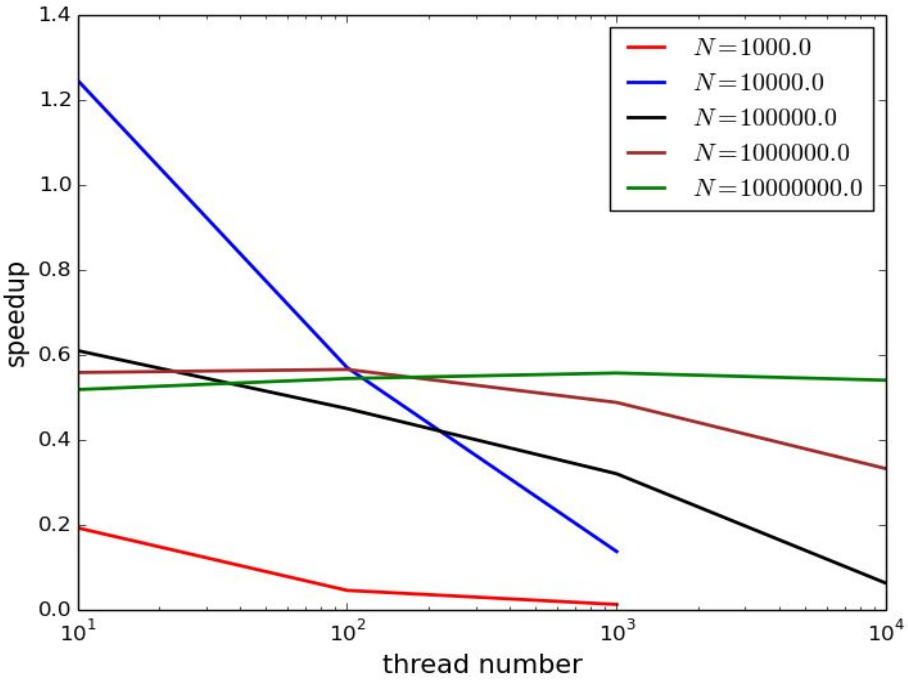


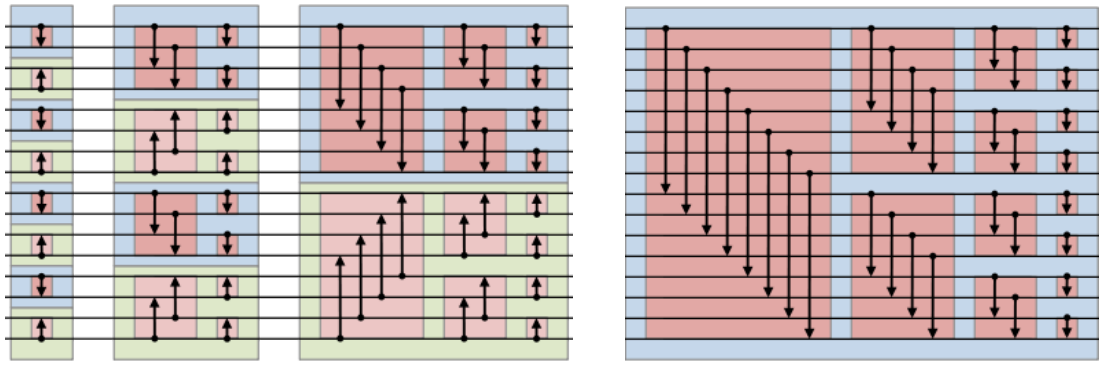
Figure 6. The plot of speedup vs thread number for different test data size N in pre-hashed radix sort parallelization. The dash lines represent serial execution time. While the solid lines represent the parallel execution time with different thread number.

- **Result 1:** The performance of the **pre-hashed parallelisation is better than serial implementation only for N=10000 with thread number 10**.
- **Result 2:** Except for that, all the other cases didn't achieve the performance boost as we expected. And they got worse when we tried to launch more threads. This is because in this algorithm, with our implementation, **the frequent context switching and overhead of thread creation consumed more time than that saved by multithreading**.

1.3 Bitonic Sort

The algorithm, created by Ken Batcher in 1968, consists of two parts. First, the unsorted sequence of size N is built into a bitonic sequence, as is shown in Figure 7 (a); Then, the series is split multiple times into smaller sequences until the input is in sorted order, as is shown in Figure 7(b). This algorithm can **only accept input array of size** 2^n , where n is an integer.

In the example illustrated in figure 7(a), there are 3 steps. Each step consists of multiple blue or green boxes. We call each blue or green box a **moving-arrow-sort process**. Each moving-arrow-sort process starts with a **chunk size L**: step 1 starts with L being 2; step 2 starts with L being 4; step 3 starts with L being 8. Each moving-arrow-sort process is a recursive process, which stopped when L is decreased to 2. Each moving-arrow-sort process has a certain sorting direction which can be determined in the beginning of each step.



(a) ***(b)***

Figure 7. Bitonic sort. (a) is the forming bitonic sequence step; (b) is the sorting bitonic sequence step.

- **Sequential bitonic sort**:
  - Forming bitonic sequence:
    - Initiate chunk size L as 2 and update it after each step by L=2*L.
    - Within each step, apply moving-arrow-sort process to recursively process chunk of size L: after each sub-step(illustrated as pink boxes in Figure 7), update L by L = L/2, until L is decreased to 2.
  - Sorting bitonic sequence:

        - Apply moving-arrow-sort process to the whole bitonic array of size N with L equals to N.
- **Parallel implementation**:
    - Our implementation of parallelism is at the initialization of each step.
    - We take the initial chunk size L as the input, and, by some predefined rules (which are fine tuned after multiple experiments), we decide whether we will parallelism the processing in this step. And the way the rules are found will be described in the Experiment part later on.
    - If L is eligible for parallelization, we create N/L threads to process the N/L chunks in parallel by moving-arrow-sort process.
- **Theoretical complexity analysis**:
    - When run in serial, the bitonic sorting network completes its work in $O(n \log^2 n)$ comparisons, which falls short of the ideal comparison-based sort efficiency of $O(n \log n)$.
    - In our parallelisation, we only parallelised the chunk processing in each step, but the moving-arrow-sort process can be further parallelised. So there still be a big part of serial processing. Thus, by ***Amdahl's law***, theoretically, this parallelisation is supposed to has time efficiency less than $(1/K)O(n \log n)$, with K threads created in total.
- **Experiment:**
    - Given N, to find out the optimum range of chunk size L where the parallel execution is faster than serial execution. We did experiments and recorded the execution time:
        - Sort the same sequence of size N with serial and parallel implementation.
        - In step of initial chunk size L, the parallel implementation uses N/L threads to execute in parallel.
        - Record the time of each step which has the initial chunk size L for both serial and parallel execution. The result is plotted in Figure 8 in log-log fashion.
        - From this figure: (1) for different N, the optimum range of L is different; (2) **the corresponding range of parallelization eligible L is found and recorded in the row 3 of Table 1**.
    - Using the found optimum range of chunk size L for each array size N, we implemented the parallelization of bitonic sort. And the result is shown in Table 1.

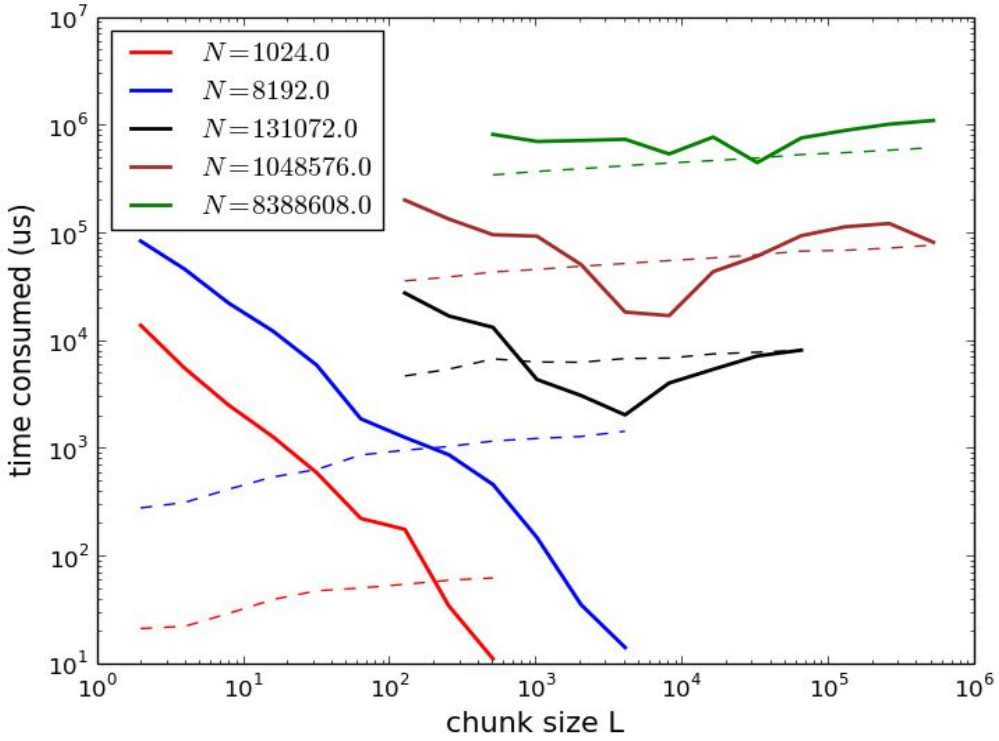


Figure 8. The plot of execution time for different chunk size L. The dash lines represent serial execution. While the solid lines represent parallel execution.

Array size N	1024 (~1K)	8192 (~10 K)	131072 (~100K)	1048576 (~1M)
Serial execution time (us)	524	3940	87792	932745
Optimum range of chunk size L for parallel execution	[2^8, 2^9]	[2^8, 2^12]	[2^10, 2^15]	[2^10, 2^15]
Thread number	6	62	252	96
Parallel execution time (us)	486	3563	69634	930425

Table 1. The serial and optimum parallel execution of bitonic sort

- **Result 1**. From Table 1, by comparing the second row and last row, it is clear that **for all the tested array size N, parallel execution achieved better results than serial execution**.
- **Result 2**. The best optimization is achieved when array size is 87792 with thread number 252, which is not as big as 252 times of speedup as calculated in theory. **This is because our computer has a local constraint in terms of catch size, overhead of creating threads, and extra time for thread context switching, which compromises some of the speed boosting from parallelisation**.

1.4 Question 1 summary

- **Question 1**: Which of the three algorithms when run sequentially has the best performance in order of Complexity?
    - The theoretical complexity for quicksort algorithms is O(nlogn). Radix sort complexity is O(wn) for n keys which are integers of word size w. Bitonic Sort have time complexity as $O(n \log^2 n)$. So in theoretical analysis we found that quicksort have the best performance among this three sorts. And from our experiment, we also find quicksort will have best performance.

- **Question 2**: Which of the three algorithms when run in parallel has the best performance in order of complexity? Why, how?
    - In theory, bitonic sort is very suitable for parallelization naturally. Because, in each step, there are many chunks to be processed in a uniform fashion without affecting other chunks. However, due to our un-fully explored parallelization(we only parallelised the chunk processing in each step, but the moving-arrow-sort process can be further parallelised) and some of the hardware constraints(maximum thread number is about 1 M) of our local machine, bitonic sort didn't achieve the biggest performance boost among the considered three.

- **Question 3:** Do you results agree with the analysis above? Why yes, why not?
    - No, the performance of our implementations are not in line with our expectations. The general case is that when we launch more threads, the speed of the program will be slower. For this phenomenon, the possible reason we guess is that there will be much overhead in the thread creating and synchronizing stage. In the meantime, the parallel implementation part only take very small proportion in the whole workflow while the input array size is also not large enough.When the datasize number is small, we find that the parallel performance have lowest speedup, only for the big data, the parallel will have some good performance.

2. Question 2 - Gaussian Elimination

- **Sequential gaussian elimination**:

```
1.      procedure GAUSSIAN_ELIMINATION (A, b, y)
2.      begin
3.          for k := 0 to n − 1 do                  /* Outer loop */
4.          begin
5.              for j := k + 1 to n − 1 do
6.                  A[k, j] := A[k, j]/A[k, k];   /* Division step */
7.              y[k] := b[k]/A[k, k];
8.              A[k, k] := 1;
9.              for i := k + 1 to n − 1 do
10.             begin
11.                 for j := k + 1 to n − 1 do
12.                     A[i, j] := A[i, j] − A[i, k] × A[k, j]; /* Elimination step */
13.                 b[i] := b[i] − A[i, k] × y[k];
14.                 A[i, k] := 0;
15.             endfor;          /* Line 9 */
16.         endfor;              /* Line 3 */
17.     end GAUSSIAN_ELIMINATION
```

- **Parallel implementation:**
  - We still do the outer loop part in sequential
  - We create some of thread in the inner loop.
  - In the division loop part:
    - We can let the for loop rewrite in this form
    - for j = k+1+threadid; j < N; j += num_threads
    - In this way, we can let the column element in k row be parallelism
    - Barrier all threads done
  - In the elimination part:
    - We can let the for loop rewrite in this form
    - for i = k+1+threadid; i < N;i += num_threads
      - m = A[i][k]/A[k][k]
      - for j = k+1; j < n; j++
        - A[i][j] = A[i][j] - m* A[k][j]
    - This make after k row, we let each row do elimination be parallelism
    - Barrier all threads done

- **Theoretical complexity analysis**:
  We have already discussed the sequential and parallel implementation in previous. From the pseudo-code we can see that in the sequential way, we use O(n^3) time complexity. And in the parallel version, because we only do parallel in division step and elimination step, so we will only use O(n^2 /thread_count) time for this part. So the ideal speedup for this parallel program is thread numbers. But in the code, we decided to implement the two version parallel code, one is not use threads in division loop part, i.e do sequential operation in the division part. And another

is use pthreads in division part, because this part we only need to process “n” datas, compared to elimination will process n^2 data, it is less data. So it may cost more time due to overhead. In this experiment we compared the results between them. Found that version 2 without pthread in division part run more fast.

- **Experiment Analysis:**
    - This is the experiment result for the sequential, different version code, different threads result.

time(us)	10	100	1000
sequential	3	1547	15579140
version_1, 2 threads	1204	11175	426208
version_1, 4 threads	2406	22489	519011
version_2, 2 threads	699	8284	411100
version_2, 4 threads	1001	11584	328878

Table 2. Gaussian Elimination with 10, 100, 1000 matrix dimension

Version 1: parallel in the division part
Version 2: sequential in the division part
Because when matrix dimension grow, the compute time will increment hugely, so the largest dimension we use is 1000.
From the above table, we can see that with the matrix dimension increase, parallel will use less time. Because in the lower dimension, the thread may have overhead time, this will waste lots of time. We believe when the dimension increase to 1000, 10000, the parallel also will use less time. And compare the performance between this two version, we found that in this low data dimension, version 2 run more fast. It may because in the division part, we only do n times data operation, if we use parallel, it will cost much more time in threads overhead. So the main parallel part for this program is in elimination part, this will do n^2 data operation, which will spend lots of time and parallel can save the time.