Assignment 1 - CS671

Performance Modeling

- Ashish Jindal (AJ523)

Resource in the submission

1. assignment0-cs671.pdf - Assignment 0, environment setup and running commands on elf.
2. Assignment1-cs671.pdf - This file.
3. Code -
   a. seqMS.h and seqMS.c - Sequential merge sort implementation
   b. parallelMS.h and parallelMS.c - Parallel Merge sort implementation
   c. Main.c - Test code that accepts 2 command line arguments data size and number of threads.
   d. Makefile
4. Scripts -
   a. Clean.sh - To remove all stats related files and all the obj files.
   b. Stats.sh - It will call ./main with predefined data-sizes and thread counts to obtain stats and dump them into two folders 1)fixed_data_stats and 2)var_data_stats
   c. Speedup.sh - It can be called after you run Stats.sh to use the stats data obtained to calculate the speedup factor.
5. Fixed_data_stats - Stats generated for fixed data size with varying number of threads
6. Var_data_stats - Stats generated per thread for varying data sizes.

How to run the code?

1. Run make
2. ./main 10000 3
3. Above command runs the sort algorithm on 10000 random items using 3 threads
4. To invoke sequential merge sort algorithm invoke the command - ./main 10000 1
5. 1st argument = data set size
6. 2nd argument = number of threads

Problem A

Ts(n) = Time for serial algorithm

Tp(n,p) = Time for parallel algorithm using p threads

Ts(n) = n

Tp(n,p) = n / p + log2(p)

Had there been no parallel overhead the value of Tp(n,p) would have been n / p

1. Expression for speedup -

Speedup = Ts(n) / Tp(n,p) = n / (n / p + log2(p)) = np / (n + p*log2(p))

Yes the calculation will hold for non uniform processors as well as we consider Tp as total time taken in parallel mode irrespective of individual time taken by cores.

2. Increase in problem size for constant efficiency -

Efficiency = Speedup / p = (np / (n + p*log2(p))) / p = n / (n + p*log2(p))

If we increase the p by a factor q then let k be increase in the problem size for constant efficiency -

Therefore -

n / (n + p*log2(p)) = k*n / (k*n + p*qlog2(p*q))

Solving the above equation we get following value of k

K = q*(1 + log2(q) / log2(p))

3. Does linear speedup imply Strong scalability -

Yes it is strongly scalable because the efficiency in such a scenario will always be one (No parallel overhead)

Ts(n) = n

Tp(n,p) = n / p

Speedup = Ts(n) / Tp(n,p) = n / (n / p) = p

Efficiency = Speedup / p = p / p = 1

Also if the problem size remains ‘n’ and number of threads is increased to q such that q > p then also the efficiency will be one.

Problem B

1. How long does sequential merge sort program take for varying data sizes -

Data Size	Time (Seconds)
10000	0.004189
100000	0.051030
1000000	0.531103
10000000	3.234650
100000000	29.433559

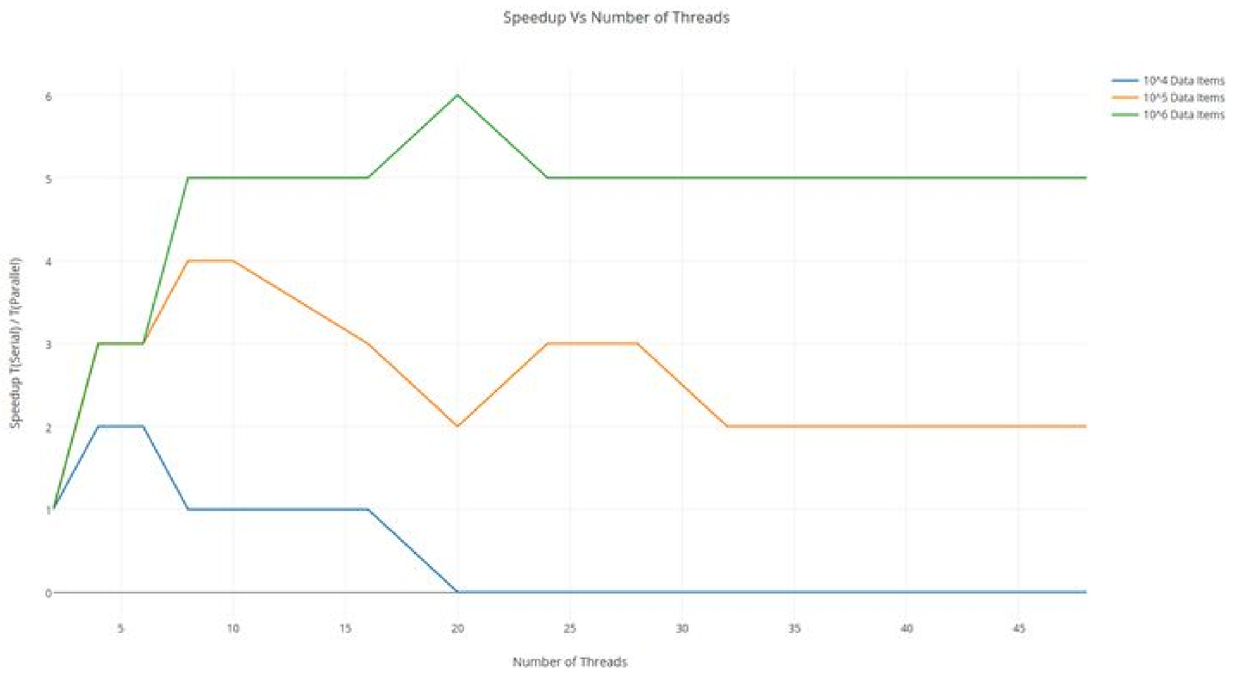
2. Pseudo Code for parallel mergesort -

```
2.1.  Void mergeSort(int* a, int l, int r, int num_threads) {
2.2.  If (l < r) {
2.3.  Int m = (l + r) / 2
2.4.  If (num_threads > 1) {
2.5.  Create a new thread and let it process left half of tree (l - m) also pass it num_threads = num_threads / 2
2.6.  Let the current thread process the right half of tree with num_threads = num_threads - num_threads / 2
2.7.  }
2.8.  Else { // Num_threads = 1
2.9.  Let current thread process both the halves of the tree as in sequential merge sort algorithm
2.10. }
2.11. Perform a merge of the sorted let sub-array and right sub-array
```

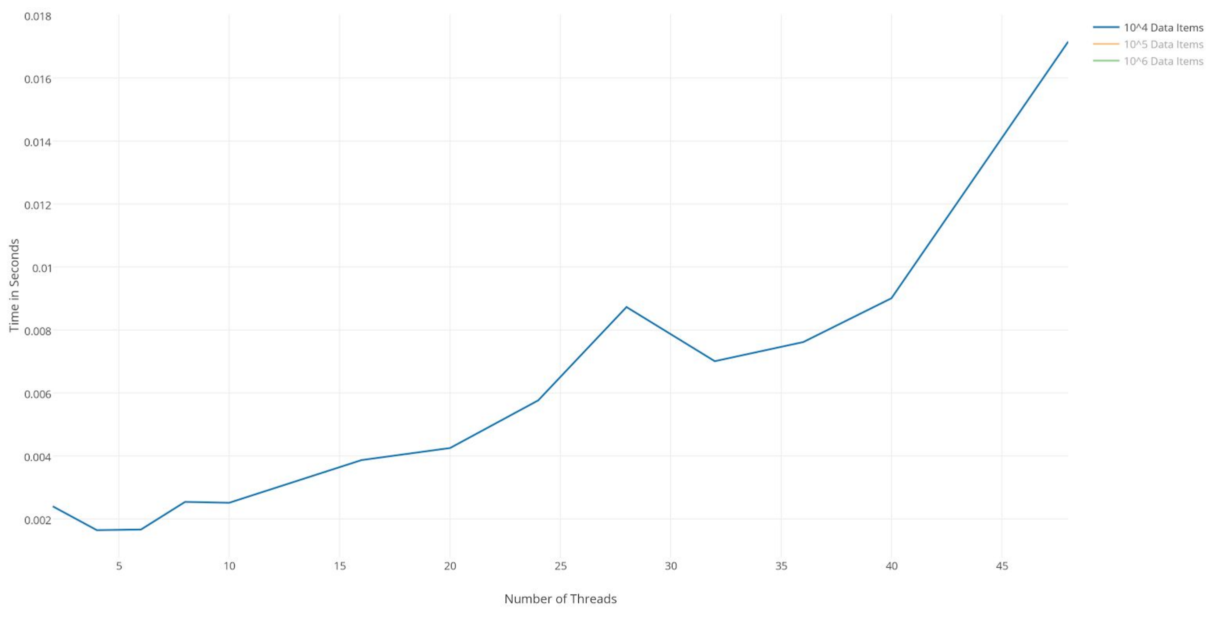
2.12. }

3. Parallel merge-sort implementation - File parallelMS.c.
4. Plots at the end of document.
5. No, One thread doesn't necessarily mean one core. It's up to the scheduler to map user threads to system cores. In a busy environment scheduler may be tempted to schedule all the threads on one core as well.

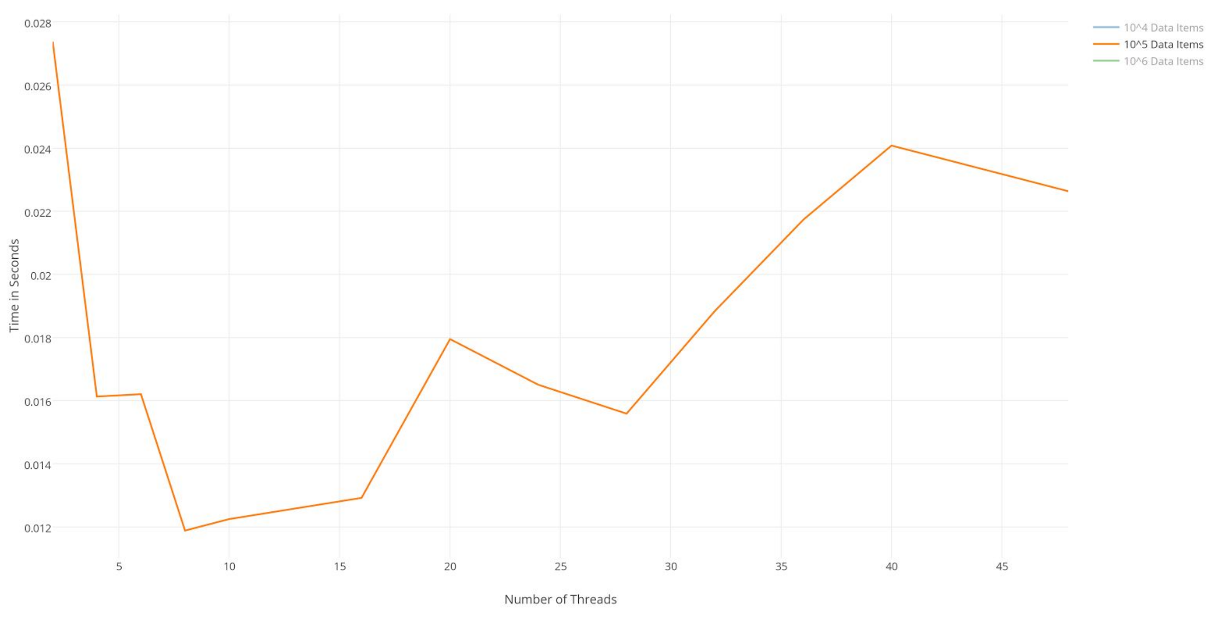
Plots

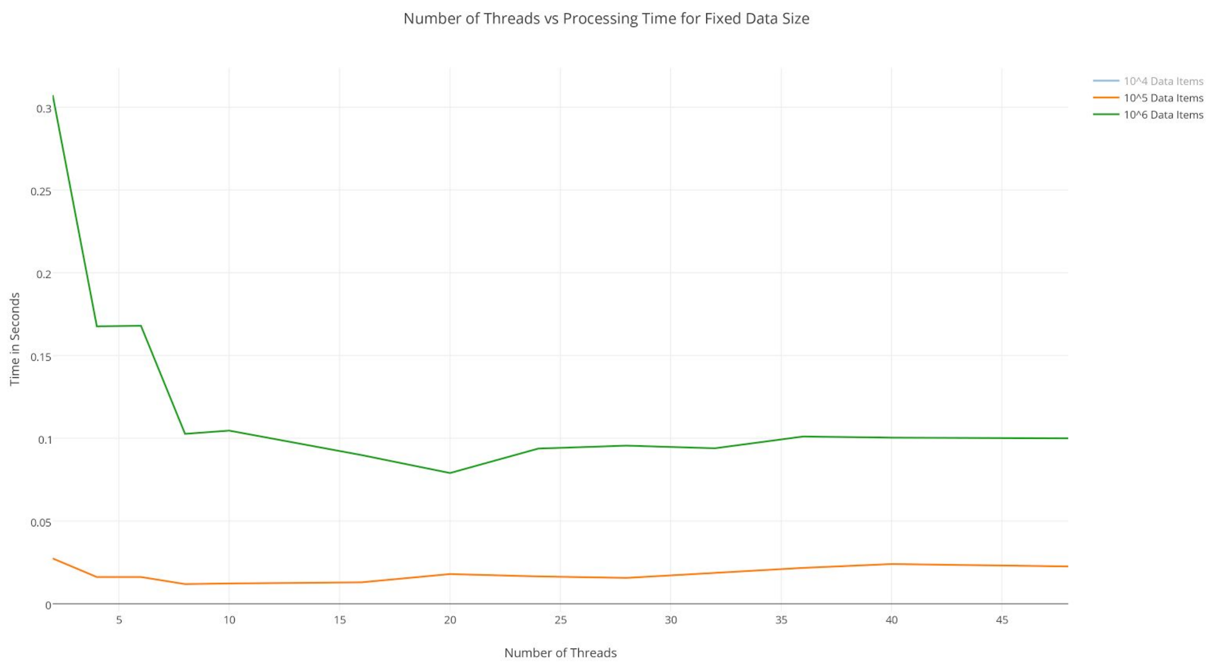


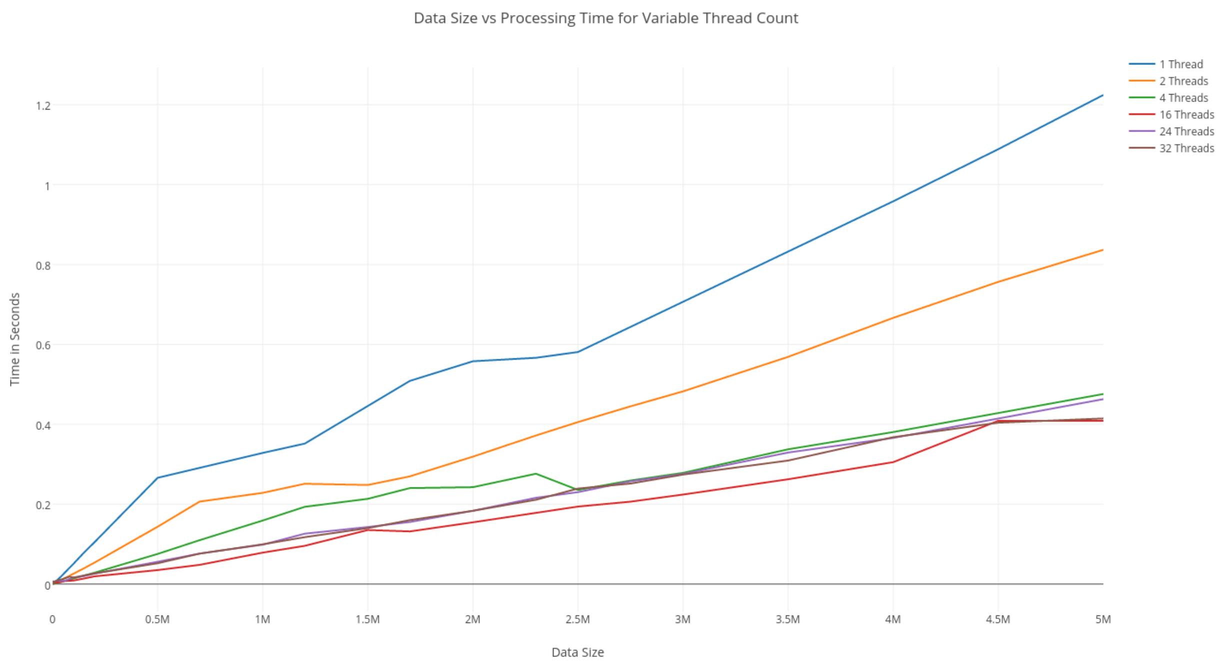
Number of Threads vs Processing Time for Fixed Data Size



Number of Threads vs Processing Time for Fixed Data Size







Judging from the above plots, parallel program is always faster if we have a sufficiently larger data size. But for smaller data sizes for e.g 10000 , we don’t benefit from more threads because the cost of creation of threads dominates the processing time.