CS-301

HPC Project

Semester 5

Mauli Shah 201501132
Ekta Bhoraniya 201501402

Monday 16th October, 2017

Analysis and Comparison of Sorting Algorith.

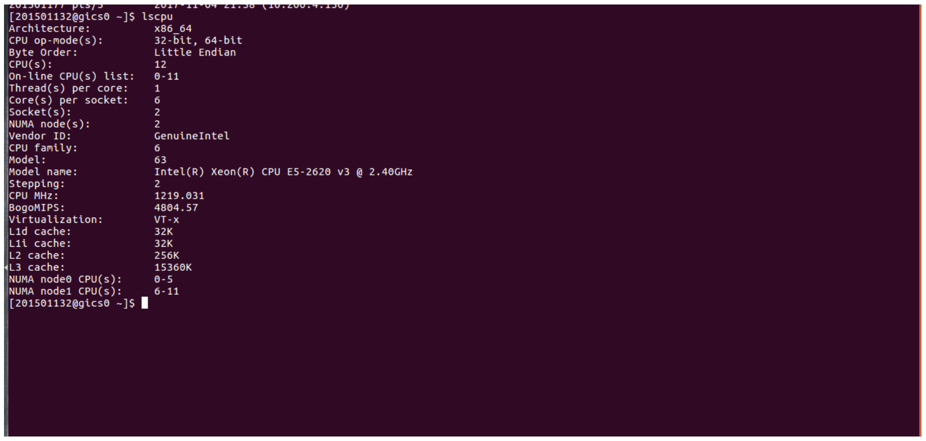
1 Abstract

Here we are trying to compare the performance of quick sort and selection sort for various input sizes ranging from 10^1 to 10^6 .
We are trying to make out the point that why quick sort is preferred over selection sort i.e O(nlog(n)) better than $O(n^2)$.

2 Introduction

Both the sorting algorithms are implemented serially as well as in parallel with the same data set for a fixed size.The data sets are generated using random assignment of the values of that range in the array.
Graph for time v/s the input size and speed-up is plotted for a fixed number of processor is plotted for both the cases for both parallel and serial is plotted.
Graph for time v/s number of processor is plotted for a fixed input size with number of processor varying from 1 to 12.
Individual plots of all are also shown.This can be helpful to get the insight for a particular sorting algorithm and when to used.

3 Hardware Details



```
201501177 pts/3          2017-11-04 21:38 (10.200.4.130)
[201501132@gics0 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                12
On-line CPU(s) list:   0-11
Thread(s) per core:    1
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
Stepping:              2
CPU MHz:               1219.031
BogoMIPS:              4804.57
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              15360K
NUMA node0 CPU(s):     0-5
NUMA node1 CPU(s):     6-11
[201501132@gics0 ~]$
```

4 Implementation Details

4.1 Brief and clear description about the Serial implementation

- Quick Sort
  QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.
  This algorithm divides the list into three main parts : Elements less than the Pivot element Pivot element(Central element) Elements greater than the pivot element.The pivot is localized to all the sub-arrays formed
  It is an in-place sorting algorithm but unstable.

- Selection Sort
  This algorithm first finds the smallest element in the array and swaps it with the element in the first position, then find the second smallest element and swaps it with the element in the second position, and continues in this way until the entire array is sorted.
  It is an in-place sorting algorithm but unstable.

4.2 Brief and clear description about the implementation of the approach (Parallelization Strategy, Mapping of computation to threads)

- Quick Sort
  Here we used the clause pragma omp task and single nowait in order to parallize the code.
  we parallelize the code for large inputs($> 10^3$). For small inputs it is sorting serially.
  Firstly partition method will be called by a single thread and then tasks will be divided among the threads. whenever a thread encounters a task construct,a new task is generated.
  The task construct is placed where array is divided into two parts. So, Task division will be like first for N elements then N/2, N/4,N/8,..,1. Threads sort that individual array and finally merge all subarrays for final output.

- Selection Sort
  Simply the clause pragma omp parallel for is used to parallel the for loop. For the outer for loop the pragma clause is used.
  By using this on outer loop it divides the whole arrays into small sub arrays which is shared by all the threads.Here we store the result in another array to avoid cache coherence.

5 Complexity and Analysis

5.1 Complexity of serial code

- Selection Sort
  It is $\Theta(n^2)$ as there are two nested loops. Though it is $\Theta(n^2)$ but for small input sizes Selection sort is faster than Quick Sort because in quick sort the divide and conquer part will bring into picture the communication overhead

- Quick Sort
  In quick sort the time complexity differs from case to case. For best and average case it is of the order of $\Theta(nlogn)$.
  For worst case it is of the order of $\Theta(n^2)$

5.2 Complexity of parallel code (split as needed into work, step, etc.)

- Selection Sort
  It is of the order of $\Theta(\frac{N^2}{p})$.The arrays are divided into small sub-arrays and each is sorted individually and then the result is stored in an different array in order to avoid synchronization issue. It takes up extra space of O(N) at the same time it reduces O(N) swap operations which are executed if another array is not used.

- Quick Sort
  It is of the order of $\Theta(\frac{NlogN}{p})$. The partition is of the order of $\Theta(N)$ and it is called $\Theta(logN)$ times by p processors making it $\Theta(\frac{NlogN}{p})$.

5.3 Cost of Parallel Algorithm

- Quick Sort
  For smaller input sizes we implement it serially to avoid the time taken for creation of tasks

- Selection Sort
  Takes up extra space of the order of N.

5.4 Theoretical Speedup (using asymptotic analysis, etc.)(i.e. making number of processors ∞)

Input Size	Theoretical Speedup for Quick Sort	Theoretical Speedup for Selection Sort
10^2	1.23	1.69
10^3	1.66	84.11
10^4	2.24	606

5.5 Estimated Serial Fraction

Input Size	Estimated Serial Fraction for Quick Sort	Estimated Serial Fraction for Selection Sort
10^2	0.7960	0.5866
10^3	0.5662	0.011
10^4	0.3961	0.019

5.6 Tight upper bound based on Amdahl's Law

Input Size	Tighter upper bound if p=12 for Quick Sort	Tighter upper bound if p=12 for Selection Sort
10^2	1.2068	1.61
10^3	1.5733	10.71
10^4	2.0303	9.92

5.7 Number of memory accesses

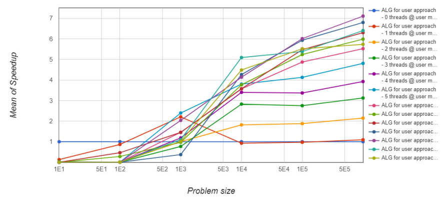
- Selection Sort
  The number of memory access is $\Theta(N^2)$.
- Quick Sort
  The number of memory access is $\Theta(NLogN)$.

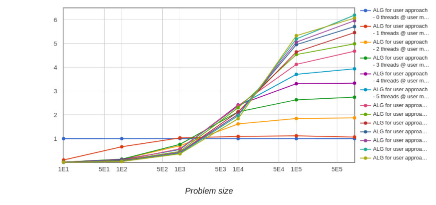
5.8 Number of computations

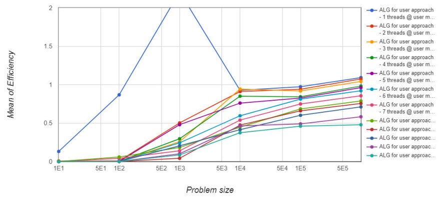
- Selection Sort
  Here the number of computation is $O(N^2)$.In the function ompsort the number of computations are approximately equal to $N*(2*N+1)$ and that in the last fill function is equal to $2*N$ thus making it of the order of $\Theta(N^2)$
- Quick Sort
  In partition computations will be $O(N)$ and it is called $O(logN)$ times. So, total computations will be $O(NLogN)$

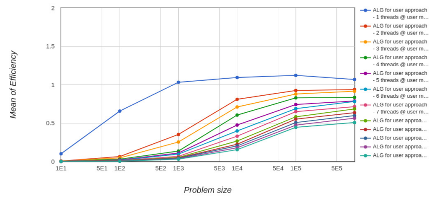
6 Curve Based Analysis

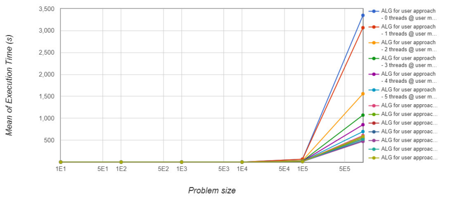
The left side graphs are of Selection sort and that on right are of quick sort.

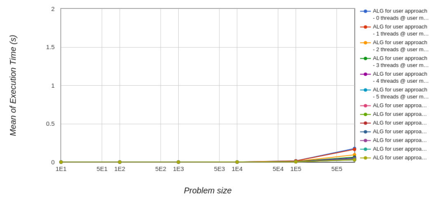












6.1 Time Curve related analysis (as no. of processor increases)

- Selection Sort
  Execution time decreases with increase in number of processor when the input size is small but when the input size increases the communication time increases hence increasing the overall time for number of processors more than 9 but still it is significantly less than the time for serial.

- Quick Sort
  The difference due to the array size is seen for the large input size where more number of tasks are formed.It takes time at the end to merge the tasks.

6.2 Time Curve related analysis (as problem size increases, also for serial)

- Selection Sort
  For any fixed number of processors as the input size increases the time taken is always increasing.
- Quick Sort
  For any fixed number of processors as the input size increases the time taken increases.Also for 10^6 more number of tasks are formed it takes more time but still less than that of serial.

6.3 Speedup Curve related analysis (as problem size and no. of processors increase)

- Selection Sort
  The speed-up increases as the input size increases for any number of processor but the rate of increase decreases as the input size decreases.
- Quick Sort
  For small inputs ($< 10^3$), we observed that speed up decreases with increase in number of processors because task are created but not used while for large inputs we get more speed up as number of processors increases since task creation takes place

6.4 Efficiency Curve related analysis

- Selection Sort
  For a fixed input size the decrease in efficiency is obtained as the number of processor increases.
- Quick Sort
  Here also a decrease in efficiency is obtained as the number of processor increases.

7 Further Detailed Analysis

7.1 Major serial and parallel overheads

- Selection Sort
  For small input sizes the number of computations increases the communications doesn't increase that much because a significant amount of data can be still accommodated in cache.At this point the overhead is majorly because of the creation of threads.But as the input size increases there is an increase in communication time thus making communication overhead come into picture.

- Quick Sort
  For small array size($<= 10^3$) sorting is done serially. As we increase the number of processors, thread creation and communication between processors takes some significant time. So, for large input size, increase in computations and communications causes overhead.

7.2 Memory wall related analysis

- Selection Sort
  Here all the threads shares the whole array though it sorts only a part of an array so there is no issue of accessing the elements

- Quick Sort
  here threads divides array into smaller parts and sort that particular array where that array is shared among all the threads so no issues of memory wall.

7.3 Cache coherence related analysis

- Selection Sort
  The frequency of cache miss is increased when the input size is 10^6. At this point the cache miss increases thus concept of spatial locality is not used very efficiently. The communication time increases .

- Quick Sort
  In quick sort array is divided into two parts, left and right. Threads will sort that small array part. So there will be independent arrays to sort for each thread. So, there will not be any dependency.

7.4 False sharing related analysis

- Selection Sort
  Since the array is divided into sub arrays and each is being sorted in parallel the result needs to be stored in different array so that a sorted element is not replaced by some other element in the list.

- Quick Sort
  Since array is divided into two independent sub parts and sorted parts are also independent. Threads joins sorted right part at the end of sorted left part. so, no false sharing will be there.

7.5 Scheduling related analysis

- Selection Sort
  There is no data dependency present so threads can run in parallel without waiting for one another.

- Quick Sort
  As we used task, it generate new task when thread encounters it.Then it may choose to execute the task immediately or defer its execution until a later time.If task execution is deferred, then the task is placed in a pool of tasks.A thread that executes a task may be different from the thread that originally encountered it.

7.6 Load balance analysis

- Selection Sort
  No load balancing related issue because each thread has to sort equal number of elements $\frac{n}{p}$.
- Quick Sort
  Here all threads might not get the same length of array to sort because load each task gets depends on the pivot element as elements less than pivot will be given to one thread and elements greater than pivot will be given to another thread. so, there won't be load balancing.

7.7 Synchronization related analysis

- Selection Sort
  Since the result is stored in a different array there is no need to update the value in current array so synchronization is not an issue.
- Quick Sort
  As the data of array given to each thread is not dependant, Synchronization won't be issue here. Even if different threads takes different time to sort it won't delay other threads' work.

7.8 Granularity related analysis

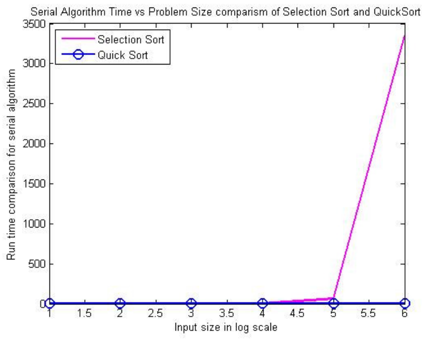
- Selection Sort
- Quick Sort

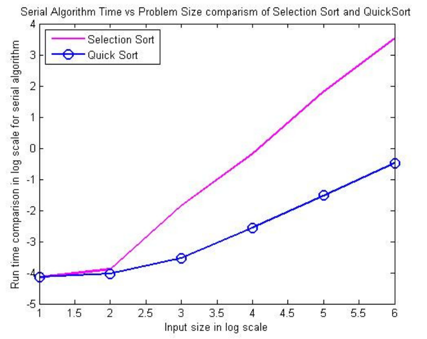
7.9 Scalability related analysis

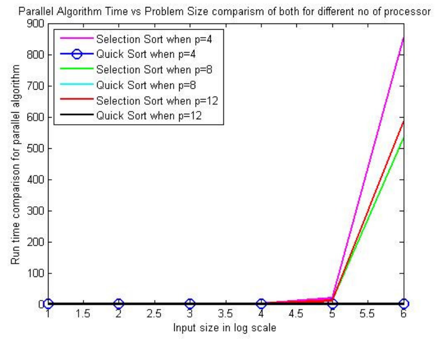
- Selection Sort
  It is weakly scalable. On increasing the input size there is a huge increase in the running time of algorithm though speed-up is obtained but if possible another algorithm should be preferred instead of this.From the graph of execution time clearly the difference is seen between the running time of $10^5 and 10^6$.

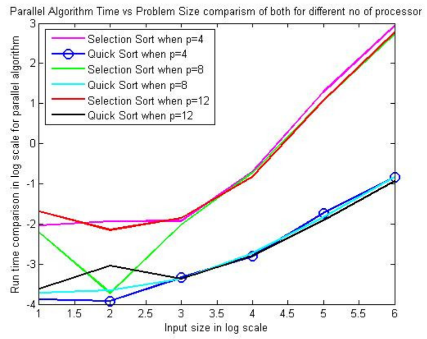
- Quick Sort

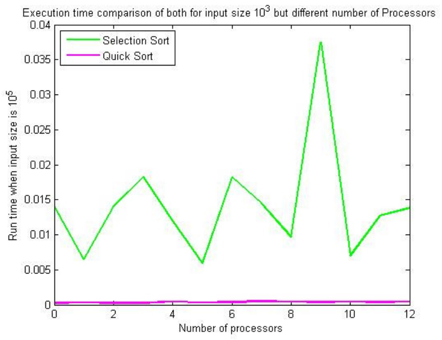
8 Comparison based Analysis

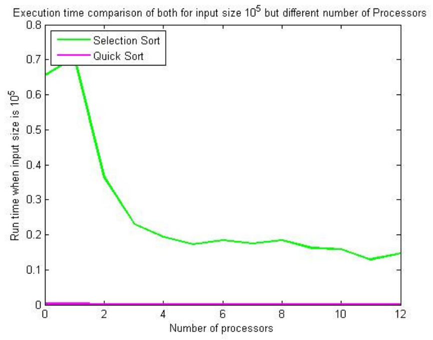


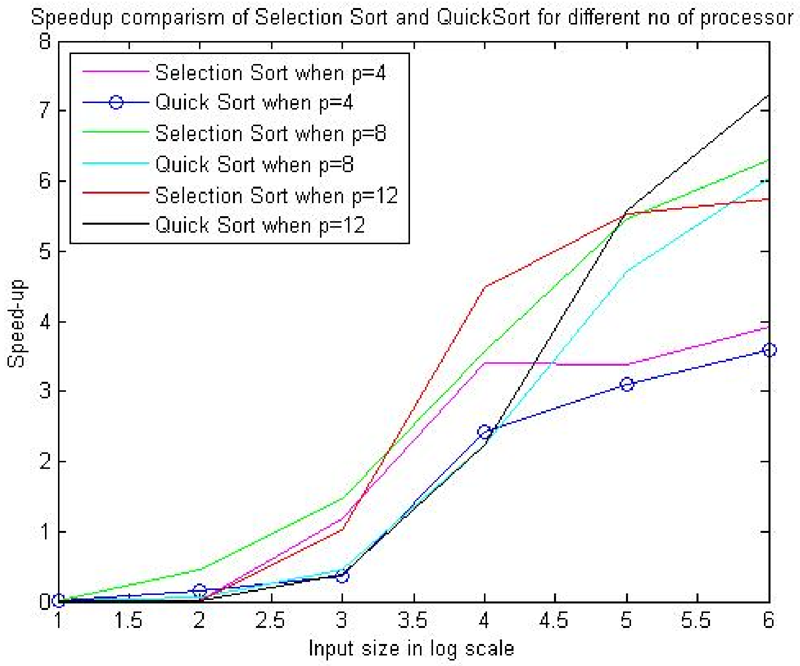












Graphs for various types of comparisons are plotted above.

It can be clearly seen from the graphs that for larger input size quick sort is better algorithm both in case of serial as well as parallel.For smaller input sizes both are efficient not much difference is observed.Those are also shown on log scale to bring out the smallest difference which is not highlighted if only the run time graphs are taken into account.
In selection sort when the number of processor increases beyond 8 for higher input sizes memory bottleneck is observed so the execution time increases a little but still it is significantly less than that on less number of processors. This is not the issue with quick sort.
For higher input sizes there is a constant decrease in time as the number of processor increases for Selection sort which is not the case with quick sort.This clearly shows that it is weakly scalable.

9 Additional Analysis

9.1 Advantages/Disadvantages of your approach

As in both the algorithms worst case complexity is $O(N^2)$, for large inputs we should think of better algorithms like heap sort with complexity of $O(NlogN)$. For small input size quick sort can be one choice but for large there are better options.

9.2 Difficulties faced while implementing this approach

In selection sort, With complexity of $O(N^2)$ it took some hours to run the code for input size of 10^6. In quick sort, first we were getting serial algorithm taking less time than parallel due to communication between threads.

10 Refrences and Appendecies

`https://www.uio.no/studier/emner/matnat/ifi/INF3380/v10/undervisningsmateriale/inf3380-week12.pdf`
`https://computing.llnl.gov/tutorials/parallel_comp/`
`http://compalg.inf.elte.hu/~tony/Oktatas/AlgofInf-Vol2-HTML/Vol2-HTML/Volume2.html#d5e11032` `https://codepad.co/snippet/T7uq4wWc`