Algorithm Performance Analysis

Aayush Jain, Will Foster, Annie Dinh*

College of Computer and Information Science
Northeastern University

August, 8 2018

Abstract

Sorting algorithms hold a critical role in the development of cutting edge algorithms in Computer Science. However, a question remains as to whether the theoretical "best" algorithms perform to high levels when running on machine. With other hardware components factoring in, do the algorithms maintain their level of efficiency? We will test quick sort, merge sort and insertion sort on different hardware parameters and try to verify that quick sort is indeed the best in terms of efficiency.

I. Introduction

An algorithm is a sequence of computational steps that transform a set of data input into output. We can also view an algorithm as a tool for solving a well-specified computational problem.
[Introduction to Algorithms, Third Edition]
When an algorithm is executed, it uses the resources of the computer to perform operations that hold the program instructions & data required for execution.

Sorting by definition is enforcing an order on a list of data objects, in an increasing or decreasing order. Sorting algorithms are widely used and have the profound impacts on problem solving. In one way or another they incorporate into the building blocks of complex algorithms. We can broadly divide the sorting algorithms into two categories: comparison sort, which does not require any prior knowledge of the data, and counting sort, which needs prior knowledge of the data to be sorted. Connected networks, pathways, financial trading, event scheduling, and information management are some of the real life problems where sorting is applied. Looking at the scale of these problems and performance demands, the user needs to select a sorting algorithm that works the fastest and uses the least resource. We need to analyze different algorithms and select the one that performance the best not only in terms of how fast it is but also in terms of resource utilization.

As we know that an algorithm is a well defined sequence of steps, we cannot draw value from it unless we implement it on a computer and measure the parameters of time and resource utilization. The performance analysis of an algorithms refers to determining the qualities of an algorithm, based on a variety of system parameters. This project will attempt to quantify the comparison of performance efficiency of three sorting algorithms, insertion sort, merge sort, and quick sort. We will measure performance parameters such time and space complexity, as well as other hardware parameters such as cache hit/miss, page faults, and branch misprediction. When taken together, this will give us an insight as to how they perform on a computer.

We have learned that theoretically quick sort has a time complexity of $nlogn$ and the best in terms of practical applications. In this project we will attempt to verify this assertion. We hypothesize that quick sort will have the best overall performance, all parameters considered.

*Team : we_need_more_cache

II. Methods

Sorting algorithms

The algorithms this project analyze are merge sort, quick sort and insertion sort. All of these algorithms are comparison sort, and subjected to same data set.

- Insertion sort: Has a time complexity of $O(n^2)$ and uses in-place sorting.
- Merge sort: Has a time complexity of $O(nlogn)$ and needs separate storage to perform the sorting.
- Quick sort: Also has a time complexity if $O(nlogn)$ and uses in-place sorting.
  Both merge sort and quick sort are examples of divide-and-conquer strategies, which involve the division of the input array into smaller subsets and recursively sort through them. Although both merge sort and quick sort have same time complexity, it would be interesting to know how they match up objectively.

Input size

The datasets given to the algorithms are of unsorted integers of size 0-1000. We do the performance analysis of the sorting algorithms on integer arrays of size 500 to 40000, in increments of 500. We use a randmam_gen.c to generate the random data.

System specification

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 4
On-line CPU(s) list: 0-3
Thread(s) per core: 2
Core(s) per socket: 2
Socket(s): 1
NUMA node(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 37
Model name: Intel(R) Core(TM) i5 CPU M 460 @ 2.53GHz
Stepping: 5
CPU MHz: 1408.503
CPU max MHz: 2534.0000
CPU min MHz: 1199.0000
BogoMIPS: 5045.47
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 3072K

Measuring parameters

- Time and Space Complexity : Time complexity of a program refers to the total amount of time the program requires to run to completion. Space complexity is the amount of memory space needed for execution. We denote these in terms of BigO

- Cache Misses : The cache is a high speed memory located very close to the CPU. When the CPU cannot find data or instruction on cache, we have a cache miss and the information is fetched from main memory. It works on the principle of locality.
- Page Faults : Page faults occur when the operating system cannot find the data in virtual memory and it has to look for it in the main memory.
- Branch Misprediction : Branch misprediction is a significant impediment to performance, and it occurs when the control jumps to new instruction and the pipeline is flushed. This incurs a substantial penalty to performance.
- Context Switches : When a CPU switches over to a different process than the one it is currently executing, this is achieved by a context switch. Context switching hurts performance due to the time in which the processor is being used but no user code is executing.
- CPU Clock : The time when the CPU is executing code.
- Hardware Instructions : The number of hardware instructions.
- L1 Data Cache Loads : The cache loads on

the cache which is closest to the CPU on which the data is loaded.
- L1 Instruction Cache Loads : The cache loads on the cache which is closest to the CPU on which the instructions are loaded.
- CPU Migration : On a multi-core processor, a process can switch between CPUs depending upon the workload. This also hurts performance.

Although we have listed all these parameters, we will study the ones that effect performance the most. These are time complexity, cache misses, page faults, context switches and branch misses.

III. Challenges

Implementation

Algorithm implementation is open ended. There are many different ways we can implement one particular algorithm. It can be implemented in a parallel environment, in different languages, it can be implemented in-place or out-of-place. The challenge we faced was to implement the algorithms in such a way that different sorting algorithms would work with our tools and existing code. We achieve this by programming the sorts in C following the same function definition. This allows us to implement different algorithms in separate files, yet make them compatible with our performance measurement tools. The function definition we used is:

```
void sort(int* array, int start, int end)
```

Profiling vs Instrumentation

Performance measurement of a program can be done in two ways:

- Non-Invasive: Where we do not change the program in any way and try to measure its system performance. Linux has an inbuilt tool to do this called *perf*. This tool directly talks to the kernel and gives us system counters for a program. This method of getting the information is light weight and does not add any overhead. The limitation of this tool is that we can only measure parameters for a complete executable and not specific parts of the program. This measurement would also include data for auxiliary functions, FILE I/O etc. But we are only interested in parameters counted for sorting.
- Invasive: Where we change the program in order to measure specific parts of the program. For this we use a library called [Libperf]. It allows us to instrument specific parts of the code/program. This increases the accuracy of our readings and ensures high output data confidence.

Accuracy

We overcame the instrumentation challenge by using the libperf library. More so, the accuracy also depends upon its implementation. It only makes sense to initialize all data structures separately and isolate the sorting function from all the other aspects of implementation. This way, only the sorting function would affect the counters. To achieve isolation and modularity of using different algorithms, we use a layer of a profiling function in a separate file to trigger the sorting and counters at the same time. We use the

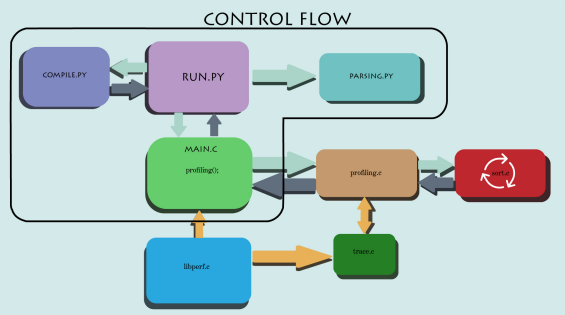
```
-finstrument-fucntions
```

flag to set up a profiling function to trigger the counters and call the sort method in it.

```
__cyg_profile_func_enter
__cyg_profile_func_exit
```

are the two functions provided in gcc we uses to trigger the counter. To increase the accuracy we even use the timespec data structure and recorded the CPU count in nanoseconds. Also, for time measurements, for each data set we ran the sorting algorithm five times and took the average of them.

IV. Code Structure



1. run.py
   This is the entry point of our code. For each sorting algorithm, it calls the compile.py which compiles all C files. Then it runs the main program on the different inputs, and finally move all output files to their respective directory.

2. compile.py
   This compiles all the C files using specific flags. We generate object files from main.C using -libperf flag. Main.C initializes and finalizes the data structure for the hardware counters. Then we generate object files from profiling.c using -finstrument-functions flag. We want the functions in profiling.c to trigger the trace functions when they are called and exit. We link all object files together to generate a binary main_out

3. main.c
   This is the core of our program. Here, we initialize the data structures, read the files and populate the arrays to be sorted, call the profile function which triggers the hardware counters and sorting since we compiled by using the flag mentioned above. When the sorting is finished and counters are turned off we generate the output from the counter data structure and write them to the file.

4. profiling.c
   This is the extra layer we are using for isolation. This has just one function which calls sorting. As we compiled this file using the above mentioned flags it triggers the counters to start and the sort to start simultaneously.

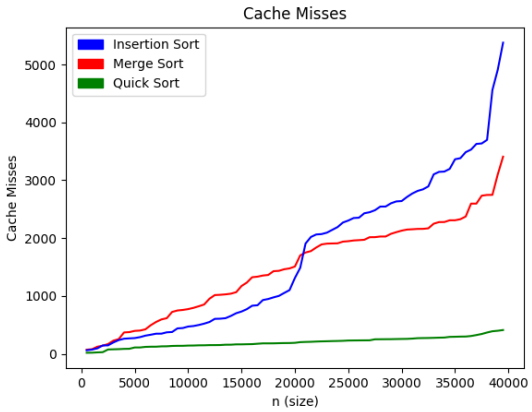
5. trace.c
   We put in the enable and disable the counters and timers here.

6. libperf.c
   This file has the functions that talk to the kernel and get the hardware counters.

7. sort.c
   This us where our implementation for the sort goes, following a predefined function definition.

V. Results & Discussion

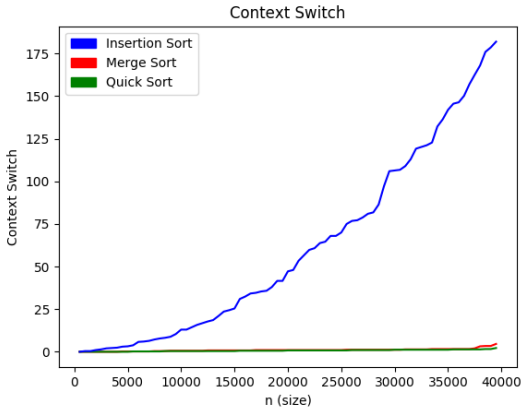
i. Cache Misses



Insertion sort starts with lower cache misses than merge sort, but as the size of the array increases, the misses goes up. One reason for this could be that insertion sort is not a divide and conquer algorithm. The complete array is attempts to be stored on the cache, but fails due to the large array size and therefore it experiences more cache misses for large inputs. But for small array size, insertion sort performs better as it loads the array and makes sequential passes over each value so we get better cache performance. Merge sort is a divide and conquer algorithm that performs better in terms of cache performance when

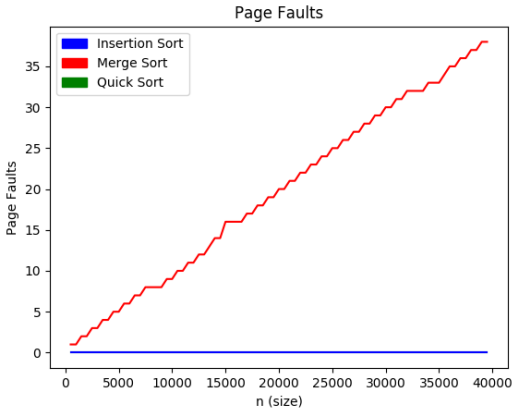
it comes to large array sizes. It breaks down the larger arrays into sub-arrays which can fit the cache lines. However, since it involves jumps between the two arrays that it merges, it initially gives a lot more cache misses than insertion sort. Quick sort works the best because it make sequential passes and also has a divide and conquer approach. Quick sort performs the best as it make sequential passes which means it has an excellent spacial locality. Since it is a divide and conquer, the sub-array is small enough to fit the cache line, this gives an excellent temporal locality.

ii. Context Switch



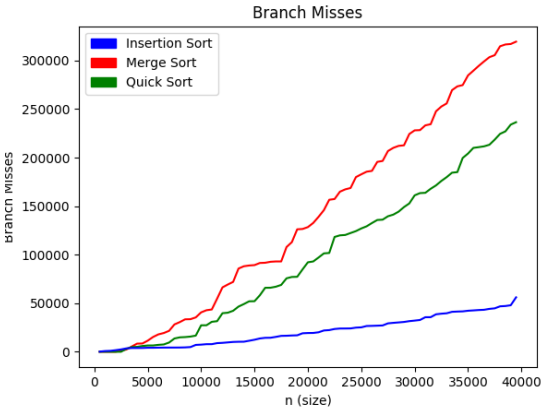
Insertion sort does not use a divide and conquer method. Only a part of the array can fit the cache line, so when it is goes through the sequential comparisons of each element with the every other, it reads through the data from the cache lines, and since the array size is large, it has to reload part of the array. This leads to a a context switch as the kernel becomes active to fetch that memory. Merge sort and quick sort perform very well and have less context switches than insertion sort. One probable reason for this is that they use divide and conquer so the sub-array can fit in the cache lines.

iii. Page Faults



Merge sort gives a high page fault count. This is because merge sort needs extra arrays (out-of-place) to hold the integers. We account for this need by doing a malloc call to get the array. Looking deeper into the cause of these page faults, we come across the fact (in the output log files) that these are minor page faults which occur due to malloc calls. Further investigation leads to here. Since the quick sort and insertion sort are in-place and no extra memory is used. We do not get page faults in them.

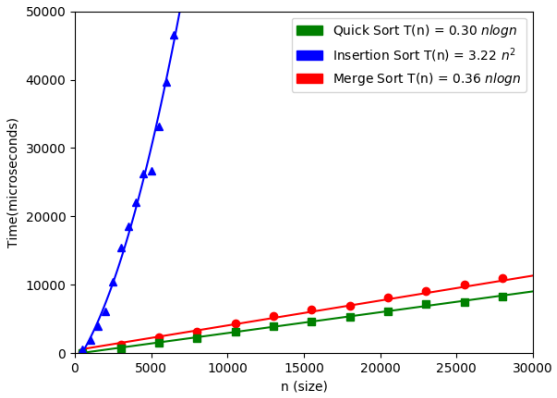
iv. Branch Misses



Branch misses are a result of conditional statements, both quick sort and merge sort have complex conditional statements that makes predicting instructions harder. While insertion

sort has just one conditional statement which eases prediction next instruction. So, we get the least branch misses.

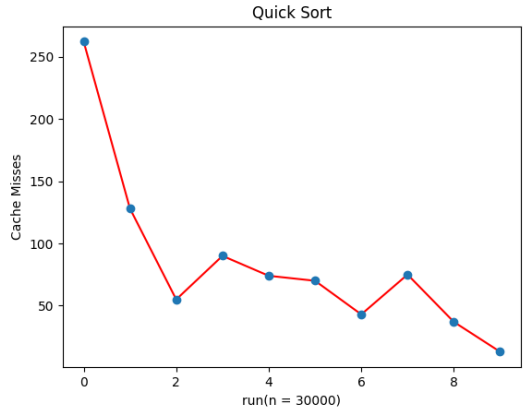
v. Time Analysis

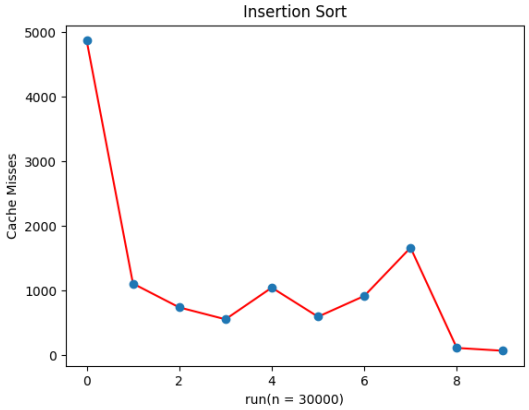


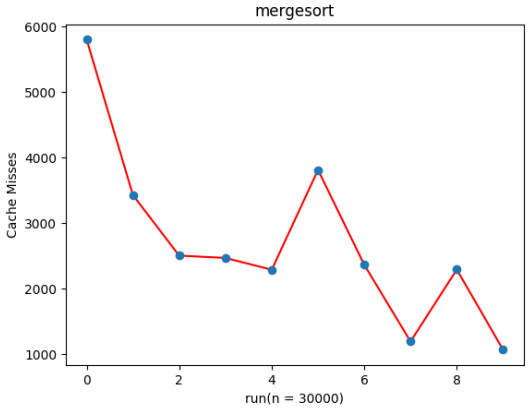
Considering all the factors that we analyzed above, quick sort performed the best in all except one case. All the factors mentioned above add to the overhead to the execution and make the process slow. We plotted the time taken by each sort against the size of the input data, and calculate the constant C in the BigO analysis. We clearly see insertion sort takes the longest time for the same array size. Although merge sort and quick sort have same time complexity, quick sort takes less time than the merge sort. This is because we have lower overheads when they are implemented on the computer.

vi. Individual cache misses

Apart from doing the comparative analysis, we performed individual analysis. We ran each algorithm 10 times on a data of size 30000 and recorded the cache misses. As we expected, we get a high cache misses when we run the sorting the first time but as the cache warms up we get fewer cache misses. These graphs also show anomalous peaks in between, which we suspect might be because our program is not the only one running on the computer. There are a lot of essential operating systems processes running in the background, such as the kernel. We have no control over the scheduling of the processes and other processes can run while our program is waiting to return from context switch. Other programs can write over the cache line our program was using. A complete isolation of a process is very difficult to an impossible task.

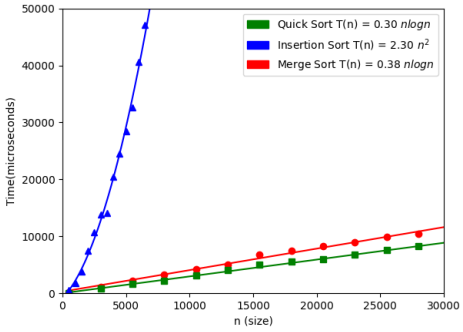


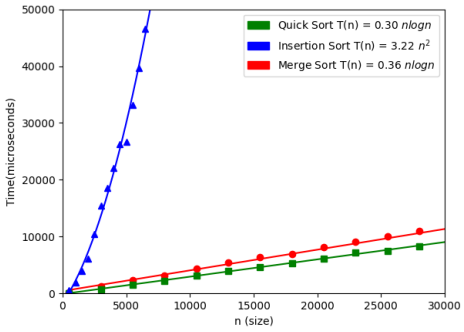




VI. Limitations

As we discuss above, the complete isolation of a process is not possible. The validity of the accuracy our results will always be in question. We can also never be fully sure about the start and stop of the counters at the very exact moment the of our sort. The tool we are using to read the data also adds to an overhead on the execution as it is talking to the kernel to get the data. In the following two figures we can see the difference when we use the libperf tool for hardware counters and when we do not. Note the reduction of C values.





VII. Way forward

We can think in two directions after the study we performed:
Come up with a better tool to record the hardware parameters which will have minimal overhead while keeping the accuracy high.

Implement the algorithms in such a way that they use the system parameters to their advantage.

References

[Introduction to Algorithms, Third Edition]
Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein

[Libperf] https://github.com/theonewolf/libperf

Lamarca, A and Ladner, RE. The Influence of Caches on Performance of Sorting. *Journal of Algorithms* , 31(1):66-104, 1999

S. Eyerman , J.E. Smith, E. Lieven. Characterizing the branch misprediction penalty. *IEEE International Symposium on Performance Analysis of System and Software; 48-58, 2006.*

U. Ferraro-Petrillo, I. Finocchi, G.F. Italiano. (Italian) The price of Resiliency: a Case study on Sorting with Memory Faults. *Algorithmica; 53: 597-620, 2009.*

C.C. McGeoch. Experimental Analysis of Algorithms. *Notices of the AMS; 48(3):304-311, 2001.*

V.P. Kulalvaimozhi, M. Muthulakshmi, R. Mariselvi, G. Santhana Devi, C. Rajalakshmi & C. Durai. *Performance Analysis of Sorting Algorithm. International Journal of Computer Science and Mobile Computing; 4(1):291-306, 2015.*