COMP35112 Chip Multiprocessors
- Lab 2 Report -

Name: Gabor Hosszu, Student ID: 8070815

Expectations & Observations

As with the previous lab, I feel that the main goal of this experimentation was to try and capture the point (if any) where increasing the number of threads used for the merge sort algorithm starts to have benefits.

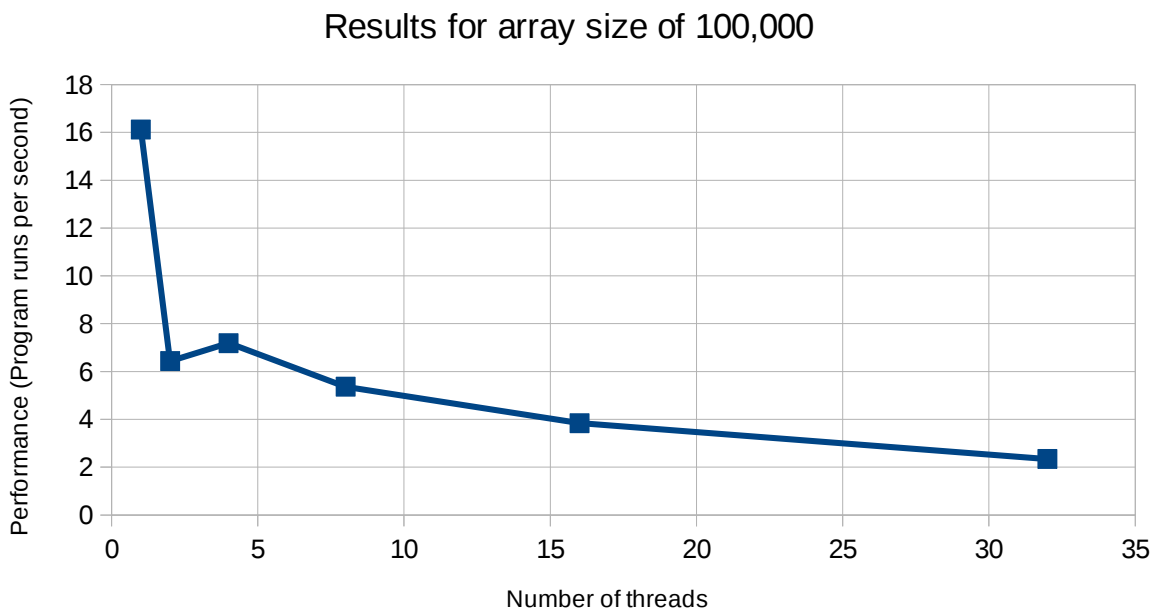
To start things off, I tried running the multi-threaded mergesort application with a small supplied array size (10 - 1000). As expected, there was no performance gain going from sorting 100 numbers iteratively, to splitting them into 2 halves and using 2 threads (combining the results in the end, of course). In fact, when using 2 threads to sort the 100 numbers, the execution time of the program is actually greater. So the performance is worse. This is due to the fact that the overhead from creating multiple threads takes longer than the actual sorting of the numbers.

Going forward, I started speculating about the real question, when am I going to see some performance increase? The results were indeed unexpected. As with the previous exercise, I thought that once greater numbers are used (such as an array size of 10 million), I am going to see the benefits of multi-threading. This was not necessarily the case, certainly not as cut and dried as it looks. Of course, I am basing these observations on the assumption that my code is syntactically and logically correct. So, the “sweet” point where I found that the number of threads starts to have an impact on execution was around an array size of 1 million. I have, therefore, selected to plot the results around that number to illustrate this, specifically 100000, 1 million, 10 million and 100 million.

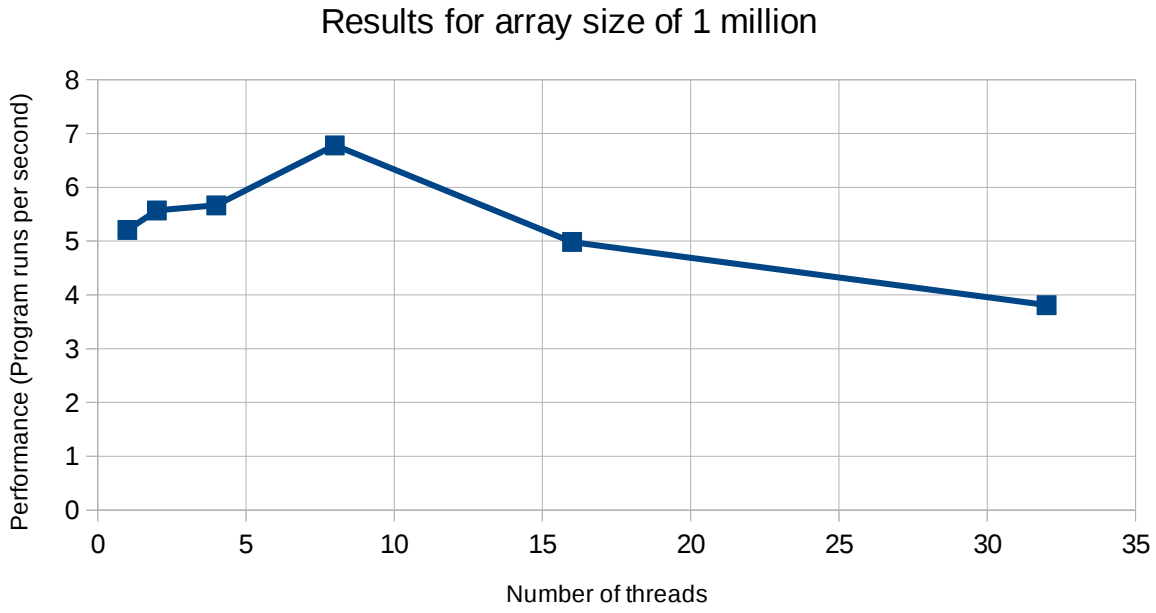
Furthermore, for each array size, the application ran with 1, 2, 4, 8, 16, 32 (powers of 2) number of threads. I used powers of 2 as input for the thread number because of the way merge sorting works and how each time the array is “halved”.

Following this are the results plotted on graphs. The recorded values are, of course, all averaged across multiple runs to decrease possible 'noise' from the system.

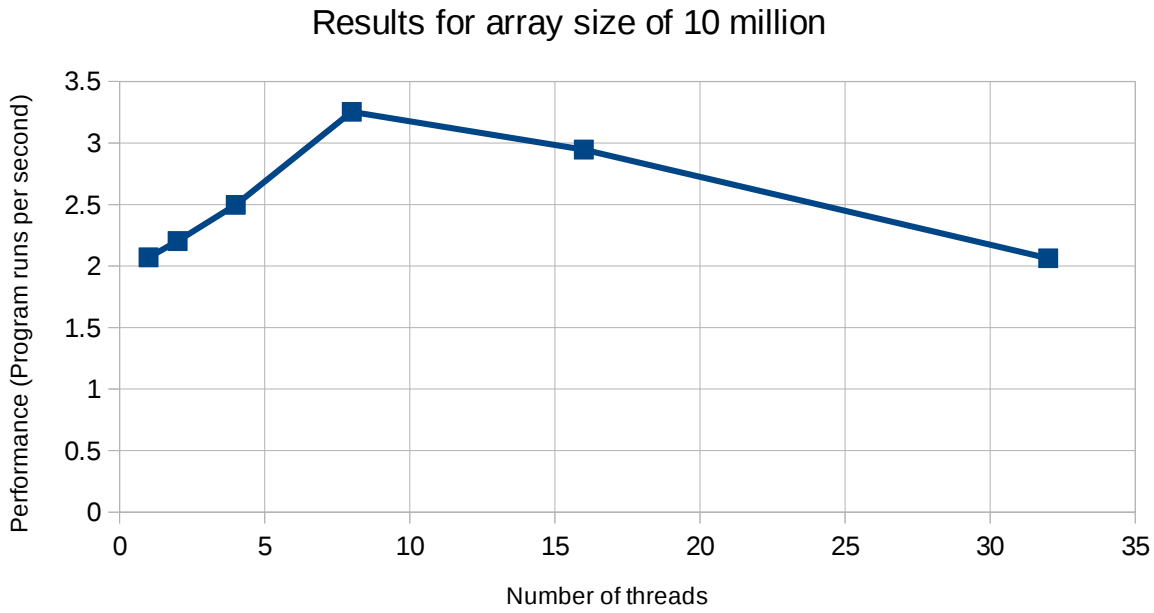
Performance



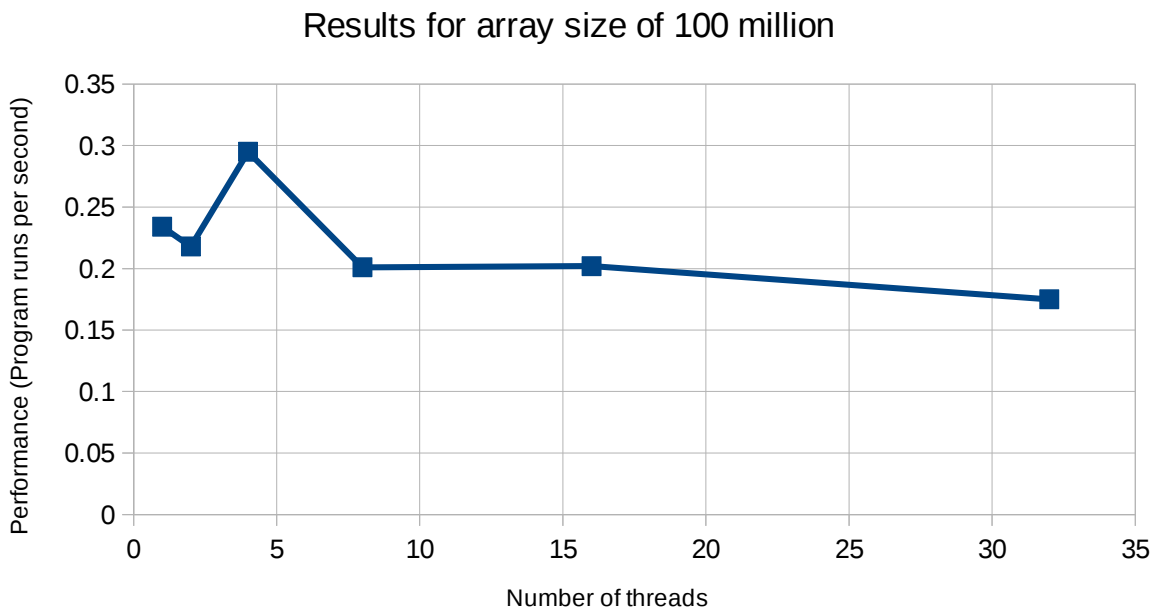
As you can see from the above graph, for an array size of 100 thousand, the best performance is still achieved by the process running on merely 1 thread. After that, the performance drops drastically and stays relatively linear from 2 to 32 threads.



When we reach an array size of 1 million, the first noticeable increase in performance can be seen. The rate grows with the number of threads relatively linearly until somewhere around a thread count of 8, after which, the performance starts to slowly decrease again as more threads are used.



Same as previously, for 10 million, the ideal number of threads to be used is around 8.



This is the part which is quite surprising. When the application is run with 100 million elements, the ideal number of threads to be used, based on a lot of testing, is now 4. It dropped from 8 to 4, when going from 10 million to 100. After that, the performance further decreased as the thread number increased until 8 (which was the previous optimum), then it stays more or less linear.

To conclude, the results I got are partially what I expected. What's clear from the above is that, generally, using more than 8 threads is redundant. However, maybe for even greater array sizes, such as 1 billion or 10 – 100 billion, the optimum number of

threads increases even more, up to 16 or maybe 32 at some point. Those test are unfortunately infeasible as they take too long to run.