Parallel algorithms and parallel computers (in4026)

Assignment B

Sverre Rabbelier
srabbelier@gmail.com
1308211

Introduction

Asked is to provide an algorithm that, given two lists with unique, sorted numbers, produces a list that is the merge of these lists, itself unique and sorted as well.
My implementation does this by first calculating the rank of log(n) elements. This step will hereafter be described as the rank step. Subsequently, it uses this ranking to figure out what the

position of each element should be in the final array, resorting to using a sequential merge to get the grunt of the work done. This second step will hereafter be described as the merge step. My algorithm requires that both input arrays are a power of two and that all numbers are globally unique (i.e., when the inputs are concatenated, the resulting list contains only unique numbers).

Time-complexity analysis

The algorithm consists of two steps, the rank and the merge step.

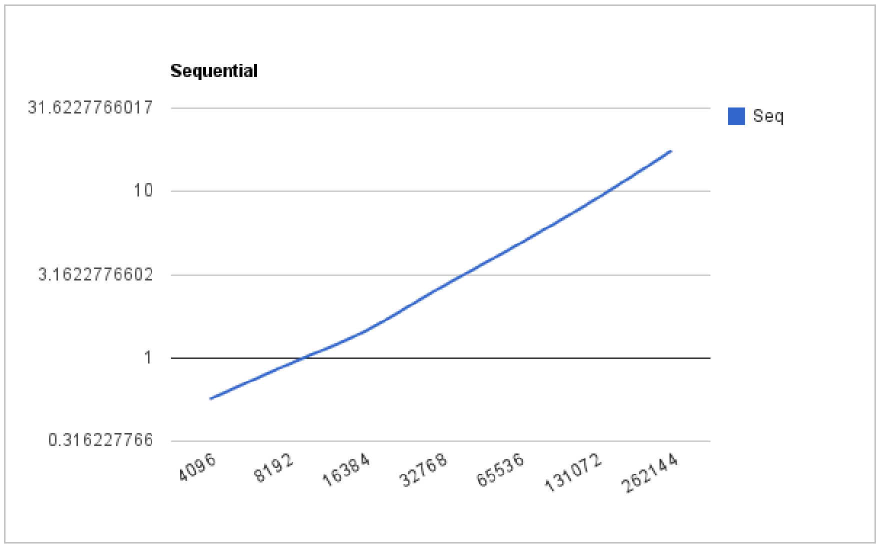
The prefix step consists of one O(1, n/log(n)) loop that contains a O(log(n), log(n)) inner loop, resulting in a O(log(n), n) work-time complexity

The prefix step also consists of one loop O(1, n/log(n)) loop which contains a O(log(n), log(n)) inner loop, resulting again in a O(log(n), n) work-time complexity.

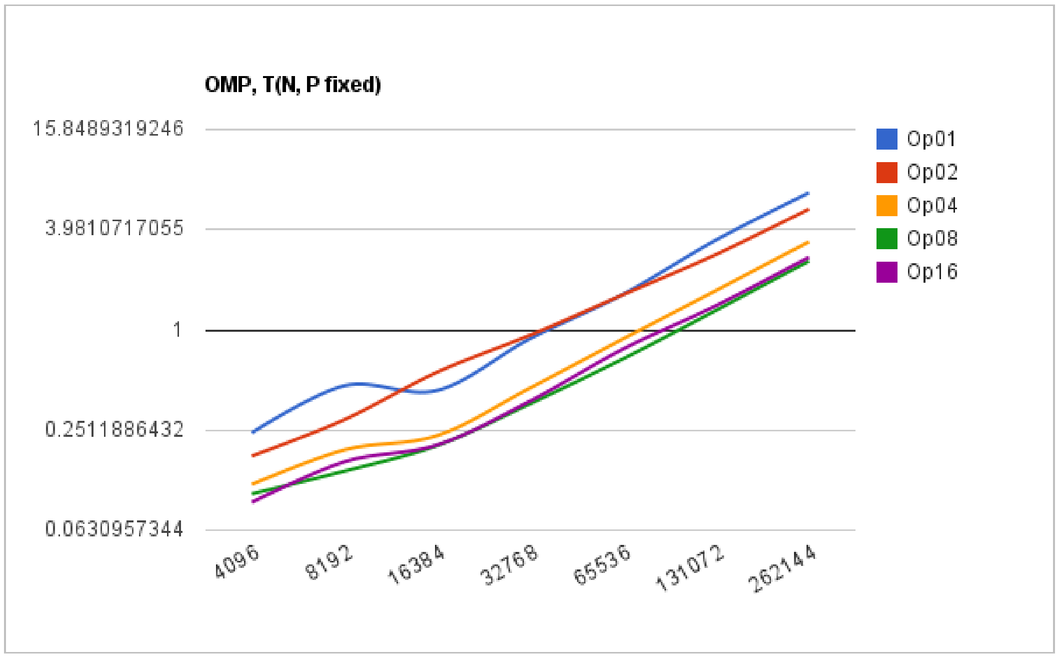
The algorithm thus has a O(log(n), n) work-time complexity.

Testing

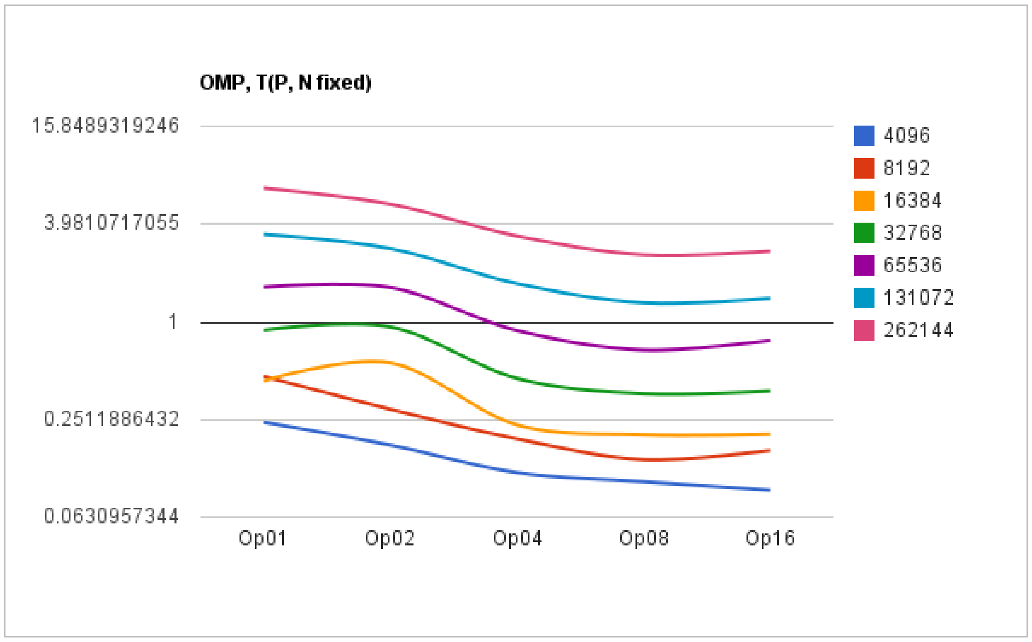
First I tested plotted the linear set to get a baseline. I chose a log scale because that works nicely with the exponentially increasing input size.



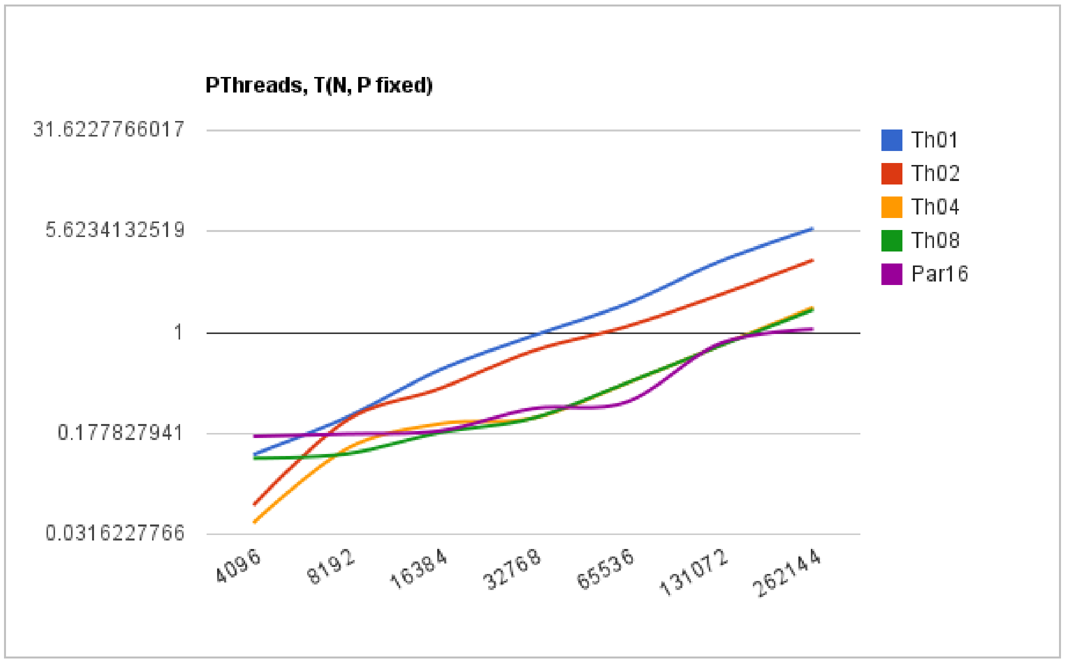
Now I'd expect the OMP to show something similar but of course with lower values for a higher thread count. There seems to be a slight hump for 1 thread for low n, but considering how fast the algorithm runs for low n, it might just be a slight increase in server load.



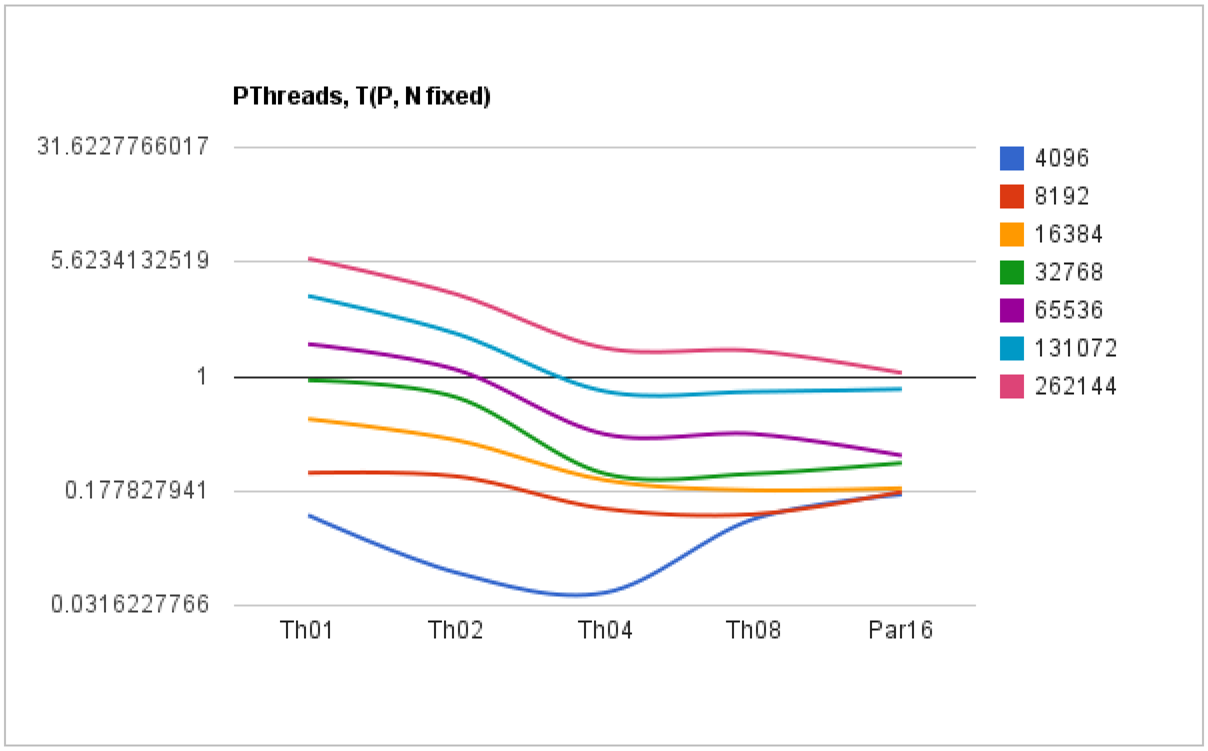
If we instead plot the threads against N we find a similar pattern, we find again a slight hump for two threads for low N, which is probably not significant.



For pthreads the results are a little more inline this time around. There is not really a clear winner between four and eight threads, and low N still results in weird discrepancies, but at least there's a clear upward trend decernable this time.



Keeping n fixed but varying the number of threads clearly shows the erraticness of the low N.



Conclusions

Translating this assignment to OpenMP and pthreads turned out to be relatively trivial. Simply adding the appropriate pragma’s and barrier waits took but a few tries to get right. I suspect that if I had increased TIMES further and started out with a higher N that the results would have been more reliable, but as it already takes quite a while to run the algorithm that wasn’t really feasible.