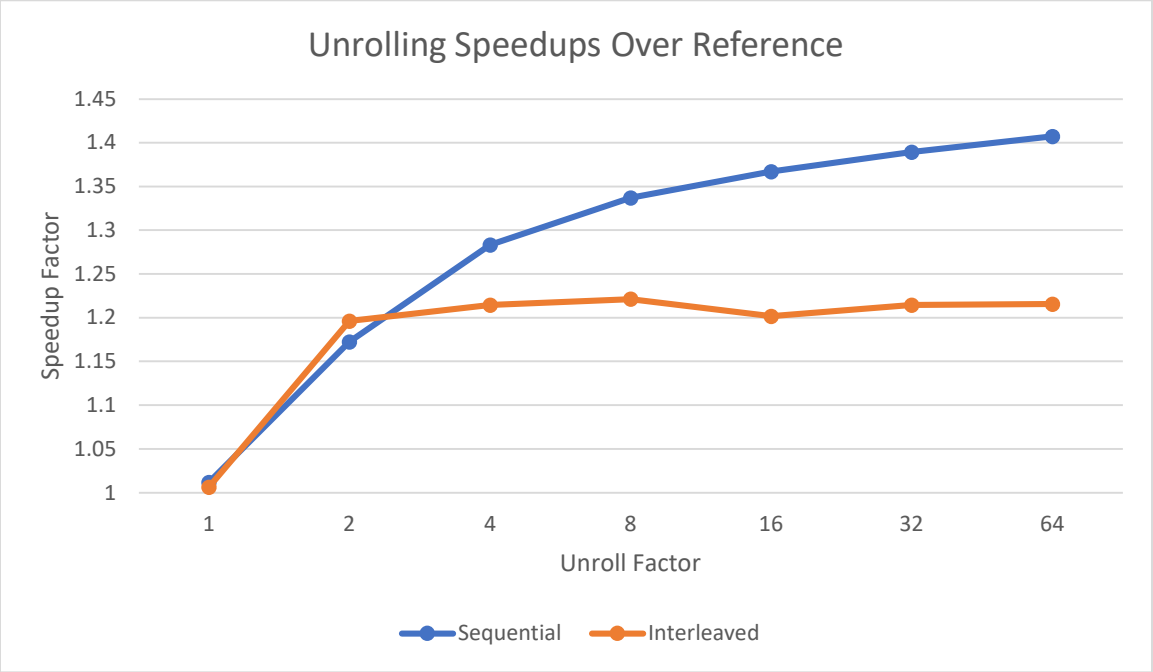
Homework1 Write-Up

Galen Turoci

CSE 113

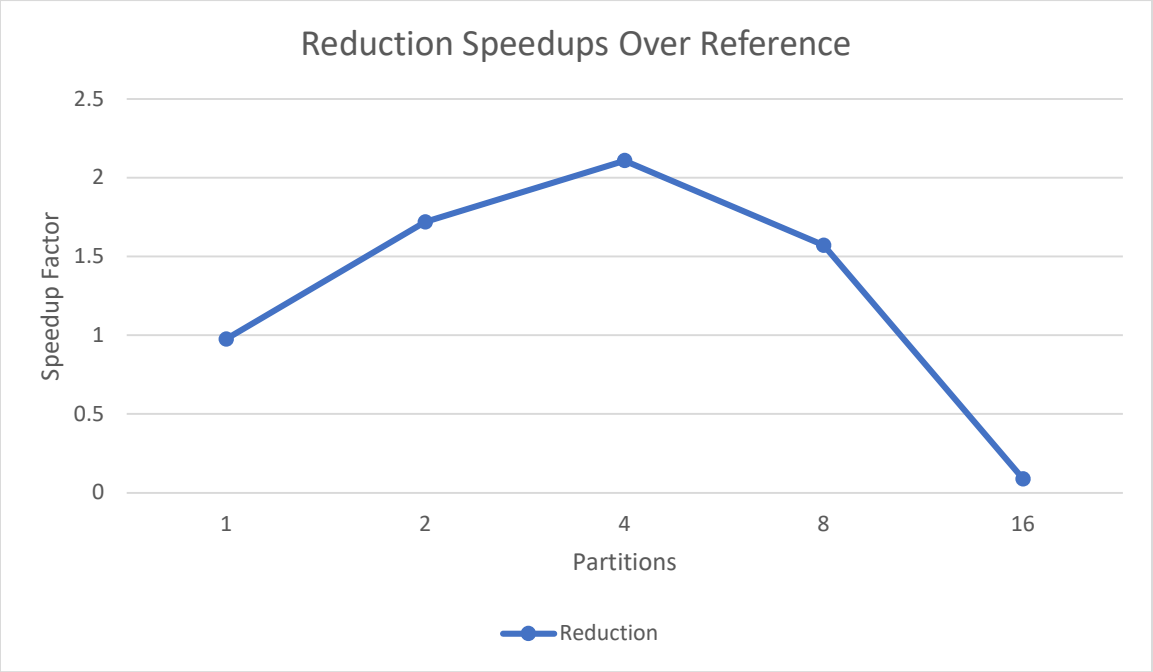
Part 1



Presented here are the results of me running both the sequential and interleaved algorithms for ILP loop unrolling against a rolled-up reference loop. For reference, I am running this on an 8-core 3700MHz processor and 16GB of RAM. The size of the to-be-unrolled loops was 1024*1024*8, and the dependency chain length for each loop iteration was 64.

Though both algorithms did present significant speedups over the reference loop, especially with unroll factors of $\geq$2, it surprised me that the speed of the interleaved algorithm largely plateaued, whereas the sequential algorithm continued to get relatively faster in what appeared to be a logarithmic fashion (it is worth noting that with sufficiently small chain lengths, the interleaved algorithm was actually significantly slower than even the reference loop). This is surprising given that it is my belief that the compiler has significantly less work to do in the interleaved algorithm, as I was able to eliminate the need for a temporary variable entirely, instead performing operations directly on each array element. Could the cost of accessing and operating on an array element be greater than that of initializing a variable from an array element, operating on that variable, and then replacing the old element with the variable? Either this is the case or I have somehow implemented the algorithm wrong, which testing has led me to believe that I have not, though I cannot rule out the possibility.

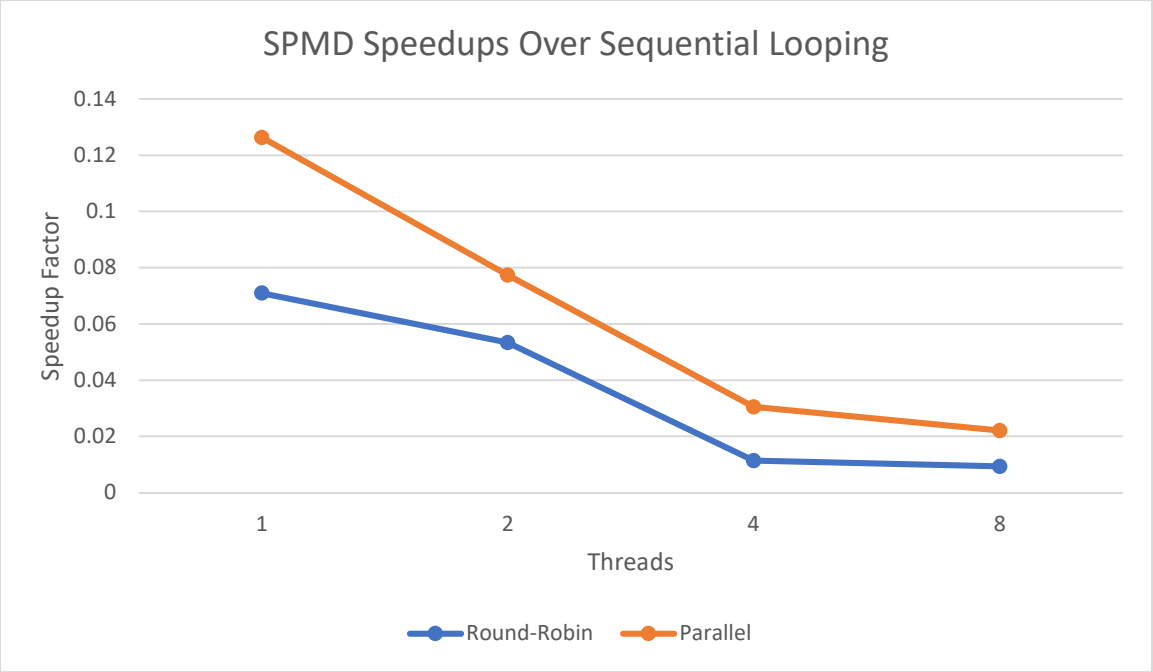
Part 2



Presented here are the results of me running the partitioning algorithm for ILP loop partitioning against an un-partitioned reference loop. For reference, I am running this on an 8-core 3700MHz processor and 16GB of RAM. The size of the to-be-unrolled loops was 1024*1024*32.

The results of my tests are… surprising to me, mostly due to the slowdowns incurred once partitions (P) became $\geq$8. I would think that my algorithm would only cause the program to go faster, but that is not the case for reasons I can only guess at. Furthermore, the speed of my algorithm when P = 1 seemed to vary quite a bit (sometimes being up to ~20% slower than the reference), which surprises me greatly as the operations performed by both loops are practically identical in such a case.

Part 3



Presented here are the results of me running two different SPMD threading algorithms against a standard sequential loop. For reference, I am running this on an 8-core 3700MHz processor and 16GB of RAM. The value of the int being stored in the loops (K) was 1048576. I do not know the name of the algorithm used for array c, but it is akin to a reduction-style ILP method as used in Part 2, with the size of the loop doing the storing being reduced according to the number of threads being used.

As is plain to see, both SPMD algorithms were significantly slower than the sequential algorithm (though my reduction-style algorithm was consistently faster than the round-robin one), for reasons I can only guess at. Had I not followed the example and explanations given in the April 8th lecture closely, I would assume I had somehow implemented the threading techniques incorrectly, but since I did I cannot see how that is possible. Equally confusing is the fact that both algorithms become slower with added threads, rather than faster. Initially I thought that the reason the SPMD algorithms appeared slower might be due to the semantics of my timing implementations, but the fact that the algorithms consistently become slower dissuaded me from that notion. I also double-checked how the speedup factor was being calculated, to ensure that I was not simply mixing up my results right at the last step, but this was also fruitless. I'm afraid I simply have no idea why my results are what they are.