Task 4 Analysis

Moritz Jasper Techen, Paul Kang, Bailiang Li, Max Spanning

January 9, 2025

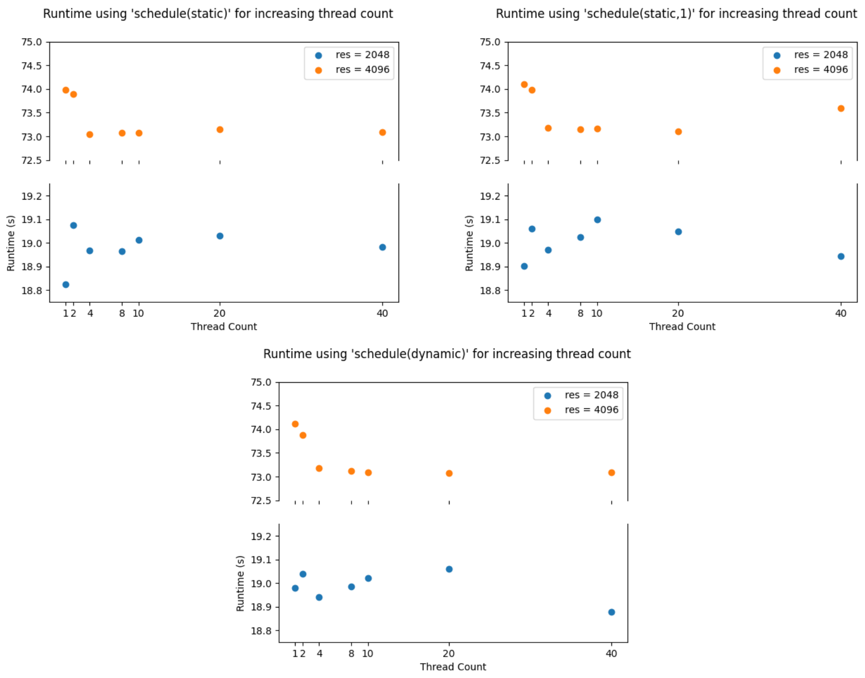

Figure 1: Plot of Runtimes for Different OpenMP Scheduling Policies with Increasing Thread Count

1 Impact of Scaling Thread Count

When considering thread optimization, one must consider the balance between the number of chunks that needs to be done and the complexity of the tasks involved in the chunks. Higher thread count is more relevant if both characteristics are large. However, as seen by all policies, at a certain point there will be diminishing returns, and potentially detrimental if the thread count is too high compared to the work involved. This is best generally seen by the calculations run with a resolution of 4096, where as thread count increases, the enhanced runtimes plateau and even run slower at 40 threads. This is due to the increased need for synchronization across threads as thread count increases. If the task is not complex enough, then the overhead becomes more apparent relative to the cost of the task itself. Examining the lower resolution shows that the runtimes are erratic with increasing thread count. This demonstrates that for smaller chunk sizes, parallelization doesn't significantly enhance performance and runtime will come down to the whims of the compiler.

2 Impact of Type of OpenMP Scheduling Policies Used

Static scheduling refers to distributing chunks as evenly as possible across all threads. Because the assignment of workload is already determined, this is decision is made at compile time. In OpenMP, the default chunk size is 1, but custom chunk size can also be set. This is shown by the plots comparing the performances of 'schedule(static)' and 'schedule(static,1)'.

Dynamic scheduling refers assigning chunks during runtime. Rather than assigning threads a set amount of chunks of work, this policy assigns new chunks as soon as a thread finishes its current one. This is beneficial for code that contains for loops with varying levels of complexity, allowing easier tasks to be completed and starting harder tasks earlier to improve runtime. However, this attempt comes at the cost of overhead as the compiler must be continuously deciding where chunks go and ensuring synchronization. A custom chunk size can also be set for this policy but defaults to 1.

Comparing the plots of static and dynamic scheduling shows the importance of understanding the code being run when considering policies. The chunk size for the jacobi process code used in this assignment depends on the size of the interior points (as the boundary points just require 2*N iterations, where N = resolution). As resolution increases, the number of threads and scheduling policy becomes more impactful. As shown by the runtime at 40 threads, the dynamic policy wins out because of this uneven distribution of complexity within the code. Had the chunks been more even in size within the code, it is likely that the static scheduling would have been better to use, as it does not have to deal with the increased overhead from dynamic scheduling.

*I noticed my runtimes was off by a factor of 4000 for some reason and was not able to rectify this issue before assignment submission, hence there is a factor of 4000 used when creating the plots for this report.