Andrew Malatak
amm8689

Homework 1: OpenMP

Introduction

Part 1 of this homework was done using C++ was compiled with the g++ intel compiler for debugging and for final code compilation for execution. The code for part 1 can be found under /home1/05893/amalatak/hw1/HW1. Part 2 of this homework was done using C and was compiled with icc. The code for this part can be found under /home1/05893/amalatak/hw1/rb; both folders are referenced in a README for understanding the code and how to execute it if desired. Since all plotting was done in Python, it is important to note that some axes are zero-indexed.

Part 1

The code in this part of the homework was executed on a the Stampede 2 skylake node using the compilation flags -O2 –xcore-avx512 –fopenmp. The proper numactl commands were issued in correspondence with the desired usage per requested under each assignment. Kernels b and c from homework 0 were each parallelized using omp directives and the formatted output was put into text files. The array size used for this part of the homework was 98,306 by 98,306.

1.1

Ensuring that the code was properly debugged, the number of elements below the threshold value for the serial and single thread execution were checked after the data was smoothed and they were roughly equal, within roughly one thousand; close considering their value was over one hundred thousand.

The serial, single thread and eight thread execution time for counting the number of elements below the threshold of .1 are seen below in the table for the smoothed and unsmoothed data.

	Count Time X (s)	Count Time Y (s)
Serial	6.33865	11.9364
One thread	21.8911	16.4429
Eight threads	8.21016	4.50051

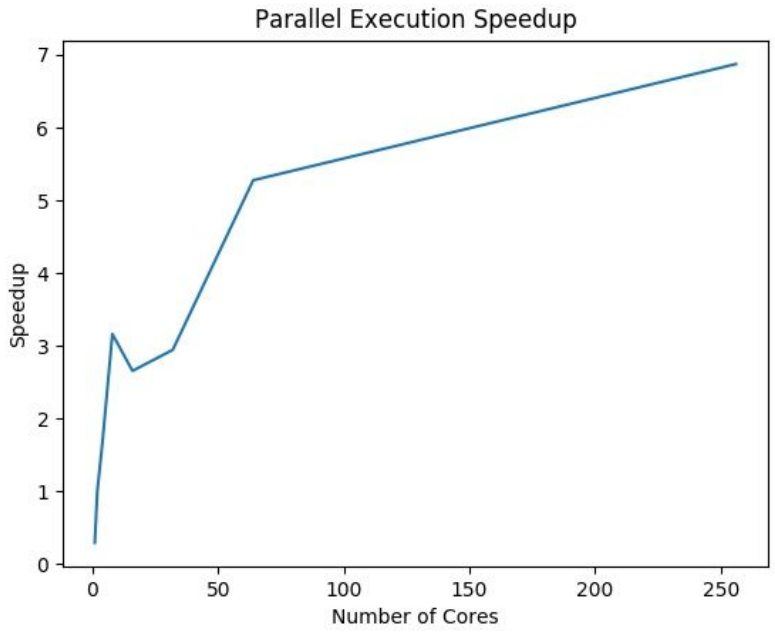
As can be seen, the results are mixed. The single thread execution took the longest for each category and both threaded code executions finished the X count in a shorter time than they did for the Y count, while the serial execution took longer to count the Y array. Overall, the eight thread execution was the fastest, but still was slower than the serial execution for the X count time.

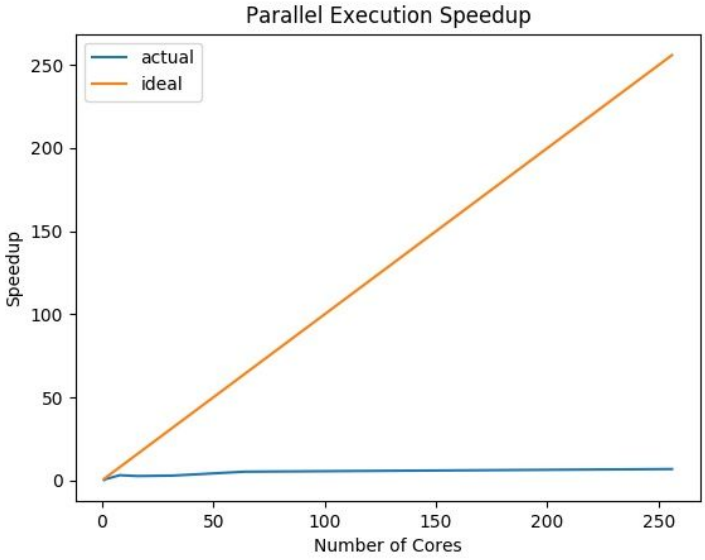
The single thread execution may be significantly slower since it is moving all of the data through a socket which may cause the program to slow down. This may also be the reason that the 8 thread program runs comparably fast to the serial execution, but since the program is running on 8 threads, it is still substantially faster than the 1 thread execution.

1.2

The array size was kept the same for this section of the analysis, but the number of cores was varied. Focusing on the smoothing kernel, the entire program was run for 1, 2, 4, 8, 16, 32, 64, 128 and 256 threads in a while loop while reusing the same initialization array to save time. All smoothing execution times were saved along with the rest of the regular output into a text file and the data was plotted using Python 3.6.8. Stampede did not run Python 3, but the code is uploaded in the general file for evaluation.

Below are the plots for the actual speedup and the actual speedup vs the ideas speedup.





As can be seen from the two plots, the actual speedup compared to the ideal speedup is very much slower than a theoretical scenario. Ideal speedup was taken to simply be equal to the number of cores used. Based on the first plot, subjectively, it seems like 64 threads is the ideal number to execute the code; for 64 cores, the output is substantially faster than 32, but not as much slower than 128 or 256 enough to warrant the use of that many cores.

The output from the run is below; to minimize time, all cases were run during the same ./a.out call. The times from left to right can be read from 1 to 256 cores in the order seen under “number of threads”. For example, the 8 core output time for smoothing was 3.84 seconds.

```
Summary
-------
Number of elements in a row/col       :: 98306
Number of inner elements in a row/col :: 98304
Total number of elements              :: 1074135044
Total number of inner elements        :: 1073741824
Memory (MB) in each Array             :: 77312.6
Threshold                             :: 0.1
Smoothing constants (a, b, c)         :: 0.05 0.1 0.4
Number of elements below threshold (X):: 28095412
Fraction of elements below threshold  :: 0.0261659
Number of elements below threshold (Y):: 105066
Fraction of elements below threshold  :: 9.78503e-05
Number of Threads                     :: 1 2 4 8 16 32 64 128 256



Action               ::   time (s)
------
CPU Alloc-X          :: 0.292851
```

```
CPU Alloc-Y          :: 0.308589
CPU Init-X           :: 172.255
CPU Smooth           :: 41.5 12.1613 7.32787 3.8458 4.57781 4.12956 2.30468 2.09364 1.76905
CPU Count-X          :: 15.4892 13.7395 10.8326 9.93979 5.62802 6.85583 5.6491 3.22705 3.19837
CPU Count-Y          :: 16.4568 4.65421 5.32625 2.23473 3.91123 0.519824 1.18618 1.16314
1.1695
```

Based on the collected data and plots, the scaling is good up to a certain point. For a low number of cores, say until 16, the speedup seems to be quite substantial for the number of added computers, but as the cores increases, the scaling gets worse. The plot where ideal speedup is shown exacerbates this by showing that the ideal speedup dwarfs the actual speedup.

1.3

Again, discussing the smoothing kernel, static and dynamic scheduling performance was analyzed for several different chunk sizes. Eight threads were used for this analysis and the entire analysis was run during one call to ./a.out to save time from reusing the same randomized matrix for all eight cases. The results are below.

Chunksize	Dynamic Scheduling Runtime	Static Scheduling Runtime
100	4.43943	11.2528
1000	4.43943	5.10699
10000	4.11104	5.56348
100000	28.0389	28.4834

As can be seen, Dynamic scheduling outperforms Static scheduling at every chunksize, and does substantially better for the lowest chunksize analyzed. Dynamic scheduling runtimes appear relatively steady from chunk sizes ranging from 100 to 10000, but changes by half an order of magnitude when the chunksize increases to 100000. The optimal chunksize for dynamic scheduling appears to be around 10000 based on the results of this test, whereas the optimal chunksize for static scheduling appears to be around 1000 based on this test. The high performance window for static scheduling is smaller than that of dynamic scheduling given that its runtime more than doubles outside of the chunksize range from 1000 to 10000. Overall, selecting a chunksize anywhere from 1000 to 10000 will yield comparable results from either scheduling type, yet dynamic scheduling will yield slightly better results.

1.4

Using eight threads, a table of runtimes for the smoothing kernel was created and is shown below for various cases of different numactl settings.

	Smoothing Runtime	Memory Allocated	Thread Locations
no numactl	17.809	0	0
numactl --cpunodebind=0 --preferred=0	18.1108	0	0
numactl --cpunodebind=0 --interleave=0,1	17.2001	0, 1	0, 1
numactl --cpunodebind=0 --preferred=1	19.8612	1	0

As can be seen from the table, different policies for managing the shared memory on different nodes can slightly affect the runtime of a program, and in two of the three cases of entering numatcl commands, the program slowed down. The execution speed differs because of the different amount of time it takes for sockets to access memory in different nodes, the closer the threads are to the memory allocation, the faster the communication is.

The --hardware option for numactl shows the inventory of the available nodes on a system, or in this case, the skylake nodes on stampede 2. It shows the number of nodes, the memory, available memory and cpus per node and information about the node relationships.

There are 2 nodes (0,1) on the skylake system and there are 96965 MB on node 0 and 98304 MB on node 1.

1.5

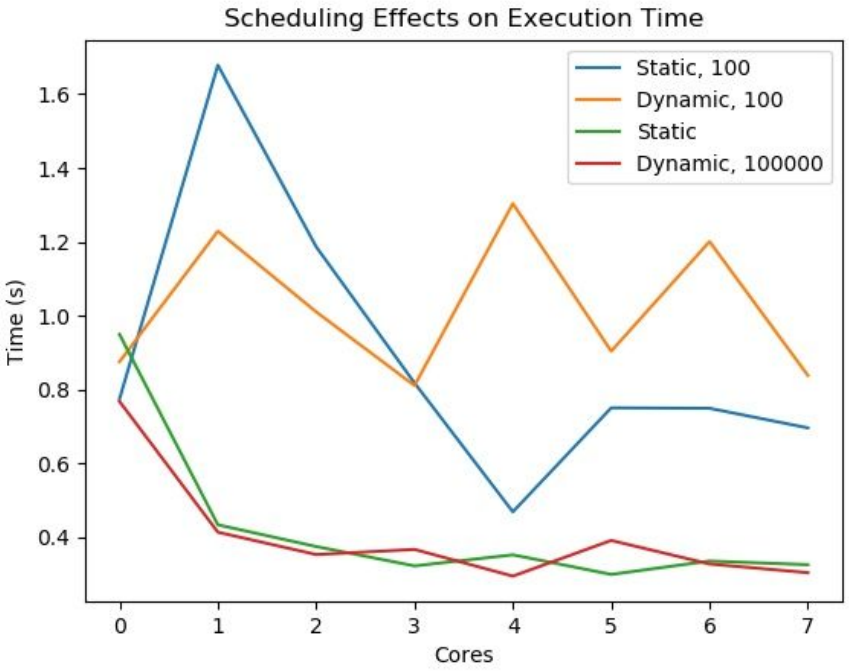
The compilation flag –xcore-avx512 is used to instruct the compiler to optimize the processor scalability and generate vector length instructions. It is important to use the highest instruction set available for this application since we are studying scalability for large matrices. The compilation flag serves to optimize the code to make it run slightly faster which would make a very big difference if using even larger arrays than we are now.

Part 2

The code in this part of the homework was executed on a the Stampede 2 idev node and was compiled and run using the provided compile and dothis files.

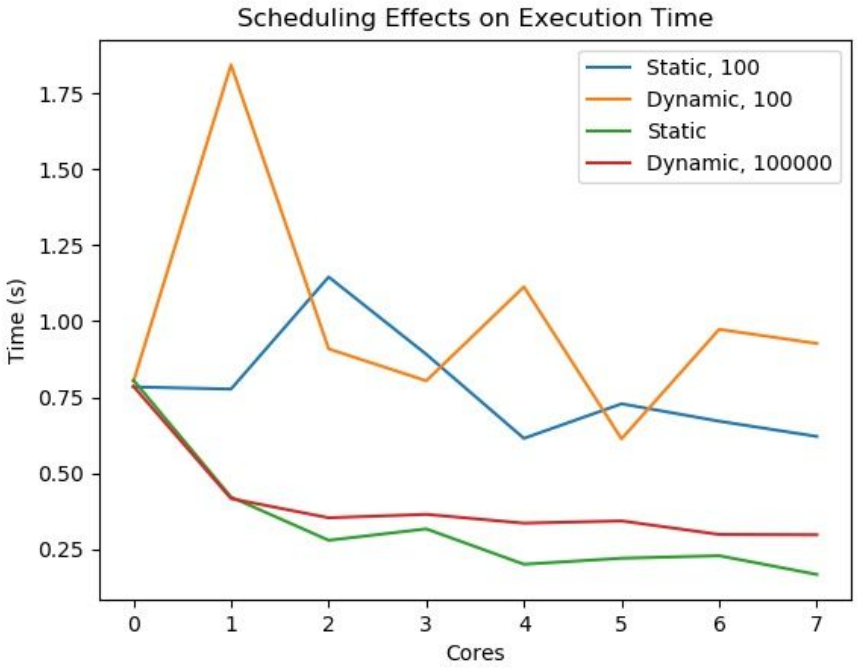
2.1

At first, the loops were parallelized using simple omp directives with a schedule(runtime) clause added to allow for the schedule to be altered. The results are shown below.

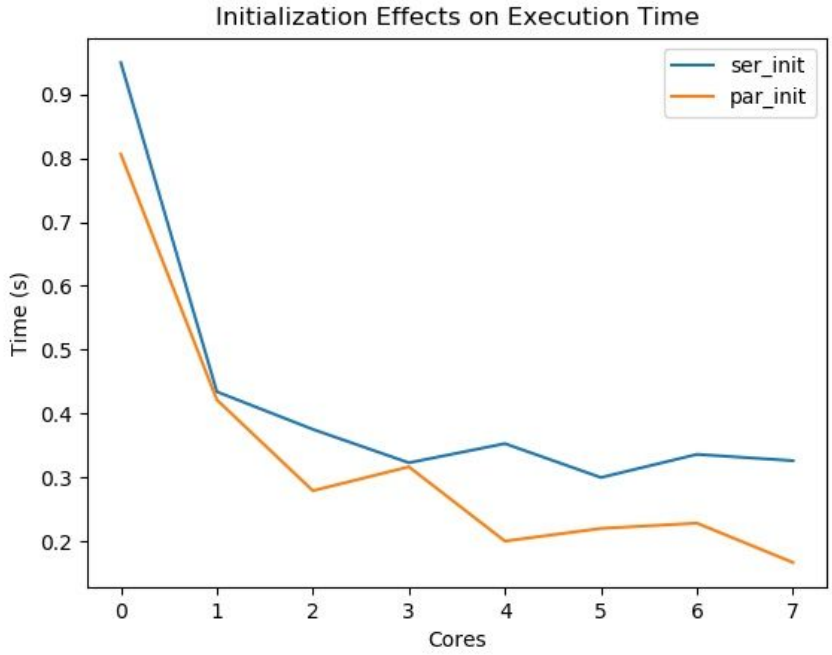


As can be seen, the default static scheduling and the large chunksize dynamic scheduling have comparable performance. The small chunk size dynamic scheduling has virtually no effect on performance as the number of cores increases. So, the type of scheduling definitely affects performance in this case.

Next, the initialization loop was parallelized using similar directives, except without the scheduling clause. The results are shown below



With the first loop parallelized, there is still a difference between the scheduling with different chunk sizes, but it appears that the small chunksize scheduling directives are more closely aligned. The results from default static directive and the large chunksize dynamic directive diverged slightly. Just by glancing at the plots, the scaling appears roughly the same, but if we take a look at the plot below, we see some clearer results.

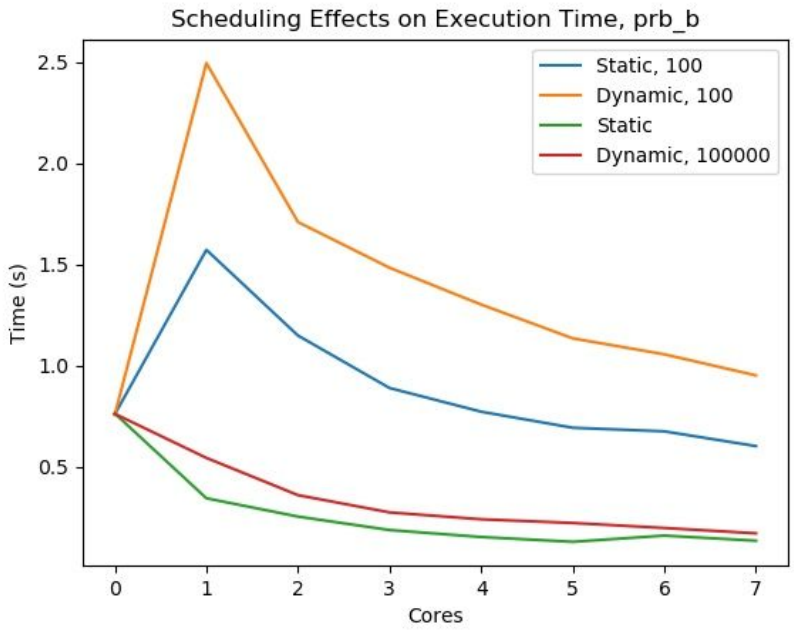


The plot above shows the runtime results for only the default static directive before and after the initialization loop was parallelized. It can be seen that parallelizing the initialization loop yielded a faster time for the while loop for every number of threads forked in the figure above. The scaling is slightly better after parallelizing the initialization loop since threads have already been formed in the program that can be reused.

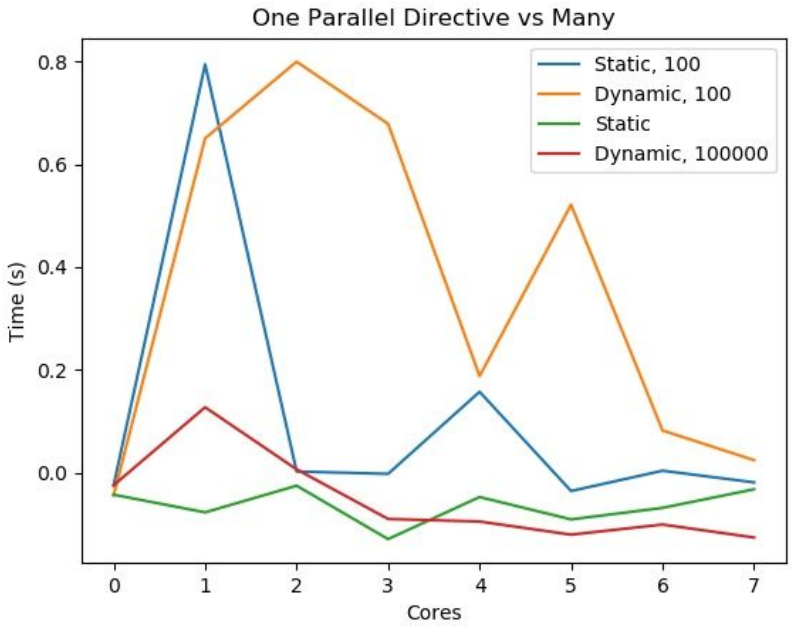
Using top, "1" to see where the threads are running shows an interesting result in that some CPUs are being used more that others, whereas other CPUs seem to be running at random. In particular, CPU 0 and 19 were consistently used, but many others would be used up to a certain percentage and then return to roughly zero.

2.2

The code used in part 2.1 was simplified by reducing the number of parallel regions to just 1 for the whole program. The results are shown below.



As a result of the streamlining of the code, the scaling not substantially better, at least for low numbers of cores. It is definitely smoother and more predictable when compared to the results seen further above. The results below show the difference in the execution time when comparing the single parallel region to the code with multiple.



A negative graph would suggest that the single parallel directive program is faster than the multiple parallel directive, which is true for the larger chunk sizes, but not immediately so for the small chunk sizes, which have a substantial gap at low core numbers, but converge to zero as the number of cores increases. The speed for each different case should run in roughly the same amount of time since forking at this small a number of cores only takes a very small amount of time, on the order of milliseconds or smaller and these executions are on the order of seconds.

2.3

All outputs for this are located in the rb folder under hw1_pt2_cXX.txt.