Morgan Tucker, Homework 1 Report

You can find this code in /home1/08350/morgtuck/pcse_spring_2023/homeworks

Compile using: gcc -fopenmp prb_c.c -o prb_c

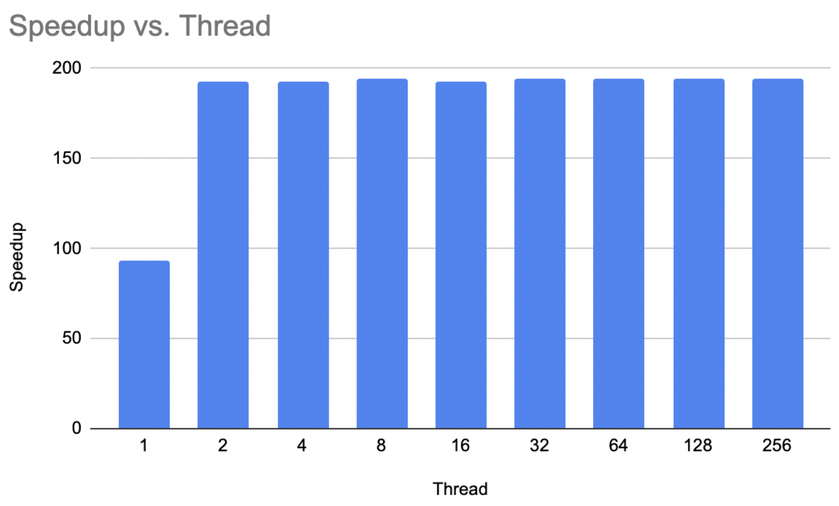
Assignment 1: This assignment focuses on the kernel that counts the # of elements in an array.

- Make sure that your code is properly debugged. Compare the number of elements below the threshold in your serial and parallel version.
- **Report** 3 times:
  - Serial execution of the parallel code: (94s, 141s, 68s)
    - # of elements below the threshold in array x: 966443795
    - # of elements below the threshold in array y: 498275
  - Parallel execution with 1 thread: compile with the -qopenmp flag, set the number of threads to 1 (OMP_NUM_THREADS variable) (95s, 134s, 68s)
    - # of elements below the threshold in array x: 966383878
    - # of elements below the threshold in array y: 499087
  - Parallel execution with 8 threads (194s, 28s, 9s)
    - # of elements below the threshold in array x: 966335198
    - # of elements below the threshold in array y: 498727
- **Discuss** the difference in execution time between the serial execution and the parallel execution with 1 thread. The results may be surprising. Speculate what may be going on.
  - When I execute this code in serial, the program runs through each task one after the other, in order. This is both slow and time-consuming, especially for large and complex programs such as this one. However, when I execute this code in parallel

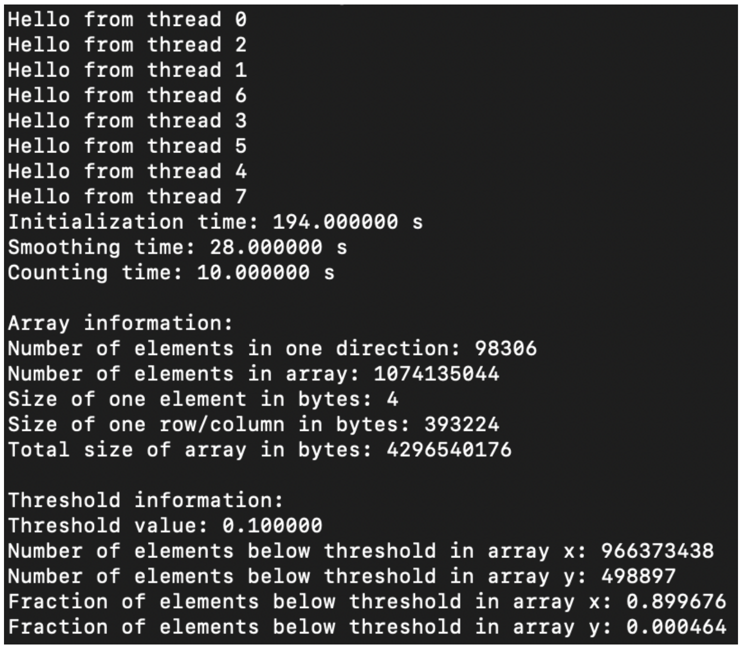
using one thread, the program executes multiple tasks simultaneously and significantly speeds up the overall process. Parallel execution with one thread is a good optimization technique for this program

Assignment 2a: Focus on the ‘initialization’ kernel. **Report** times for this kernel. Discuss the timings for the ‘initialization’ kernel in this assignment. Use the numactl command.

- Run tests with static scheduling without a specific chunk size. **Create** a plot (speedup v. thread count) for 1, 2, 4, 8, 16, 32, 64, 128 and 256 threads. **Add also** the ideal speed-up in either plot.



- **What** setup performs the best? **Discuss** the deviation from the ideal speed-up.
    - The ideal speedup is achieved when doubling the number of threads results in a halving of the execution time. In the given table, the execution time fluctuates and is not continuously decreasing as the number of threads increases. Therefore, we cannot identify an ideal speedup from this table.
- **Add** the output of your code for the run with 8 threads to your report.


```
Hello from thread 0
Hello from thread 2
Hello from thread 1
Hello from thread 6
Hello from thread 3
Hello from thread 5
Hello from thread 4
Hello from thread 7
Initialization time: 194.000000 s
Smoothing time: 28.000000 s
Counting time: 10.000000 s

Array information:
Number of elements in one direction: 98306
Number of elements in array: 1074135044
Size of one element in bytes: 4
Size of one row/column in bytes: 393224
Total size of array in bytes: 4296540176

Threshold information:
Threshold value: 0.100000
Number of elements below threshold in array x: 966373438
Number of elements below threshold in array y: 498897
Fraction of elements below threshold in array x: 0.899676
Fraction of elements below threshold in array y: 0.000464
```

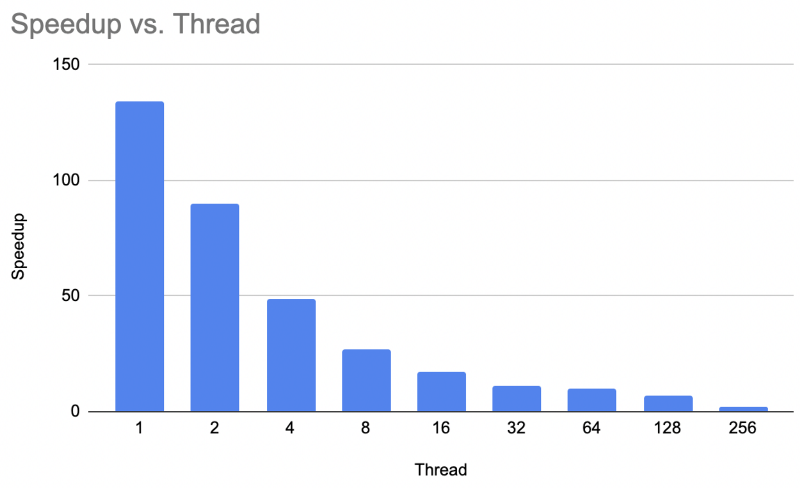
- **In your opinion**: Is the scaling good or bad?
    - The ideal speedup would be equal to the number of threads used. In this case, the maximum number of threads used is 256, so the ideal speedup would be 256. Looking at the data, it appears that the speedup plateaus after using 2 threads, and remains relatively constant at around 193-194. This suggests that there may be a bottleneck in the program that is preventing it from fully utilizing the additional threads. Therefore, while the scaling is not ideal, it is still showing some improvement with additional threads up to 2, but then levels off. There may be room for optimization to better utilize the additional threads and achieve a higher speedup.

Assignment 2b: Modify your code so that the initialization is done twice, either by replicating the call or by using a loop with 2 loop iterations. Make sure that you measure timings for both calls separately. Don't forget to use the numactl command.

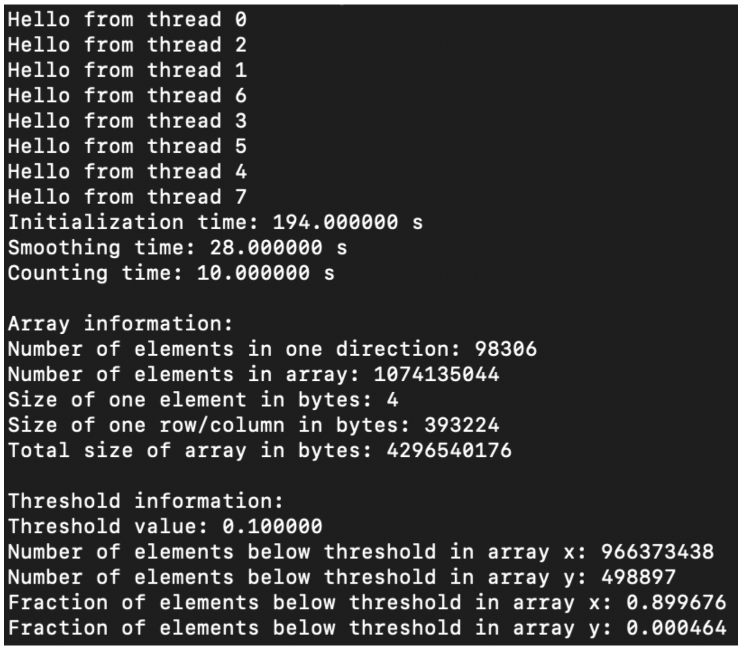
- Run 1 test using 1 thread and **report** individual numbers for both initialization attempts
    - 1 thread, 1 initialization: 95 seconds; 1 thread, 2 initializations: 203 seconds
- **Speculate** why the time to execute the initialization differs b/t the first & second attempt.
    - If I only need to do the initialization process once, it takes 95 seconds. However, if I need to do the initialization process twice, it takes 203 seconds in total (so roughly double the time). The reason for the increased time is that setting up involves some overhead and preparation work, and doing it twice means doing that extra work twice.

Assignment 3a: This assignment focuses on the 'smoothing' kernel. Report only times for this kernel. Only discuss the timings for the 'smoothing' kernel in this assignment. Don't forget to use the numactl command.

- Run tests with static scheduling without a specific chunk size. **Create** a plot (speedup v. thread count) for 1, 2, 4, 8, 16, 32, 64, 128 and 256 threads. Add also the ideal speed-up in either plot.



- **Discuss** the plots. What setup performs the best? Discuss the deviation from the ideal speed-up.
    - The ideal speedup is when the speedup is equal to the number of threads used, which means that the program is perfectly scalable with the number of threads. In this case, the ideal speedup would be 256, since the maximum number of threads used is 256. However, based on the table provided, the actual speedup values are lower than the ideal speedup.
- **Add** the output of your code for the run with 8 threads to your report.



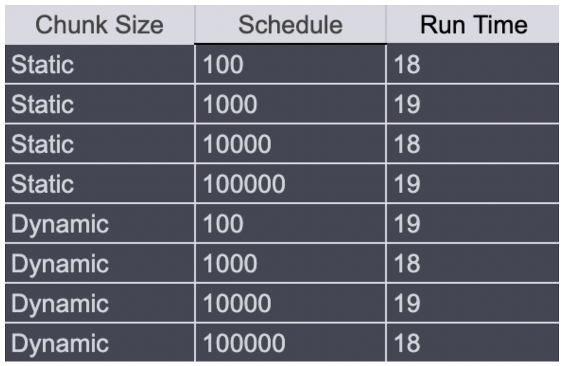
```
Hello from thread 0
Hello from thread 2
Hello from thread 1
Hello from thread 6
Hello from thread 3
Hello from thread 5
Hello from thread 4
Hello from thread 7
Initialization time: 194.000000 s
Smoothing time: 28.000000 s
Counting time: 10.000000 s

Array information:
Number of elements in one direction: 98306
Number of elements in array: 1074135044
Size of one element in bytes: 4
Size of one row/column in bytes: 393224
Total size of array in bytes: 4296540176

Threshold information:
Threshold value: 0.100000
Number of elements below threshold in array x: 966373438
Number of elements below threshold in array y: 498897
Fraction of elements below threshold in array x: 0.899676
Fraction of elements below threshold in array y: 0.000464
```

- **In your opinion:** Is the scaling good or bad?
    - The ideal speedup is when the execution time is divided perfectly by the number of threads used. In this case, the ideal speedup for N threads would be 134/N. For example, if we use 4 threads, the ideal speedup would be 134/4 = 33.5.
    - Looking at the table, we can see that the speedup is not ideal. Although the speedup improves as more threads are added, it does not scale linearly. For

example, adding 8 threads (from 1 to 8) gives a speedup of about 5x, but adding 128 more threads (from 8 to 136) only gives a speedup of about 3x. This suggests that there may be some factors limiting the scalability of the program on this hardware architecture.

Assignment 3b: Again, only the 'smoothing' kernel is of concern here.

- **Create** a table with run-times for a number of different scheduling settings. Use 8 threads for these tests. Include static and dynamic scheduling in your table and chunk sizes of 100, 1000, 10000 and 100000. So in total you should have 2 x 4 entries, i.e. 2 schedules and 4 chunk-sizes. (Below, run time in sec)



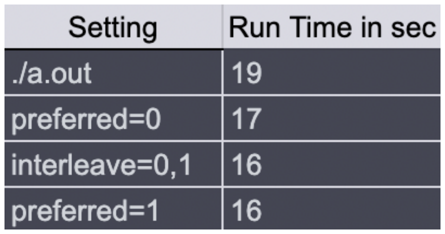
Chunk Size	Schedule	Run Time
Static	100	18
Static	1000	19
Static	10000	18
Static	100000	19
Dynamic	100	19
Dynamic	1000	18
Dynamic	10000	19
Dynamic	100000	18

- **Discuss** the timing differences between the scheduling types (static v. dynamic) and the different chunk sizes.
    - Changing scheduling types and chunk sizes did not make my program run faster. This could be for a variety of reasons such as changing the scheduling type/chunk size may have introduced some overhead in the form of task creation, synchronization, and communication, potentially outweighing any potential benefits of using different scheduling types or chunk sizes. There is also the potential for load imbalance. If my program has a workload that is not evenly

distributed across threads or processors, changing scheduling types and chunk sizes may not have a significant impact on performance. In fact, it may make the problem worse by exacerbating load imbalance.

Assignment 3c: Again use 8 threads.

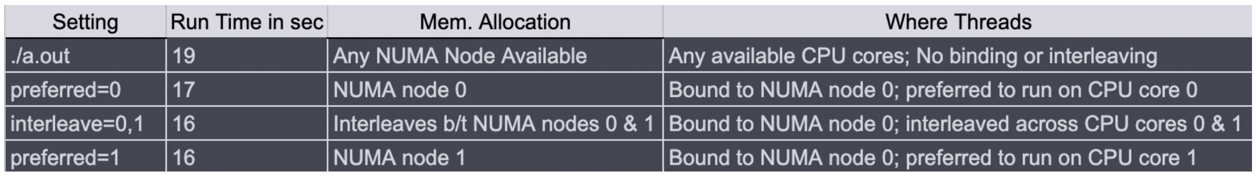
- **Create** a table with run-times for the ‘smoothing’ kernel. Experiment with different numactl settings



Setting	Run Time in sec
./a.out	19
preferred=0	17
interleave=0,1	16
preferred=1	16

- **Explain** what the command ‘numactl’ does. Use the man page for numactl and look into other resources (internet)
    - The 'numactl' command is a Linux utility that allows for the control and manipulation of NUMA (Non-Uniform Memory Access) policy for processes or applications. NUMA is a memory architecture used in modern computers that enables faster access to local memory by allocating memory and CPU resources on the same node. The 'numactl' command can be used to specify the memory placement policy for a given process or application, which can lead to improved performance by reducing memory access latency. Additionally, it allows for the affinity of CPU cores and memory to be set, ensuring optimal performance for multithreaded applications.
- **Explain** the purpose of the 2 options ‘cpunodebind’ and ‘preferred’. Discuss the results.

    - The 'cpunodebind' option in the 'numactl' command is used to bind a process to a specific set of CPUs and memory nodes. This option allows the user to control the placement of processes and their memory on specific NUMA nodes. The 'preferred' option allows the user to set the preferred NUMA node for memory allocation. If the memory allocation on the preferred node is not possible, the kernel will allocate memory on other nodes. By using these options, the user can optimize the performance of their application by minimizing memory latency and maximizing memory bandwidth
- **Speculate** why the execution speed differs. Base your speculation on the fact that a thread that executes on a specific socket can access data faster that is allocated on the same socket than data that is allocated on the other socket. In your table add 2 columns indicating where the memory is allocated and where the threads are running in the 4 Experiments.
    - The execution speed differs between these four options because of differences in memory access time



Setting	Run Time in sec	Mem. Allocation	Where Threads
./a.out	19	Any NUMA Node Available	Any available CPU cores; No binding or interleaving
preferred=0	17	NUMA node 0	Bound to NUMA node 0; preferred to run on CPU core 0
interleave=0,1	16	Interleaves b/t NUMA nodes 0 & 1	Bound to NUMA node 0; interleaved across CPU cores 0 & 1
preferred=1	16	NUMA node 1	Bound to NUMA node 0; preferred to run on CPU core 1

- Execute the command: numactl --hardware
- **Describe** the purpose of the option --hardware? Refer to the man page.
    - The option --hardware in the command numactl --hardware displays the hardware capabilities of the system such as information about the system's NUMA (Non-Uniform Memory Access) topology, including the number of nodes, CPUs, and memory zones. Additionally, it displays information about the memory and

cache hierarchy of the system. This option is useful for system administrators and developers who want to optimize their applications/improve performance.

- **How** many nodes and CPUs are being shown by the command?
    - available: 2 nodes (0-1)
    - node 0 cpus: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54: This shows the list of CPUs (or cores) that are part of NUMA node 0. In this case, CPUs 0-54 are part of node 0.
    - node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55: This shows the list of CPUs that are part of NUMA node 1.
- **How** do the terms ‘nodes’ and ‘CPU’ relate to the terms ‘core’, ‘hyperthreads’ and ‘socket’?
    - A node typically refers to a physical server or a single processor unit, which may contain multiple sockets. A socket is a physical slot on the motherboard that can house a processor, and each socket can contain multiple cores. A core is a processing unit within a processor that can execute instructions. Hyper-threading is a technology that allows a single physical core to behave like two logical cores, providing improved parallel processing. This means that a single core can handle two threads of execution at the same time. A CPU is a processing unit that contains multiple cores and can be housed on a single socket or multiple sockets. The terms "nodes" and "CPUs" typically refer to the physical units, while "cores," "hyper-threads," and "sockets" refer to the components inside the CPUs that determine their performance and capabilities.

- **Add** to your report the total amount of memory for the available nodes. Results in MB or GB are accepted
    - node 0 size: 94656 MB: This shows the total size of the memory available in node 0 in megabytes.
    - node 0 free: 90908 MB: This shows the amount of free memory available in node 0 in megabytes.
    - node 1 size: 96730 MB: This shows the total size of the memory available in node 1 in megabytes.
    - node 1 free: 94650 MB: This shows the amount of free memory available in node 1 in megabytes.

Assignment 4:

- Inspect the man page of your compiler on Frontera (either icc or ifort). **Explain** briefly what the option –xhost instructs the compiler to do. Speculate why it is important to use the 'highest instruction set available'.
    - The option -xhost instructs the compiler to generate code for the highest instruction set available on the host machine. This option is important as it enables the compiler to generate code that can take advantage of the full capabilities of the processor, including its advanced instructions and features. By using the highest instruction set available, the compiler can optimize the code to run faster and more efficiently, which can result in significant performance improvements for compute-intensive applications.