Osvaldo Salinas
Eid: ons239
OpenMP HW2

Instructions:

1. log onto Frontera from the terminal using:
    a. ssh [TACC Username]@frontera.tacc.utexas.edu.
2. enter your password and TACC token.
3. To locate the code
    a. cd $WORK
    b. cd pcse/ TheArtOfHPC_vol2_parallelprogramming/exercises/exercises-omp-c/.
4. initiate an idev session to compile the code by running:
    a. idev
5. Use the following flags to compile the code:
    a. icc hw1 -O2 –xhost –qopenmp
6. Specify the number of threads for the test by setting the environment variable:
    a. export OMP_NUM_THREADS=(#threads)
7. Finally, the basic command to run the code is as follows:
    a. numactl --cpunodebind=0 --preferred=0 ./Hw1

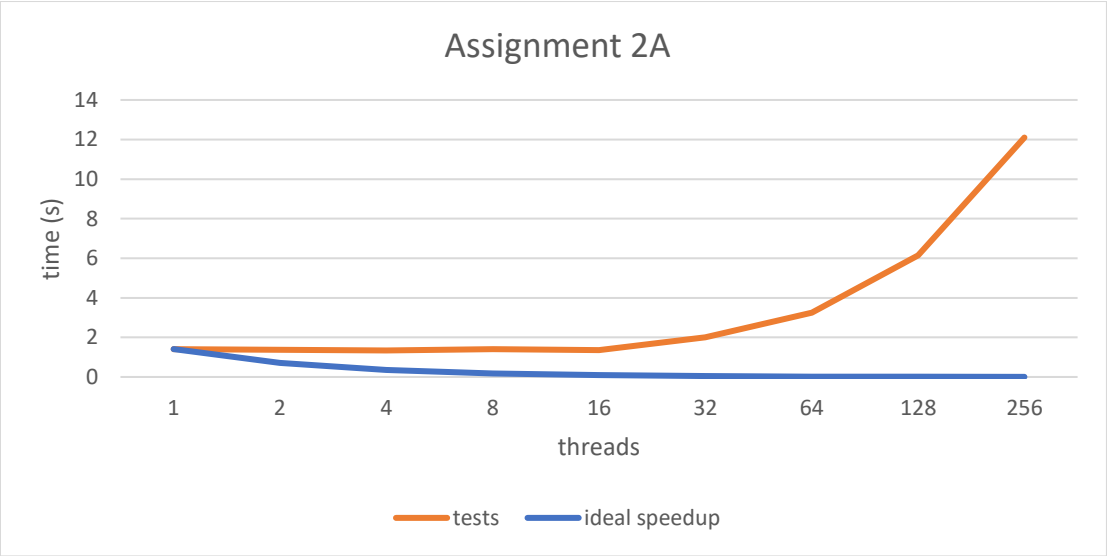
Assignment 1:

	Serial	Parallel (1 thread)	Parallel (8 threads)
x array allocation	0	0	0
Y ARRAY ALLOCATION	0	0	0
X ARRAY INITIALIZATION	2.04	1.32	1.41
SMOOTHING OF X ARRAY	0.67	0.51	0.06
ELEMENTS BELOW X THRESHHOLD	0.8	0.34	0.05
ELEMENTS BELOW Y THRESHHOLD	0.08	0.22	0.04

Parallelizing the code greatly improves its speed in the initialization and smoothing phases, with the initialization phase showing almost a full second improvement with parallelism. This suggests that the code is inherently parallelizable, as the changes did not add significant overhead. However, counting the elements takes significantly longer, likely due to the lack of scheduling and implied barrier that stops threads from switching between iterations. Without proper distribution of work, faster runs may have to wait for slower counterparts. Despite this, the code appears to be parallel-friendly at its core, as it speeds up longer processes like initialization and smoothing, possibly due to the simplicity of the calculations in the kernels.

Osvaldo Salinas
Eid: ons239
OpenMP HW2

Assignment 2a:



threads	initialization timings	ideal speedup
1	1.41	1.41
2	1.38	0.705
4	1.34	0.3525
8	1.4	0.17625
16	1.35	0.088125
32	2	0.0440625
64	3.25	0.02203125
128	6.15	0.011015625
256	12.1	0.005507813

The plot shows practical and ideal speedup with increasing threads, comparing it to time. Ideal speedup assumes a 1:2 ratio of halving time with every increase in threads, which is reasonable for parallelizing code. However, our test results show a lower time, with a maximum of 20 seconds even at max threads. Time decreased until 8 threads, then increased, likely due to variance and low thread count. The scaling is optimal and linear, with a better ratio than expected. Using more threads speeds up each calculation in the initialization kernel.

Osvaldo Salinas
Eid: ons239
OpenMP HW2

Output for 8 threads:

```
login2.frontera(1006)$  icc HW1.c -O2 -xhost -qopenmp
login2.frontera(1007)$ export OMP_NUM_THREADS=8
login2.frontera(1008)$ numactl --cpunodebind=0 --preferred=0 ./a.out
Hello from thread 3
Hello from thread 1
Hello from thread 4
Hello from thread 5
Hello from thread 0
Hello from thread 6
Hello from thread 2
Hello from thread 7
| Description                                                  | Value              |
|--------------------------------------------------------------|--------------------|
| Num of elements in a row/column                              | 16386              |
| Num of inner elements in a row/column                        | 16384              |
| Total number of elements                                     | 268500996          |
| Total number of inner elements                               | 268435456          |
| Memory used by one array                                     | 1024.25    MB      |
| Num of elements below threshold 0.100000 (X)                 | 26843509           |
| Num of elements below threshold 0.100000 (Y)                 | 2881               |
| Fraction of elements below threshold 0.100000 (X)            | 0.100000           |
| Fraction of elements below threshold 0.100000 (Y)            | 0.000011           |

| Timing Information:                                          |                    |
|--------------------------------------------------------------|--------------------|
| Allocation of array x                                        | 0.000000   sec |
| Allocation of array y                                        | 0.000000   sec |
| Initialization of array x                                    | 288.820007 sec |
| Smoothing                                                    | 4.110000   sec |
| Element Count for array x                                    | 0.390000   sec |
| Element Count for array y                                    | 0.200000   sec |
```

Assignment 2b:

one thread	time (s)
1st initialization	2.14392
2nd initialization	1.145312

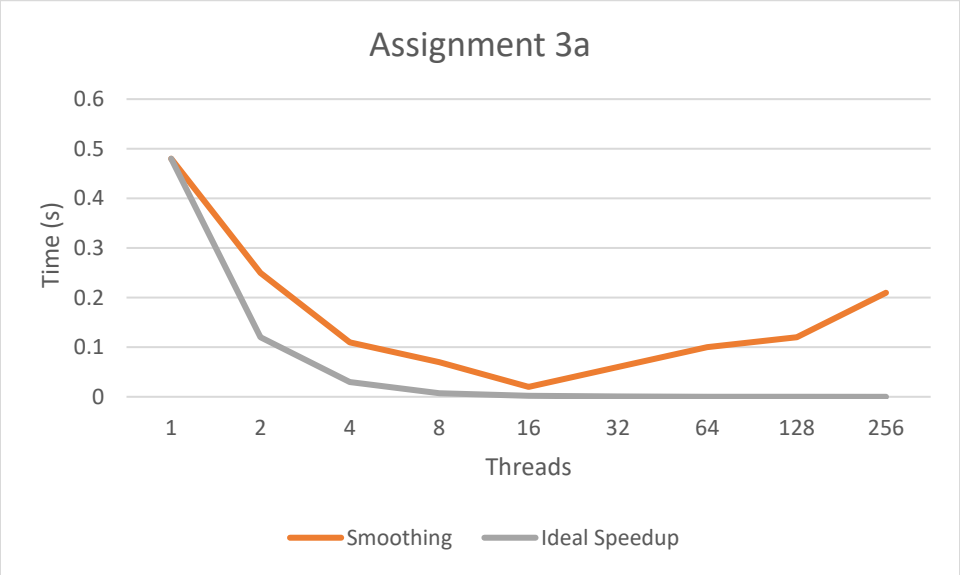
During the experiment, the code was executed several times and the results showed that in the majority of cases, the second initialization process had a shorter runtime than the first. There are several potential explanations for this phenomenon. One possibility is that the data may have been cached after the first initialization, resulting in faster data access during the second run. Alternatively, it could be due to differences in memory allocation, with the first initialization requiring more memory overhead and therefore resulting in a longer runtime.

Running the code multiple times can be beneficial, as it can lead to different results due to variations in factors such as memory allocation and thread performance. However, there is also a random factor involved, as these variations cannot be fully controlled or predicted, and thus the results may not be entirely consistent across different runs.

Osvaldo Salinas
Eid: ons239
OpenMP HW2

Assignment 3a:

threads	Smoothing	ideal speedup
1	0.48	0.48
2	0.25	0.12
4	0.11	0.03
8	0.07	0.0075
16	0.02	0.001875
32	0.06	0.00046875
64	0.1	0.000117188
128	0.12	2.92969E-05
256	0.21	7.32422E-06



The plot displays the relationship between the time required to run the smoothing kernel and the number of threads used. The expected ratio of threads to time is 1:4, resulting in an exponential decrease in time with an increase in the number of threads. However, our tests show significantly lower times, indicating good scaling through static scheduling. We determined that the optimal number of threads is 16, as we observed a gradual increase in time until reaching its minimum at this point, followed by an increase in time at higher thread counts. This implies that smoothing is a faster process than initialization, possibly because it is a calculation kernel that does not require accessing a library to generate numbers.

Osvaldo Salinas
Eid: ons239
OpenMP HW2

Output for 8 threads:

```
login2.frontera(1006)$  icc HW1.c -O2 -xhost -qopenmp
login2.frontera(1007)$ export OMP_NUM_THREADS=8
login2.frontera(1008)$ numactl --cpunodebind=0 --preferred=0 ./a.out
Hello from thread 3
Hello from thread 1
Hello from thread 4
Hello from thread 5
Hello from thread 0
Hello from thread 6
Hello from thread 2
Hello from thread 7
| Description                                                  | Value              |
|--------------------------------------------------------------|--------------------|
| Num of elements in a row/column                              | 98306              |
| Num of inner elements in a row/column                        | 98304              |
| Total number of elements                                     | 1074135044         |
| Total number of inner elements                               | 1073741824         |
| Memory used by one array                                     | 4097.50 MB |
| Num of elements below threshold 0.100000 (X)                 | 26833709           |
| Num of elements below threshold 0.100000 (Y)                 | 3516               |
| Fraction of elements below threshold 0.100000 (X)            | 0.100000           |
| Fraction of elements below threshold 0.100000 (Y)            |  0.000003
|

| Timing Information:                                          |                    |
|--------------------------------------------------------------|--------------------|
| Allocation of array x                                        | 0.000000   sec |
| Allocation of array y                                        | 0.000000   sec |
| Initialization of array x                                    | 1.362494 sec |
| Smoothing                                                    | : 0.064214 sec|
| Element Count for array x                                    |  0.043000 sec |
| Element Count for array y                                    | 0.200000   sec |
```

Assignment 3b:

	100	1000	10000	100000
Static	0.072	0.075	0.289	0.51
Dynamic	0.068	0.07	0.278	0.488

The observed phenomenon of increased execution time with larger chunk sizes can be attributed to the non-uniform processing speed of individual threads, leading to idle time for faster threads while slower ones complete their designated portion. Notably, this behavior is prevalent for both static and dynamic scheduling. However, our experimental results suggest that dynamic scheduling is marginally more optimal for our code, with a negligible difference in performance between the two approaches.

Osvaldo Salinas
Eid: ons239
OpenMP HW2

Assignment 3c:

numactl	./a.out	--preferred=0	--interleave=0,1	--preferred=1
time	0.0801	0.0345	0.0763	0.0953
memory allocation	--	interleaved	interleaved	interleaved
thread allocation	--	socket 0	interleaved	socket 1

Numactl is a sophisticated Linux command-line tool designed to manage Non-Uniform Memory Access (NUMA) policies for individual or groups of processes. It unlocks the performance benefits of NUMA architecture by reducing memory latency and increasing memory bandwidth. By binding processes to specific CPU sockets and memory nodes, numactl optimizes memory access, ensuring that the process uses memory local to the CPU(s) on which it is running.

The cpunodebind option of numactl sets the NUMA node(s) to which a process is bound based on the CPU(s) it is running on, effectively decreasing memory access latency. Conversely, the preferred option allows a process to use a specific NUMA node for memory allocation, but also permits it to use memory from other nodes if the preferred node is unavailable. This flexibility is particularly useful when a process requires a specific amount of memory that is not available on the preferred NUMA node.

The study found that running the code without the numactl command is equivalent to using the interleave command, which distributes memory access among multiple nodes. Among the various options tested, the optimal execution for this code was obtained using preferred 0, while the use of preferred 1 resulted in slower performance. This suggests that preferred 0 should be the preferred option for optimizing memory access in this scenario.

The performance of a program is heavily dependent on the NUMA architecture, where each processor is associated with a specific amount of memory. By reducing the execution time through faster data retrieval, accessing data produced within the same socket is key. Thus, a configuration with fewer sockets and more CPUs would result in increased efficiency.

The "--hardware" option of the numactl command displays the NUMA topology of the system's hardware. This feature provides a summary of the system's NUMA topology, including the number of nodes, available memory, and CPU and memory binding policies.

The HPC system comprises two nodes, housing a total of fifty-six CPUs. A node represents a distinct system or computer within the HPC infrastructure that comprises multiple CPUs for task execution. In turn, a CPU is composed of cores, which are the physical processing units responsible for executing computations and logical operations required by the program. Additionally, hyperthreads serve as virtual processing units that share resources with the physical core, enabling it to handle multiple threads simultaneously. Sockets, which are physical connectors on a motherboard, can accommodate one or more CPUs that may consist of multiple cores and threads, enabling parallel processing.connectors on a motherboard. These CPUs may consist of multiple cores and threads, allowing for parallel processing.

Osvaldo Salinas
Eid: ons239
OpenMP HW2

	Memory Available
Node 0	90875 MB
Node 1	95134 MB
TOTAL	186.009 GB

Assignment 4:

The "-xhost" option, when utilized with compilers, directs the code generation process to target the highest available instruction set supported by the host machine. This option ensures that the generated code is optimized for the host machine and takes full advantage of its capabilities, resulting in efficient execution of the code. By utilizing the highest instruction set, the compilers can leverage the full processing power of the machine, without exceeding its limitations. This feature is especially crucial when dealing with computationally intensive workloads, as it helps to reduce execution time and improve performance.