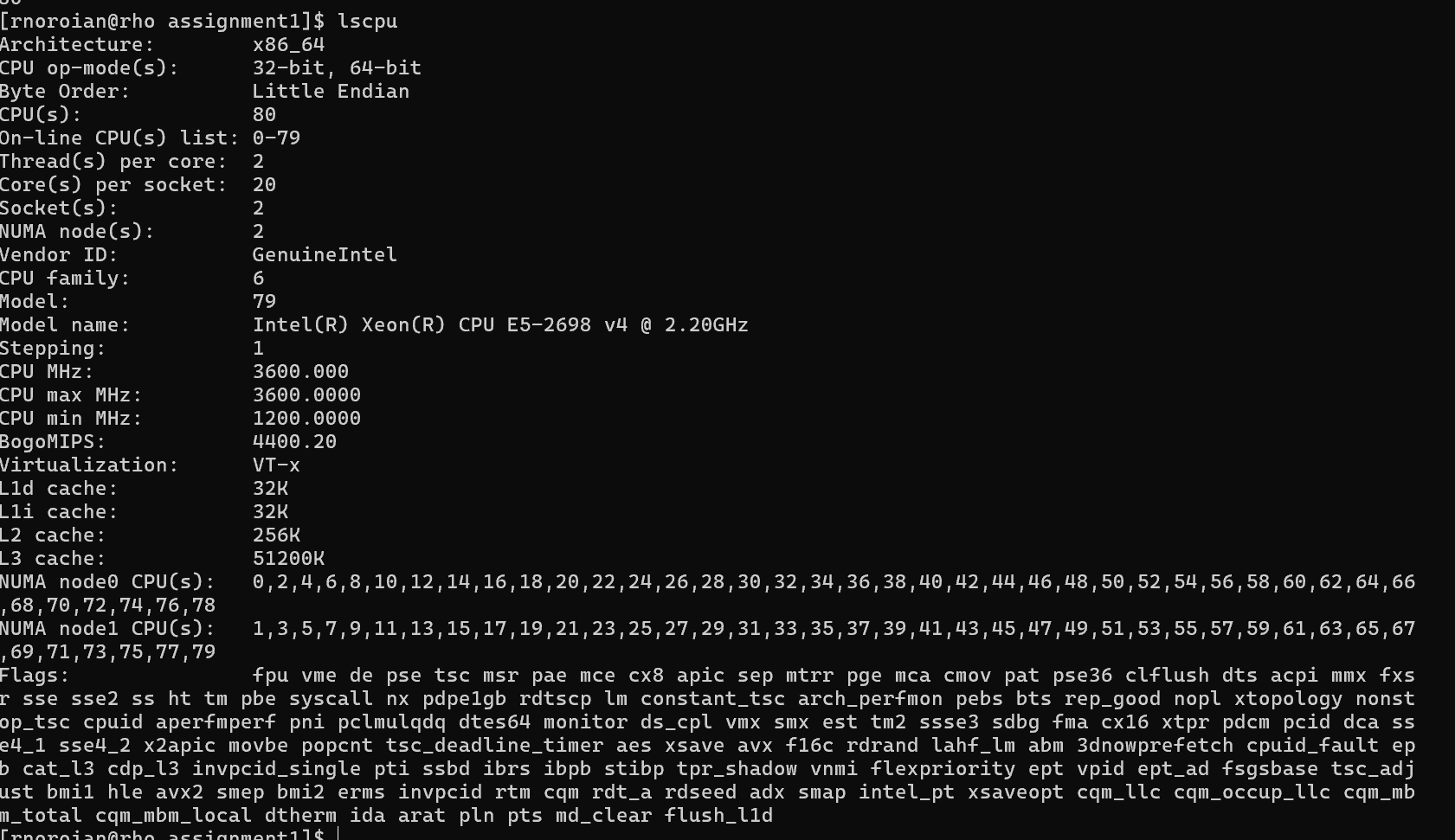
2. Sorting Algorithm to sort 10,000 random integers with values 1-10,000 using pthreads.

I used the merge sort algorithm for this problem. In this sorting algorithm, the array is repeatedly halved and then progressively merged in a sorted fashion. It repeatedly sorts smaller portions of the array until the entire array is sorted. I was working with a CPU with the following specifications. It exceeds the minimum of 4 cores. 

a) Program with 1, 2, 4, 8, 32 threads

I ran, at minimum, 5 trials for each thread count.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Thread Count: | 1 | 2 | 4 | 8 | 32 |
| Speed (ms): | 1.65 | 1.21 | 0.85 | 0.92 | 2.12 |

Optimal performance was measured with 4 threads, as time increased due to thread overhead after forking more threads.

b) Describe challenges faced when performing sorting with multiple threads

In merge sort, there are several challenges, and the primary challenge exists when merging sections of the array sorted by each thread. Each thread sorts a section of about the same size from 1 to 10,000. After they complete it, it would require additional overhead to introduce threads that would merge the sorted sections of the array. For example, if there were 8 threads sorting 4 integers each, then 4 threads could be introduced to merge these 8 sections into 4. However, after this, another 2 threads would need to be introduced to merge the sections in 2, and so forth.

In my program, I included this merging in the time evaluation, though it didn’t include multi-threading with pthreads. It requires iteratively sorting sections of the array of increased size until there are two sections of the array merged into one.

c) Evaluate both the weak scaling and strong scaling properties

In this sorting problem, since integers are the datatype being sorted, the process is simple and 10,000 for the array size means there is increased overhead with threaded parallelization.

This program exbibits poor strong scaling, as the number of threads increases while the array size remains the same. For my sorting implementation, forking new threads is too costly at a greater scale for this simple problem. Introducing a greater number of threads would increase the speed of merge sort, as demonstrated by the time difference between 32 and 8 threads. 4 threads were shown to be the most efficient, with an average run time of 0.85 ms.

In weak scaling, the number of threads would increase proportionally to the size of the array. To test this at optimal performance of 4 threads, the size of the array was increased. Based on previous performance, it is expected that at 50,000 integers, the optimal thread count should be about 20, and for 100,000 integers, it should be about 40. This is based on the proportional 0.85 ms:4 ratio for the original tests. The previous time per thread was about 0.2 ms.

50,000 Integer Array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Thread Count: | 8 | 16 | 20 | 32 |
| Speed (ms): | 3.03 | 3.44 | 3.58 | 4.54 |

100,000 Integer Array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Thread Count: | 16 | 32 | 40 | 64 |
| Speed (ms): | 5.78 | 6.84 | 7.33 | 9.88 |

The time per thread at 20 threads for 50,000 integers was about 0.179 ms, and for 40 threads at 100,000 integers, was 0.18 ms. This means that this program exbibits good weak scaling, as the time per thread is relatively the same. However, for these different problem sizes, the scaled thread count isn’t the most efficient, as not all amounts of threads have a constant time per thread, and the lower thread counts, with less overhead, have more efficient completion times.