Supporting Parallelism in Operating Systems & Programming Languages

Exercise 1: Warm-Ups

Technical Report

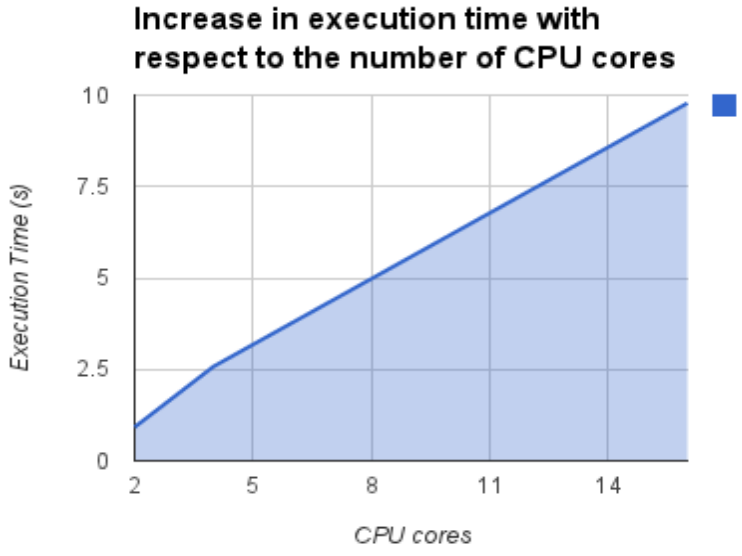
1. Parallel generation of pseudo-random numbers

Q1.1: Can you determine the performance of a parallel program using only the execution time of each of its threads in the general case?

If we consider the ideal case, when assigning task to threads and merging the results from the threads is instantaneous, the performance (execution time) of the parallel program can be measured as the time interval between the start of the first thread and the completion of the last thread.

However, in the non-ideal case, things are different:

- the threads might not run in the same time and on different cpus, which means that 2 threads could have very different execution times, which will affect the total program execution time.
- creating a very large number of threads can cause the task assignment period (when threads are created and assigned some work) and result merging (when results from the threads are used towards some computations) to take much longer as the number of threads increases.
- creating a large number of threads, even when each thread runs on its dedicated CPU will eventually create a bottleneck. This happens because of a constant contention for resources like the memory and IO bus or the network.



For the first exercise, we analyzed the execution time of a program which performs a parallel

generation of pseudo random numbers. In our experiments we created a number of threads equal to the number of CPU's available in the system and affined each thread to a CPU. Each of the aforementioned threads generates a random number in a loop, for 10^5 iterations. We noticed that the execution of the program increases dramatically with the number of threads created. For example, when only 2 threads generate random numbers, the execution time is close to 1 s, while when 16 threads perform the same work, the execution time is almost 10 s. This increase in the execution time is caused by the contention of all the created threads for the memory bus.

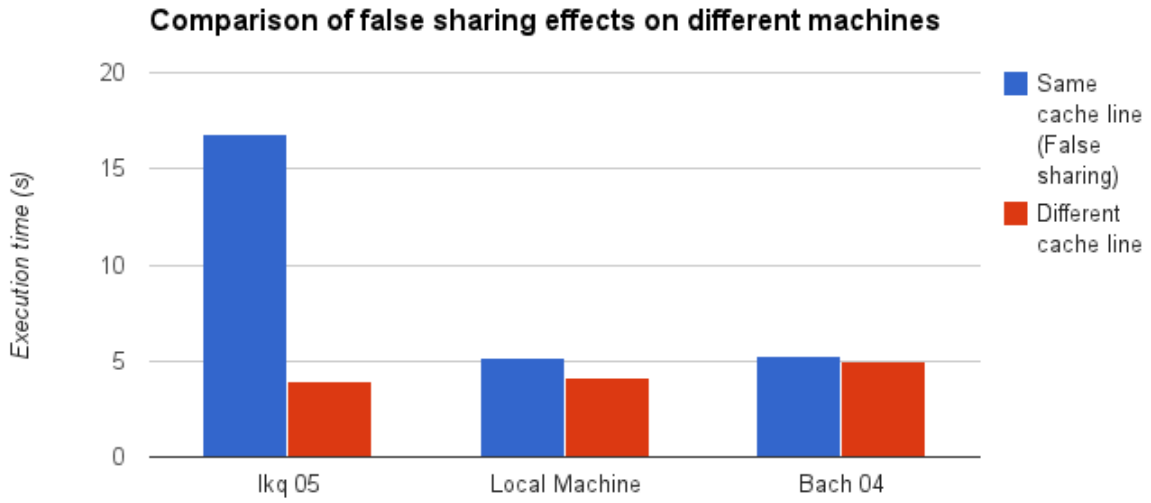
2. False sharing

False sharing is a phenomena that appears in computer science when 2 or more process access neighbouring memory locations. If each process is executed by a different CPU, each CPU can hold a cache line which contains both of these memory locations. Because the cache operates at the level of cache lines, whenever a CPU performs an operation on its memory location, the whole cache line is marked as dirty. Because of this, each CPU must go to the main memory to check the value at the memory locations it is working on.

In our experiments, we compared the effects of false sharing on different machines. Our approach was the following:

- we allocated an array of values, as array locations are allocated contiguously.
- we created a number of threads, each affined to a specific CPU. Each thread would increment the value corresponding to an array index, in a loop, for a large number of times.

We expected to see the effects of false sharing when the indexes of the array elements are close or neighboring.



The graph plotted shows the difference in the execution time of the same program when 2

threads are incrementing different memory locations from the same integer array for 10^9 iterations. For the first case, represented in blue, we chose 2 neighbouring index, i and i+1. In this case, the memory locations are located in the same cache line. For the second case, the indexes are further apart. In our experiments, we noticed that a difference in execution time appears only if the difference between array indexes chose is greater than 7. The explanation for this is that a distance of 8 integer memory locations, given the size of a integer memory location to be 4 B, guarantees that the two memory locations are 64 B apart, which means that each location is loaded in a different cache line.

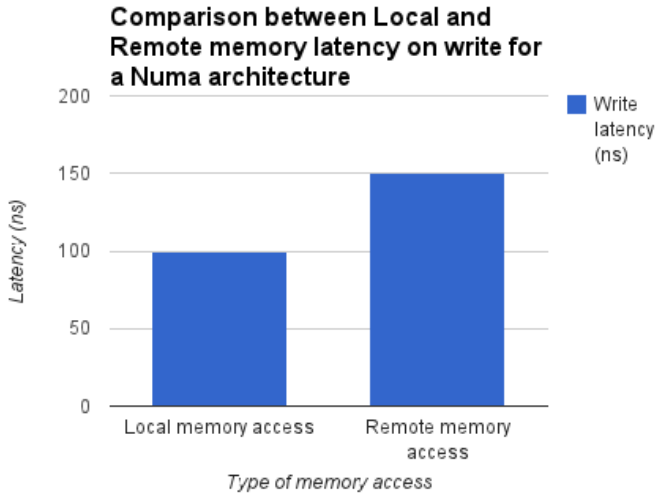
3. Numa allocation

NUMA - (Non - Uniform Memory Access) - is an architecture where the memory access time depends on the memory location relative to each processor. This architecture contains a number of NUMA nodes, each containing a separate memory and a number of CPU's. We can therefore distinguish between local memory access (the memory accessed is on the same node) and remote memory access, when the memory resides on a different node.

For our experiments, we aimed to test the latency of a memory for write operations. We chose to measure the write access time because:

- write access usually takes longer and a difference between write access times can be spotted more easily.
- C compilers can usually perform optimizations for code that performs a large number of read access in a row and eliminate memory reads if the result is not used.

Our approach consisted in a series of writes to random indexes of a very large array (10^8 elements). For this, we created an array of integer values in the interval [0, n] which we then shuffled. At each step, we marked the value of the current array location as the next index and then performed a write on the current location. By doing this n times we try to access a memory location which is not in the cache at each iteration.



The graph shows the write access time for local and remote memory as measured by our

program. It can be easily seen that accessing remote memory is much slower than accessing local memory.