Multicore Processors - Spring 2019
Homework 1

Daniel Rivera Ruiz
Department of Computer Science
New York University
drr342@nyu.edu

1. There are several types of parallelism that we can find in different programs. What are they? For each one, specify whether exploiting that type needs programmer involvement or the hardware/-compiler is enough to exploit it.

   There are four phases of microprocessor design trends that are characterized by the internal use of parallelism:

   (a) *Parallelism at bit level.* Incrementing the word size used by the processor (e.g. 32 or 64 bits in modern systems).

   (b) *Parallelism by pipelining.* Overlapping of the execution of multiple instructions by partitioning them into multiple steps.

   (c) *Parallelism by multiple functional units.* The processor uses several independent functional units like ALUs, FPUs, load/store units and branch units that work in parallel

   (d) *Parallelism at process or thread level.* Alternative approach that uses the increasing number of transistors on a chip by putting multiple, independent processor cores onto a single processor chip.

   The first three techniques assume a single sequential control flow which is provided by the compiler and determines the execution order if there are dependencies between instructions. For the programmer, this has the advantage that *a sequential programming language can be used nevertheless leading to a parallel execution of instructions.* However, the degree of parallelism obtained by pipelining and multiple functional units is limited.

   The last technique results in processor chips called multicore processors. Each of the cores of a multicore processor must obtain a separate flow of control, which means parallel programming techniques must be used. The cores of a processor chip access the same memory and may even share caches. Therefore, memory accesses of the cores must be coordinated.

   Within the context of the Flynn's Taxonomy for Parallel Architectures, the first three techniques correspond to a SISD architecture, while the fourth one corresponds to the MIMD architecture. The MISD execution model is very restrictive and no commercial parallel computer of this type has ever been built. The SIMD approach can be very efficient for applications with a significant degree of data parallelism and is the default model used to program GPUs.

2. Multiprocessor systems, where we have several chips each of which is a single core, have been around for several decades now. This means we should already have good experience dealing with parallel systems. Yet, we are facing challenges dealing with multicore processors.

   (a) List all the differences you can think of between traditional multiprocessor systems and the current multicore processors.

   - Traditional multiprocessor systems can only take advantage of instruction-level parallelism (ILP) like the one described in the first three techniques of the previous question.
   - Multicore processors can additionally implement thread-level parallelism (TLP) by running separate threads in parallel in addition to exploiting ILP among individual instructions within each thread.
   - Due to the much lower communication latencies between cores, multicore processors can incorporate new features such as speculative thread mechanisms.

- Traditional multiprocessor systems are distributed memory machines (DMM), where each processor can only access its local memory directly and all other communications among processors is accomplished through message passing.
- Multicore processors are shared memory machines (SMM), which consist of several processors or cores and a shared physical memory (global memory). Data can be exchanged between processors via the global memory by reading or writing shared variables.

(b) List one or more cases where our expertise with traditional multiprocessor systems is helpful in dealing with multicore processors.

- A programmer can take advantage of a multiprocessor system by dividing a program into tasks that can be parallelized and assigning them to different processors. The same principle can be applied to a multicore system, assigning tasks to different cores.

(c) List one or more cases where our expertise with traditional multiprocessor systems is NOT helpful in dealing with multicore processors.

- A multiprocessor system can easily handle many independent sequential applications. This is possible because each processor has its own resources and shares nothing with the rest of the system. The same is not true in multicore systems, where some resources are shared and must be managed carefully (e.g. memory).

3. What do you think are the factors that can make an application very hard (or sometimes impossible) to parallelize?

- There are dependencies among instructions that make the code inherently sequential.
- The algorithm used is inherently sequential and no parallel alternative is available.
- The application requires a very specific sequential memory access pattern that cannot be modified.
- The tasks performed by the application are intertwined and are hard to differentiate from one another.
- The application relies on a lot of third party libraries which only exist in a sequential version.

4. If you are given a sequential program that you are required to parallelize, first you need to find the parts that are parallelizable. However, in some cases, it is not worth it to parallelize those parts, why?

   If the parallelizable parts of a program account for a very small portion of the overall execution time of the program, it is not worth parallelizing them because the overall speedup will be negligible. In terms of Amdahl's law:

$$S_P = \frac{1}{F + \frac{1-F}{P}}$$

   where P is the number of cores and $F \in [0, 1]$ is the inherently serial portion of the code. Therefore, as F gets closer to 1 so does the speedup S_P, which means that the parallelizing has no remarkable effect.

5. Suppose you have two parallel programs that solve the same problem. State two factors that can make you pick one program over the other (beside price of course).

- *Portability:* a program that is not strongly tied to a particular hardware, OS, network architecture, etc. is preferable, since it will be easier to implement it in different systems.
- *Scalability:* a program that is designed to scale easily with the size of the problem and the available hardware is preferable, since it will require less modifications to maintain the desired performance level.

6. Suppose, for a specific problem, we know the best algorithm for it for a single core (e.g. quicksort is best for sorting). Does this mean that this algorithm is also the best for multicore? Justify.

   Not necessarily. The best algorithm for a sequential program may have some of the properties listed in question 3, and therefore parallelizing it could be very hard. It could also be the case that the portions of the algorithm for a single core that can be parallelized account for a very small fraction of the overall execution time, and therefore parallelizing it will not yield any significant improvements (Amdahl's law).

   These factors, along with the challenges introduced by parallel programming such as communication among cores, synchronization, etc. must be considered to determine the best algorithm for a multicore solution.

7. Suppose we have the algorithm (assume N is a large even number):

```
for(i = 0; i < N/2; i++)
    a[i] += a[i+ N/2];
```

   (a) Can we parallelize the above algorithm? If no, why not? If yes, explain.
   The algorithm can be parallelized because each iteration of the for loop is independent from all the others. Assuming that the array `a` has N elements, what the for loop does is basically an element-wise addition of the first half and the second half of the array, overwriting the first half with the result of the addition.

   (b) What is the maximum number of cores after which no performance enhancement can be seen? Justify.
   Since there are $N/2$ additions to perform, having more than $N/2$ cores will not yield any performance enhancement (the additional cores will remain idle while the $N/2$ cores perform the additions).