Department of Information and Computer Science
Utrecht University

INFOB3CC: Parallelism (solutions)

Trevor L. McDonell

December 15, 2022

Introduction

Use these tasks to practice the topics of the lectures. You may have to do some research to read up on terms or topics not (yet) covered in the lectures.

Questions

1. We discussed Moore's curve and the difficulties in increasing processor performance, specifically the limitations in power dissipation, memory speed, and instruction-level parallelism. To gain more understanding on these topics read the following articles and answer the questions:

   - The Free Lunch is Over: A Fundamental Turn Towards Concurrency in Software
     `https://edu.nl/yrk68`
   - The Future of Computers (Part 1): Multicore and the Memory Wall
     `https://edu.nl/tvcyd`
   - The Future of Computers (Part 2): The Power Wall
     `https://edu.nl/9khdt`

   (a) Why is it unlikely that we will ever get a 10GHz CPU?

   > **Solution:** Limitations on how much power a processor can consume (and thus dissipate) and the speed at which the processor can be fed with instructions and data (memory limitations).

   (b) Sandia Labs reports that as the number of cores increases, CPU power *decreases* at an exponential rate. Why?

   > **Solution:** The extra processor cores were starved of memory bandwidth, causing them to substantially sit idle waiting for memory and consequently not consume any power (but also not get any work done).

   (c) What does "embarrassingly parallel" mean?

   > **Solution:** A problem which can be broken into many individual pieces which are all independent and can be worked on in parallel (with each other).

   (d) According to the article, caches are reaching practical limits. Instead of using caches, why do we not directly increase memory bandwidth; for example, by increasing memory frequency?

   > **Solution:** Increasing memory frequency increases power consumption. There are also physical limitations related to the proximity of memory to the CPU itself (that is, the layout of the motherboard and the signal delay in communicating between components outside the CPU).

(e) In what way(s) does temperature affect performance?

Solution: Increasing temperature of the processor decreases the speed at which the transistors operate, as well as increasing the amount of leakage (waste) heat generated by the processor (further exacerbating the problem). Over-heating processors are more likely to fail.

(f) What is Dennard Scaling?

Solution: As transistors get smaller, both their voltage and current requirements decrease proportional to the linear size of the transistor, meaning that the power density per unit area remains constant.

(g) What is Dynamic Voltage Scaling?

Solution: The ability of the processor to change its internal voltage level depending on what the processor is doing; when idle the processor can lower the core voltage to save power, and increase it when there is work to do.

2. Definitions of concurrency and parallelism.

(a) What is concurrency? Give an example of a problem which can be solved using concurrency.

Solution: Concurrency is about dealing with lots of things at once. Concurrency is about the structure of the problem, where in a concurrent program there are multiple independently executing processes where two or more threads are making progress. Concurrency can be achieved on a single processor with threads taking turns executing (time slicing). An example would be a web server where the interaction with each client is handled by a separate thread.

(b) What is parallelism? Give an example of a problem which can be solved using parallelism.

Solution: Parallelism is about doing lots of things at once. Parallelism is one possible way to execute a concurrent program when multiple execution units (processors) are available. Parallelism requires the simultaneous execution of multiple (possibly related) parts of the program. An example would be a sorting algorithm such as quicksort, where the sub-tasks are broken up over multiple processors using a divide-and-conquer strategy to be solved in parallel. It is important to note that the goal of parallelism is to reduce the overall (wall-clock) running time of a program.

(c) What does it mean for an application to be concurrent but not parallel?

Solution: An operation/application which is concurrent but not parallel is one that does not require multiple processors to execute correctly. Concurrent execution is possible on a single processor.

(d) What does it mean for an application to be parallel but not concurrent?

Solution: An operation/application which is parallel but not concurrent is one in which multiple operations must be executed simultaneously; there is no ability to split the operations to execute sequentially on a single processing unit.

3. Improving applications through the use of parallelism.

(a) In what ways can parallelism be used to improve the performance of an application?

Solution: Examples include:

- Having multiple processors coordinate on computing a single value, in order to reduce the total execution time (latency).
- Have multiple processors work on different problems (or, different instances of the same problem), in order to increase the rate at which a series of results are computed (throughput)
- Reducing the power consumption of a computation, for example if a computation can be offloaded to specialised hardware which is more efficient for a particular task (such as the GPU for rasterisation)

(b) What does Amdahl say about the maximum performance improvement which can be achieved by a parallel application?

Solution: Amdahl considered parallel application time to be split between time spent doing sequential (non-parallelisable) serial work, and time spent doing parallel work. In this framework the speedup is limited by the degree of sequential execution in the program. Depending on the fraction of sequential work in the program, at some point adding more cores will no longer improve the performance of the application.

(c) What does the Gustafson-Barsis formula say about the maximum performance improvement of a parallel application?

Solution: Gustafson makes the observation that in practice a larger multiprocessor usually allows a larger-size problem to be undertaken in a reasonable amount of time. Hence, the size of the problem is selected based on the number of available processors (and memory, etc.), while keeping the (desired) parallel execution time fixed. This assumes that the serial section of the code is fixed and does not increase with the size of the problem.

(d) Suppose we have a program which has a serial portion of 5%. What is the maximum possible speedup when running this program on 20 processors according to Gustafson? What is the speedup according to Amdahl? What definitions of the word "speedup" are used here? What are the assumptions that each of these approximations make?

Solution:

- Gustafson: $S_{20} = 1 + (20 - 1) * 0.95 = 19.05$
- Amdahl: $S_{20} = 1/(0.05 + 0.95/20) = 10.26$

Note that the slides use the notation S_P for the speedup; for $P = 20$, this is S_{20}.

For both formulas, the assumption is that the work can be split into a completely serial portion and a completely parallel portion. Gustafson assumes that this serial portion does *not* scale with the problem size, whereas Amdahl assumes that it does.

In Gustafson's case, we take the maximum problem size that we can solve using 20 processors within a given time budget, and we observe that the (constant) serial portion is then 5% of the total runtime. The computed S_{20} is the speedup as compared to running this same problem size on a single processor. This is called *weak scaling*, or *scaled speedup*.

Amdahl's formula is, by its assumptions, not dependent on the chosen problem size (because we assume that both the serial and the parallel component scale with the problem size). Its S_{20} is the speedup we get when we run any particular problem size on 20 processors, as compared to running that same problem on one processor. This is the *relative speedup*.

The *absolute speedup* would be computing the Amdahl formula, but comparing to a good sequential algorithm instead of our parallel algorithm executed on one processor.

4. Methods of parallelisation.
   (a) What is task parallelism? Give an example.

   Solution: Task parallelism considers breaking a problem into separate tasks, and distributing these tasks over the available processors. The tasks may individually communicate and synchronise with each other. An example would be pipeline parallelism, which consists of a single set of data moving through a series of separate tasks that can each execute independently of the others.

   (b) What is data parallelism? Give an example.

   Solution: Data parallelism considers breaking up the input data into separate chunks and executing the *same* operation on each of the chunks in parallel. An example would be, say we want to sum all of the values along each row of a matrix; we can split the input data to have a different processor to each sum the values along a different row of the matrix in parallel.

   (c) What is the difference between task parallelism and data parallelism?

   Solution:

   - In task parallelism, the program is broken into different pieces (tasks) which may be executed concurrently, and the operation of each task *may* be different.
   - In data parallelism, every thread executes the same task (function), applied to a different data element.

   (d) What is the difference between parallelism and concurrency? Give an example of a problem/computation which is parallel but not concurrent.

   Solution:

   - Concurrency is the ability for the computer to compute tasks out-of-order with respect to each other. A concurrent application can be executed on a single processor.
   - Parallelism means to execute multiple concurrent tasks at the same time, with the goal to reduce the overall running time of the program. Requires multiple processing elements.

5. Applications of parallelism.
   (a) Amdahl's law considers the execution time of a program as a combination of (a) time spent doing serial work and (b) time spent doing parallel work (and nothing in between). Explain what this implies for the maximum parallel scalability of a program.

Solution: The speedup bound is determined by the degree of sequential execution of the program, not the number of processors. Where f is the fraction of sequential work, the parallel speedup S on P processors is given as:

$$\lim_{P \to \infty} S_P \leq 1/f$$

(b) Can you think of an example of a (part of a) parallel program that does get faster when more processors are added, but not linearly (i.e. it does not parallelise perfectly)? In other words, can you think of a program for which the assumptions of Amdahl's formula are not satisfied?

Solution: A typical example is a parallel reduction. Consider computing the sum of a large array of numbers of length n. On a single processor this takes linear time ($O(n)$), but there is no parallel algorithm that scales so perfectly that with n processors we can compute the sum in time $O(1)$.

The standard parallel algorithm for this takes time $O(\log(n))$: a tree reduction. First, in parallel, add each adjacent pair of numbers, resulting in an array of length $\frac{n}{2}$. Then do this again in the shorter array, resulting in $\frac{n}{4}$ numbers. Repeat until you have 1 number left; this is the sum. This algorithm takes about $\log_2(n)$ steps. Because the steps themselves parallelise perfectly, given sufficiently many processors, each step is constant-time, resulting in a total parallel runtime of $O(\log(n))$.

(c) What does "embarrassingly parallel" mean? Give an example of a problem/pattern which is embarrassingly parallel.

Solution: The operation to be performed in parallel is completely independent and can be executed without cooperation with other threads. Example: `map`.

(d) What is the difference between throughput and latency? Consider throughput and latency both for an entire application, and for execution of instructions in a processor. Give examples of how parallelism can be used to address each of these concerns.

Solution: Throughput is the number of results computed per unit of time. For example, if your program needs to transform a list of objects, and each object's result is individually interesting, then you can improve the throughput of your program by transforming multiple objects in parallel. You can do this, for example, by using multiple threads (transforming one object per thread), or (if possible) by vectorising your computation and doing each step of the transformation on multiple objects at once.

Latency is how long it takes to compute a single result. In the same example where you are transforming a list of objects, latency refers to how long it takes to transform a *single* object. To improve latency of this computation, transforming multiple objects in parallel does not help: at worst you may *increase* latency because your processor gets hotter, or you saturate your cores. You will need to find, and exploit, parallelism *inside* the transformation of a single object. This can be either using threads or using e.g. SIMD.

For example, think of a build system. If your goal is to compile a large project (with many files) quickly, then computing one file per processor core works very well: this is easy parallelisation, and what you really care about is just the number of files compiled per second. However, if you are working on a single source file and recompiling each time to test your changes, you care most about the time it takes to compile that single file (assuming that that's enough to test the resulting application). In this case, the compiler itself (working on one file) will need to be

parallelised to speed things up (i.e. improve the compilation latency). This is often possible, but much harder than just running the (single-file) compiler multiple times in parallel.

On the level of a processor, throughput and latency are also relevant but look slightly differently. Here, throughput is the number of *instructions* executed per second; this can be improved using pipelining and super-scalar execution. While these hardware improvements improve throughput and thus make your code run faster, they generally increase the time for a *single* instruction to run to completion: the hardware is more complex and has to go through more (pipelined) steps to complete one instruction.

(e) How does a "divide-and-conquer" algorithm work? Give an example of an algorithm which can be implemented in this way.

Solution: Recursively break down a problem into smaller sub-problems, each of which can be executed in parallel, until the problem becomes simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. Examples: quicksort, mergesort.

6. Hardware design of CPUs and GPUs.

(a) In what ways do the designs of CPU and GPU hardware differ?

Solution: CPUs are designed to be general purpose processors. They excel at executing a few serial threads. A significant fraction of the CPU transistors are dedicated to non-computational tasks, in particular instruction scheduling and caches.

GPUs are more specialised processors, and excel at executing many (data-)parallel threads. They are optimised for highly scalable parallel execution and have high-bandwidth parallel memories.

(b) What are the performance considerations relating from the differences in CPU and GPU hardware?

Solution: CPUs are optimised for fast sequential execution and have low-latency cached memories. In contrast, the execution speed of an individual thread on the GPU is much lower, with high latency for each individual instruction. GPUs hide this latency by allowing parallel execution of huge numbers of threads.

(c) What kinds of computations are best suited for the CPU?

Solution: Serial computations with irregular control flow.

(d) What kinds of computations are best suited for the GPU?

Solution: Data-parallel computations with regular (non-branching) control flow.

7. We discussed GPU hardware and its programming model (CUDA, OpenCL, etc.)

(a) How does the GPU programming model map to the underlying hardware?

Solution: The core idea of the GPU programming model is the kernel function: a single function that is executed by every thread on the device in data-parallel. The model scales as

we can execute as many threads as we like with the same kernel function over the available processing cores.

The kernel threads are arranged in thread blocks. The thread blocks are individually assigned to the compute units (streaming multiprocessors) of the GPU for execution. Once scheduled onto a compute unit a work group is executed to completion on that compute unit (logically) without preemption (i.e. work groups do not migrate and are not de-scheduled). Threads are mapped onto the underlying SIMD hardware in logical SIMD groups called warps; all threads in the warp execute a common instruction at a time. Full efficiency is reached when all threads in the warp agree on which instruction to execute.

(b) Give an example of how code in a sequential loop could be reimplemented to run on the GPU.

Solution: An example would be replacing a sequential loop in the `map` pattern; the body of the loop then becomes the body of the kernel function.

(c) What are the consequences of having deeply nested conditionals in kernel code? Compare the case of (nested) `if` statements and a `while` look, such as that which can be used to compute elements of the Mandelbrot set.

Solution: Since work items executed in SIMD groups (warps), when all threads in the warp do not agree on which instruction to execute next (e.g. some threads take the true branch of a conditional while others take the false branch), only the SIMD lanes executing the current instruction will be active, while the other threads will be disabled (predicated/masked execution). Once the threads in the true branch complete, the execution mask flips and the remaining threads execute the false branch while the others are disabled. Deeply nested conditionals with divergent branching could result in all threads taking different pathways resulting in execution of threads in the warp being sequentialised.

In the case of a `while` loop, threads which have finished the loop will be disabled and must wait for the remaining threads of the SIMD group to also complete (so, the hardware will similarly not be fully utilised). Once all threads in the warp complete the loop, the will continue execution as a group; after the loop there is no need for threads to execute different (divergent) code paths.

(d) What methods exist for communication and synchronisation between GPU threads? What are the restrictions on these and why is this?

Solution: Only threads within the same thread block can communicate and synchronise. Communication (using Cuda as an example) is via `__shared__` memory (several special warp shuffle instructions exist on NVIDIA devices as well). Synchronisation is via the `__syncthreads` operation. Every thread must execute the barrier, and all memory transactions to local and global memory respectively will become visible before threads proceed.

(e) What is thread occupancy and when is it an important consideration?

Solution: Thread occupancy is the number of threads which can be scheduled onto a streaming multiprocessor given the resource usage of the kernel, compared to the maximum number of simultaneous threads that compute unit supports. Increasing occupancy is important for memory-bound kernels, as GPUs use parallelism to hide latency; whenever the threads of a warp are stalled, instructions from a different warp will be executed. The warp context switch has zero overhead.

8. The canonical implementation of a spin-lock has the following form:

```
do {
  old = atomic_exchange(&lock[i], 1);
} while (old == 1);

/* critical section */

atomic_exchange(&lock[i], 0);
```

Here `lock` is an array of integers where a value of zero at index `i` indicates that the element at that index is unlocked, and a value of one means that another thread currently has the lock on that element. The initial loop repeatedly attempts to take the lock at index `i` by writing `1` into the slot at that index. Once the `old` state of the lock returns `0`, this thread has acquired the lock and continue to the critical section, after which it releases the lock by writing `0` back into the slot.

Will this code execute correctly when run on the GPU? Motivate your response.

Solution: No. Because the threads of a work unit are mapped in groups onto the underlying SIMD hardware, several threads will be executing in lockstep, with predicated execution used to handle divergent control flow. In the above code, once a thread acquires its lock, it will be disabled and stop participating in the loop, waiting for all other threads to acquire their locks before continuing program execution. If two threads in the same warp attempt to acquire the same lock, then once the lock is acquired by one thread, that thread will sit idle at the end of the loop, waiting for the second thread to acquire the lock it currently holds, but never able to proceed to execute the critical section and release the lock. The threads of the warp are deadlocked.

To prevent this situation, we must invert the algorithm so that threads always make progress, until every thread in the warp has committed their result:

```
done = 0;
do {
  if ( atomic_exchange(&lock[i], 1) == 0 ) {

    /* critical section */

    done = 1;
    atomic_exchange(&lock[i], 0);
  }
} while (done == 0);
```

9. What is the difference between correlation and causation?

Solution: This question is really asking whether or not you see why the term "Moore's Law" is really not quite accurate. This term is used to describe the observation that the number of transistors in an integrated circuit doubles roughly every two years (correlation), but there is no fundamental law of nature that says "more time makes more transistors for integrated circuits" (causation); saying this out loud makes it clear that this is nonsense, but for some reason the word *law* is still used for something that just a (temporary!) correlation.