Note 22: Amdahl's Law, Thread-level Parallelism, OpenMP

Kevin Moy

Abstract

We want to increase CPU performance- but adding more and more transistors, as we covered last note, won't quite do it. We know domain-specific hardware or parallelism were some viable alternatives. Specifically, we will talk about parallel threads (processes) in a CPU in this note.

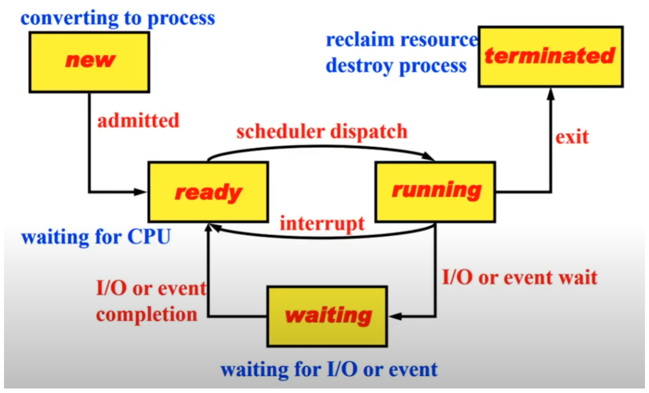
1 Multi-processing

Multi-processing enters the MIMD stage of Flynn's taxonomy: multiple instructions streams, multiple data streams.

The goal is to increase performance with parallel tasks. We can either use multiple cores to run these tasks in parallel, OR run multiple tasks on a SINGLE core- concurrently.

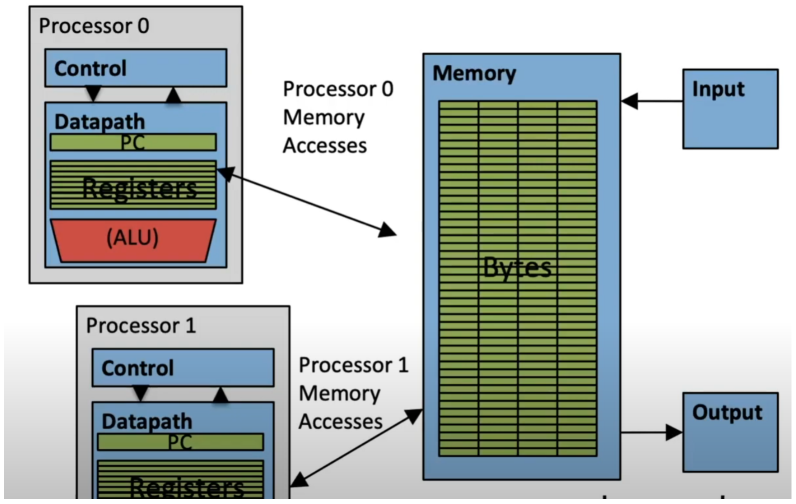
Let's first talk about multiple cores. Remember a **program** is a compiled, assembled, and linked piece of code in executable format ready to be loaded. On the other hand, a **process** is a program that's actually currently running (on a "processor"). This process has its own PC and memory space and state.

There is a corresponding *process state diagram* to map the possible states of a process. A generic diagram is shown below:



So far, we've covered **single-core systems**: one processor that communicates with memory. This memory also interacts with IO devices through an IO-memory interface.

Now if we ADD more processors- exact carbon copies of each other- we can now run multiple tasks at once! Specifically, a **multiprocessor system** is a computer with multiple processors/cores. Each core has its own independent PC and registers, and executes independent instruction (streams). Additionally, processors share the same system memory. Communication between processors occurs between loads and stores to some agreed-upon location in memory.



Remember we are now considering **task-level parallelism**: the goal of this, then, is to increase **throughput** rather than clock speed.

2 Multi-Threading

Let's now talk about the second solution: **multithreading**, or running multiple tasks on a single core concurrently.

A "software task" is called a **thread**: a (smallest) unit of execution within a process. A single process could actually have multiple threads- doing multiple tasks inside that process. Each thread must contain some independent memory or data: separate PC, register, and stack memory!

An example of threads: let's say we are implementing Chrome such that every tab opens up a new thread. Since the same code loads each website, all threads should, of course, share the code segment of memory. Each tab also shares the static memory segment: i.e. global settings. However, since each tab has independent and separate state, we want our own stack and registers for each thread now.

Processor resources are expensive and should not be idle. For example, we have high memory latency on a cache miss, since we have to travel to main memory now. However, instead of waiting doing nothing, we can **switch threads** such that we are doing some other work while waiting for the data to be retrieved. There is one thing we must make sure of: the cost of switch MUST be less than the cost of cache miss latency- i.e. we don't waste more time context switching than we would by just waiting.

For multithreading, instead of having a *single* PC and registers, we now have MULTIPLE sets of PCs and registers inside our processor- all independent of each other. From the perspective of software, it looks like multiple processors. The control logic actually decides which thread to execute instruction(s) from next.

Let's compare multithreading to multi-core. Multithreading is better for utilization: around 5 percent more hardware for about 1.3x better performance. A multicore system really just has a bunch of processor *copies*: requires much more hardware, but around 2x better performance. The memory components that multicore systems share differ as well. Modern machines have multiple cores with multiple threads per core.

3 Ahmdahl's Law

How much speedup can we really get from parallelism? How hard is it to write parallel-processing programs?

Let's say we had a 100-s long program. Let's say part A takes 95s and part B takes 5s to run. In general, speedup with improvement = runtime without improvement / runtime with improvement- which should give you some number between 1 and 2. So if we speed up part B to 2.5s, say, our performance improvement would be 100s/97.5s = 1.026 or about a 2.6 percent increase. Performance improvements are really only useful if they are on part A in this case- the **important** parts of the program that take up the most time!

Amdahl's Law is a heartbreaking law:

$$\text{Speedup} = \frac{1}{(1-F) + \frac{F}{S}}$$

This law tells us limitations of speedup on parts of a program. F represents the fraction of execution time sped up, and S represents the scale of improvement. $1 - F$ represents the portion of the code NOT parallelizable (cannot improve), F represents the portion of the code that is parallelizable. For example, for a 2x improvement to 25 percent of code, we simply set parameters F = 0.75 (code NOT sped up), and S = 2 to give a 1.14x speedup.

The amount of speedup that can be achieved with parallelism is actually limited by your *non* parallel code.

To achieve a linear speedup with more processors, **none** of the original computation can be non-parallelizable (scalar)- i.e. everything must be parallelizable. Thus to get a speedup of 90x from 100 processors, we'd have to have less than 0.1 percent of our program be scalar!

4 Data Races

Now we focus on actually writing parallel programs.

Remember that thread scheduling is non-deterministic: no guarantees one thread executes over another at any point in time! If different threads write to the same variable, this is called a **data race**.

Let's look at an example with sum reduction. SAy we want to sum 10000 numbers in array `A` using 10 processors. Each processor has processor ID `Pn` from 0 to 9. We ideally would give each processor 1000 numbers to sum. The first step would be to calculate these partial sums for each processor:

```
sum[Pn] = 0
for (i = 1000*Pn; i < 1000*(Pn+1); i++)
    sum[Pn] = sum[Pn] + A[i]
```

This is ok: we don't really have any *data dependencies* and we also don't write to any shared memory addresses.

The next step, we actually want to utilize parallelism to add these partial sums, so we'll do just that. Half the processors will add pairs, then a quarter of them, until finally we are left with one processor which computes our final sum. We just add P0 to P5, P1 to P6... so on until now we only need to utilize 5 processors. Then we want to cut to two processors: P0 calculates P0 + P2 + P4, and P1 calculates P1 + P3. Finally, P0 calculates P0+P1 as our final sum.

Below is the pseudocode for this sum reduction:

```
half = 10 //number of processors/elmts to sum and halve again.
repeat
    synch();
    //Handle odd elements
    half = half/2
    if (Pn < half)
        sum[Pn] = sum[Pn + half]
until (half == 1);
```

5 Synchronization

We can always avoid data races by **synchronizing** reads and writes to get deterministic behavior.

In comes the **lock**. The lock grants a thread access to a region (**critical section**) of code so that ONLY THAT THREAD can operate on it at a time. We need all processors to be able to check the lock- so we use a shared memory location for this lock. Processors will either read lock and wait OR set the lock (if it isn't there) and go into critical section. Traditionally, 0 is unlocked, 1 is locked.

Below is some pseudocode for how lock synchronization works:

```
Check lock
Set Lock (if lock not set)
Critical Section (change shared variables)
Unset lock
```

Let's consider the above process with multiple threads. Say two threads had this process going, and both their lock unset checks happen at around the same time:

```
while (lock != 0);
lock = 1
//Critical Section
lock = 0
```

So we could have TWO threads running in the critical section at the same time. What's worse, once the first thread finishes it unsets the lock, allowing ANOTHER thread to enter the critical section while the second thread is still in it! Really bad.

What if we had another lock to try and prevent this?

```
while (locklock != 0); //Check if locklock free
locklock = 1;
    //wait for lock release
    while (lock != 0);
    //lock == 0 (unlocked)

    //set lock
    lock = 1;
```

```
locklock = 0;

    //access shared resource
lock = 0;
```

Basically, `locklock` locks our ability to even get the lock. But this won't work. The LOCK(LOCK) ITSELF is shared memory, and as long as that's the issue, there is always going to be an issue with synchronization.

However, there might be a solution with **hardware synchronization**. Two possible solutions via upgrading hardware:

1. Atomic read/write (both in one clock cycle)
2. Reads/writes in a SINGLE instruction- with nothing allowed in between.

There are a couple of ways to implement this. The first is a swap of registers in memory- reading and writing in a single operation. We can also have a linked read-write (pair of instructions).

There are several RISC-V atomic operations (AMOs) that perform both an operation on an operands in memory (write) AND set `rd` to the original memory value (read). They are all a sort of modified R-type instruction.

Let's take a look at `amoadd.w rd rs2 (rs1)`:

```
amoadd.w rd rs2 (rs1)
temp = M[R[rs1]]
M[R[rs1]] = temp + R[rs2]
```

Basically `rd` is where we read our data from memory into, `rs2` is the data we are storing INTO memory, and then we add the value from rs1 into rs2 and store THAT into the address specified by `rs1`.

Remember that atomic instructions include *memory ordering*. With an atomic CPU, we might have out-of-order reads and writes to memory. `aq` is acquire lock, and `rl` is releasing lock. We cannot execute ANY memory accesses until an `aq` instruction completes, and we must FINISH ALL memory accesses BEFORE an `rl` instruction BEGINS.

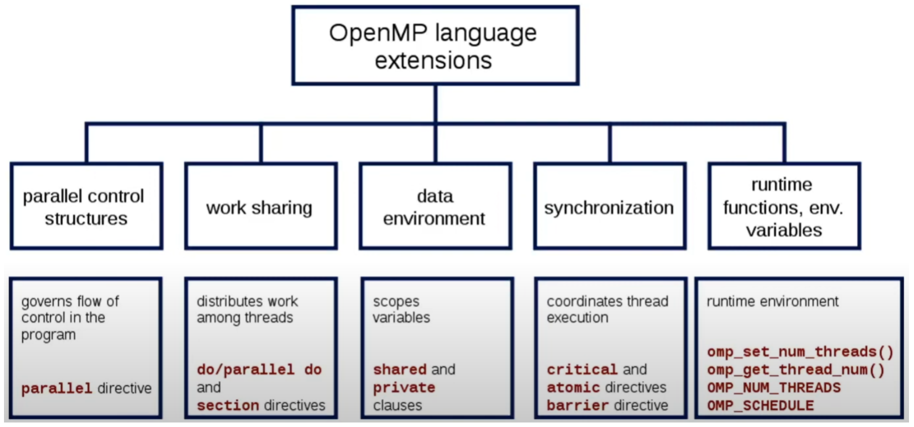
Let's look at now the fixed hardware synchronization code that ensured that only one thread could possibly enter the critical section at a time (lock at `a0`):

```
addi t0 x0 1
Try:
    amoswap.w.aq t1 t0 (a0)
    bnez t1 Try
Locked:
    #Do stuff in critical section
Unlock:
    amoswap.w.rl x0 x0 (a0)
```

6 OpenMP

OpenMP is an API used for multi-threaded, shared memory parallelism applications. They consist of compiler directives (instructions for the compiler), runtime library routines, environment variables.

OpenMP is a *specification*, and consists of multiple different parts, each which contain helpful coding language extensions:


OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives
barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

OpenMP is a **shared memory model** with explicit thread-based parallelism. There are multiple threads in a shared memory environment. It is an explicit programming model that gives the programmer full control over parallelization.

Some pros:

- Take advantage of shared memory- programmer doesn't need to worry (that much) about where data is in memory.
- Compiler directives simple and easy usage
- Portability: legacy serial code does not need to be rewritten

Some cons:

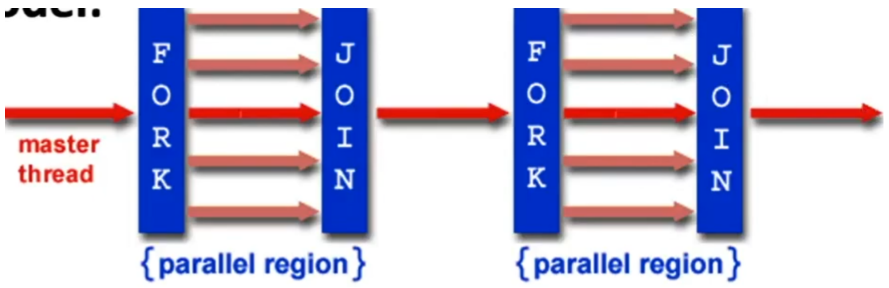
- Code can only be ran on a **shared memory environment**.
- Compiler must support OpenMP explicitly

OpenMP is actually just an extension of C- we just need `#include <omp.h>` and remember to compile with the `-fopenmp` flag. The usefulness? We can now specify certain blocks of our C code to be parallel!

Some of the key ideas of OpenMP are variable scope (shared vs. private variables) as well as parallelization, work sharing, and synchronization directives.

6.1 OpenMP Programming Model

OpenMP runs on a **fork-join model**: openMP programs begin as a single process (master thread) and execute sequentially (normally) until it hits the first (explicitly specified) parallel region of code. Then, the program **forks**: it splits our single thread into multiple threads and executes. Once ALL threads are finished, we **join** them back (via synchronization and termination), leaving only the single master thread running like normal. Below is a diagram of how this works:



The fork-join model meshes well with `for` loops!

6.2 Pragmas

C allows usage of **pragmas**: preprocessor mechanisms for language extensions. Compilers that don't understand what a pragma is will simply ignore them and just run code sequentially.

Below is the basic OpenMP construct for parallelization:

```
#pragma omp parallel
{ //Curly brace MUST be on separate line than pragma!
    //Parallel code.
}
```

Each thread that runs will then run a copy of the parallel code block. Remember that thread scheduling is non-deterministic: we have no idea when a thread executes or finishes (before another).

Variables declared OUTSIDE of a pragma are **shared variables**. To declare them private, we must declare this with a pragma as well:

`#pragma omp parallel private (x)`

6.3 Thread Creation

The MAXIMUM number of threads we can create is defined by `OMP_NUM_THREADS`. Usually this number is the same as the number of processor cores- however, we can specify more. In this case, the OS must *multiplex* these threads onto the CPU.

There are also a few useful OpenMP intrinsics:

- `omp_set_num_threads(x)`: set x threads.
- `omp_get_num_threads()`: get number of threads.
- `omp_get_thread_num()`: get thread ID.

Let's look at an example of some OpenMP code:

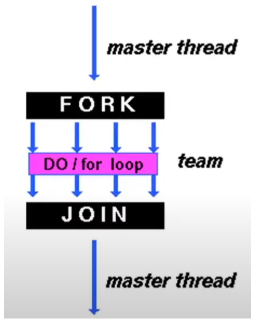
```c
#include <stdio.h>
#include <omp.h>
int main(){
    int nthreads, tid;
    /* Fork threads with private thread id (tid) */
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num(); //get thread ID
        printf("Hello world from thread = %d\n", tid);
        //Only for MASTER thread, with id 0
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }
    /* Here, all threads join master and terminate. */
}
```

Note variables `nthreads` and `tid` are declared OUTSIDE of the `pragma`, so they are shared across all threads. However, since we declared `tid` private, it now becomes like an instance variable!

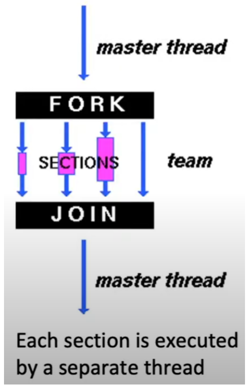
7 OpenMP Work Sharing

Work sharing divides execution of a code block among different threads.

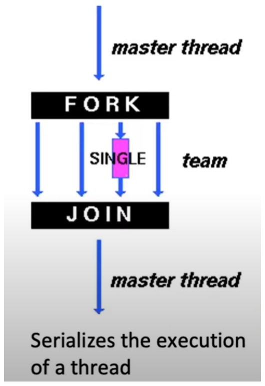
With a `for` loop, iterations of the for loop can be shared across different threads:



Another model separates sections of code block to be executed by separate threads (see prev. example code with master thread printing number of threads):



Our final model *serializes* code block to execute in a single thread:



Normally with for loops in a parallel block, we have:

```
#pragma omp parallel
{
    #pragma omp for //only directive allowed in parallel code
    for (i=0;i<len;i++){...}
}
```

We can actually use `#pragma omp parallel for` as a shorthand:

```
#pragma omp parallel for
for (i=0; i<len; i++){...}
```

So how does OMP actually parallelize a for loop? It will break it up into chunks, into a separate thread. For example, if `len` = 100 with 2 threads, thread 0 will handle iterations 0-49, and thread 1 will handle iterations 50-99.

There are some requirements, however. The for loop ”shape” must be simple enough for an openMP compiler to parallelize it and assign appropriate amounts to threads. Another requirement is NO PREMATURE EXITS (no `break,return,exit,goto`). Also no `do-while` loops. In general, we should NEVER (prematurely) jump outside a `pragma` block- we need to make sure the fork-join process is completed.

Also note that loop index is actually a private variable as well: we certainly don't want one thread's changing of loop index to affect another thread's usage of that index.

We also want to divide contiguous iterations per thread. Why? Remember that each thread has separate caches (since they are on separate cores). We can run into a problem called **false sharing**- not writing to the same location in memory, invalidating other thread caches.

Finally, remember there is an *implicit synchronization* at the end of the for loop- the join.

8 Summary

Parallel programming improves performance. We can run multiple threads on multiple cores, or multiple threads on a single core. We need to be careful of data races (reads/writes to the same address fucking up because of timing): but fortunately there are some synchronization techniques (atomic instructions, locking) that can help. We know Ahmdal's law measures speedup and indicates the maximal overall speedup we can actually have by just speeding up a portion of our code. Finally, we discuss OMP: a C extension that allows for parallel programming.