CS 61C
Fall 2023

Parallelism

Discussion 10

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 SIMD is ideal for flow-control heavy tasks (i.e. tasks with many branches/if statements).

False. Data-level parallelism really shines through when we need to repeatedly perform the same operation on a large amount of data. Flow control statements disrupt the continuous flow of computation, which makes programs with them hard to take advantage of SIMD.

1.2 Intel's SIMD intrinsic instructions invoke large registers available on the architecture in order to perform one operation on multiple values at once.

True. For example, we can pack four 32-bit integers in a single 128-bit register and perform the same arithmetic operation on all four integers in one go, using an instruction such as `__m128i _mm_add_epi32(__m128i a, __m128i b)`.

1.3 The pipelined datapath is an example of parallelism because it performs different stages of instructions in parallel.

True. While a pipelined datapath doesn't execute multiple instructions at the same time, it makes use of each part of the processor at the same time with different instructions, implementing instruction-level parallelism. This can be contrasted with data-level parallelism, which takes advantage of larger registers to do simultaneous memory accesses, and thread-level parallelism, which forks into multiple parallel threads and joins the tasks together.

1.4 The most effective way of increasing performance on a modern PC is to increase its clock speed.

False. Modern clock speeds have almost reached their physical limits, and so there's not much room to improve our performance with faster clock speeds. To improve performance, the current best way is to parallelize onto multiple cores (thread-level parallelism).

1.5 In thread-level parallelism, the amount of speedup is directly proportional to the increase in number of cores.

False, usually there is some overhead in parallelizing an operation. Additionally, Amdahl's Law shows that true speedup is affected not only by the number of threads but also by the amount of code that cannot be sped up.

1.6 In thread-level parallelism, threads may run in any order and can start while other threads are partway through their execution.

True. We must ensure that whichever order the threads execute in, the behavior of the program is correct, which includes handling any potential data races.

2 Write-Back Caches

When it comes to writing data to cache memory, there are multiple write policies to consider that offer different options when building our system. Some of them you might encounter are:

1. **Write-through**: In this policy, when we have a write we write to both the cache and the memory. This is the case for every write, so the main memory always has the updated data. This is simple to implement, but writing to main memory every single time is slow.

2. **Write-back**: On a write, the data is only updated/written in the cache. The main memory only receives the data upon eviction. This means the cache has more up to date data most of the time. While this is faster as there is less accesses to main memory, it is harder to implement as we have to include more overhead, such as dirty bits and so on.

2.1 Considering the above information, lets consider a direct mapped cache with a capacity of 16B and a block size of 4B. Lets also assume that the memory addresses are 8 bits each. Assuming the cache is completely empty in the beginning, we make memory accesses to the following locations:

- 0x6A, Write
- 0x0F, Write
- 0x38, Read
- 0x6B, Read
- 0x81, Read
- 0x87, Write
- 0x68, Write
- 0x6B, Read

Fill out the metadata of the cache if we use a write-back policy.

Index	Valid	Dirty	Tag
0			
1			
2			
3			

Short Answer:

```
Index: 0    Valid: 1    Dirty: 0    Tag: 0x8
Index: 1    Valid: 1    Dirty: 1    Tag: 0x8
Index: 2    Valid: 1    Dirty: 1    Tag: 0x6
Index: 3    Valid: 1    Dirty: 1    Tag: 0x0
```

Long Answer:

- 0x6A: 0110 1010 Tag: 0x6, Index: 2 Offset: 2, Valid Bit: 1, Dirty Bit: 1 The dirty bit is a 1 because we do a write.

- 0x0F: 0000 1111 Tag: 0x0, Index: 3, Offset: 3, Valid Bit: 1, Dirty Bit: 1 The dirty bit is a 1 because we do a write.

- 0x38: 0011 1000 Tag: 0x3, Index: 2, Offset: 0, Valid Bit: 1, Dirty Bit: 0 For this memory access, we will evict the block that is at index 2 and bring in a new block into our cache (block with tag = 0x3, index = 2). The block that is getting evicted has dirty bit = 1 (data in that cache block is updated and correct, but the corresponding contents in main memory are not). Therefore, when we evict the block (of tag = 0x6, index = 2) from the cache, we update the corresponding block (addresses with tag 0x6, index 2) in memory. When we load the cache with the new block (tag = 0x3, index = 2), our dirty bit will initially be set to 0 since we're doing a read.
- 0x6B: 0110 1011 Tag: 0x6, Index: 2, Offset: 3, Valid Bit: 1, Dirty Bit: 0 For this memory access, we will evict the block that is at index 2 and bring in a new block into our cache.
- 0x81: 1000 0001 Tag: 0x8, Index: 0, Offset: 1, Valid Bit: 1, Dirty Bit: 0
- 0x87: 1000 0111 Tag: 0x8, Index: 1, Offset: 3, Valid Bit: 1, Dirty Bit: 1
- 0x68: 0110 1000 Tag: 0x6, Index: 2, Offset: 0, Valid Bit: 1, Dirty Bit: 1 This is a cache hit, but because we do a write, we will update our dirty bit to be a 1.
- 0x6B: 0110 1011 Tag: 0x6, Index: 2, Offset: 3, Valid Bit: 1, Dirty Bit: 1

2.2 How many times will we write to main memory in this case?

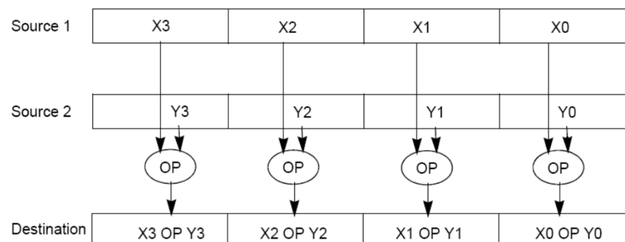
1 time. For the write-back policy, we only write to main memory when a cache block gets evicted and its dirty bit is set to 1. In this case, when we access memory address 0x38 (has index = 2), we will evict the current block at index 2, which has tag = 0x6. Since the dirty bit for this block (tag = 0x6) was 1, we write to memory the updated data when evicting.

2.3 Now suppose we had a write-through policy. How many times will we write to main memory in this case?

4 times. For the write-through policy, every time we do a write, we will write to both the cache and main memory.

3 Data-Level Parallelism

The idea central to data level parallelism is vectorized calculation: applying operations to multiple items (which are part of a single vector) at the same time.



Some machines with x86 architectures have special, wider registers, that can hold 128, 256, or even 512 bits. Intel intrinsics (Intel proprietary technology) allow us to

use these wider registers to harness the power of DLP in C code.

Below is a small selection of the available Intel intrinsic instructions. All of them perform operations using 128-bit registers. The type __m128i is used when these registers hold 4 ints, 8 shorts or 16 chars; __m128d is used for 2 double precision floats, and __m128 is used for 4 single precision floats. Where you see "epiXX", epi stands for **e**xtended **p**acked **i**nteger, and XX is the number of bits in the integer. "epi32" for example indicates that we are treating the 128-bit register as a pack of 4 32-bit integers.

- `__m128i _mm_set1_epi32(int i)`:
  Set the four signed 32-bit integers within the vector to `i`.
- `__m128i _mm_loadu_si128( __m128i *p)`:
  Load the 4 successive ints pointed to by `p` into a 128-bit vector.
- `__m128i _mm_mullo_epi32(__m128i a, __m128i b)`:
  Return vector $(a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, a_3 \cdot b_3)$.
- `__m128i _mm_add_epi32(__m128i a, __m128i b)`:
  Return vector $(a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- `void _mm_storeu_si128( __m128i *p, __m128i a)`:
  Store 128-bit vector `a` at pointer `p`.
- `__m128i _mm_and_si128(__m128i a, __m128i b)`:
  Perform a bitwise AND of 128 bits in a and b, and return the result.
- `__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)`:
  The ith element of the return vector will be set to 0xFFFFFFFF if the ith elements of a and b are equal, otherwise it'll be set to 0.

3.1 SIMD-ize the following function, which returns the product of all of the elements in an array.

```
static int product_naive(int n, int *a) {
    int product = 1;
    for (int i = 0; i < n; i++) {
        product *= a[i];
    }
    return product;
}
```

Things to think about: When iterating through a loop and grabbing elements 4 at a time, how should we update our index for the next iteration? What if our array has a length that isn't a multiple of 4? What can we do to handle this tail case?

```
static int product_vectorized(int n, int *a) {
    int result[4];
    __m128i prod_v = __mm_set1_epi32(1);
    for (int i = 0; i < n/4 * 4; i += 4) { // Vectorized loop
        prod_v = __mm_mullo_epi32(prod_v, __mm_loadu_si128((__m128i *) (a + i)));
    }
    _mm_storeu_si128((__m128i *) result, prod_v);
    for (int i = n/4 * 4; i < n; i++) { // Handle tail case
        result[0] *= a[i];
    }
    return result[0] * result[1] * result[2] * result[3];
}
```

4 Thread-Level Parallelism

OpenMP provides an easy interface for using multithreading within C programs. Some examples of OpenMP directives:

- The `parallel` directive indicates that each thread should run a copy of the code within the block. If a for loop is put within the block, **every** thread will run every iteration of the for loop.

  ```
  #pragma omp parallel
  {
      ...
  }
  ```

  NOTE: The opening curly brace needs to be on a newline or else there will be a compile-time error!

- The `parallel for` directive will split up iterations of a for loop over various threads. Every thread will run **different** iterations of the for loop. The exact order of execution across all threads, as well as the number of iterations each thread performs, are both non-deterministic, as the OpenMP library load balances threads for performance. The following two code snippets are equivalent.

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

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

There are two functions you can call that may be useful to you:

- `int omp_get_thread_num()` will return the number of the thread executing the code
- `int omp_get_num_threads()` will return the number of total hardware threads executing the code

4.1 For each question below, state and justify whether the program is **sometimes incorrect**, **always incorrect**, **slower than serial**, **faster than serial**, or **none of the above**. Assume the number of threads can be any integer greater than 1. Assume `arr` is an `int[]` of length `n`.

(a)
```
// Set element i of arr to i
#pragma omp parallel
{
    for (int i = 0; i < n; i++)
        arr[i] = i;
}
```

Slower than serial: There is no `for` directive, so every thread executes this loop in its entirety. Without considering overhead, `n` threads running `n` loops at the same time will actually execute in the same time as 1 thread running 1 loop.

The values should all be correct at the end of the loop since each thread is writing the same values. Furthermore, the existence of parallel overhead due to the extra number of threads, along with coherence misses, will slow down the execution time.

(b)
```
// Set arr to be an array of Fibonacci numbers.
arr[0] = 0;
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < n; i++)
    arr[i] = arr[i-1] + arr[i - 2];
```

Sometimes incorrect: There are dependencies between threads. Suppose $n = 8$, there are 2 threads, and thread 0 is assigned to execute the loop for $i = 2$ to 4, and thread 1 is assigned to execute the loop for $i = 5$ to 7. At $i = 5$, thread 1 requires the values of arr[4] and arr[3] to correctly compute arr[5]. However, arr[4] and arr[3] may not have been computed yet by thread 0. The program can be correct in a interweaved scheme where the threads take turns completing each iteration in sequential order. Each thread will have the correctly updated shared `arr` to compute the next Fibonacci number. Note that this scheme would still be slower than serial due to the amount of overhead required as the threads need to wait for each other's execution to finish as well as deal with coherency issues regarding the shared data.

(c)
```
// Set all elements in arr to 0;
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
    arr[i] = i;
```

Faster than serial: The **`for`** directive automatically makes loop variables (such as the index) private, so this will work properly. The **`for`** directive splits up the iterations of the loop to optimize for efficiency, and there will be no data races.

(d)
```
// Set element i of arr to i;
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
    *arr = i;
    arr++;
```

Sometimes incorrect: There is a data race to increment the array pointer (which is shared). To better visualize an execution order of threads that will result in incorrect behavior, it is helpful to convert the C code to assembly. Assume that two atomic instructions cannot be executed at the same time. Assume:

- s1 contains the value of arr
- t0 contains the value of n
- t1 contains the value of i

```
1 loop : beq t1 t0 exit
2 sw t1 0(s1)
3 addi t1 t1 4
4 addi s1 s1 4
5 exit :
```

Suppose n = 2, and there are two threads. Thread 0 is assigned to execute the loop with i = 0, and thread 1 is assigned to execute the loop with i = 1.

Correct final contents of arr: arr[0] = 0, arr[1] = 1

Execution order that would result in incorrect behavior:

```
1 sw t1 0(s1)  (thread 0) // the private variable i of thread 0 = 0, and arr[0] is set = 0
2 sw t1 0(s1) (thread 1) // the private variable i of thread 1 = 1, and arr[0] is overwritten and set =
     1
3 addi t1 t1 4 (thread 0) // the private variable i of thread 0 is incremented to 1
4 addi t1 t1 4 (thread 1) // the private variable i of thread 1 is incremented to 2
5 addi s1 s1 4 (thread 1) // the pointer is incremented to arr + 4
6 addi s1 s1 4 (thread 0) // the pointer is incremented to arr + 8
```

Not only is arr[0] incorrect, but arr[1] is never set to a value

Execution order that would result in correct behavior:

```
1 sw t1 0(s1) thread 0 // the private variable i of thread 0 = 0, and arr[0] is set = 0
2 addi t1 t1 4 (thread 0) // the private variable i of thread 0 is incremented to 1
3 addi s1 s1 4 (thread 0) // the pointer is incremented to arr + 4
4 sw t1 0(s1) (thread 1) // the private variable i of thread 1 = 1, and arr[1] is set = 1
5 addi t1 t1 4 (thread 1) // the private variable i of thread 1 is incremented to 2
6 addi s1 s1 4 (thread 1) // the pointer is incremented to arr + 8
```

This order will be slower than serial due to the amount of overhead required as the threads need to wait for each other's execution to finish as well as deal with coherency issues regarding the shared data.

5 Amdahl's Law

In attempting to parallelize a program, the overall performance speedup will always be limited by the fraction of the program that cannot be sped up. This overall speedup can be formulated by Amdahl's Law, which states that

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

Speedup refers to the theoretical speedup of the program compared to its naive implementation. Note that $\textbf{Speedup} > 1$ since we're making our program faster than the original.

F refers to the fraction of the program that can be optimized;

S is the speedup factor for how much that portion of the program can be optimized by, where $(\textbf{S} > 1)$

5.1 Derive Amdahl's Law using the ratio: **Speedup** $= t_{\text{naive}}/t_{\text{optimized}}$

First, we can split the overall time a program takes into the time it takes for the part of the program that can be optimized and the rest of it. Letting F represent the fraction that can be sped up, we have:

$t_{\text{naive}} = F(t_{\text{naive}}) + (1-F)t_{\text{naive}}$

Then, we can implement the optimization, known as the speedup factor S into our equation by dividing the optimizable portion to get:

$t_{\text{optimized}} = \frac{F(t_{\text{naive}})}{S} + (1-F)t_{\text{naive}}$

Solving for the ratio **Speedup** $= t_{\text{naive}}/t_{\text{optimized}}$ leads to Amdahl's Law.

5.2 Assuming we have infinite threads and resources, what would our overall speedup be for a program with some fraction of our code that can be parallelized F?

With infinite scaling factor S, our total speedup will approach $\frac{1}{1-F}$. However, in reality there would be some non-zero overhead that is required to properly split up work.

5.3 You write code that will search for the phrases "Hello Sean", "Hello Jon", "Hello Dan", "Hello Man", "Bora is the Best!" in text files. With some analysis, you determine you can speed up 40% of the execution by a factor of 2 when parallelizing your code. What is the true speedup?

Using Amdahl's Law with F=0.4, S=2:

$$\frac{1}{0.6 + \frac{0.4}{2}} = \frac{1}{0.8} = 1.25$$

5.4 You run a profiling program on a different program to find out what percent of time within the program each function takes. You get the following results:

Function	% Time
`f`	30%
`g`	10%
`h`	60%

(a) Assuming that each of these functions can be parallelized by the same speedup factor, which one, if parallelized, would cause the most speedup for the entire program?

`h`

(b) What speedup would you get if you parallelized just this function with 8 threads? Assume that work is distributed evenly across threads and there is no overhead for parallelization.

$1/(0.4 + 0.6/8) \approx 2.1$