CS 61C Summer 2023

Data- and Thread-Level Parallelism

Discussion 9

1 Precheck

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.

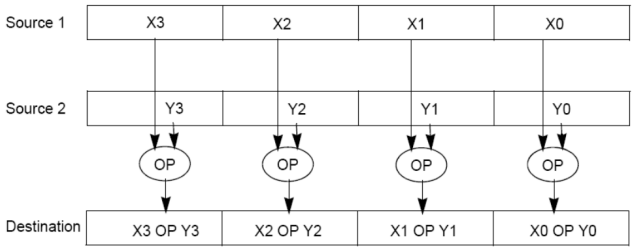
False, usually there is some overhead in parallelizing an operation. Additionally, Amdahl's Law shows that true speedup is affected not only by the number of threads but also by the amount of code that cannot be sped up.

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 SIMD

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.

2.1 You have an array of 32-bit integers and a 128-bit vector as follows:

```
1 int arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};
2 __m128i vector = _mm_loadu_si128((__m128i *) arr);
```

For each of the following tasks, fill in the correct arguments for each SIMD instruction, and where necessary, fill in the appropriate SIMD function. Assume they happen independently, i.e. the results of Part (a) do not at all affect Part (b).

(a) Multiply `vector` by itself, and set `vector` to the result.

```
vector = _mm_mullo_epi32(vector, vector);
```

(b) Add 1 to each of the first 4 elements of the `arr`, resulting in `arr = {2, 3, 4, 5, 5, 6, 7, 8}`

```
__m128i vector_ones = _mm_set1_epi32(1);
__m128i result = _mm_add_epi32(vector, vector_ones);
_mm_storeu_si128((__m128i *) arr, result);
```

(c) Add the second half of the array to the first half of the array, resulting in `arr = {1 + 5, 2 + 6, 3 + 7, 4 + 8, 5, 6, 7, 8} = {6, 8, 10, 12, 5, 6, 7, 8}`

```
__m128i result = _mm_add_epi32(_mm_loadu_si128((__m128i *) (arr + 4)), vector);
_mm_storeu_si128((__m128i*) arr, result);
```

(d) Set every element of the array that is not equal to 5 to 0, resulting in `arr = {0, 0, 0, 0, 5, 0, 0, 0}`. Remember that the first half of the array has already been loaded into vector.

```
__m128i fives = _mm_set1_epi32(5);
__m128i mask = _mm_cmpeq_epi32(vector, fives);
__m128i result = _mm_and_si128(mask, vector);
__mm_storeu_si128((__m128i *) arr, result);
vector = _mm_loadu_si128((__m128i *) (arr + 4));
mask = _mm_cmpeq_epi32(vector, fives);
result = _mm_and_si128(mask, vector);
_mm_storeu_si128((__m128i *) (arr + 4), result);
```

3 TLP

As powerful as data level parallelization is, it can be quite inflexible, as not all applications have data that can be vectorized. Multithreading, or running a single piece of software on multiple hardware threads, is much more powerful and versatile.

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

3.1 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 no thread will complete in its entirety before another thread starts executing. 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. `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 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: While the loop has dependencies from previous data, in a interweaved scheme where the threads take turns completing each iteration in sequential order (e.g.

```
1 for (int i = omp_get_thread_num(); i < n; i += omp_get_num_threads())
```

is the work allocation per thread and the order of execution is based on the shared variable `i` from 2 to `n`), 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] = 0;
```

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: Because we are not indexing into the array, there is a data race to increment the array pointer. If multiple threads are executed such that they all execute the first line, *arr = i; before the second line, arr++;, they will clobber each other's outputs by overwriting what the other threads wrote in the same position. However, taking a similar interweaved scheme as in 4.1b, there is an order that will not encounter data races, though it will be slower than serial.

3.2 Consider the following multithreaded code to compute the product over all elements of an array.

```
1  // Assume arr has length 8*n.
2  double fast_product(double *arr, int n) {
3      double product = 1;
4      #pragma omp parallel for
5      for (int i = 0; i < n; i++) {
6          double subproduct = arr[i*8]*arr[i*8+1]*arr[i*8+2]*arr[i*8+3]
7                          * arr[i*8+4]*arr[i*8+5]*arr[i*8+6]*arr[i*8+7]
8          product *= subproduct;
9      }
10     return product;
11 }
```

(a) What is wrong with this code?

The code has the shared variable product, which can cause data races when multiple threads access it simultaneously.

(b) Fix the code using **#pragma** omp critical. What line would you place the directive on to create that critical section?

```
1  double fast_product(double *arr, int n) {
2      double product = 1;
3      #pragma omp parallel for
4      for (int i = 0; i < n; i++) {
5          double subproduct = arr[i*8]*arr[i*8+1]*arr[i*8+2]*arr[i*8+3]
6                          * arr[i*8+4]*arr[i*8+5]*arr[i*8+6]*arr[i*8+7]
7          #pragma omp critical
8          product *= subproduct;
9      }
10     return product;
11 }
```