CS 61C Summer 2023

Process Level Parallelism, Caches

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 Both the multithreading in data-level parallelism and the manager-worker framework used in multiprocess code do not use shared memory.

False. Multithreaded programs can access main memory across threads, causing data races if written incorrectly. On the other hand, multiprocess programs have completely independent and distinct instances of the program starting from `MPI_Init`.

1.2 Replacing `amoswap.w rd rs2 (rs1)` with `lw rd 0(rs1)` and `sw rs2 0(rs1)` results in equivalent behavior.

False. These "atomic" instructions are labeled such because they cannot be divided into separate instructions. The use of `amoswap.w` in data synchronization and only allowing one thread to have the lock at a time doesn't work if the swapping happens in multiple instructions. For example, if two threads execute the `lw` instruction before one of them executes the `sw` instruction, then both threads will have the lock at the same time.

1.3 Because the manager-worker framework requires one process to deal with load balancing the rest of the work across programs, process-level parallelism is mostly useful for large-scale tasks.

True. Open MPI requires massive amounts of overhead, moreso than any other form of parallelism discussed in this course, with an entire dedicated manager process and the expensive communication across individual nodes.

1.4 Because process-level parallelism already takes advantage of multiple nodes, utilizing the OpenMP library in the Open MPI framework results in a performance decrease, as each thread will do the same, redundant work.

False. Thread-level parallelism does its multi-threaded work onto one node, as all its work is done onto one shared memory, while process-level parallelism can work across nodes. While both forms of parallelism allow for multiple operations to be done concurrently, the resources each require and can use are different. If allocated correctly, OpenMp and Open MPI can end up being complementary to each other, and are necessary optimizations in supercomputers, where much more resources are available and operations are done on a massive scale.

2 Open MPI

Beyond multithreading, the idea of process-level programming is to run one program on multiple processes at once.

The Open MPI project provides a way of writing programs which can be run on multiple processes. We can use its C libraries by calling their functions. Then, when we run the program, Open MPI will create a bunch of processes and run a copy of the code on each process. Here is a list of the most important functions for this class:

- `int MPI_Init(int* argc, char*** argv)` should be called at the start of the program, passing in the addresses of argc and argv.
- `int MPI_Finalize()` should be called at the end of the program.
- `int MPI_Comm_size(MPI_COMM_WORLD, int *size)` gets the total number of processes running the program, and puts it in `size`.
- `int MPI_Comm_rank(MPI_COMM_WORLD, int *rank)` gets the ID of the current process ($0 \sim$ total number of processes - 1) and puts it in `rank`.
- `int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, 0, MPI_COMM_WORLD)` sends a message in `buf`, which consists of `count` things with data type `datatype` to the process with ID `dest`.
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source , 0, MPI_COMM_WORLD, MPI_Status *status)` receives a message consisting of `count` things with data type `datatype` from the process with ID `source`, and puts the message into `buf`. Some additional information is put into a struct at `status`.
  - If you want to receive a message from any source, set the source to be `MPI_ANY_SOURCE`.
  - The source of the message can be found in the `MPI_SOURCE` field of the outputted status struct.
  - If you don't need the information in the status struct (e.g. because you already know the source of the message), set the status address to `MPI_STATUS_IGNORE`.

Note: Unlike OpenMP, the MPI functions will always put their results into an address which you provide as their arguments. The return value of the function is not an output, but rather the error code of the function.

In this section, we will implement the ManyMatMul example from lecture using a manager-worker approach.

We have n pairs of matrices available in input files `Task0a.mat, Task0b.mat, Task1a .mat, Task1b.mat`, ..., and we want to multiply each pair of matrices together, with their outputs written to the output files `Task0ab.mat, Task1ab.mat`, ...

We want to accomplish this task using multiple processes such that one process (the manager) assigns work to all other available processes (the workers).

2.1 First, perform the overall setup required for Open MPI to function. Fill out the following skeleton of the program:

```
#define TERMINATE -1
#define READY 0

/**
 * Takes in a number i. Reads files Taskia.mat, Taskib.mat,
 * multiplies them, then outputs to Taskiab.mat.
*/
int matmul(int i) {
    // omitted
}

int main(int argc, char** argv) {
    int numTasks = atoi(argv[1]); // read n from command line
    MPI_Init(&argc, &argv); // initialize
    // get process ID of this process and total number of processes
    int procID, totalProcs;
    MPI_Comm_size(MPI_COMM_WORLD, &totalProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &procID);
    // are we a manager or a worker?
    if (procID == 0) {
        // manager node code (see Q2.3)
    } else {
        // worker node code (see Q2.2)
    }
    MPI_Finalize(); // clean up
}
```

2.2 Next, fill in what the worker needs to do. Worker processes should repeatedly ask the manager for more work, then perform the work the manager asks of it. If it receives a message that there's no work to be done, it should stop. Let us define a simple communication protocol between the manager and worker:

- When the worker is free, it will send the READY(0) message to the manager.
- The manager will send one number back, which is the task number the worker should work on next.
- If there are no more tasks to done, then instead the manager will send back the TERMINATE(-1) message to the worker.

We will use a single 32-bit signed integer as the message, which corresponds to the MPI data type `MPI_INT32_T`.

```
// worker node code
int32_t message;
while (true) {
    // request more work
    message = READY;
    MPI_Send(&message, 1, MPI_INT32_T, 0, 0, MPI_COMM_WORLD);
    // receive message from manager
    MPI_Recv(&message, 1, MPI_INT32_T, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    if (message == TERMINATE) break; // all done!
    matmul(message); // do work
}
```

2.3 Finally, fill in the code for the manager process. While there's still more work to do, the manager should wait for a message from any worker and respond with the next task for the worker to work on. When all work has been allocated, the manager should wait for another message from each worker (meaning the worker is done with all work), and respond to each with the TERMINATE(-1) message. The manager shouldn't exit before sending TERMINATE to every worker!

```
// manager node code
int nextTask = 0; // next task to do
MPI_Status status;
int32_t message;
// assign tasks
while (nextTask < numTasks) {
    // wait for a message from any worker
    MPI_Recv(&message, 1, MPI_INT32_T, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    int sourceProc = status.MPI_SOURCE; // process ID of the source of the message
    // assign next task
    message = nextTask;
    MPI_Send(&message, 1, MPI_INT32_T, sourceProc, 0, MPI_COMM_WORLD);
    nextTask++;
}
// wait for all processes to finish
for (int i = 0; i < totalProcs - 1; i++) {
    // wait for a message from any worker
    MPI_Recv(&message, 1, MPI_INT32_T, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    int sourceProc = status.MPI_SOURCE; // process ID of the source of the message
    message = TERMINATE;
    MPI_Send(&message, 1, MPI_INT32_T, sourceProc, 0, MPI_COMM_WORLD);
}
```

3 Open MPI with Dependencies

Now that we have a working Open MPI implementation of our ManyMatMul task, lets extend this to account for data dependencies! Let's change our task to have an additional step: multiply n output matrices `Task0ab.mat`, `Task1ab.mat`, etc. in place with a set matrix `kernel.mat`.

Here we provide a new function to use in the worker process:

```
/**
 * Takes in a number i. Reads files Taskiab.mat and
 * multiplies them with kernel.mat in place. If file
 * does not exist, return -1
*/
int final_matmul(int i) {
    //omitted
}
```

3.1 Provided below is the pseudocode for the manager process in our new implementation. Assume that our program and workers are set up in the same way as described in Q3.

```
// manager node pseudocode
counter = 0;
while (counter < n) {
    Wait for a message from any worker;
    Assign worker with the next pair of matrices to multiply,
        worker will call matmul(counter);
    counter++;
}
counter = 0;            // start in-place multiplication
while (counter < n) {
    Wait for a message from any worker;
    Assign worker with next in-place multiplication,
        worker will call final_matmul(counter);
    counter++;
}
// wait for all processes to finish
for each process {
    Wait for a message from any worker;
    Send worker message to TERMINATE;
}
```

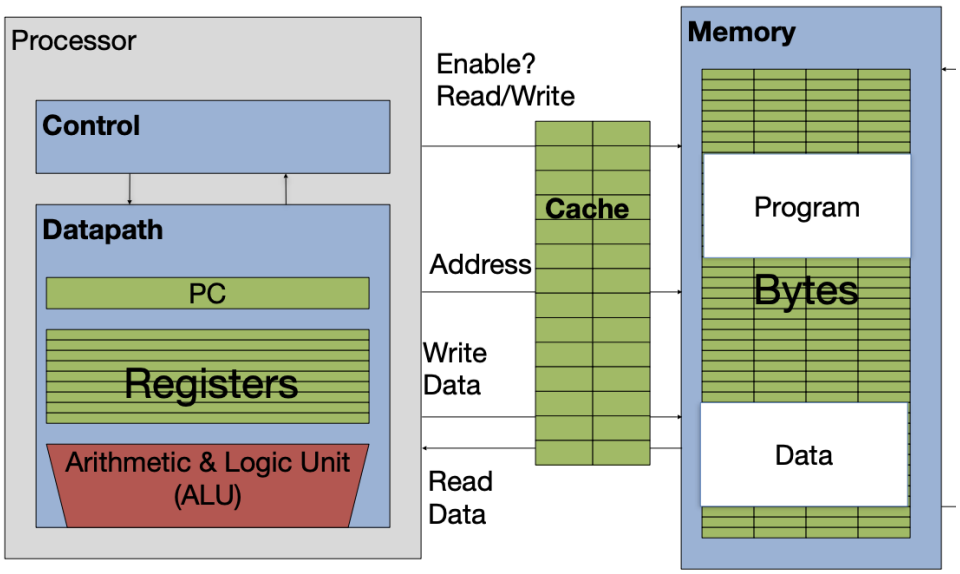
Will this program successfully output the correct matrix files? If it doesn't, explain why. If it does, does it optimally parallelize our desired task? You may assume that if `final_matmul` returns -1, the worker will wait some amount of time before sending the manager another READY message.

As the second while loop does its work in sequential order, the program will be forced to wait for the corresponding first task to finish before attempting any additional

`final_matmuls`. For example, if Task1 was a massive, high-dimensional calculation, each other process would need to wait for the Task1 to finish before attempting any of the in-place multiplications in the second while loop, creating a performance bottleneck.

4 Understanding T/I/O

We use caches to make our access to data faster. When working with main memory (RAM), the main problem faced is the fact that access to the main memory is very slow. In fact, modern processors take about 100 instructions cycles or more to access the main memory, meaning memory accesses become the bottleneck of our programs. Caches help fix this problem for us - they hold a portion of the data in main memory, that we might access again later on. They are closer to the processor in the memory hierarchy, and thus accessing a cache is much faster than accessing the main memory.



As seen above, the access to cache is the middle step between the CPU asking for a memory bit, and the actual main memory access - if the data is not found in the cache, only then is main memory accessed. This way unnecessary trips to main memory are avoided. One important detail is that caches are much smaller in size than main memory - this is why we have to be efficient in what we hold in caches. When we are saving data in caches, we need to be as efficient as possible. In order to do this, we make use of locality. We have two different kinds of locality to consider.

- **Temporal Locality:** If we have accessed a piece of information recently, it is possible that we will access it again. So, we hold this data in the cache.
- **Spatial Locality:** If we have accessed a memory location recently, it is probable that we will access the neighbouring addresses as well. So, we also keep the neighbouring addresses within the cache. An example is array accesses - if we access the 0th element of an array, it is probable we will also access the 1st one.

Note that caches hold the data in blocks that have a size equal to the block size of

the cache.

When working with caches, we have to be able to break down the memory addresses we work with to understand where they fit into our caches. There are three fields:

- **T**ag - Used to distinguish different blocks that use the same index. Number of bits: (# of bits in memory address) - Index Bits - Offset Bits
- **I**ndex - The set that this piece of memory will be placed in. Number of bits: $\log_2$(# of indices)
- **O**ffset - The location of the byte in the block. Number of bits: $\log_2$(size of block)

Given these definitions, the following is true:

$$\log_2(\text{memory size}) = \text{address bit-width} = \#\text{ tag bits} + \#\text{ index bits} + \#\text{ offset bits}$$

Another useful equality to remember is:

$$\text{cache size} = \text{block size} * \text{num blocks}$$

4.1 Assume we have a direct-mapped byte-addressed cache with capacity 32B and block size of 8B. Of the 32 bits in each address, which bits do we use to find the index of the cache to use?

We can determine the number of index bits we need from the number of sets our cache has. Since our cache is direct-mapped, the number of sets is the same as the number of blocks, so we just need to figure out how many blocks our cache has. Using the equality from above, we see that num blocks = cache size/block size, so our cache has $32/8 = 4$ blocks. We need $\log_2(4) = 2$ bits to differentiate the 4 blocks, so we have 2 index bits.
In order to determine where exactly the index bits are, we need to calculate the number of offset bits and tag bits we have. The number of offset bits is just dependent on the block size, so since our blocks are size 8B, we need $\log_2(8) = 3$ bits to differentiate the 8 bytes in the block, so we have 3 offset bits.
Our offset bits take up the least significant bits, with the index bits being the set of next most significant bits. Denoting the most significant bit (MSB, on the left) as 31 and the least significant bit (LSB, on the right) as 0, having 3 offset bits means our index bits start at bit 3, and thus we use bits 3 and 4 as the index bits.

4.2 Which bits are our tag bits? What about our offset?

The offset (in this case) is the 3 least significant bits, so reusing the convention from the previous quesiton, the offset bits are bits 0, 1, and 2. Our tag is the remaining high-order bits, so our tag bits are bits 5-31.

4.3 Classify each of the following byte memory accesses as a cache hit (H), cache miss

(M), or cache miss with replacement (R). Tip: Drawing out the cache can help you see the replacements more clearly.

Address	T/I/O	Hit, Miss, Replace
0x00000004		
0x00000005		
0x00000068		
0x000000C8		
0x00000068		
0x000000DD		
0x00000045		
0x000000CF		
0x000000F3		

Ignore miss types (compulsory/conflict/capacity) until Q4.

```
0x00000004     Tag 0, Index 0, Offset 4: M, Compulsory
0x00000005     Tag 0, Index 0, Offset 5: H
0x00000068     Tag 3, Index 1, Offset 0: M, Compulsory
0x000000C8     Tag 6, Index 1, Offset 0: R, Compulsory
0x00000068     Tag 3, Index 1, Offset 0: R, Conflict
0x000000DD     Tag 6, Index 3, Offset 5: M, Compulsory
0x00000045     Tag 2, Index 0, Offset 5: R, Compulsory
0x000000CF     Tag 6, Index 1, Offset 7: R, Conflict
0x000000F3     Tag 7, Index 2, Offset 3: M, Compulsory
```

Note that the M and R distinction here is for student understanding, and that the cache doesn't behave differently for these cases.