CS 61C Spring 2023 — Parallelism II, 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, however, 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 cores, 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 core, as all its work is done onto one shared memory, while process-level parallelism can work across cores. 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 Locks and Critical Sections

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

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

(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?

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

In order to implement critical sections, we can use the idea of uninterrupted execution, also known as **atomic** execution.

In RISC-V, we have two categories of atomic instructions:

1. **Amoswap**: allows for uninterrupted memory operations within a single instruction
2. **Load-reserve, store-conditional**: allows us to have uninterrupted execution across multiple instructions

Both of these can be used to achieve atomic primitives. Here we'll focus on the former with this example:

```
Test-and-set

Start:  addi         t0 x0 1 # Locked = 1
        amoswap.w.aq t1 t0 (a0)
        bne          t1 x0 Start
# If the lock is not free, retry

        ... # Critical section

        amoswap.w.rl x0 x0 (a0) # Release lock
```

amoswap rd, rs2, (rs1): Atomically, loads the word starting at address **rs1** into **rd** and puts **rs2** into memory at address **rs1**. Data races are avoided using the *aq* and *rl* flags, which *acquire* a lock that forces multiple threads to wait their turn until the lock is *released*.

Test-and-set: We have a lock stored at the address specified by `a0`. We utilize `amoswap` to put in 1 and get the old value. If the old value was a 1, we would not have changed the value of the lock and we will realize that someone currently has the lock. If the old value was a 0, we will have just "locked" the lock and can continue with the critical section. When we are done, we put a 0 back into the lock to "unlock" it.

We've experimented with data synchronization across threads in C, but now let's take a look at how to parallelize and avoid data races in RISC-V!

We want to parallelize a program that finds the sum of the integers in an array pointed to by `a0` (array length = `a2`) and places it in memory at address `a1`. There is a free word of memory initialized to zero (i.e. result of `calloc(4, 1)`) pointed to by `a3`. For the sake of simplicity, assume there is a function `get_thread_num` that returns the current thread's number and a function `get_num_threads` that returns the total number of threads.

2.2 Here is some skeleton code to parallelize this operation. Note the use of `amoswap`. Fill out the skeleton code accordingly.

```
1      #Prologue
2      ...
3      mv s0 a0        #s0 points to the array
4      mv s1 a1        #s1 points to the global sum
5      mv s2 a2        #s2 has the length of array
6      mv s3 a3        #s3 holds our lock
7      jal get_num_threads
8      mv s4 a0        #s4 has the total number of threads
9      jal get_thread_num
10     mv s5 a0        #s5 has the current thread number
11     li t0 0         #t0 holds our local sum
12 Loop:
13     bge s5 s2 Exit
14     slli t1 s5 2
15     add t1 s0 t1    #index into array
16     lw t2 0(t1)
17     add t0 t0 t2    #add to local sum
18     add s5 s5 s4    #process indices which are equal to s5, modulo s4
19     j Loop
20 Exit:
21     li t2, 1        #try to swap a nonzero value into the lock
22 Try:
23     lw t1 0(s3)     #check if lock is held by other thread
24     bnez t1 Try
25     amoswap.w.aq t1 t2 (s3)
26     bnez t1 Try     #must try again if we fail to acquire lock
27     lw t2 0(s1)
28     add t2 t2 t0
29     sw t2 0(s1)     #add to the global sum in critical section to avoid data races
30
31     amoswap.w.rl x0 x0 (s3) # release lock
32     #Epilogue
33     ...
```

2.3 Why do we want to use an atomic instruction in our parallelized implementation?

Without using some sort of atomic instruction, we encounter a data race when multiple threads could write to the global sum at `s1`. This results in non-deterministic behavior in `s1`.

2.4 Between which lines in the program above should threads start to run in parallel on separate copies of code? (Equivalent to where we put `#pragma omp parallel` in C)

Between lines 6 and 7, after we store all arguments but before we find the total number of threads we are running. This is because we want to store the arguments only once for efficiency, but we don't know the number of threads until we spawn them.

3 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 ∼ 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).

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

```c
#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 Q3.3)
    } else {
        // worker node code (see Q3.2)
    }
    MPI_Finalize(); // clean up
}
```

3.2 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
}
```

3.3 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);
}
```

4 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
}
```

4.1 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.