CS 61C Summer 2024 — Thread-Level Parallelism, PLP — Discussion 10

1 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

1.1 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.

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;
}
```

2.2 When added to a `#pragma omp parallel for` statement, the `reduction(operation : var)` directive creates and optimizes the critical section for a for loop, given a variable that should be in the critical section and the operation being performed on that variable. An example is given below.

```
// Assume arr has length n
int fast_sum(int *arr, int n) {
    int result = 0;
    #pragma omp parallel for reduction(+: result)
    for (int i = 0; i < n; i++) {
        result += arr[i];
    }
    return result;
```

```
}
```

Fix the code by adding the `reduction(operation: var)` directive to the **`#pragma`** `omp parallel` **`for`** statement. Which variable should be in the critical section, and what is the operation being performed?

```
double fast_product(double *arr, int n) {
    double product = 1;
    #pragma omp parallel for reduction (*:product)
    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;
}
```

3 Multi-Process Code

One advantage of process-level parallelism is that we have freedom to do complex tasks without worrying about race conditions in memory due to processes not sharing memory. Examine the code snippet below to answer the questions.

```
int x = 10;
int y = 0;

// Split into two processes

if (/* Is Process 1 */) { y++; }
if (/* Is Process 2 */) { x--; }
```

3.1 After the code segment completes, what will be the values of `x` and `y` for Process 1?

```
x = 10;
y = 1;
```

Notice that only the value of `y` changes. This is because when we create a new processes, it is given a separate address space. This enforces the separation between processes that provides security within a system.

3.2 After the code segment completes, what will be the values of `x` and `y` for Process 2?

```
x = 9;
y = 0;
```

Notice that only the value of `x` changes. This is when a new process is created, it is initialized with a separate address space.

4 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).

4.1 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
}
```

4.2 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
}
```

4.3 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);
}
```

5 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
}
```

5.1 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.