

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 ~ 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:

```

1  #define TERMINATE -1
2  #define READY 0
3
4  /**
5   * Takes in a number i. Reads files Taskia.mat, Taskib.mat,
6   * multiplies them, then outputs to Taskiab.mat.
7   */
8  int matmul(int i) {
9      // omitted
10 }
11
12 int main(int argc, char** argv) {
13     int numTasks = atoi(argv[1]); // read n from command line
14     _____; // initialize
15     // get process ID of this process and total number of processes
16     int procID, totalProcs;
17     MPI_Comm_size(MPI_COMM_WORLD, _____);
18     MPI_Comm_rank(MPI_COMM_WORLD, _____);
19     // are we a manager or a worker?
20     if (_____ ) {
21         // manager node code (see Q3.3)
22     } else {
23         // worker node code (see Q3.2)
24     }
25     _____; // clean up
26 }

```

- 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.

```

1 // worker node code
2 int32_t message;
3 while (true) {
4     // request more work
5     message = _____;
6     MPI_Send(_____);
7     // receive message from manager
8     MPI_Recv(_____);
9     if (message == TERMINATE) _____; // all done!
10    _____; // do work
11 }

```

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!

```

1 // manager node code
2 int nextTask = 0; // next task to do
3 MPI_Status status;
4 int32_t message;
5 // assign tasks
6 while (_____ ) {
7     // wait for a message from any worker
8     MPI_Recv(_____);
9     int sourceProc = _____; // process ID of the source of the message
10    // assign next task
11    message = _____;
12    MPI_Send(_____);
13    nextTask++;
14 }
15 // wait for all processes to finish
16 _____ {
17     // wait for a message from any worker
18     MPI_Recv(_____);
19     int sourceProc = _____; // process ID of the source of the message
20     message = TERMINATE;
21     MPI_Send(_____);
22 }

```

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:

```

1  /**
2   * Takes in a number i. Reads files Taskiab.mat and
3   * multiplies them with kernel.mat in place. If file
4   * does not exist, return -1
5   */
6  int final_matmul(int i) {
7      //omitted
8  }
```

- 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.

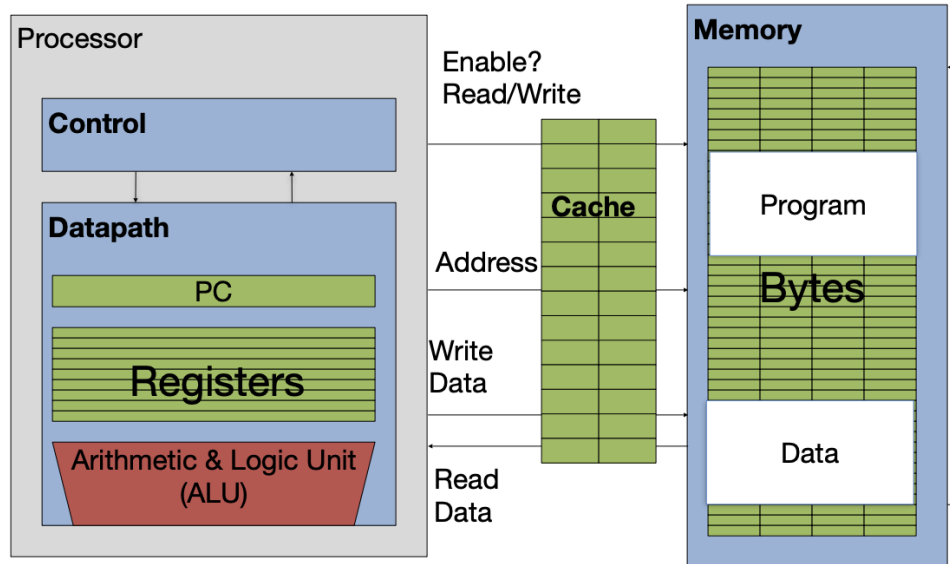
```

1  // manager node pseudocode
2  counter = 0;
3  while (counter < n) {
4      Wait for a message from any worker;
5      Assign worker with the next pair of matrices to multiply,
6      worker will call matmul(counter);
7      counter++;
8  }
9  counter = 0;          // start in-place multiplication
10 while (counter < n) {
11     Wait for a message from any worker;
12     Assign worker with next in-place multiplication,
13     worker will call final_matmul(counter);
14     counter++;
15 }
16 // wait for all processes to finish
17 for each process {
18     Wait for a message from any worker;
19     Send worker message to TERMINATE;
20 }
```

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.

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:

- **Tag** - Used to distinguish different blocks that use the same index. Number of

bits: (# of bits in memory address) - Index Bits - Offset Bits

- **Index** - The set that this piece of memory will be placed in. Number of bits: $\log_2(\# \text{ of indices})$
- **Offset** - The location of the byte in the block. Number of bits: $\log_2(\text{size of block})$

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?

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

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		