

1 Background

This document will develop the concepts of so-called *hybrid parallel programming*, where MPI and OpenMP are both used in the same program in an attempt to gain the benefits of both parallel frameworks. The hybrid parallel programs will be executed on a Raspberry Pi cluster that has already been developed. As such, it is assumed that the previous assignments on building the Pi cluster, executing MPI programs on the cluster, and programming in OpenMP have been covered. In particular, a shared directory is used to run programs on the cluster, which was set up in the MPI cluster assignment.

Recall that OpenMP is a *shared-memory* framework: it is designed to efficiently run parallel programs on a single multi-core computer by allowing the sharing of memory between cores where possible. The basic unit of parallelism in an OpenMP program is a thread, which is relatively lightweight and efficient. On the other hand, MPI is a *distributed-memory* framework: while it has no concept of shared memory, it is capable of running parallel programs across multiple computers simultaneously. The basic unit of parallelism in MPI is a process, which requires more memory than a thread, but two MPI processes on different computers can interact with each other as if they were on the same computer.

In view of this, there is a natural way to combine OpenMP and MPI to run efficient parallel programs across a cluster of computers. We will create exactly one MPI process per computer in the cluster, and use the message passing provided by MPI to communicate between these processes. From within each node's MPI process, we will parallelize that node's work using OpenMP, taking advantage of its lightweight threads and efficient memory sharing capabilities. This approach should thus provide the benefits of both programs. In practice,

2 Hybrid Hello World

As usual, our first attempt at hybrid parallelism will be a "Hello World" program that prints an identifying message from each thread, provided as `hybrid-hw.c`. The program is built from features of MPI and OpenMP that we have used before:

- This program begins with the call `MPI_Init(&argc, &argv)` and ends with the call `MPI_Finalize()`, as must be the case with every program that uses MPI.
- The `MPI_Comm_rank` and `MPI_Get_processor_name` functions are used to compute the MPI process ID and the hostname of the executing MPI process.
- The `omp_get_thread_num` function is used to compute the OpenMP thread ID of the executing thread.

- The `for` loop is parallelized using the OpenMP compiler directive:

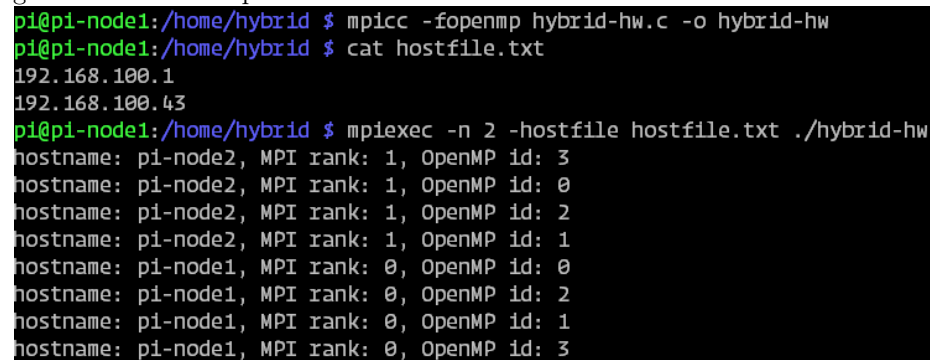
```
#pragma omp parallel for
for (int i = 0; i < 4; i++)
```

This directive creates separate tasks for each iteration of the loop then distributes each task to a thread as threads are made available to OpenMP, resulting in the parallelization of the loop.

Recall that the `mpicc` compiler should be used to compile an MPI program. However, `mpicc` is implemented as a *wrapper* around the `gcc` compiler, in that calling `mpicc` is ultimately just calling `gcc` with certain extra command-line arguments that make the MPI library available. Thus we can use the `-fopenmp` flag to compile our program with OpenMP functionality, just as we have done previously when compiling with `gcc`.

Once we have compiled our program to an executable, it remains to actually execute it. As before, we will execute our MPI programs using `mpiexec`. To run the program on multiple nodes in a cluster, we must also pass a hostfile containing the IP addresses of nodes on which to create processes. Since we want to run only one MPI process on each node, our hostfile should contain the IP address of each node only once.

To correctly use the hostfile, we must pass it to `mpiexec` with the `-hostfile` command-line argument. Additionally, the `-n` argument must be used to tell MPI how many processes to create in total. The following image shows how the program would be compiled and executed on a two-node cluster:



```
pi@pi-node1:/home/hybrid $ mpicc -fopenmp hybrid-hw.c -o hybrid-hw
pi@pi-node1:/home/hybrid $ cat hostfile.txt
192.168.100.1
192.168.100.43
pi@pi-node1:/home/hybrid $ mpiexec -n 2 -hostfile hostfile.txt ./hybrid-hw
hostname: pi-node2, MPI rank: 1, OpenMP id: 3
hostname: pi-node2, MPI rank: 1, OpenMP id: 0
hostname: pi-node2, MPI rank: 1, OpenMP id: 2
hostname: pi-node2, MPI rank: 1, OpenMP id: 1
hostname: pi-node1, MPI rank: 0, OpenMP id: 0
hostname: pi-node1, MPI rank: 0, OpenMP id: 2
hostname: pi-node1, MPI rank: 0, OpenMP id: 1
hostname: pi-node1, MPI rank: 0, OpenMP id: 3
```

Notice that both nodes print four lines, corresponding to the four iterations of the `for` loop that each node executes. Since at least four threads are available to OpenMP by default, each iteration of the loop is executed by a different thread, with thread IDs ranging from 0 to 3. Since the MPI process running on each cluster node has its own copy of the program and its own threads, we see that both processes have their own thread with ID 0. Thus, in order to uniquely identify a given thread in our hybrid program, we will need both its MPI rank as well as its OpenMP thread ID.

If we instead explicitly set the available threads to lower than 4, we will see that some OpenMP threads will execute more than one iteration of the loop:

```

pi@pi-node1:/home/hybrid $ export OMP_NUM_THREADS=2
pi@pi-node1:/home/hybrid $ mpiexec -n 2 -hostfile hostfile.txt ./hybrid-hw
hostname: pi-node2, MPI rank: 1, OpenMP id: 0
hostname: pi-node2, MPI rank: 1, OpenMP id: 0
hostname: pi-node2, MPI rank: 1, OpenMP id: 1
hostname: pi-node2, MPI rank: 1, OpenMP id: 1
hostname: pi-node1, MPI rank: 0, OpenMP id: 1
hostname: pi-node1, MPI rank: 0, OpenMP id: 1
hostname: pi-node1, MPI rank: 0, OpenMP id: 0
hostname: pi-node1, MPI rank: 0, OpenMP id: 0

```

3 Hybrid prime counting

This section will revisit the prime counting program that was originally written purely using MPI. We will now modify it to use a hybrid parallel approach. Namely, it will use OpenMP directives to parallelize the work on each individual node, while still using MPI to coordinate the final results of each node.

When trying to parallelize existing serial code with OpenMP, a natural place to start is to check each of the program's loops to see if their iterations can be easily parallelized. If later loop iterations generally depend on previous iterations, the loop may be difficult to parallelize, but if the loop iterations are mostly independent from each other, then a single OpenMP directive can often be quite effective.

There are two `for` loops in the original `primes.c` program that are candidates for parallelization. The first is within the `is_prime` function:

```

for (int i = 2; i <= sqrt((double)number); i++) {
    if (number % i == 0) return 0;
}

```

While these loop iterations are indeed independent of each other, each iteration represents a very small amount of work, amounting to essentially only one modulo operation and one numeric comparison. Parallelizing such short iterations can often incur more overhead from scheduling than is worthwhile. The second loop, however, is much more promising:

```

int primes_found = 0;
for (int n = a; n <= UPPER_LIMIT; n += k) {
    if (is_prime(n)) primes_found++;
}

```

This loop body calls `is_prime(n)`, which for large `n` represents enough work to warrant parallelization. However, since the loop iterations write to the external variable `primes_found`, we need to take care to update this variable in a thread-safe way. In particular, the following approach *does not work*, since multiple threads may attempt to increment to `primes_found` simultaneously, resulting in a race condition:

```
// this results in a race condition
#pragma omp parallel for
for (int n = start; n <= end; n++) {
    if (is_prime(n)) primes_found++;
}
```

There are several ways to fix the above directive to correctly compute `primes_found`, but the simplest is to use a sum reduction, a technique we previously used in the parallel selection sort algorithm. With a sum reduction, each thread will maintain its own local copy of `primes_found` that isn't vulnerable to a race condition. When the loop ends, the values from each thread will be added serially, resulting in the correct value. The following shows the correct directive that includes the sum reduction:

```
#pragma omp parallel for reduction(+:primes_found)
for (int n = start; n <= end; n++) {
    if (is_prime(n)) primes_found++;
}
```

Exercise 1. Compile and execute `hybrid-primes.c` using one MPI process and four OpenMP threads. You will need to use the `-lm` command-line argument when compiling, which links the math library containing the `sqrt` function used by `is_prime`. Time the execution using the `time` utility.

```
pi@pi-node1:/home/hybrid $ mpicc -lm -fopenmp hybrid-primes.c -o hybrid-primes
pi@pi-node1:/home/hybrid $ export OMP_NUM_THREADS=4
pi@pi-node1:/home/hybrid $ time mpiexec -n 1 ./hybrid-primes
process 0 found 348513 primes
found 348513 primes less than or equal to 5000000

real    0m10.798s
user    0m31.100s
sys     0m0.078s
```

Exercise 2. Execute `hybrid-primes` using 1 MPI process for each node in the cluster and 4 OpenMP threads per node.

```
pi@pi-node1:/home/hybrid $ export OMP_NUM_THREADS=4
pi@pi-node1:/home/hybrid $ cat hostfile.txt
192.168.100.1
192.168.100.43
192.168.100.124
pi@pi-node1:/home/hybrid $ time mpiexec -n 3 -hostfile hostfile.txt ./hybrid-primes
process 0 found 125796 primes
process 1 found 113323 primes
process 2 found 109394 primes
found 348513 primes less than or equal to 5000000

real    0m4.219s
user    0m8.089s
sys     0m0.221s
```

Exercise 3. Execute `hybrid-primes` using 4 MPI processes for each node in the cluster and only 1 OpenMP thread per node. Compare the execution time to Exercise 2.

4 Hybrid matrix multiplication

This section demonstrates a matrix multiplication program written using hybrid parallel programming. The program is provided as `hybrid-matmul.c` and is adapted from University of Arizona course materials that are cited at the end of the section.

First, this program allocates N-by-N matrices **A**, **B**, and **C** on the root node (i.e. the node with MPI rank 0). The root node also initializes **A** and **B** to have all entries equal to 1. The goal of the program is then to write the matrix product **AB** into the uninitialized matrix **C**. The matrix product computation is distributed to all available cluster nodes, each of which computes a *strip* (a number of consecutive rows) of **C**. MPI message passing is used to coordinate all partial results on the root node, which has the completed result **C** by the end of the program. Since the input matrices **A** and **B** are only initialized on the root node, MPI message passing must be used to send these matrices to non-root nodes as well.

In order to compute an entry $C[i][j]$ of the matrix product (that is, the entry of **C** in the *i*-th row and *j*-th column), one needs the *i*-th row of **A** and the *j*-th column of **B**. This follows from the definition of the matrix product:

$$C[i][j] = A[i][0]*B[0][j] + A[i][1]*B[1][j] + \dots + A[i][N]*B[N][j];$$

From this, we can deduce that the entirety of **B** is needed to compute the *i*-th row of **C**, since one element of the row occupies every column. However, since all those elements lie on the same row of **C**, they can be computed from only the *i*-th row of **A**. Thus, in order for each node to compute its rows of **C**, it only needs to be sent the corresponding rows of **A** from the root node. This is an optimization which thus saves both data transfer time and memory on the non-root nodes.

With the above in mind, the methods `MPI_Send` and `MPI_Recv` are used to transfer only a strip of rows of **A** from the root node to each non-root node, as we have seen before. These same methods are used to transfer the resulting strip of **C** back to the root node. Since the entirety of **B** is needed on every node, the `MPI_Bcast` method is used to share **B** between every node in the cluster. As was shown previously, broadcasting in this way is more efficient than using send and receive methods when the same data needs to be shared among every node.

Finally, note that OpenMP is used to parallelize each node's computation of its strip of **C** using the standard `parallel for` directive:

```
#pragma omp parallel for shared(A,B,C)
for (i=0; i<stripSize; i++) {
    for (j=0; j<N; j++) {
        for (k=0; k<N; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Note that since the directive is applied to the outer loop, only the outer loop's iterations will be divided into parallel tasks; each such task will consist of the entirety of the inner two loops for a particular value of `i`.

Exercise 4. Compile `hybrid-matmul.c` and execute it using both one node and the full cluster (with four OpenMP threads per node in both cases). Compute the speedup factor when using the full cluster and compare it to the number of nodes in the cluster.

Exercise 5. The `collapse(n)` clause can be added to an OpenMP `parallel for` directive to *collapse* `n` nested loops, meaning that OpenMP schedules the iterations of `n`-th inner loop in parallel, rather than the iterations of the outer loop. Experiment with this directive by adding a `collapse` clause to the `parallel for` directive above the main loop of `hybrid-matmul.c`. Test the new program on the full cluster. Record two separate execution times: one when `collapse(2)` is used, and one when `collapse(3)` is used. Compare the results to Exercise 4.

Additional resources

- Original matrix multiplication program:
<http://dkl.cs.arizona.edu/teaching/csc522-fall16/examples/hybrid-openmp-mm.c>
(Local copy saved to `resources/hybrid-openmp-mm.c`)

5 Hybrid sort

This section develops the provided program `hybrid-sort.c`, which sorts an array of integers with a hybrid parallel approach. It uses as subroutines the quicksort and mergesort programs that have been covered previously.

At a high level, the program first generates an input array on the root node, then distributes equal strips of it between the available cluster nodes using MPI message passing. Each node sorts its strip using the same `quicksort` subroutine that was developed using OpenMP tasks, then sends the now-sorted strip back to the root node, again as an MPI message. Finally, the root node uses the same `merge` subroutine that was originally run as part of a CUDA mergesort kernel to merge the sorted strips into a single fully sorted array.

Since the pattern of distributing equal-sized strips of an input array is very common, this functionality was implemented into the MPI library functions `MPI_Scatter` and `MPI_Gather`. First, `MPI_Scatter` is used to distribute equal-sized portions of the input array:

```
MPI_Scatter(data, ELTS_PER_NODE, MPI_INT,  
            strip, ELTS_PER_NODE, MPI_INT,  
            ROOT_RANK, MPI_COMM_WORLD);
```

The function arguments denote that `ELTS_PER_NODE` values of type `MPI_INT` are being sent from the node of rank `ROOT_RANK` to every cluster node in the communicator `MPI_COMM_WORLD` (which includes all cluster nodes by default). The values that are sent are composed of equal-sized segments from the input `data` array, and each node receives the values in the `strip` array. Note that since the root node is also in the communicator `MPI_COMM_WORLD`, the above call will also write a segment of `data` into the `strip` array on the root node, without using any inter-node message passing.

The function `MPI_Gather` has the inverse functionality of `MPI_Scatter`, and is used to return the strips back to the root node once they have been sorted:

```
MPI_Gather(strip, ELTS_PER_NODE, MPI_INT,
           data, ELTS_PER_NODE, MPI_INT,
           ROOT_RANK, MPI_COMM_WORLD);
```

MPI is designed so that this call, with the same arguments as `MPI_Scatter` above (other than transposing `strip` and `data`), will write the equal-sized `strip` arrays on each node back into the same segments of `data` that they originated from. After the above call is completed in the `hybrid-sort.c` program, the `data` array will be composed of a number of *sorted* segments of size `ELTS_PER_NODE` equal to the number of nodes in the cluster.

Finally, note that the number of elements sorted per node should be passed as a command line argument to the program. The number is parsed from the command line arguments using the call:

```
int ELTS_PER_NODE = atoi(argv[1]);
```

Command line arguments following the name of a program being executed by `mpiexec` are passed to the program itself rather than `mpiexec`. This is seen in the following example, which sorts 10 million elements per node:

```
pi@pi-node1:/home/hybrid $ mpicc -fopenmp hybrid-quicksort.c -o qs
pi@pi-node1:/home/hybrid $ mpiexec -n 3 -hostfile hostfile.txt ./qs 10000000
Duration: 4.81199 sec
Input size: 30000000
Results sorted: yes
```

Exercise 6. Compile `hybrid-sort.c`. Record the execution times when sorting 30 million integers with 1, 2, and 3 nodes and four OpenMP threads per node. Note that since the program's command line argument only describes the elements sorted *per node*, different arguments will need to be used to sort 30 million integers total, depending on the number of nodes.