OpenMPI Assessable Task

Liam Ryan — 19769811

August 5, 2022

1 Introduction

The Mandelbrot Set is a fascinating mathematical entity that has garnered the interest of mathematicians and computer scientists alike since it's discovery in the late 1970s. The Mandelbrot Set is defined as the set of κ for which the following series does not diverge

$$z_{i+1} = z_i^2 + \kappa \tag{1}$$

where z and κ are complex and $z_0 = \kappa$. Although this is a fairly simple definition, the function gives rise to incredibly detailed, recursive patterns. The Mandelbrot Set will be used as the focal point of an investigation into fundamental concepts and partitioning strategies within the OpenMPI message-passing framework. All results are produced on Pawsey Supercomputing Centre's Zeus cluster with 10 MPI processes on an 8000x8000 grid on the interval $[-2, 2] + [-2, 2]i$.

2 Programming Notes

I programmed in C for this project, all programs were compiled through gcc/4.8.5. Through the addition of compilation flags I adhered to -ansi, -Wall and -Werror standards. I utilised conditional compilation throughout so that image functionality (.ppm file output) and timing functionality (MPI timing functions, timing results to .csv file) can be easily swapped through the addition of the definition of the 'TIME' directive at compile time (add `-D TIME` to compilation flags). As an example snippet:

```
#ifdef TIME
time1 = MPI_Wtime();
#endif
```

I will omit all of the `#ifdefs` and `#endifs` when presenting the code snippets throughout the report for clarity's sake - the 'timed' version is what will be presented. Each program generates a .csv file, all visuals are produced by Python scripts on this data using `matplotlib` and NumPy's `np.genfromtxt()`. The full code can be found at `https://github.com/LDRyan0/mandelbrot`.

3 Block Partitioning

The first portioning philosophy employed was to statically decompose the work into even rectangular blocks. The 8000x8000 grid was decomposed into 8000x800 blocks horizontal blocks - one for each of the 10 MPI processes. After variables are declared and memory is allocated, each MPI process calculates the values on the required set points. `calcTime` is calculated through timers directly above and below the computation for loop.

```
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &ncpu);
```

```
groupSize = N * N / ncpu;
x=(float *)malloc(groupSize*sizeof(float));
if (rank == 0) {
    y = (float *)malloc(N*N*sizeof(float));
}

time1 = MPI_Wtime();
for (loop=0; loop<groupSize; loop++) {
    pos = loop+groupSize*rank;
    i=pos/N;
    j=pos%N;
    z=kappa= (4.0*(i-N/2))/N + (4.0*(j-N/2))/N * I;
    k=1;
    while ((cabs(z)<=2) && (k++<MAXITER))
        z= z*z + kappa;
    x[loop]= log((float)k) / log((float)MAXITER);
}
time2 = MPI_Wtime();
calcTime = time2 - time1;
```

It is important to note that this solution only works when `N*N` is a multiple of `ncpu`. A tailored solution could be produced which handles the extra computation necessary fairly easily but was not required in our case. Now that all of the computation has been done we use `MPI_Gather()` to collect the work from all processes. With regards to timing it is important to note that different processes will process their allocated computational block at different speeds. For proper timing of the time spent waiting for other processes to finish calculating and execute the `MPI_Gather`, an `MPI_Barrier()` was added beforehand. This gives us `waitTime` and `commTime`.

```
ierr = MPI_Barrier(MPI_COMM_WORLD);
time1 = MPI_Wtime();
waitTime = time1 - time2;

ierr = MPI_Gather(x, groupSize, MPI_FLOAT, y, groupSize, MPI_FLOAT, 0,
    MPI_COMM_WORLD);

time2 = MPI_Wtime();
commTime = time2 - time1;
```

As it can be seen in Figure 1, clearly different parts of the Mandelbrot Set require significantly greater amounts of computation. Regions within the set take much longer as they are iterated on until MAXITER (1000) whereas regions that diverge rapidly are exited out of the while loop quickly. This is especially highlighted in the comparison of process 4 which took 105.28s to calculate while process 9 which took only 0.53s to complete. Process 4 is holding up all the other process which are simply left idling until it arrives at its `MPI_Gather()`. A further note is that communication time is of no significance in this case - 10-100 ms. Although block partitioning across 10 MPI processes offered an ≈2.7x speedup over the serial version, clearly this partitioning scheme has significant load imbalance and subsequently poor resource utilisation.

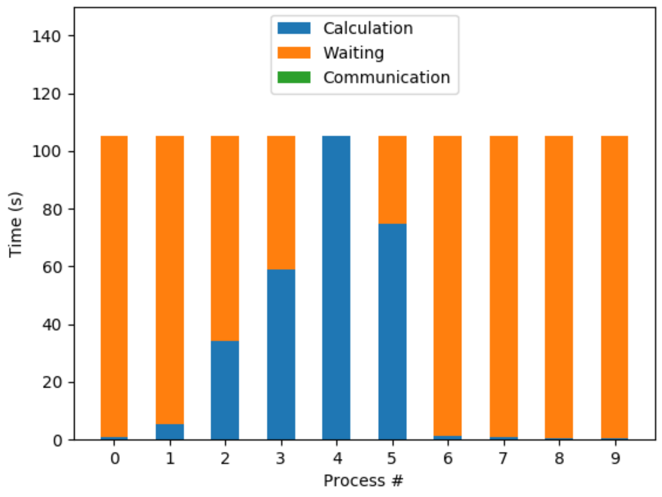


Figure 1: Timing results for static block decomposition on 8000x8000 grid with 10 MPI Processes

4 Master-Worker Partitioning

A form of dynamic work partitioning, master-worker parallelism is a scheme where 1 process (the master) is retracted from computation and instead distributes computing work to the various workers. The master ”dispenses” work for the worker to complete, to which the worker will send back to the master once it is completed. The master then gives the back new work until all computation is complete. This scheme of dynamic partitioning allows for a high guarantee of resource utilisation, as if any processes are completely work quicker than others then the process is simply given new work. Master-worker schemes have some downsides; one process being reserved entirely for communication (spending much of it’s time waiting), higher amounts of communication are required and it is a more difficult partitioning methodology to develop.

The program begins by setting up MPI and allocating an array (x) of size `chunkSize` for both the master and workers. Beginning with the master, we first need to deal with the case where `N*N` is not a multiple of `chunkSize`. There are multiple solutions to this issue, many would require some form of dynamic checking on the master and worker end as to whether the `N*N` limit has been reached. Ultimately I decided that it would be a cleaner solution (and potentially faster as I have come across before) to extend the domain and perform some redundant computation by extending out the domain to be the next multiple of `chunkSize` after `N*N`. There will always be trade-offs in performance with dynamic checks vs redundant computation but this is the route I decided to take. We calculate past the NxN tile but the file I/O still only writes NxN so this issue becomes easily resolved. Ultimately I believe I much cleaner solution is derived with this methodology.

```
if (rank == MASTER) {
    domainSize = ((N * N - 1) / chunkSize + 1) * chunkSize;

    float *y;
    y = (float *)malloc(domainSize*sizeof(float));

    /* Workers already complete one "chunk" of work without communication */
    startIdx = (ncpu - 1) * chunkSize;

    while (startIdx < domainSize) {
        /* Receive work from worker */
        ierr = MPI_Recv(x, chunkSize, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
            MPI_COMM_WORLD, &status);

        /* copy in work, index stored in tag */
        memcpy(y + status.MPI_TAG, x, chunkSize*sizeof(float));

        /* Send startIdx back to worker we just received from for new work */
        ierr = MPI_Send(&startIdx, 1, MPI_INT, status.MPI_SOURCE, 0,
            MPI_COMM_WORLD);
        startIdx += chunkSize;
    }
```

Each worker performs it's first 'chunk' of computation without any communication from the worker based off `chunkSize` and it's rank, so we start of the variable that keeps track of how many pixels we have calculated (`startIdx`) at `(ncpu - 1) * chunkSize`. I devised of a solution where we only need one `MPI_Send()` and one `MPI_Recv()` for each master and worker iteration. This is achieved by the worker sending through the data (x) to the master with the `startIdx` given in the flag. Since all processes are aware of `chunkSize`, we only need to communicate the starting index of the compute chunk and "both sides" can know exactly which locations on the grid were calculated. It is crucial that `MPI_ANY_SOURCE` is used in the source field of `MPI_Recv()` so we can receive work from any worker. We also require `MPI_ANY_FLAG` as since the flag stores `startIdx` which is different for each communication. The starting index of the incoming work is retrieved through the status struct with `status.MPI_TAG` for the 'loading' in of the data through C's `memcpy()` function. The worker is then sent the new work by simply being sent the new `startIdx`, in the data field this time. The starting index is incremented by chunkSize with all of this being enclosed by a while loop until `startIdx >= domainSize`. Once this while loop is executed we need to make sure that the buffer is cleared and that all workers know to finish.

```
        startIdx = -1;
        loop = 1;
        while (loop < ncpu) {
            ierr = MPI_Recv(x, chunkSize, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
            /* copy in work, index stored in tag */
            memcpy(y + status.MPI_TAG, x, chunkSize*sizeof(float));
            /* Send -1 to worker so it knows to stop */
            ierr = MPI_Send(&startIdx, 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
            loop += 1;
        }
```

We set `startIdx` to -1 so that we indicate to the worker that computation has finished. We have a while loop to ensure that the final chunks come can come in any order from the processes. We receive the final chunk, copy it in and then send the -1 out. The loop variable is incremented until we have processed all of the workers.

```
    if (rank != MASTER) {
        startIdx = (rank - 1) * chunkSize; /* For initial work */

        while (startIdx >= 0) {
            time1 = MPI_Wtime();
            for (loop=0; loop < chunkSize; loop++) {
                pos =  startIdx + loop;
                i=pos/N;
                j=pos%N;
                z=kappa= (4.0*(i-N/2))/N + (4.0*(j-N/2))/N * I;
                k=1;
                while ((cabs(z)<=2) && (k++<MAXITER))
                z= z*z + kappa;

                x[loop]= log((float)k) / log((float)MAXITER);
            }
            time2 = MPI_Wtime();
            calcTime += time2 - time1;

            /* startIdx in tag */
            ierr = MPI_Send(x, chunkSize, MPI_FLOAT, MASTER, startIdx,
                MPI_COMM_WORLD);
            ierr = MPI_Recv(&startIdx, 1, MPI_INT, MASTER, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);

            time1 = MPI_Wtime()
            commTime += time1 - time2;
        }
    }
```

From the worker's end, it receives in the new work specified by `startIdx`, completes it, sends the chunk back with `startIdx` in the flag field and repeats this until it receives the -1 from the master. Overall I believe this to be a fairly clean solution, there is the issue of the added computation but there are several reasons why I think this would trump added dynamic checks/communications required in other solutions. Firstly, in the case of the Mandelbrot Set these redundant computations will be performed in a region that is already rapidly diverging, thus adding minimal time. Secondly, for `chunkSize` sufficiently smaller than `N*N` (we want sufficiently small chunks anyway for load balancing) this amount of redundant computation is negligible compared to the overall size of the grid. Finally, the reduced communication time of only having 1 send/receive per iteration is a notable benefit, particularly for small chunk sizes where the run-time becomes "communication dominant". I achieved the testing of the chunkSizes by isolating the Mandelbrot Set calculation into it's own function and creating a main function which looped over set values of `chunkSize`.

```
    FILE *fp;
    fp = fopen("results-master.csv","w");
    fprintf(fp, "chunkSize, execTime\n");

    int chunks[] = {10, 30, 100, 300, 1000, 3000, 10000, 30000, 100000, 300000,
        1000000, 3000000};
    for (i = 0; i < sizeof(chunks) / sizeof(chunks[0]); i++) {
        if (rank == 0) {
            time1 = MPI_Wtime();
        }
        mb_master_worker(chunks[i]);
        MPI_Barrier(MPI_COMM_WORLD);

        if (rank == 0) {
            time2 = MPI_Wtime();
            execTime = time2 - time1;
            fprintf(fp,"%d,%lf\n", chunks[i], execTime);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
```

Note the importance of the `MPI_Barrier()` - I ran into some large issues with certain processes moving ahead to larger chunk sizes whilst others where still on the previous before I implemnted correct synchronisation.

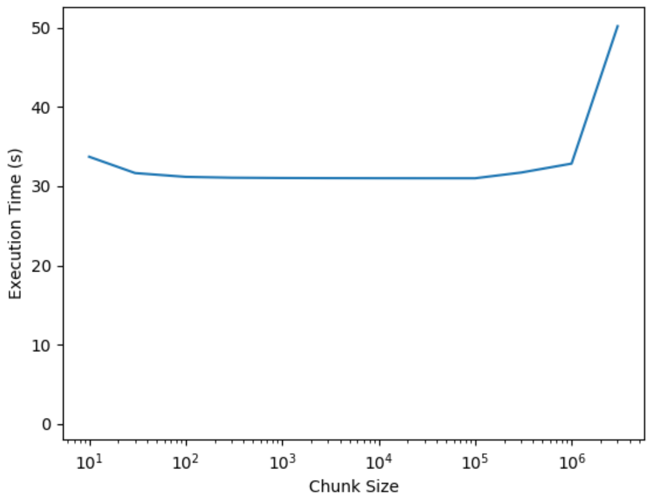
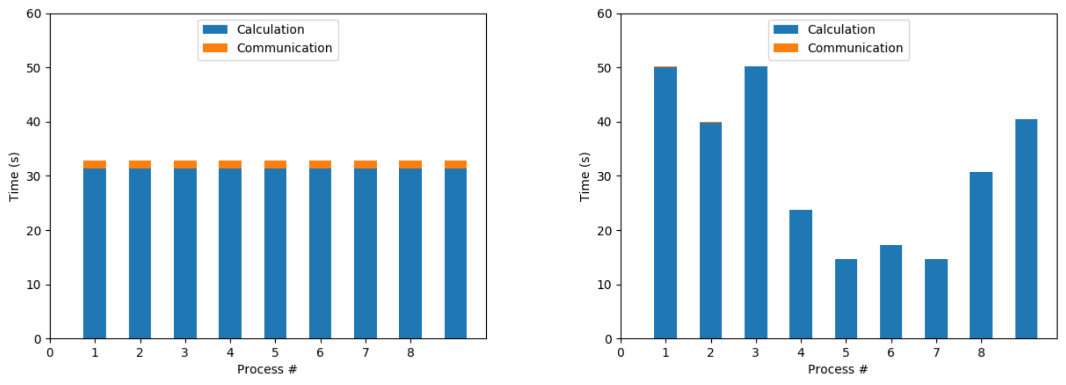


Figure 2: Execution time vs. chunk size, 8000x8000 grid 10 MPI processes

As we can see, the overall execution time begins slightly higher (`chunkSize`=10) due to added communication overhead. The overall time taken is fairly consistent until ≈1,000,000 when it begins to climb. This is because the load balancing is reducing as the chunks get larger and larger - there is less and less opportunity to 'balance' the workload out and processes get stuck in computationally difficult regions whilst others 'race ahead'. This is well illustrated by looking at the calculation and communication times of each process at both ends as can be seen in Figure 3



(a) chunkSize=10 (b) chunkSize=3,000,000

cd

Figure 3: Master-worker inter-process timings with 8000x8000 grid and 10 MPI processes

This affirms that indeed the overhead with small values of `chunkSize` is associated with added communication time whilst overhead with large values is associated with reduced load balancing. It was confirmed that when when `chunkSize` was one tenth of the total grid (6,400,000) the same results were achieved as outlined in Section 3. It is also worth noting that we are seeing lesser performance due to the master process not contributing to the brunt calculations. There is a large rage of chunk sizes that could be deemed suitable, but `chunkSize = 100000` seems to be in the middle of the range and thus would be chosen as the ideal value.

5 Cyclic Partitioning

Cyclic partitioning is a form of static partitioning where the work is distributed in cyclical structures instead of in contiguous blocks. In my implementation each process has an `N * N / ncpu` array, whilst rank 0 also has two `N * N` arrays.

```
procSize = N * N / ncpu;

x = (float *)malloc(procSize*sizeof(float));
if (rank == 0) {
    y = (float *)malloc(N*N*sizeof(float));
    out = (float *)malloc(N*N*sizeof(float));
}
```

Each process works out the pixels to calculate based of it's rank and the number of MPI processes. Processes start at position `rank` and then 'leap' forward `ncpu` steps each iteration (`pos = rank + loop * ncpu`).

```
for (loop=0; loop < procSize; loop++) {
    pos = rank + loop * ncpu;
    i=pos/N;
    j=pos%N;
    z=kappa= (4.0*(i-N/2))/N + (4.0*(j-N/2))/N * I;

    k=1;
    while ((cabs(z)<=2) && (k++<MAXITER))
        z = z*z + kappa;

    x[loop]= log((float)k) / log((float)MAXITER);
}
```

We now have all of our work computed and will bring them to a single process using `MPI_Gather()`

```
MPI_Gather(x, procSize, MPI_FLOAT, y, procSize, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

however our data is out of order, so we must rearrange this into the correct order. Since `MPI_Gather()` can only bring data back in contiguously I decided to go with an 'out-of-place' rearrange solution. I did look into solutions, including functions like `MPI_Gatherv`, that had the potential to not require this extra memory/processing to no avail. The algorithm below selects data from the old (non-arranged) array into a new array.

```
for (i = 0; i < procSize; i++) {
    for (j = 0; j < ncpu; j++) {
        out[i*ncpu + j] = y[i + j*procSize];
    }
}
```

I created a test program to help develop and display the logic of the rearrange (`https://github.com/LDRyan0/mandelbrot/blob/main/concepts/cyclic.c`. The program shows each process indicated at the start of each print statement, their cyclical indices in their host arrays `x`, the gather together into array `y` and 'out-of-place' rearrange into `out`.

```
(0): x = [  0  4  8 12 16 20 ]
(1): x = [  1  5  9 13 17 21 ]
(2): x = [  2  6 10 14 18 22 ]
(3): x = [  3  7 11 15 19 23 ]

(0): y = [  0  4  8 12 16 20  1  5  9 13 17 21  2  6 10 14 18 22  3  7 11 15 19 23 ]
(0): z = [  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 ]
```

Cyclic partitioning had fantastic load balancing, as can be seen in Figure 4. Each process spending nearly the exactly the same amount of time calculating, with very little time waiting. Cyclic outperformed the serial implementation by $\approx$ 10x, as would be expected for an ideally partitioned system. These results are far better than those achieved by block partitioning ($\approx$3.75x improvement), where the work distribution placed computationally expensive areas on single processes. Cyclic partitioning is also more highly performant than master-worker parallelism, predominantly due to the fact that all 10 processes are running unlike the master which is retracted from computation.

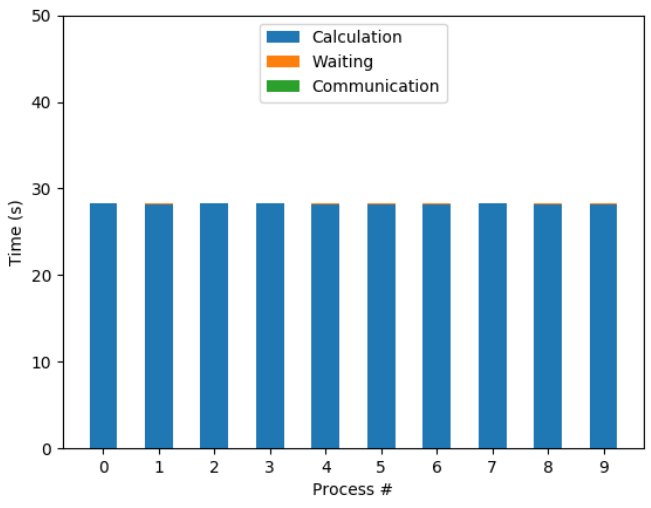


Figure 4: Timing results for static cyclic decomposition on 8000x8000 grid with 10 MPI processes

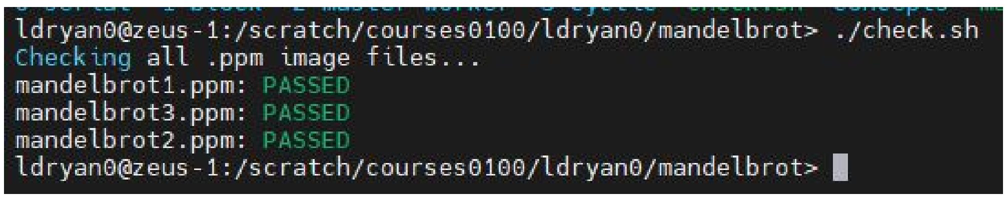
6 Additional Work

6.1 Demonstration Programs

I created a variety of scripts to ensure my understanding of key topics, available at `https://github.com/LDRyan0/mandelbrot/tree/main/concepts`. These include the `memcpy()` functionality, the added domain size calculation for the master-worker partitioning, the logic of the cyclic implementation and a basic demonstration of `MPI_Gather`.

6.2 Checking Script

I created a bash script that checks the .ppm files against the .ppm file generated by the serial implementation, utilising the `diff` command. Sample output:



```
ldryan0@zeus-1:/scratch/courses0100/ldryan0/mandelbrot> ./check.sh
Checking all .ppm image files...
mandelbrot1.ppm: PASSED
mandelbrot3.ppm: PASSED
mandelbrot2.ppm: PASSED
ldryan0@zeus-1:/scratch/courses0100/ldryan0/mandelbrot>
```

Figure 5: check.sh sample output

7 Conclusion

Three partitioning techniques were demonstrated. Block partitioning was found to have very high load imbalance and resulted in many of the process waiting for nearly the entire duration of the execution time. Master-worker was the most difficult solution to implement but found extremely consistent good performance across as large range of chunk sizes, tapering off at either of the extremes. Cyclic partitioning was found to be the best solution for the Mandelbrot Set problem, it's load balancing was fantastic, it produced an extremely clean and maintainable solution with excellent performance.

8 Appendix: Raw Results

The following are the .csv output files with all raw timing information.

8.1 Block

```
rank,calcTime,waitTime,commTime
0,0.824172,104.459228,0.111175
1,5.520548,99.763341,0.015987
2,34.281864,71.001933,0.024628
3,58.868919,46.414938,0.035129
4,105.283648,0.000136,0.043654
5,74.836979,30.446883,0.056611
6,1.241128,104.042655,0.065021
7,0.803758,104.480074,0.079537
8,0.685165,104.598618,0.087838
9,0.531178,104.752675,0.111074
```

8.2 Master Worker

```
chunkSize, execTime
10,33.707487
30,31.645738
100,31.170377
300,31.063463
1000,31.025801
3000,31.010219
10000,30.999505
30000,30.994260
100000,30.992962
300000,31.712996
1000000,32.842545
3000000,50.180797
```

8.3 Cyclic

```
rank,calcTime,waitTime,commTime
0,28.243030,0.002635,0.085039
1,28.231218,0.015308,0.010280
2,28.246393,0.000129,0.018700
3,28.244473,0.002019,0.026974
4,28.203907,0.042611,0.035103
5,28.187266,0.059208,0.043704
6,28.175890,0.070551,0.052007
7,28.239515,0.007012,0.060420
8,28.224188,0.022284,0.068636
9,28.181145,0.065272,0.085047
```