HPC (PHYS4004)

Assignment 1

Alec R. Dorrington

Student ID 19133047

Q1. Static decomposition

The source code for this task is located in 'mandelbrot_block.c'. This can be compiled on Zeus using 'mpicc -lm mandelbrot_block.c -o mandelbrot_block'. To run the executable, use 'srun -n X mandelbrot_static', where X is the number of processors. The image resolution, output file and maximum iterations are defined in the source file as preprocessor constants. By default, these values are 4000x4000, 'mandelbrot.ppm' and 1000 respectively.

Using static decomposition, that is, where the problem is divided evenly among all processors once at the start, one might expect good parallelization to occur. Interestingly, this isn't the case, due to the fact that not all equal-sized parts are equally difficult to evaluate. In this case, almost all of the work is done by processors 3, 4, 5 and 6. The complex plane is divided into 10 horizontal strips, each of which is assigned to processors 0 to 9 from top to bottom. The central region occupied by strips 3 through 5 is where most of the mandelbrot set is located. When a value *is* included in the mandelbrot set, up to 'MAX_ITR' iterations are required to validate this fact. Whereas values further away can be excluded from the set after just a couple of iterations. This accounts for the discrepancy between the work done by each of the processors. Ideally, the size of a strip would vary inversely with the difficulty of processing the strip, however, this isn't necessarily known in advance. As it stands, the program makes very ineffective use of parallelisation.

Below are the timings for a 10-processor trial run, with 'overhead time' being the time spent waiting behind communication locks. Notice how all workers end up waiting for their slowest sibling anyway.

Processor	Pixels	Processing Time (secs)	Overhead Time (secs)
0	1600000	0.159	24.422
1	1600000	0.252	0.003
2	1600000	0.593	0.002
3	1600000	9.998	0.002
4	1600000	24.499	0.002
5	1600000	24.559	0.004
6	1600000	10.000	14.567
7	1600000	0.641	23.922
8	1600000	0.247	24.329
9	1600000	0.208	24.358

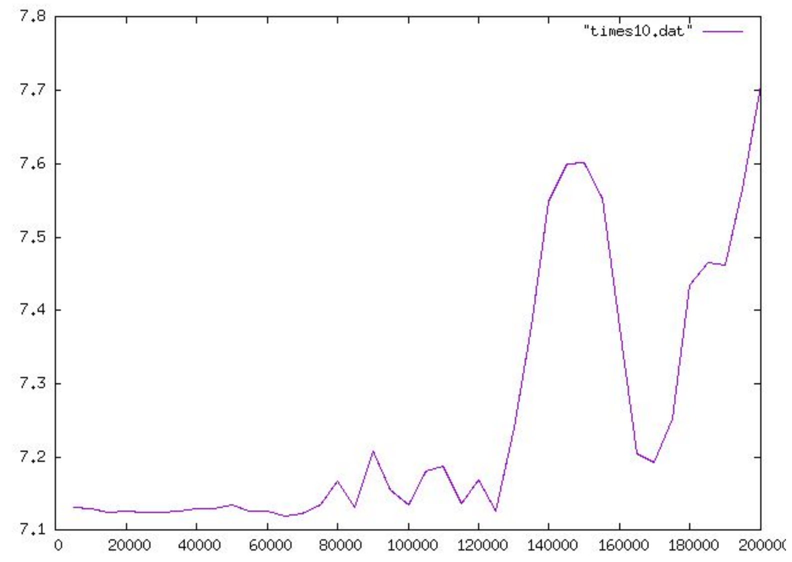
In this trial, writing the results to the file took 4.091 seconds, during which time the workers accomplished nothing. This task cannot be effectively parallelised, but that also doesn't matter since (a) it takes little time compared to the computation itself and (b) it only has to happen once.

Q2. Dynamic decomposition

The source code for this task is located in 'mandelbrot_dynamic.c'. This can be compiled and run much like the static version. An additional BLOCK_SIZE preprocessor constant is included in this version.

To solve the shortcomings associated with static decomposition, dynamic decomposition is used instead. Using a fixed block size, the task is broken up into parts and distributed among the variable processors. This time, when a processor has finished its task, it receives a new task from the master process. This ensures all processors remain fully utilised. A smaller block size maximises processor utilisation, whereas a larger block size minimises communication overhead. Some balance between the two will produce optimal results, depending on the specific problem, and the number of processors.

The total time taken is graphed against the block size below, for the 10-processor case. It appears that, within limits, a small block size is ideal (around 20,000 to 40,000), and that the communication overhead is negligible as compared to the wasted cycles caused by larger blocks. The large dip shortly after the '160,000' mark can likely be explained as 160,000 (the block size) * 10 (the number of workers) being a factor of 16,000,000, the total number of pixels, ensuring an even distribution between processors.

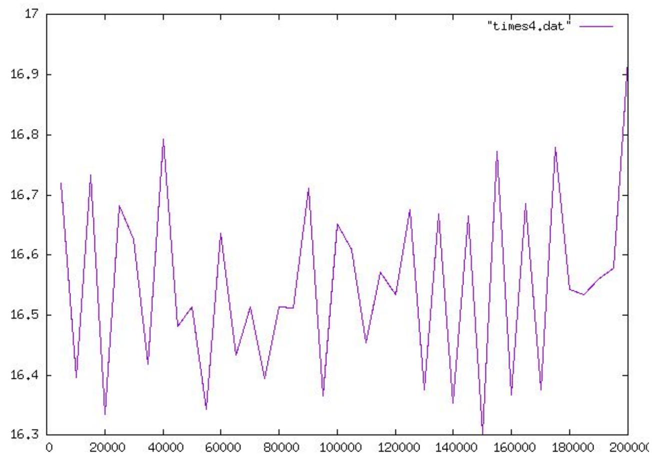


If there are n workers, and the block size is equal to 1/nth of the total problem size, this effectively becomes static decomposition. To test this, we can try a dynamic decomposition with 10 workers (11 processors total) and a block size of 1600000 pixels, 1/10th of the total number of pixels. This took 24.516 seconds, roughly equal to the time taken by the slowest processor in the static decomposition case, 24.559 seconds (processor 5).
This task is very effectively parallelizable, since regions can be computed independently. This means that varying the number of processors by some factor will offer a nearly equivalent amount of speedup compared to the serial case.

Q3. Amdahl's law

Amdahl's law refers to speedup of a program upon parallelisation when only part of the problem can be parallelised. The parallel part can be made to take an arbitrarily small amount of time, given an arbitrarily large number of processors, however, the serial portion of the program remains a limiting factor. Assuming perfectly efficient parallelisation, the time taken will be inversely proportional to the number of processors used. However, the time taken by the serial portion of the program must be added to the total time taken. From this, Amdahl's law can be derived. Where P is the portion of the program which can benefit from speedup, N is the number of processors and S is the expected speedup relative to the serial case, Amdahl's law states that S = 1 / ((1-P) + P/N).

In the specific case where the block size is 50,000 and the total image size is 4,000x4,000, when the program runs serially, file IO, the portion of the problem which can't be parallelised, took 4.139 seconds, or about 6.2% of the total time, giving a P value of 100% - 6.2% = 93.8%. This means that, with an unlimited number of processors, the maximum possible value of S, the relative speedup, becomes 1 / 0.062 = 16.2-fold. Meaning, the program can never be made to run any more than 16 times faster, given the file IO bottleneck.



To compare the actual speedup achieved in the 4 and 10 worker cases, for instance, we must first determine the optimal chunk size for the 4 worker case, just as we did for the 10 worker case before. Interestingly, the chunk size graph for the 4 worker case is a lot more chaotic than that of the 10 worker case. I suspect this has something to do with how a smaller number of workers will be more frequently able to divide the problem *evenly* into chunks of a given size. While it's difficult to see any trend in this data, for the sake of argument I will use the fastest chunk size tested, 150,000.

According to Amdahl's law, the 4 and 10 processor cases represent, respectively, speedups of 1/(0.062+(0.938/4)) = 3.37x and 1/(0.062+(0.938/10)) = 6.62x. Given that these cases take 20.17 and 11.26 seconds, giving actual speedups of 66.76/20.17 = 3.31x and 66.76/11.26 = 5.93x, where 66.76 seconds is the time taken in the serial case, the results given by Amdahl's law appear reasonable.

Q4. Arbitrary chunksize

The code for Q2 was originally written so as to allow for chunk sizes which do not evenly divide W * H, so no additional modification was `necessary`. This was easily accomplished by letting the final chunk be smaller than the rest if necessary. The entire source code from Q2 is given below.

```
// Mandelbrot set visualiser using MPI for use on a supercomputer.
// Written by Alec Dorrington for HPC at Curtin University.
// Uses dynamic decomposition.

// Horizontal pixel resolution of image.
#define WIDTH 4000

// Vertical pixel resolution of image.
#define HEIGHT 4000

// Maximum number of iterations before giving up.
#define MAX_ITR 1000

// Number of pixels to process in each chunk.
#define CHUNK_SIZE 50000

// Output image file.
#define OUTPUT "mandelbrot.ppm"

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include <mpi.h>

// Compute the number of mandelbrot iterations
// required before a value escapes to infinity.
int mandelbrot(float complex z, int maxItr) {

    float complex z0 = z;
    int itr = 0;

    // A value is guaranteed to escape to infinity
    // once its absolute value equals or exceeds 2.

    // If this never occurs, stop after maxItr.

    while(cabs(z) < 2.0 && ++itr < maxItr) {
        z = z * z + z0;
    }
    return itr;
}

// Convert an index to a value in the complex plane, within the region
// -2 < x < 2, -2 < y < 2, with the given horizontal/vertical resolution.
float complex toComplex(int i, int width, int height) {

    // Convert index to cartesian coordinates.
    int x = i % height;
    int y = i / height;

    // Convert coordinates to complex number.
```

```c
    float re = 4.0F * ((float) x + 0.5F) / (float) width - 2.0F;
    float im = 4.0F * ((float) y + 0.5F) / (float) height - 2.0F;

    return re + im * I;
}

// Compute the number of mandelbrot iterations for all pixels in a range.
int* processChunk(int base, int limit, int width, int height, int maxItr) {

    int *results = malloc(limit * sizeof(int));
    int i;

    // For each pixel in this block.
    for(i = 0; i < limit; i++) {
        results[i] = mandelbrot(toComplex(base + i, width, height), maxItr);
    }
    return results;
}

// Output the results of the computation to a PPM (image) file.
void writeFile(char *name, int *image, int width, int height, float maxItr) {

    FILE *file = fopen(name, "w");
    int i, r, g, b, x;

    // Write image resolution.
    fprintf(file, "P3\n%d %d\n255\n", width, height);

    // For each pixel.
    for(i = 0; i < width * height; i++) {
        // Determine colour based on number of iterations.
        x = (int) (255.0F * log1p(image[i]) / log(maxItr));
        r = image[i] < 127 ? 255 - 2*x : 0;
        g = image[i] < 127 ? 2*x : (511 - 2*x);
        b = image[i] < 127 ? 0 : (2*x - 255);
        // Write colour in (r, g, b) format.
        fprintf(file, "%3d\n%3d\n%3d\n", r, g, b);
    }
    fclose(file);
}

// Print statistics.
void printStats(int workers, int width, int height,
        int chunkSize, int maxItr, char *file, double time) {

    printf("  IMAGE SIZE: %dx%d px\n", width, height);
    printf("  CHUNK SIZE: %d px\n", chunkSize);
    printf("  WORKERS: %d (+1)\n", workers);
    printf("  MAX ITERATIONS: %d\n", maxItr);
    printf("  FILE NAME: %s\n", file);
    printf("  TOTAL TIME: %2.3f secs\n", time);
}

// Entry point for master process.
void master(int workers, int width, int height,
        int chunkSize, int maxItr, char *file) {

    MPI_Status status;
    int base = 0, source, offset, completed = 0;
    int *image = (int*) malloc(width * height * sizeof(int));
    int *tasks = (int*) calloc(workers * 2, sizeof(int));
    double t0 = MPI_Wtime();

    // While there remains work to be done.
    while(base < width * height || completed++ < workers) {
        // Wait for worker to send chunk.
        MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

        source = status.MPI_SOURCE;
        offset = (source - 1) * 2;

        // Load new chunk into existing image array.
        MPI_Recv(image + tasks[offset], tasks[offset+1], MPI_INT,
                MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

        // Create a new task for the worker to complete next.
        tasks[offset] = base;
        tasks[offset+1] = chunkSize < width * height - base ?
               chunkSize : width * height - base;

        // Send new task to worker.
        MPI_Send(tasks + offset, 2, MPI_INT, source, 0, MPI_COMM_WORLD);
        base += tasks[offset+1];
    }
    // Print statistics, and write image to file.
    printStats(workers, width, height, chunkSize,
            maxItr, file, MPI_Wtime() - t0);
    writeFile(file, image, width, height, maxItr);
}

// Entry point for worker process.
void worker(int rank, int size, int width, int height, int maxItr) {

    MPI_Status status;
    int task[2] = {0, 0}, *results;

    // While there remains work to be done.
    while(task[0] < width * height) {

        // Send the previous chunk (if any) to master.
        MPI_Send(results, task[1], MPI_INT, 0, 0, MPI_COMM_WORLD);

        // Receive next task from master.
        MPI_Recv(task, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

        if(results) free(results);
        // Perform calculations for this chunk.
        results = processChunk(task[0], task[1], width, height, maxItr);
    }
    if(results) free(results);
}

// Mandelbrot visualiser.
int main(int argc, char *argv[]) {

    int rank, size;

    // Initialise MPI.
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Call master() or worker() accordingly.
    if(rank == 0) master(size-1, WIDTH, HEIGHT, CHUNK_SIZE, MAX_ITR, OUTPUT);
    else worker(rank, size, WIDTH, HEIGHT, MAX_ITR);

    MPI_Finalize();
    return 0;
}
```

Q5. Idle time

The masters job is to allocate tasks to the workers. In my implementation, the master does none of the mandelbrot calculations itself. Therefore, it stands to reason that the master would spend most of its time waiting for workers to send calculation results and request more work. Furthermore, it makes sense that, the less workers there are, the easier it is for the master to keep up, and so the more time it spends waiting for communications. In the extreme case where there are sufficiently many workers to keep the master busy, the master would spend almost no time waiting for incoming messages (assuming adequate buffering). This condition certainly is not met. In the case where there is a master and one single worker, the master spent 62.72 of 63.00 seconds (excluding file IO), 99.6% of the time, waiting for incoming messages. When there are 10 workers, this number becomes *slightly* less extreme, but still very significant, with the master spending 6.72 of 7.02 seconds, or 95.7% of the time, waiting for messages.

Q6. Cyclic decomposition

The source code for this task is located in 'mandelbrot_cyclic.c'. Cyclic partitioning is another form of static partitioning, except instead of dividing the plane up into blocks to assign to each processor, individual pixels are allocated to the processors in a repeating, cyclical pattern. This serves to prevent unbalanced work distributions, since no single processor is restricted to any particular region. On average, each processor should receive an even mix of easy and difficult pixels, just by the law of large numbers. But this approach also has an advantage over dynamic partitioning, in that it cuts back on unnecessary communication overhead. So, in theory, this technique should perform the best.

Ignoring file IO, dynamic decomposition with 11 processors took about 6.62 seconds, using an optimal chunk size. Cyclic decomposition, on the other hand, took about 7.33 seconds. This is within a close margin of error, and the difference between the two is negligible. Of course, both dynamic decomposition and cyclic decomposition are far better than block decomposition. As predicted, cyclic decomposition has a slight advantage over dynamic decomposition. However, since this difference is negligible, and, in my opinion, dynamic decomposition was easier to implement, it remains my prefered strategy.