CS 5500 HW6: Mandelbrot

Marshal Taylor

March 30, 2022

Abstract

A concise report describing my implementation if Parallel Mandelbrot

1 Introduction

2 Commands

1) Compile: VS code Build

```
2) mpiexec -np {number of processors} .\LoadBalancing.exe
```

3 Implementation

This program is quite a bit more complex than the other programs that we have tackled in class. If we are process 0, we need to generate the total number of tasks and send that information out to each of the processes.

There are 6 kinds of messages that this program can send. 1) Passed work over to another processor
2) Pass the tokens
3) Completed all work
4) Initial total tasks
5) Current total tasks
6) Updating total tasks
Each processor have a vector of a backlog of work. If the processor has an item in the vector, it will spend the time necessary to do one unit or work. If the processor is odd, it will create 1-3 new tasks. If any processor has a backlog of greater than 16, it will randomly pick a new processor to send a unit of work to, popping it off the back of the vector. If a processor has no work to do, it increments the idle tick variable, tracking how long the processor was without work.

The dual pass user initiated terminating system is pretty easy. If processor zero is without work, it sends the token to the next processor. If the processor ever sent work back, the token then becomes black. If the token is returned black to processor 0, it sends the token again. Once the token is returned white, processor 0 sends out a message to each processor to terminate as all work is done.

3.1 Load Balancing

```
int main(int argc, char** argv) {
    int rank, size;
    srand(1969);
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MCW, &rank);
    MPI_Comm_size(MCW, &size);

    MPI_Request request;
    MPI_Status status;
    int flag = 0;
```

```
vector<int> work_queue;
int work_queue_max = 16;

int work = 0;
int work_max = 0;
int total_tasks = 0;

int random_location = rank;

int tasks_to_generate = 0;

int tasks_generated = 0;
int tasks_generated_max = 0;

int WHITE = 0;
int BLACK = 1;
int color = WHITE; //0 = White, 1 = Black
int token_color = WHITE;
bool has_token = true;
bool finalized = false;

int idle_ticks = 0;

int recv_process = rank - 1;
auto start = high_resolution_clock::now();

if (!rank) {

    recv_process = size - 1;
    tasks_generated_max = rand() % 1024 + 1024;
    cout << "# " << tasks_generated_max << endl;
    for (int i = 1; i < size; i++) {
        MPI_Send(&tasks_generated_max, 1, MPI_INT, i, 4, MCW);
    }
}
else {
     MPI_Recv(&tasks_generated_max, 1, MPI_INT, 0, 4, MCW, MPI_STATUS_IGNORE);
}

while (1) {

    if (!rank) {
        MPI_Iprobe(MPI_ANY_SOURCE, 6, MCW, &flag, &status);
        if (flag) {
            while (flag) {
                tasks_to_generate = 0;
                MPI_Recv(&tasks_to_generate, 1, MPI_INT, MPI_ANY_SOURCE, 6, MCW, MPI
                total_tasks += tasks_to_generate;
                MPI_Iprobe(MPI_ANY_SOURCE, 6, MCW, &flag, &status);
            }
            for (int i = 0; i < size; i++) {
                if (i % 2 == 1) {
                    MPI_Send(&total_tasks, 1, MPI_INT, i, 5, MCW);
                }
            }
```

```
    }
}
MPI_Iprobe(0, 5, MCW, &flag, &status);
while (flag) {
    MPI_Recv(&total_tasks, 1, MPI_INT, 0, 5, MCW, MPI_STATUS_IGNORE);
    MPI_Iprobe(0, 5, MCW, &flag, &status);
}

MPI_Iprobe(0, 3, MCW, &flag, &status);
if (flag) {
    MPI_Recv(&work, 1, MPI_INT, 0, 3, MCW, MPI_STATUS_IGNORE);
    break;
}


MPI_Iprobe(MPI_ANY_SOURCE, 1, MCW, &flag, &status);
while (flag) {
    MPI_Recv(&work, 1, MPI_INT, MPI_ANY_SOURCE, 1, MCW, MPI_STATUS_IGNORE);
    //cout << rank << " added work: " << work << endl;
    work_queue.push_back(work);
    MPI_Iprobe(MPI_ANY_SOURCE, 1, MCW, &flag, &status);
}

if (work_queue.size() > work_queue_max) {
    //cout << rank << " too much work " << endl;
    for (int i = 0; i < 2; i++) { /* execute twice*/
        random_location = rand() % size;
        while (random_location == rank) {
            random_location = rand() % size;
        }
        work = work_queue.back();
        //cout << rank << " sent " << work << " to " << random_location << endl;
        MPI_Send(&work, 1, MPI_INT, random_location, 1, MCW);
        work_queue.pop_back();

        if (random_location < rank) { /* turns process black if it sends backwar
            //cout << "BLACK" << endl;
            color = BLACK;
        }
    }
}

if (work_queue.size()) {
    work = work_queue.back();
    work_max = work * work;

    //cout << rank << " doing work " << work << endl;

    for (int i = work; i < work_max; i++) {
        work++;
    }
    work_queue.pop_back();
}

if (rank % 2 == 1) {
    //cout << total_tasks << endl;
```

```
    if (total_tasks < tasks_generated_max) {
        tasks_to_generate = rand() % 3 + 1;
        MPI_Send(&tasks_to_generate, 1, MPI_INT, 0, 6, MCW);

        for (int i = 0; i < tasks_to_generate; i++) {
            if (total_tasks < tasks_generated_max) {
                work = rand() % 1024;
                //cout << rank << " creating work " << work << endl;
                work_queue.push_back(work);
                total_tasks++;
            }
            else {
                signed int x = -1 * work_queue.size();
                work_queue.clear();
                MPI_Send(&x, 1, MPI_INT, 0, 6, MCW);
            }
        }
    }
    else {
        signed int x = -1 * work_queue.size();
        work_queue.clear();
        MPI_Send(&x, 1, MPI_INT, 0, 6, MCW);
    }
}

if (rank == 0 && work_queue.size() == 0 && has_token) {
    cout << "**************************** 0 sending " << (rank + 1) % size << "
    MPI_Send(&token_color, 2, MPI_INT, (rank + 1) % size, 2, MCW);
    has_token = false;
}

if (work_queue.size() == 0) {
    idle_ticks++;
    MPI_Iprobe(recv_process, 2, MCW, &flag, &status);
    if (flag) {
        MPI_Recv(&token_color, 2, MPI_INT, recv_process, 2, MCW, MPI_STATUS_IGNOR

        cout << " ********************************** " << rank << " Recieved

        if (color == BLACK) {
            token_color = BLACK;
        }

        if (token_color == BLACK) {
            token_color = BLACK;
            color = WHITE;
        }

        if (!rank && token_color == WHITE) {
            cout << "Done" << endl;
            finalized = true;
            break;
        }

        if (!rank) {
            token_color = WHITE;
```

```
                }

                MPI_Send(&token_color, 2, MPI_INT, (rank + 1) % size, 2, MCW);
            }
        }

        if (!rank && token_color == WHITE && finalized) {
            break;
        }
    }

    if (!rank) {
        auto stop = high_resolution_clock::now();
        for (int i = 1; i < size; ++i) {
            work = -1;
            MPI_Send(&work, 1, MPI_INT, i, 3, MCW);
        }
        cout << "# " << total_tasks << endl;
        auto duration = duration_cast<microseconds>(stop - start);
        cout << "Duration " << duration.count() << endl;
    }
    cout << rank << " " << work_queue.size()<< " Idle Ticks: "<<idle_ticks << endl;
    MPI_Finalize();
    return 0;
}
```

3.2 Timings

Listed below is timings for the entirety of the file to run in microseconds. 2 Processors: 250685
4 Processors: 188518*
8 Processors: 208933
16 Processors: 1734902
*4 Processors was the quickest on my PC.

4 Expected Output

A command output with the total number of tasks.
A log of the the dual pass algorithm with black and white tokens.
The total number of ticks where the processor had a work queue of zero, and the final size of each processors work queue (should always be 0)
The total number of tasks completed.
The time of the whole program.