1. **알고리즘 구현 방법**

**Static, dynamic 공통사항:**

한번에 처리한 연산량은 블락사이즈인 N\_Rectangle\*height=40\*400=16000으로 하였다.

Slave가 처리한 데이터는 display의 좌표이다 (Real과 imaginary 축 좌표가 아니라 비율을 고려해 display 좌표 (0~400)까지 처리해 보낸다.)

Display는 Master 프로세서가 이미지파일을 만들고 종료하는 것이 아니라, 본인이 직접 그래픽창을 띄우는 형식으로 하였다.(그래픽창 종료시 Master 프로세서 종료)

**Static**

static의경우 가능한 한 균등하게 블락을 배분하기 위해 다음과 같은 과정을 거쳤다. 먼저 블락수= width/N\_rectangle 로 구하고, MPI\_Size를 이용해 프로세서의 개수를 샌 다음에 프로세서 수를 블락수로 나눈 몫 Q 와 나머지 R을 구한다.

Master 프로세서가 본인에게 Q+1개의 블락을 할당하고 다른 R-1개의 프로세서에게 Q+1 개의 블락을 할당시키고 나머지 프로세서에게는 Q개의 프로세서를 할당한다. 이때 Slave에게는 시작하는 column의 위치와 처리해야하는 블락수를 보낸다.

Slave 프로세서는 블락수와 시작할 위치를 Recv받고 1블락을 처리할 때마다 블락 크기만큼의 데이터를 Master 프로세서 에게 send한다. 각 좌표를 처리해서 보내는 것이 아니라 통신량을 줄이기 위하여 한 번 send시에 블락크기만큼의 데이터를 한번에 보낸다. 그리고 배열의 마지막에 시작위치를 함께 보내 Mater 프로세서가 어디서부터 처리해야하는지 알 수 있도록 한다. 할당받은 블락을 다처리한 slave는 MPI\_Finalize()를 통해 종료시킨다.

Master 프로세서는 이 후 Q-1만큼의 블록을 Recv 후 데이터배열에 저장할 때마다 본인의 블록을 1개를 처리해 데이터 배열에 저장한다. 이는 다른 프로세서가 Send가 미뤄지지 않고 마스터 프로세서 또한 idle한 상태가 되지 않기 위해서다. (만일 마스터프로세서가 본인의 프로세서를 전부처리하고 오면 다른 프로세서는 Send를 기다리게 된다. 반대로 다른 프로세서의 send를 전부 기다리고 본인 것을 처리할 시 다른 프로세서가 처리하는 동안 마스터 프로세서가 놀게 된다.) 이는 마스터프로세서와 다른 프로세서의 처리시간이 비슷하다고 가정하기 때문에 가능하다.

그렇게 모든 블락의 처리가 끝나고나면 Master 프로세서는 혼자 남아 SFML모듈을 이용해 처리한 데이터를 display시킨다. 만일 display된 그림을 종료하게 되면 master 프로세서의 역할이 끝나고 모든 프로세서가 종료하게 된다

**Dynamic:**

다이나믹의 경우 싱글프로세서와 멀티프로세서일 때 마스터 프로세서의 역할이 다르다.

싱글프로세서일 경우 통신이 필요 없고 연산을 해줄 다른 프로세서가 없으므로 Master 프로세서 본인이 모든 연산을 처리한다. 모든 연산처리 후 display도 도맡아 한다.

Static의 경우와 다르게 다이나믹은 Master 프로세서가 다른 프로세서의 요청을 항상 기다리게 하였다. 이는 마스터 프로세서가 데이터 처리시 다른 프로세서의 작업이 밀릴 수 있기 때문에 그렇게 하였다. 따라서 멀티프로세서일 때 마스터프로세서는 연산을 처리하지 않고 오직 통신과 데이터 배열 저장, display를 위해 사용된다.

멀티프로세서의 경우 먼저 Master 프로세서는 나머지 Slave에게 블락을 하나씩 할당한 후 Recv()로 slave의 처리를 기다린다. 그리고 Slave의 연산이 결과가 오면 Master는 그 slave에게 다음 블락을 할당한 후, 방금 받았던 데이터를 전부 데이터 배열에 저장한다. Slave 로 받은 데이터의 크기는 블락사이즈보다 2만큼 더 큰 데 나머지 두칸은 어떤 Slave로부터왔는지 slave의 인덱스가 담겨져 있고 다음 나머지 한 칸에는 어떤 블락이 처리되었는지 알기 위해 해당 블락의 첫번째 column이 담겨져있다. 이렇게 모든 블락을 할당하고 나면 Master 프로세서는 나머지 프로세서에게 종료신호를 준다. 종료신호는 slave에게 column의 마지막 좌표인 width(즉 400)을 줌으로써 실현한다. 종료신호를 보낼 때 마다 count를 줄여 모든 count가 종료되면 통신을 종료한다.

Slave 프로세서는 본인이 처리해야할 블락의 첫번째 column을 Recv받고 블락을 처리 후 블락사이즈 +2 만큼의 데이터를 보낸다. 마지막 두개는 위에 언급했듯이 시작위치와 자신의 인덱스이다. 데이터를 보낸 후 곧 바로 마스터로부터 다음블록 할당을 기다리고, 만약에 종료신호가 온다면 slave는 프로세서를 종료한다.

이후 Master 프로세서는 혼자남아 display를 붙들고 있고, display된 그래픽창이 종료되면 Master 프로세서가 종료되며 모든 프로세서가 종료된다.

1. Static code

코드의 경우 cpp 파일을 함께 추가로 첨부하였습니다.

#include <stdio.h>

#include <SFML/Graphics.hpp>

#include <mpi.h>

#include <time.h>

using namespace sf;

int width = 400;

int height = 400;

int N\_rectangle = 40;

struct complex {

float real;

float imag;

};

int cal\_pixel(complex c) {

int count = 0;

int max = 256;

complex z;

z.real = 0;

z.imag = 0;

float temp, lengthsq;

do {

temp = z.real \* z.real - z.imag \* z.imag + c.real;

z.imag = 2 \* z.real \* z.imag + c.imag;

z.real = temp;

lengthsq = z.real \* z.real;

count++;

} while ((lengthsq < 4.0) && (count < max));

return count;

}

int temblock[16001];

int data[400][400];

int block[16001];

int main(int argc, char\*\* argv) {

MPI\_Init(&argc, &argv);

int myid;

int numprocs;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numprocs);

MPI\_Status status;

float minR = -2;

float minI = -2;

float maxR = 2;

float maxI = 2;

float scale\_real, scale\_imag;

scale\_real = (maxR - minR) / width;

scale\_imag = (maxI - minI) / height;

complex c;

int color;

if (myid == 0) {

clock\_t total\_start\_time, total\_end\_time;

total\_start\_time = clock();

int column = 0;

int numblock = width / N\_rectangle;

int blocksize = N\_rectangle \* height;

int R = numblock % numprocs;

int Q = numblock / numprocs;

int numSlaveBlock = 0;

if (numprocs > 1) {

int B;//각프로세스에 할당되는 블락수

for (int i = 1; i < numprocs; i++) {

if (i < R) {

B = Q + 1;

}

else {

B = Q;

}

numSlaveBlock += B;

MPI\_Send(&B, 1, MPI\_INT, i, 1, MPI\_COMM\_WORLD);

MPI\_Send(&column, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);

column = column + N\_rectangle \* B;

}

}

int s = column;

int e;

clock\_t starttime, endtime;

int index = 0;

int start, end, cur;

for (int i = 0; i < numblock; i++) {

//본인블록을 처리하는 루틴

//프로세스 갯수만큼 처리할때마다 마스터는 본인도 하나의 블록을 처리함.

if (index == 0) {

starttime = clock();

for (int y = 0; y < height; y++) {

c.imag = minI + ((float)y \* scale\_imag);

e = s + N\_rectangle;

for (int x = s; x < e; x++) {

c.real = minR + ((float)x \* scale\_real);

color = cal\_pixel(c);

data[y][x] = color;

}

}

endtime = clock();

printf("process %d complete a Block, time: %fs\n", myid, (double)(endtime - starttime) / CLOCKS\_PER\_SEC);

s += N\_rectangle;

}

// 다른 블록으로부터 데이터를 받아 저장하는 루틴

else {

MPI\_Recv(&temblock, blocksize + 1, MPI\_INT, MPI\_ANY\_SOURCE, 2, MPI\_COMM\_WORLD, &status);

start = temblock[16000]; //0~15999는 데이터, 16000은 블락의 시작위치

end = start + N\_rectangle;

cur = 0;

for (int j = 0; j < height;j++)

for (int k = start; k < end;k++) {

data[j][k] = temblock[cur];

cur += 1;

}

}

// index는 마스터가 본인차례를 알기 위해 사용

index = (index + 1) % numprocs;

}

total\_end\_time = clock();

printf("\ntotal time in static allocation:%fs\n\n", (double)(total\_end\_time - total\_start\_time) / CLOCKS\_PER\_SEC);

//마스터가 처리한 데이터를 display하는 과정

RenderWindow window(VideoMode(width, height), "Mandlebrot set");

Image image;

image.create(width, height);

Texture texture;

Sprite sprite;

while (window.isOpen()) {

Event event;

while (window.pollEvent(event))

{

if (event.type == Event::Closed) window.close();

}

for (int y = 0; y < height; y++)

for (int x = 0; x < width; x++) {

image.setPixel(x, y, Color(255 - data[y][x], 255 - data[y][x], 255 - data[y][x]));

}

window.clear();

texture.loadFromImage(image);

sprite.setTexture(texture);

window.draw(sprite);

window.display();

}

}

//slave

else {

clock\_t starttime, endtime;

double term;

starttime = clock();

int BN;

int start;

MPI\_Recv(&BN, 1, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&start, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &status);

int cur;

for (int k = 0; k < BN;k++)

{

cur = 0;

block[16000] = start; //0~15999는 데이터, 16000은 블락의 시작위치

for (int y = 0; y < height; y++) {

c.imag = minI + ((float)y \* scale\_imag);

for (int x = start; x < start + N\_rectangle; x++) {

c.real = minR + ((float)x \* scale\_real);

color = cal\_pixel(c);

block[cur] = color;

cur += 1;

}

}

MPI\_Send(&block, 16001, MPI\_INT, 0, 2, MPI\_COMM\_WORLD);

start = start + N\_rectangle;

endtime = clock();

term = (double)(endtime - starttime) / CLOCKS\_PER\_SEC;

printf("process %d complete a Block, time: %fs\n", myid, term);

starttime = endtime;

}

}

MPI\_Finalize();

return 0;

}

1. dynamic code

#include <stdio.h>

#include <SFML/Graphics.hpp>

#include <mpi.h>

#include <time.h>

using namespace sf;

int width = 400;

int height = 400;

int N\_rectangle = 40;

struct complex {

float real;

float imag;

};

int cal\_pixel(complex c) {

int count = 0;

int max = 256;

complex z;

z.real = 0;

z.imag = 0;

float temp, lengthsq;

do {

temp = z.real \* z.real - z.imag \* z.imag + c.real;

z.imag = 2 \* z.real \* z.imag + c.imag;

z.real = temp;

lengthsq = z.real \* z.real;

count++;

} while ((lengthsq < 4.0) && (count < max));

return count;

}

clock\_t starttime, endtime;

clock\_t total\_start\_time, total\_end\_time;

int data[400][400];

int block[16002];

int temblock[16002];

int main(int argc, char\*\* argv) {

MPI\_Init(&argc, &argv);

int myid;

int numprocs;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numprocs);

MPI\_Status status;

float minR = -2;

float minI = -2;

float maxR = 2;

float maxI = 2;

float scale\_real, scale\_imag;

scale\_real = (maxR - minR) / width;

scale\_imag = (maxI - minI) / height;

complex c;

int cl;

if (myid == 0)

{

total\_start\_time = clock();

int count = 0;

int column = 0;

int numblock = width / N\_rectangle;

int blocksize = N\_rectangle \* height;

//멀티프로세서 일 때

if (numprocs > 1) {

for (int k = 1; k < numprocs && column < width; k++) {

MPI\_Send(&column, 1, MPI\_INT, k, 1, MPI\_COMM\_WORLD);

count++;

column += N\_rectangle;

}

int cur = 0;

int k;

int start, end;

do {

MPI\_Recv(&temblock, 16002, MPI\_INT, MPI\_ANY\_SOURCE, 0, MPI\_COMM\_WORLD, &status);

//16001은 slave의 인덱스, 16000은 시작위치, 0~15999는 블록 데이터

cur = temblock[16001];

if (column < width) {

MPI\_Send(&column, 1, MPI\_INT, cur, 1, MPI\_COMM\_WORLD);

column += N\_rectangle;

}

else {

//모든 블락처리시 slave에게 종료신호를 보냄 종료조건은 coulmn==width

count--;

MPI\_Send(&column, 1, MPI\_INT, cur, 1, MPI\_COMM\_WORLD);

}

start = temblock[16000];

end = start + N\_rectangle;

k = 0;

//데이터배열에 저장

for (int x = start; x < end; x++)

for (int y = 0; y < height; y++) {

data[y][x] = temblock[k];

k++;

}

} while (count > 0);

}

//싱글프로세서 일 때 마스터 코어가 모든 연산을 함.

else {

starttime = clock();

for (int b = 0; b < numblock;b++) {

int s = b \* N\_rectangle;

int e = s + N\_rectangle;

for (int x = s; x < e; x++)

{

c.real = minR + ((float)x \* scale\_real);

for (int y = 0; y < height; y++) {

c.imag = minI + ((float)y \* scale\_imag);

cl = cal\_pixel(c);

data[y][x] = cl;

}

}

endtime = clock();

printf("process %d complete a Block, time: %fs\n", myid, (double)(endtime - starttime) / CLOCKS\_PER\_SEC);

starttime = clock();

}

}

total\_end\_time = clock();

printf("\ntotal time in dynamic allocation:%fs\n\n", (double)(total\_end\_time - total\_start\_time) / CLOCKS\_PER\_SEC);

RenderWindow window(VideoMode(width, height), "Mandlebrot set");

Image image;

image.create(width, height);

Texture texture;

Sprite sprite;

while (window.isOpen()) {

Event event;

while (window.pollEvent(event))

{

if (event.type == Event::Closed) window.close();

}

for (int y = 0; y < height; y++)

for (int x = 0; x < width; x++) {

image.setPixel(x, y, Color(255 - data[y][x], 255 - data[y][x], 255 - data[y][x]));

}

window.clear();

texture.loadFromImage(image);

sprite.setTexture(texture);

window.draw(sprite);

window.display();

}

}

//slave

else {

int start, end;

int x, y, t;

starttime = clock();

MPI\_Recv(&start, 1, MPI\_INT, MPI\_ANY\_SOURCE, 1, MPI\_COMM\_WORLD, &status);

while (start != width) { //종료조건 받은데이터가 column의 마지막좌표 일 때

end = start + N\_rectangle;

t = 0;

for (x = start; x < end; x++)

{

c.real = minR + ((float)x \* scale\_real);

for (y = 0; y < width; y++) {

c.imag = minI + ((float)y \* scale\_imag);

block[t] = cal\_pixel(c);

t++;

}

}

block[16000] = start;

block[16001] = myid;

// 0~15999는 데이터 16000은 블락시작위치 16001은 slave의 인덱스

MPI\_Send(&block, 16002, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

endtime = clock();

printf("process %d complete a Block, time: %fs\n", myid, (double)(endtime - starttime) / CLOCKS\_PER\_SEC);

starttime = clock();

MPI\_Recv(&start, 1, MPI\_INT, MPI\_ANY\_SOURCE, 1, MPI\_COMM\_WORLD, &status);

}

}

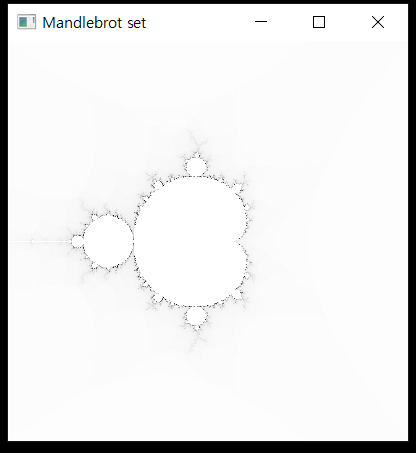
MPI\_Finalize();

return 0;

}

1. 분석

아래는 SFML 모듈로 띄운 결과창이다. 실수축과 허수축은 각각 -2~2사이로 설정했다.



Static 1 프로세서

텍스트이(가) 표시된 사진

자동 생성된 설명

Static 4 프로세서

텍스트이(가) 표시된 사진

자동 생성된 설명

Static 8프로세서

텍스트이(가) 표시된 사진

자동 생성된 설명

Dynamic 1 프로세서

텍스트이(가) 표시된 사진

자동 생성된 설명

Dynamic 4 프로세서

텍스트이(가) 표시된 사진

자동 생성된 설명

Dynamic 8 프로세서

텍스트이(가) 표시된 사진

자동 생성된 설명

표로 정리하면 다음과 같다.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| processor | Block1 | Block 2 | Block 3 | Block 4 | Block 5 | Block 6 | Block 7 | Block 8 | Block 9 | Block 10 | toal time |
| static1 | 0.001 | 0.0005 | 0.003 | 0.004 | 0.008 | 0.005 | 0.001 | 0.0005 | 0.0005 | 0.0005 | 0.022 |
| static4 | 0.007 | 0.008 | 0.002 | 0.001 | 0.002 | 0.009 | 0.0005 | 0.0005 | 0.0005 | 0.0005 | 0.015 |
| static8 | 0.004 | 0.001 | 0.009 | 0.005 | 0.012 | 0.006 | 0.005 | 0.007 | 0.001 | 0.0005 | 0.012 |
| dynamic1 | 0.001 | 0.001 | 0.002 | 0.004 | 0.008 | 0.005 | 0.0005 | 0.0005 | 0.0005 | 0.0005 | 0.021 |
| dynamic4 | 0.002 | 0.004 | 0.0005 | 0.0005 | 0.001 | 0.0005 | 0.006 | 0.006 | 0.003 | 0.007 | 0.012 |
| dynamic8 | 0.003 | 0.002 | 0.0005 | 0.011 | 0.006 | 0.001 | 0.011 | 0.007 | 0.007 | 0.006 | 0.011 |

위의 시간은 <time.h>의 clock(); 으로 쟀는데, clock의 최소 단위가 1/1000초=1ms 단위이기 때문에 1ms 보다 작게 걸린 경우에는 0.0005초로 처리했다.

표로 보건데 일단 static과 dynamic의 경우 모두 프로세서수가 증가함에 따라 속도가 빨라지는 양상을 보였다. 운영체제에 쓰레싱이 오거나 각 프로세서에 할당하는 시간이 급격히 줄지 않는 이상 프로세서수가 하여 멀티프로그래밍 정도가 올라가 speedup이 이루어졌다.

또한 각 블락 별로 처리시간은 천차만별이었는데 이는 블락 당 연산량이 작기 때문에 CPU 스케줄링의 영향으로인해 특정 블락은 빠르게, 특정블락은 느리게 처리되었지만 모두 0.000~0.012초 사이로 처리되었다. 이렇게 연산량이 적어 코드처리나 CPU의 스케줄링의 영향이 크기 때문에 각 블락당 평균을 구하는 것은 의미가 없을 것 같아서 하지 않았다.

추가적으로 10블락의 총시간을 쟀는데 이수치는 여러 번 돌려도 비슷하게 나타났으며 대체적으로 static 알고리즘에 비해 Dynamic 알고리즘이 빠르게 나타났다. 이는 static 알고리즘도 최대한 균등하게 배분했음에도 특정 프로세서의 속도가 느린 bottleneck 현상을 해결하지 못했기 때문으로 보인다. dynamic의경우 4프로세서와 8프로세서 경우를 보면 처음으로 할당한 1번 프로세서에 많은 처리가 이루어진 것을 볼 수 있는데 이는 A/L 처리속도보다 라우팅속도가 더 걸릴 만큼 연산량이 작기 때문으로 보인다. 왜냐하면 프로세서 1이 A/L을 처리하는 속도가 매우 빨라 거의 바로 다음 라우팅을 받을 수 있는 상태가 되었기 때문이다. (특히 dynamic 4의 경우 첫번째 프로세서가 10개중 6개의 블락을 처리했다. 나머지 프로세서는 각 2개씩) 만일 연산량이 훨씬 많아진다면 모든 프로세서가 할당받는 양이 거의 동등하게 바뀔것으로 보인다.