Assignment 2 & 3

# Solve PDE for Steady state temperature distribution problem. (Using MPI and Win32 Pthreads)

Platform: Visual Studio with MS MPI Implementation and Win32 Pthreads

Source Code for MPI Implementation

#include <iostream>

#include <fstream>

#include <string>

#include <cstdlib>

#include <math.h>

#include <mpi.h>

#define EPSILON 0.05

using namespace std;

void copyArrayValues(int, int, double \*, double\*\*);

void copyValuestoStrip(int, int, double \*, double\*\*);

void calcOddPoints(int, int, double\*\*);

void calcEvenPoints(int, int, double\*\*);

void broadCastToAll(int, int, int, int, double \*);

void printStep(int, int);

void printStripInfo(string, int, int, int, int, double\*\*);

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

int plateSize = 0;

int numThreads = 0;

int numberOfIterations = 0;

double epsilonValue = 0.0;

int rank, numProcesses;

MPI\_Status status;

double \*\*array;

double \*strip;

int stripNumber = 0;

int receivedStripNumber = 0;

/\*if (argc == 3){

plateSize = stoi(argv[1], nullptr, 10);

numThreads = stoi(argv[2], nullptr, 10);

cout << "Plate Size " << plateSize << endl;

cout << "Number of Threads " << numThreads << endl;

}

else {

plateSize = 64;

numThreads = 4;

}\*/

//Initialize the PlateSize to 64 and Number of Threads to 4

plateSize = 64;

numThreads = 4;

//calculate the value of episilon

epsilonValue = -1.0 \* log10(EPSILON);

//Calculate Number of Iterations necessary

numberOfIterations = (int)ceil(((double)plateSize \* epsilonValue) / (3.0));

//Initialize MPI

MPI\_Init(&argc, &argv);

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

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

//Allocate Memory for array having steady state temperature values

array = (double\*\*)malloc((plateSize + 2)\* sizeof(double \*));

for (int i = 0; i < (plateSize + 2); i++){

array[i] = (double \*)malloc((plateSize + 2) \* sizeof(double));

}

//Set up all values to 50

for (int i = 0; i < (plateSize + 2); i++) {

for (int j = 0; j < (plateSize + 2); j++){

array[i][j] = 50.0;

}

}

// Set the Top Most Portion 0

for (int j = 0; j < (plateSize + 2); j++){

array[0][j] = 0.0;

}

//Set other edges to values 100

for (int j = 1; j < (plateSize + 2); j++){

array[j][0] = 100.0;

array[j][plateSize + 1] = 100.0;

array[plateSize + 1][j] = 100.0;

}

if (rank == 0){

//Print Initial state of the problem in console

for (int i = 0; i < (plateSize + 2); i++) {

for (int j = 0; j < (plateSize + 2); j++){

if (array[i][j] == 0.0){

cout << array[i][j] << " ";

}

else if (array[i][j] == 100.0){

cout << array[i][j] << " ";

}

else if (array[i][j] == 50.0){

cout << array[i][j] << " ";

}

}

cout << endl;

}

cout << endl << "Computation for Solving Partial Differential Equation Started. " << endl << endl;

cout << "Number of Iterations " << numberOfIterations << endl << endl;

}

//Allocate Value for Strip

strip = (double \*)malloc((plateSize + 2) \* sizeof(double));

//Loop for numberOfIterations times

//Starts from k=0 to k=numberOfIterations-1

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

{

//Initialize Strip Number to 0 again

//Break array into Strip

//Communicate the Strip to each Process

//Each Process Calculates the value of Strip

//Broadcast the Strip Number first

//BroadCast the value of Each Strip to other processes

//Update the value of the strip as per the stripNumber in other corresponding processes

//Initialize stripNumber to 0 for each iteration

stripNumber = rank + 1;

while (true){

//Calculation Logic Started

/\*

If this if rank 0 process and the first Strip then

no need to receive values from any other process just wait for the

rank 0 to calculate the first strip value and

the broadcast the value of the strip to all other

processes.

\*/

if (rank == 0 && stripNumber == 1){

//Breaking up array into strip for setting

//up communication

for (int j = 0; j < (plateSize + 2); j++)

{

//As we want to copy the strip 1 for process 0 and consecutively.

strip[j] = array[stripNumber][j];

}

//copy value from strip to final array

copyArrayValues(stripNumber, plateSize, strip, array);

//Calculate odd points

calcOddPoints(stripNumber, plateSize, array);

//Calculate even points

calcEvenPoints(stripNumber, plateSize, array);

//Set information onto the buffer

copyValuestoStrip(stripNumber, plateSize, strip, array);

for (int i = 1; i < numProcesses; i++)

{

//cout << "Broadcasting Strip " << stripNumber << " From Process " << rank << " to " << i << endl;

//Broadcast StripNumber to all the other processors

MPI\_Send(&stripNumber, 1, MPI\_INTEGER, i, 50, MPI\_COMM\_WORLD);

//Broadcast the Strip information to all the processors so they can work accordingly.

MPI\_Send(strip, (plateSize + 2), MPI\_DOUBLE, i, 60, MPI\_COMM\_WORLD);

}

//Distribute the the strip number on the basis of process

stripNumber += numProcesses;

}

else {

//reInitialize receivedStripNumber to ZERO

receivedStripNumber = 0;

//Receive the stripNumber

MPI\_Recv(&receivedStripNumber, 1, MPI\_INTEGER, MPI\_ANY\_SOURCE, 50, MPI\_COMM\_WORLD, &status);

//Receive the Strip Values

MPI\_Recv(strip, (plateSize + 2), MPI\_DOUBLE, MPI\_ANY\_SOURCE, 60, MPI\_COMM\_WORLD, &status);

//copy value from strip to final array

copyArrayValues(receivedStripNumber, plateSize, strip, array);

if (receivedStripNumber == plateSize) {

break;

}

if (((receivedStripNumber + 1) == stripNumber) && (stripNumber <= (plateSize))){

//Calculate odd points

calcOddPoints(stripNumber, plateSize, array);

//Calculate even points

calcEvenPoints(stripNumber, plateSize, array);

//Set information onto the buffer

copyValuestoStrip(stripNumber, plateSize, strip, array);

//Broadcast Strip Info

broadCastToAll(rank, numProcesses, plateSize, stripNumber, strip);

if (stripNumber == plateSize) {

break;

}

//Distribute the the strip number on the basis of process

stripNumber += numProcesses;

}

}

}

cout << "Iteration Step : " << k << endl;

if (rank == 0){

//Open file for debugging purposes.

ofstream myfile("pdeSolverParallel.csv", ios::out | ios::app);

//Write output to file for each iteration State

myfile << endl << "Iteration " << k << endl;

for (int j = 0; j < (plateSize + 2); j++){

myfile << "Plate" << j << ",";

}

myfile << endl;

for (int i = 0; i < (plateSize + 2); i++) {

for (int j = 0; j < (plateSize + 2); j++){

myfile << array[i][j] << ",";

}

myfile << endl;

}

//close the file

myfile.flush();

myfile.close();

}

}

if (rank == 0){

//Print results

for (int i = 0; i < (plateSize + 2); i++) {

for (int j = 0; j < (plateSize + 2); j++){

cout << array[i][j] << " ";

}

cout << endl;

}

cout << endl << "Computation for Solving Partial Differential Equation Ended. " << endl;

}

MPI\_Finalize();

return 1;

}

//Method to values from strip to array

void copyArrayValues(int stripNumber, int plateSize, double \*strip, double\*\* array){

for (int i = 0; i < (plateSize + 2); i++)

{

array[stripNumber][i] = strip[i];

}

}

//Method to copy values from array to strip

void copyValuestoStrip(int stripNumber, int plateSize, double \*strip, double\*\* array){

for (int i = 0; i < (plateSize + 2); i++)

{

strip[i] = array[stripNumber][i];

}

}

//Method to calculate odd points for array according to strip number

void calcOddPoints(int stripNumber, int plateSize, double\*\* array){

double vDash = 0.0;

for (int j = 1; j < (plateSize + 1); j++)

{

vDash = 0.0;

if ((stripNumber + j) % 2 == 0){

vDash = (array[stripNumber - 1][j] + array[stripNumber][j - 1] + array[stripNumber + 1][j] + array[stripNumber][j + 1]) / 4.0;

array[stripNumber][j] = (vDash + array[stripNumber][j]) / 2.0;

}

}

}

//Method to calculate even points for array according to strip number

void calcEvenPoints(int stripNumber, int plateSize, double\*\* array){

double vDash = 0.0;

for (int j = 1; j < (plateSize + 1); j++)

{

vDash = 0.0;

if ((stripNumber + j) % 2 != 0){

vDash = (array[stripNumber - 1][j] + array[stripNumber][j - 1] + array[stripNumber + 1][j] + array[stripNumber][j + 1]) / 4.0;

array[stripNumber][j] = (vDash + array[stripNumber][j]) / 2.0;

}

}

}

//Broadcast strip values to other processor nodes from source node

void broadCastToAll(int rank, int numProcesses, int plateSize, int stripNumber, double \*strip){

for (int i = 0; i < numProcesses; i++)

{

//No need to send same value for strip if same processor

if (i == rank) continue;

//cout << "Broadcasting Strip " << stripNumber << " From Process " << rank << " to " << i << endl;

//Broadcast StripNumber to all the other processors

MPI\_Send(&stripNumber, 1, MPI\_INTEGER, i, 50, MPI\_COMM\_WORLD);

//Broadcast the Strip information to all the processors so they can work accordingly.

MPI\_Send(strip, (plateSize + 2), MPI\_DOUBLE, i, 60, MPI\_COMM\_WORLD);

}

}

//Print the strip information for a particular processor

//used for debugging

void printStep(int rank, int stripNumber){

cout << "Strip Number " << stripNumber << " Processed by Process " << rank << endl;

}

//Print information at a given iteration step for a given strip

//this method is more specific

void printStripInfo(string str, int iteration, int iterationStep, int stripNumber, int plateSize, double\*\* array){

if (iteration == iterationStep){

cout << str << endl;

for (int i = 0; i < (plateSize + 2); i++)

{

cout << array[stripNumber][i] << " ";

}

cout << endl;

}

}

Source Code for Pthreads

#include <iostream>

#include <fstream>

#include <string>

#include <cstdlib>

#include <math.h>

#include <sys/types.h>

#include <pthread.h>

#define EPSILON 0.05

using namespace std;

typedef struct {

int threadId;

double \*\*array;

int \*stripNumber;

int \*numberOfIterations;

int \*plateSize;

int \*numThreads;

int \*turn;

} parameter;

pthread\_mutex\_t mutex;

void calcOddPoints(int, int, double\*\*);

void calcEvenPoints(int, int, double\*\*);

void printStep(int, int);

void printStripInfo(string, int, int, int, int, double\*\*);

void \*hello(void \*arg)

{

parameter \*p = (parameter \*)arg;

printf("Thread %d ; Plate Size %d; Number of Theads %d; Number of Iterations %d; Address of Array %d; Address of Strip %d; Adress of StripNumber %d \n", p->threadId, \*(p->plateSize), \*(p->numThreads), \*(p->numberOfIterations), p->array, p->stripNumber);

return (NULL);

}

void \*solve(void\* arg){

parameter \*p = (parameter \*)arg;

int numberOfIterations = 0;

while (numberOfIterations < \*(p->numberOfIterations))

{

\*(p->stripNumber) = 1;

\*(p->turn) = 0;

while (true)

{

//Must be bound by Mutex

//Critical Section

pthread\_mutex\_lock(&mutex);

if (\*(p->turn) == p->threadId){

if (\*(p->turn) == p->threadId){

//calculate odd points

calcOddPoints(\*(p->stripNumber), \*(p->plateSize), p->array);

//calculate even points

calcEvenPoints(\*(p->stripNumber), \*(p->plateSize), p->array);

//Debug Info

if (\*(p->stripNumber) == 1 && numberOfIterations == 0){

cout<<"Computation by"<<p->threadId << "Value is " << p->array[1][1]<<endl;

}

//cout << "Thread ID " << p->threadId << " has StripNumber " << \*(p->stripNumber) << endl;

\*(p->stripNumber) += 1;

}

if (!(\*(p->stripNumber) < (\*(p->plateSize) + 1))){

numberOfIterations++;

pthread\_mutex\_unlock(&mutex);

break;

}

\*(p->turn) +=1;

if (\*(p->turn) == \*(p->numThreads)){

\*(p->turn) = 0;

}

}

pthread\_mutex\_unlock(&mutex);

}

}

if (p->threadId==3){

//Open file for debugging purposes.

ofstream myfile("pdeSolverPThreadTest.csv", ios::out | ios::app);

//Write output to file for each iteration State

myfile << endl << "Iteration " << numberOfIterations << endl;

for (int j = 0; j < (\*(p->plateSize) + 2); j++){

myfile << "Plate" << j << ",";

}

myfile << endl;

for (int i = 0; i < (\*(p->plateSize) + 2); i++) {

for (int j = 0; j < (\*(p->plateSize) + 2); j++){

myfile << p->array[i][j] << ",";

}

myfile << endl;

}

//close the file

myfile.flush();

myfile.close();

}

return (NULL);

}

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

pthread\_t \*threads;

pthread\_attr\_t pthread\_custom\_attr;

int plateSize, numThreads, numberOfIterations, stripNumber = 0, turn = 0;

double \*\*array;

double epsilonValue = 0.0;

parameter \*param;

//Initialize the PlateSize to argv[1] and Number of Threads to argv[2]

if (argc == 3){

plateSize = stoi(argv[1], nullptr, 10);

numThreads = stoi(argv[2], nullptr, 10);

}

else {

plateSize = 64;

numThreads = 4;

}

//Set things Up

{

//Allocate Memory for Thread

threads = (pthread\_t \*)malloc(numThreads\*sizeof(\*threads));

//Initialize Thread attribute

pthread\_attr\_init(&pthread\_custom\_attr);

//Allocate Memory for param

param = (parameter \*)malloc(sizeof(parameter)\*numThreads);

//Allocate Memory for parameter members

array = (double\*\*)malloc((plateSize + 2)\* sizeof(double \*));

for (int i = 0; i < (plateSize + 2); i++){

array[i] = (double \*)malloc((plateSize + 2) \* sizeof(double));

}

//Set up all values to 50

for (int i = 0; i < (plateSize + 2); i++) {

for (int j = 0; j < (plateSize + 2); j++){

array[i][j] = 50.0;

}

}

// Set the Top Most Portion 0

for (int j = 0; j < (plateSize + 2); j++){

array[0][j] = 0.0;

}

//Set other edges to values 100

for (int j = 1; j < (plateSize + 2); j++){

array[j][0] = 100.0;

array[j][plateSize + 1] = 100.0;

array[plateSize + 1][j] = 100.0;

}

//Calculate the value of episilon

epsilonValue = -1.0 \* log10(EPSILON);

numberOfIterations = (int)ceil(((double)plateSize \* epsilonValue) / (3.0));

//Print Initial state of the problem in console

/\*for (int i = 0; i < (plateSize + 2); i++) {

for (int j = 0; j < (plateSize + 2); j++){

if (param->array[i][j] == 0.0){

cout << param->array[i][j] << " ";

}

else if (param->array[i][j] == 100.0){

cout << param->array[i][j] << " ";

}

else if (param->array[i][j] == 50.0){

cout << param->array[i][j] << " ";

}

}

cout << endl;

}\*/

//Assign Number of PlateSize, Number of Iterations and Number of Threads

for (int i = 0; i < numThreads; i++)

{

param[i].numberOfIterations = &numberOfIterations;

param[i].numThreads = &numThreads;

param[i].plateSize = &plateSize;

param[i].stripNumber = &stripNumber;

param[i].turn = &turn;

param[i].array = array;

}

}

//Setup Mutex

pthread\_mutex\_init(&mutex, NULL);

cout << "Computation Started" << endl;

//Start Threads Up

for (int i = 0; i < numThreads; i++)

{

param[i].threadId = i;

pthread\_create(&threads[i], &pthread\_custom\_attr, solve, (void \*)(param + i));

}

cout << "Computation Finished" << endl;

getchar();

return 1;

}

void calcOddPoints(int stripNumber, int plateSize, double\*\* array){

double vDash = 0.0;

for (int j = 1; j < (plateSize + 1); j++)

{

vDash = 0.0;

if ((stripNumber + j) % 2 == 0){

vDash = (array[stripNumber - 1][j] + array[stripNumber][j - 1] + array[stripNumber + 1][j] + array[stripNumber][j + 1]) / 4.0;

array[stripNumber][j] = (vDash + array[stripNumber][j]) / 2.0;

}

}

}

void calcEvenPoints(int stripNumber, int plateSize, double\*\* array){

double vDash = 0.0;

for (int j = 1; j < (plateSize + 1); j++)

{

vDash = 0.0;

if ((stripNumber + j) % 2 != 0){

vDash = (array[stripNumber - 1][j] + array[stripNumber][j - 1] + array[stripNumber + 1][j] + array[stripNumber][j + 1]) / 4.0;

array[stripNumber][j] = (vDash + array[stripNumber][j]) / 2.0;

}

}

}

void printStep(int rank, int stripNumber){

cout << "Strip Number " << stripNumber << " Processed by Process " << rank << endl;

}

void printStripInfo(string str, int iteration, int iterationStep, int stripNumber, int plateSize, double\*\* array){

if (iteration == iterationStep){

cout << str << endl;

for (int i = 0; i < (plateSize + 2); i++)

{

cout << array[stripNumber][i] << " ";

}

cout << endl;

}

}

Test Results

Test Results are enclosed in the Attached Excel File.



Conclusion

In conclusion, I was very influenced by the idea of using Shared Memory. In case of MPI, where shared memory was not an option I had to pass the array strip values from one process to another, this introduced the overhead of processing as well as passing the strip. The co-ordination to keep the process going in a serial fashion was a much harder task as well. Process co-ordination was done using assigning and passing corresponding strip numbers. However, MPI also introduced an important concept of how truly distributed systems work and how truly parallel programs are to be designed. Working through the project I felt the implementation of concepts introduced in class.

Working on the assignment 3, threaded applications turned out to be much simpler than using MPI. Partly, the reason was because we could share memory in between threads. So we could work directly with the array containing values and no strip communication was necessary. The threads were processed in a serial fashion using a mutex. The mutex helped avoid possible race conditions within the threads and also helped maintain computational integrity.