

**Consolidated Lab Report**

**Name: Samridh Anand Paatni**

**Registration Number: 20BCE1550**

**Subject: CSE4001 Parallel Distributed Computing**

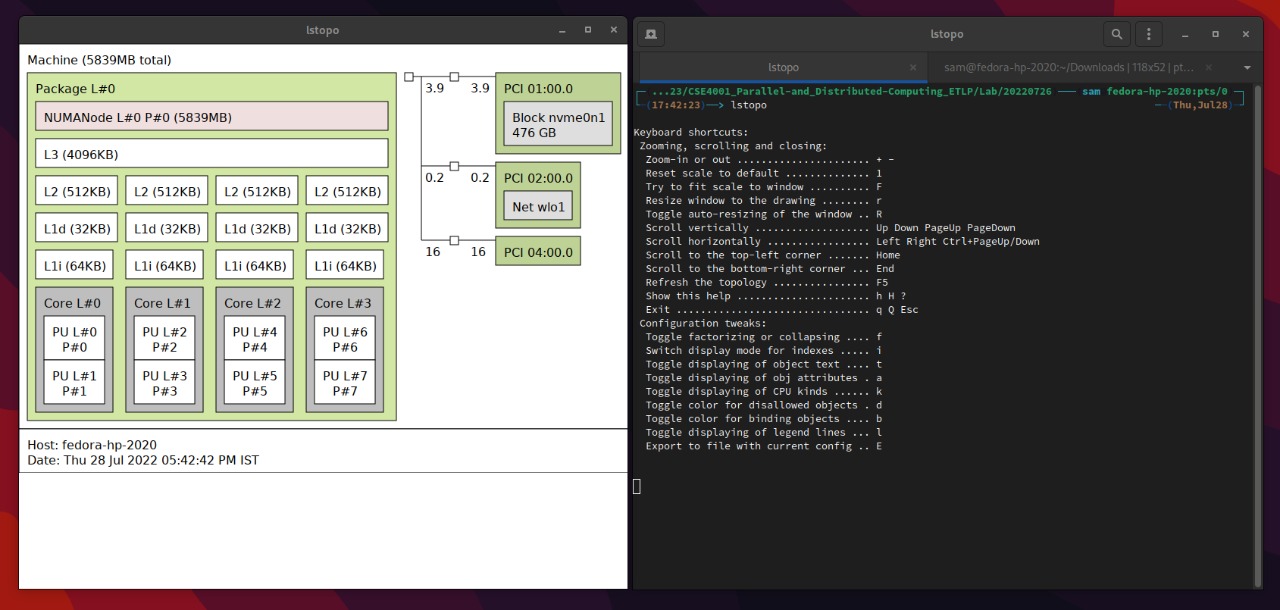
**Slot: L9+L10**

**Professor: Dr. Kumar R**

**Lab 01: PThreads**

**Q1. Display the processors layout of your system.**

The output for the command **lstopo**, after installing the hwloc package:



**Q2. Write a multithreaded program in C to create 10k, 20k and 50k threads and measure the time taken for each thread group.**

C Code:

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<time.h>

void \* void\_function(void \*message) {}

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

pthread\_t \* threads;

int num\_threads = atoi(argv[1]); // because the cli argument is an ASCII code

threads = (pthread\_t \*) calloc(num\_threads, sizeof(pthread\_t));

clock\_t t = clock();

for (int i = 0; i < num\_threads; i++) {

pthread\_create(&threads[i], NULL, void\_function, NULL);

}

for (int i = 0; i < num\_threads; i++) {

pthread\_join(threads[i], NULL);

}

t = clock() - t;

printf(

"took %f seconds to make %d threads\n",

((double) t)/CLOCKS\_PER\_SEC,

num\_threads

);

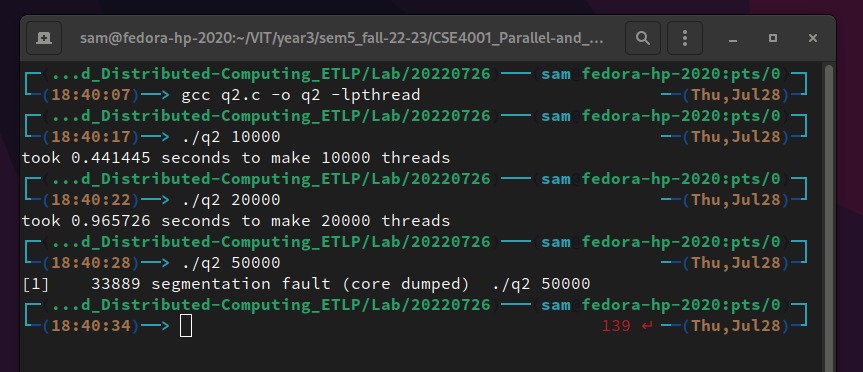
free(threads);

return 0;

}

Output:

Creating 5000 pthreads resulted in a segmentation fault.



**Q3. Write a program to create 2 threads. Thread 1 has to print “PDC” and thread 2 has to print “lab”.**

C Code:

#include<stdio.h>

#include<pthread.h>

void \* message\_function(void \*message) {

printf("%s\n", ((char \*) message));

}

int main() {

pthread\_t t1, t2;

char \*m1 = "Thread 1: PDC";

char \*m2 = "Thread 2: Lab";

pthread\_create(&t1, NULL, message\_function, (void \*) m1);

pthread\_create(&t2, NULL, message\_function, (void \*) m2);

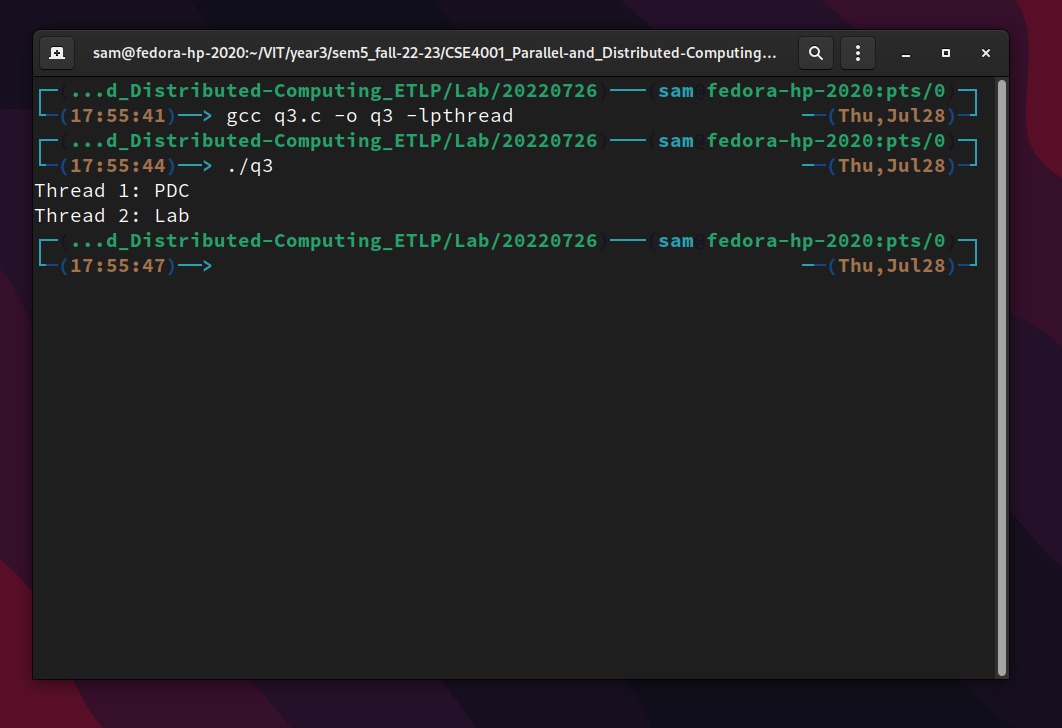
pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

return 0;

}

Output:



**Lab 02: Open MP Programming**

**Q1. Create a hundred threads using:**

1. **Runtime Library Routines**

**Code:**

// using runtime library routines

#include<stdio.h>

#include<omp.h>

// compile using: `gcc filename -fopenmp`

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

int tid, numThreads;

omp\_set\_num\_threads(100);

# pragma omp parallel private (tid, numThreads)

{

tid = omp\_get\_thread\_num();

printf("welcome to PDC %d\n", tid);

if (tid == 0) {

numThreads = omp\_get\_num\_threads();

printf("%d threads have been created\n", numThreads);

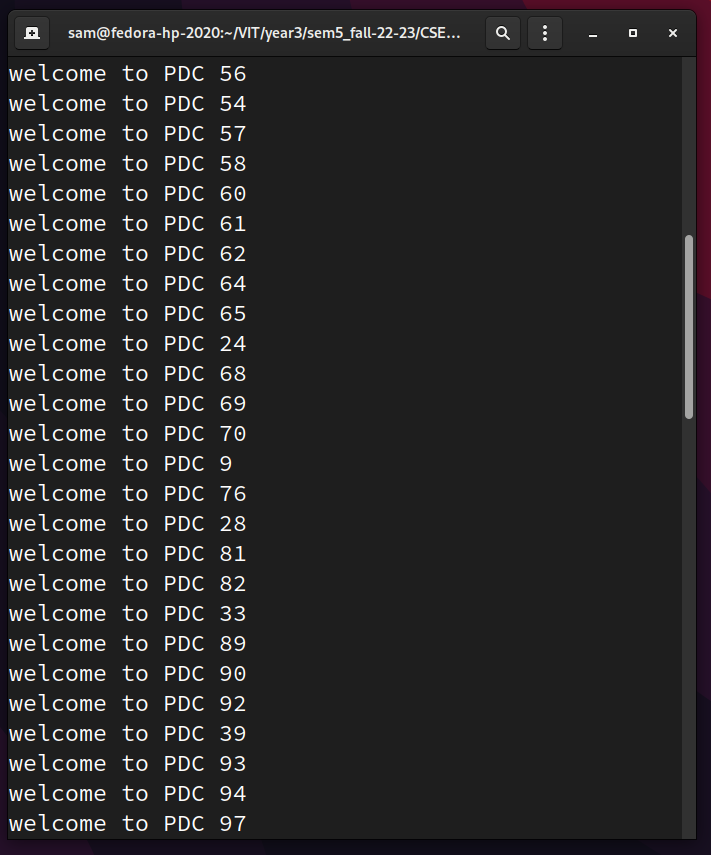
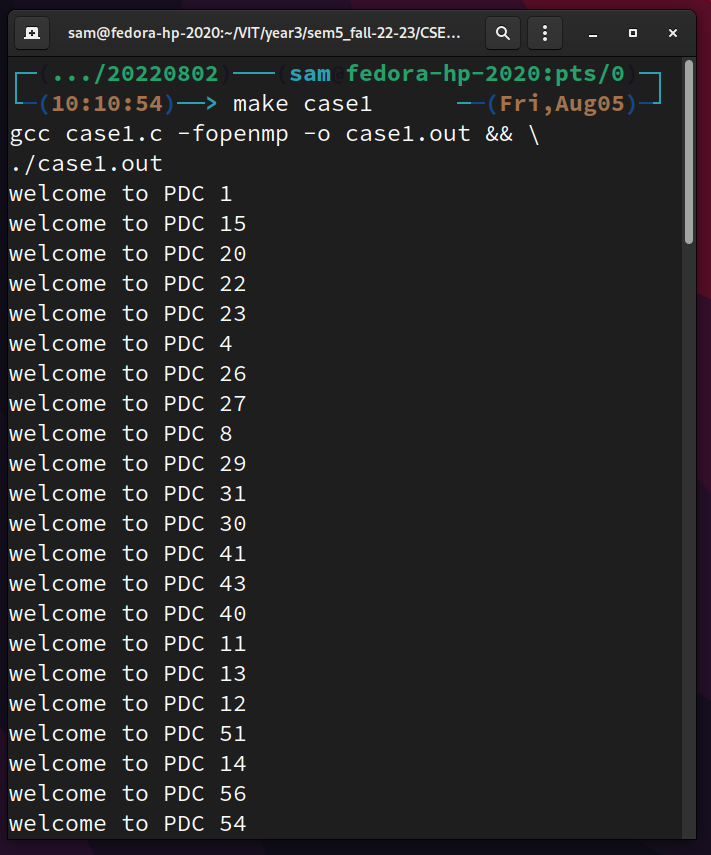
}

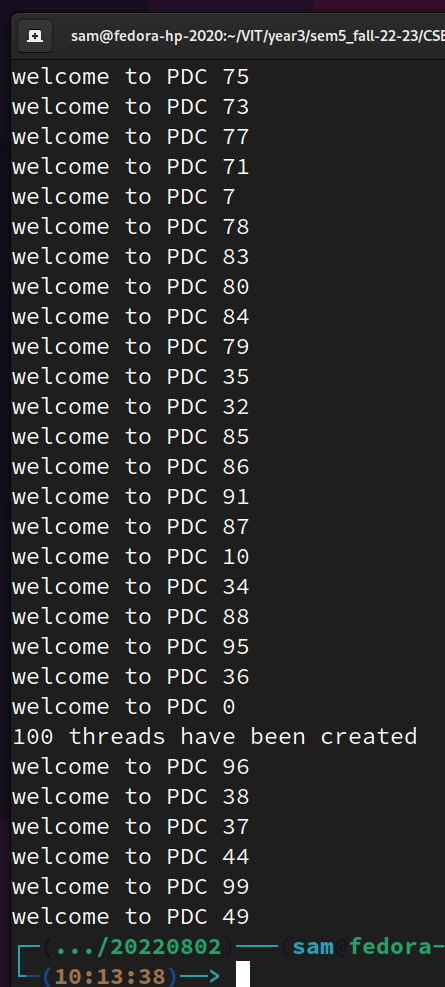
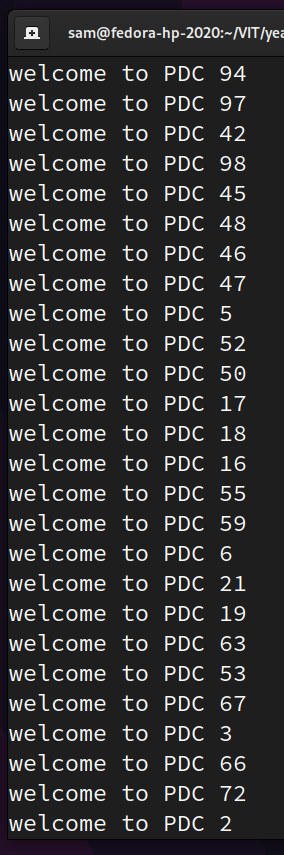
}

return 0;

}

**Output:**

****

****

1. **Compiler Directives**

**Code:**

// using compiler directives

#include<stdio.h>

#include<omp.h>

// compile using: `gcc filename -fopenmp`

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

int tid, numThreads;

# pragma omp parallel private (tid, numThreads) num\_threads(100)

{

tid = omp\_get\_thread\_num();

printf("welcome to PDC %d\n", tid);

if (tid == 0) {

numThreads = omp\_get\_num\_threads();

printf("%d threads have been created\n", numThreads);

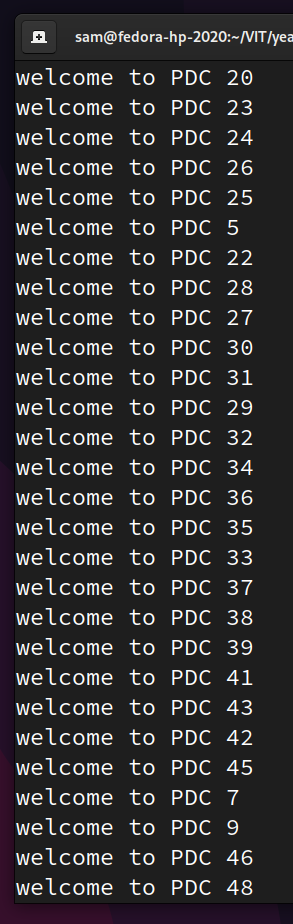
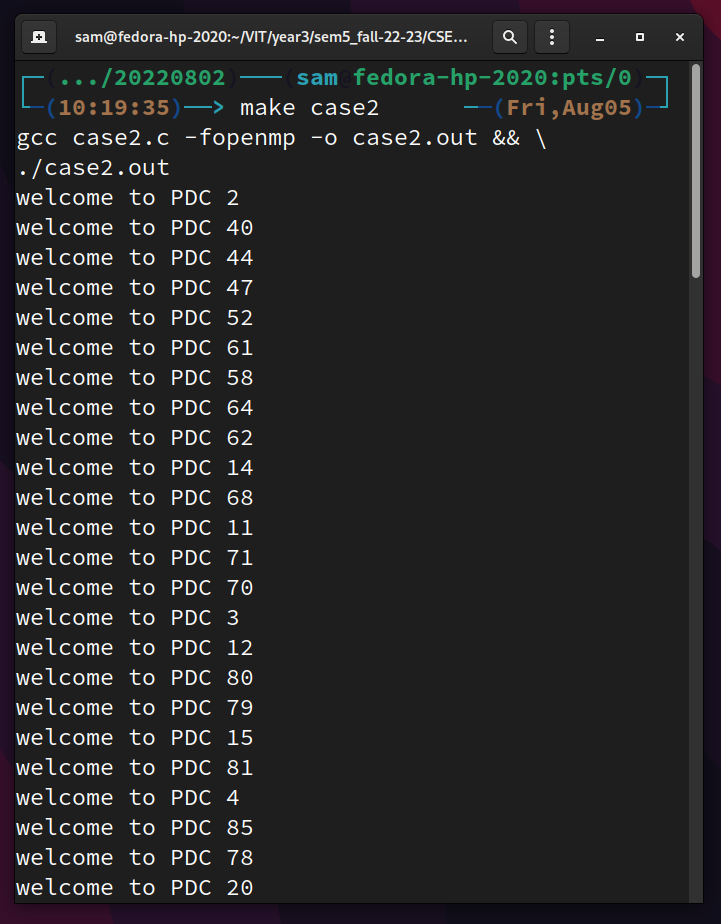
}

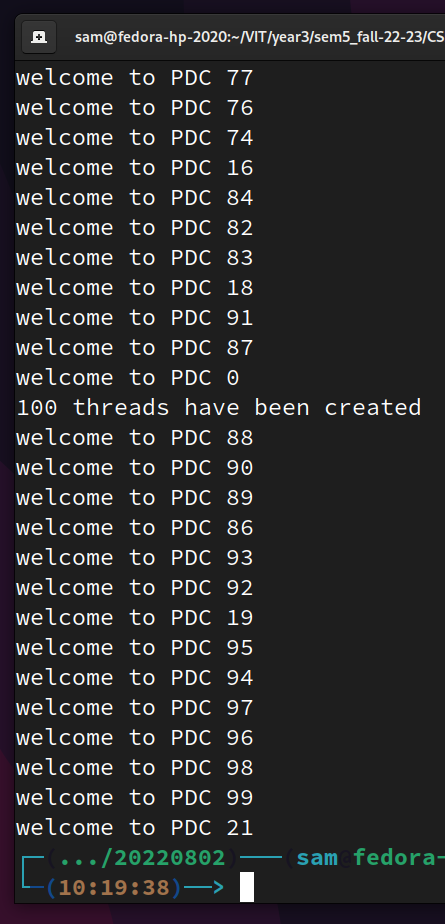
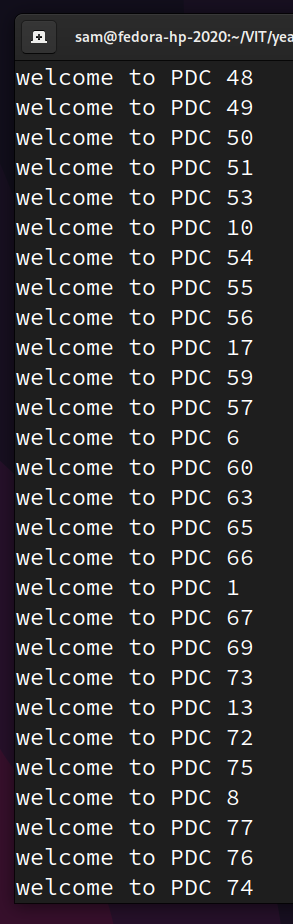
}

return 0;

}

**Output:**

****

****

1. **Environment Variables**

**Code:**

// using environment variables

#include<stdio.h>

#include<omp.h>

// compile using: `gcc filename -fopenmp`

// before running, give the command: `export OMP\_NUM\_THREADS=100` in bash

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

int tid, numThreads;

# pragma omp parallel private (tid, numThreads)

{

tid = omp\_get\_thread\_num();

printf("welcome to PDC %d\n", tid);

if (tid == 0) {

numThreads = omp\_get\_num\_threads();

printf("%d threads have been created\n", numThreads);

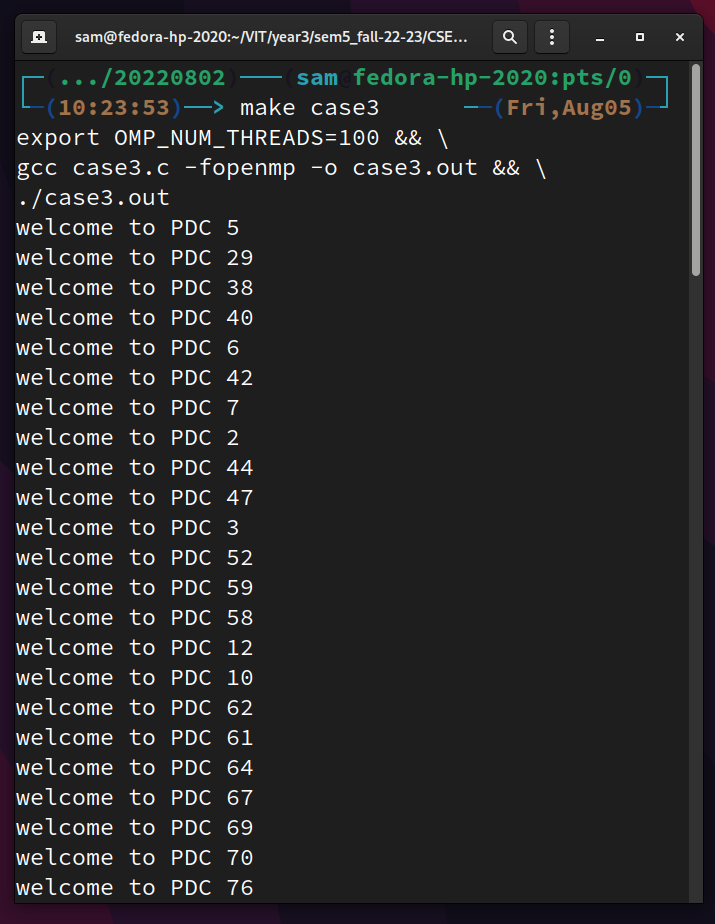
}

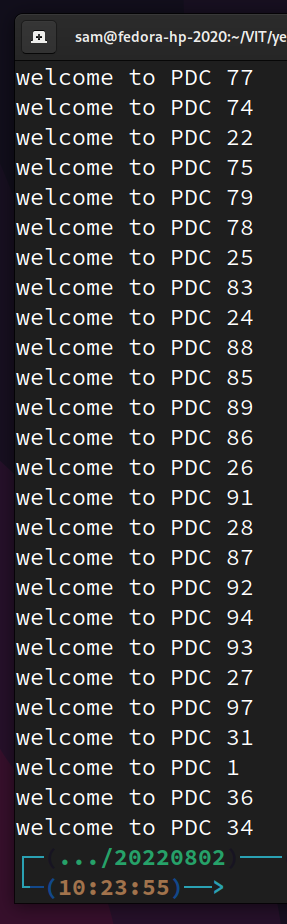
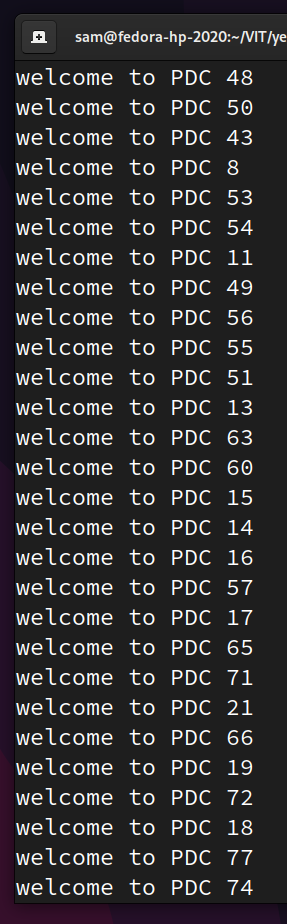
}

return 0;

}

**Output:**

****

****

**Q2. Implement vector addition in serial and parallel and compare the results. Do the parallel computation using a 1000 threads.**

**Code:**

**Serial addition:**

#include <stdio.h>

#include <time.h>

#define VECTOR\_SIZE 100000

int main() {

// make the vectors

int a[VECTOR\_SIZE], b[VECTOR\_SIZE], c[VECTOR\_SIZE];

for (int i = 0; i < VECTOR\_SIZE; i++) {

a[i] = VECTOR\_SIZE - i;

b[i] = i;

}

// serially add the vectors

clock\_t tSerial = clock();

for (int i = 0; i < VECTOR\_SIZE; i++) {

c[i] = a[i] + b[i];

}

tSerial = clock() - tSerial;

// show the output

printf(

"Serial addition took %f seconds\n",

((double) tSerial)/CLOCKS\_PER\_SEC

);

return 0;

}

**Parallel addition:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <time.h>

#define VECTOR\_SIZE 100000

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

// make the vectors

int a[VECTOR\_SIZE], b[VECTOR\_SIZE], c[VECTOR\_SIZE];

for (int i = 0; i < VECTOR\_SIZE; i++) {

a[i] = VECTOR\_SIZE - i;

b[i] = i;

}

int nThreads = atoi(argv[1]);

// paralelly add the vectors:

clock\_t tPar = clock();

// the part of the vector one thread will access

int slice\_size = VECTOR\_SIZE / nThreads;

int slice\_start, slice\_end;

int tid;

// make threads

omp\_set\_num\_threads(nThreads);

#pragma omp parallel private (tid, slice\_start, slice\_end)

{

// allot a slice to the particular thread

tid = omp\_get\_thread\_num();

slice\_start = tid \* slice\_size;

slice\_end = slice\_start + slice\_size;

// perform addition for the elements in the allotted slice\_size

for (int i = slice\_start; i < slice\_end; i++) {

c[i] = a[i] + b[i];

}

}

tPar = clock() - tPar;

// show the output

printf(

"Parallel addition took %f seconds with %d threads\n",

((double) tPar)/CLOCKS\_PER\_SEC,

nThreads

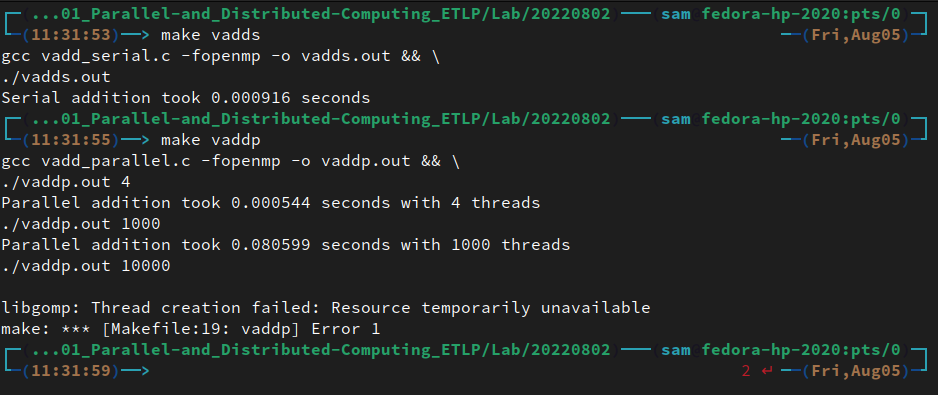
);

return 0;

}

**Output:**

Running the computation with 4 threads is the fastest (because I have a quad-core laptop). The parallel computation with a 100 more threads is slower than the serial execution.



**Lab 4: Sections**

**Q1.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

void function1() {

// int n = 100000;

int n = 16;

int \*nums = (int \*)calloc(n, sizeof(int));

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

nums[i] = rand() % 100;

}

int i = 0;

int local\_min\_1, local\_min\_2, local\_min\_3, local\_min\_4;

int min = 100;

#pragma omp parallel shared(nums, n, min) private(i, local\_min\_1, local\_min\_2, local\_min\_3, local\_min\_4) num\_threads(4)

{

#pragma omp sections

{

min = 100;

#pragma omp section

{

local\_min\_1 = 100;

for (i = 0; i < n/4; i++) {

if (nums[i] < local\_min\_1) local\_min\_1 = nums[i];

}

#pragma omp critical

{

if (local\_min\_1 < min) min = local\_min\_1;

}

}

#pragma omp section

{

local\_min\_2 = 100;

for (i = n/4; i < 2 \* n/4; i++) {

if (nums[i] < local\_min\_2) local\_min\_2 = nums[i];

}

#pragma omp critical

{

if (local\_min\_2 < min) min = local\_min\_2;

}

}

#pragma omp section

{

local\_min\_3 = 100;

for (i = 2 \* n/4; i < 3 \* n/4; i++) {

if (nums[i] < local\_min\_3) local\_min\_3 = nums[i];

}

#pragma omp critical

{

if (local\_min\_3 < min) min = local\_min\_3;

}

}

#pragma omp section

{

local\_min\_4 = 100;

for (i = 3 \* n/4; i < n; i++) {

if (nums[i] < local\_min\_4) local\_min\_4 = nums[i];

}

#pragma omp critical

{

if (local\_min\_4 < min) min = local\_min\_4;

}

}

}

}

for (i = 0; i < n; i++) {

printf(

"%s%d%s",

i == 0 ? "[" : " ",

nums[i],

i == n - 1 ? "]\n" : ","

);

}

printf("\n%d is the minimum element", min);

free(nums);

}

int main() {

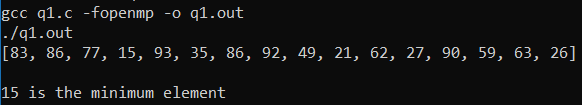
function1();

printf("\n");

return 0;

}

**Output:**



**Q2.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

typedef int \*\* matrix;

#define N 1100 // the size of the matrices

void multiply(matrix A, matrix B, matrix ans) {

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

for (int j = 0; j < N; j++) {

for (int k = 0; k < N; k++) {

ans[i][j] += A[i][k] \* B[k][j];

}

}

}

}

// case 1 = only the outermost loop is parallelized

void multiplyCase1(matrix A, matrix B, matrix ans, int numThreads) {

printf("The outermost loop is parallelized\n");

int i = 0;

#pragma omp parallel for shared(numThreads, A, B, ans) private(i)

for (i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

for (int k = 0; k < N; k++) {

ans[i][j] += A[i][k] \* B[k][j];

}

}

}

}

void multiplyCase2(matrix A, matrix B, matrix ans, int numThreads) {

printf("The outer 2 loops are parallelized\n");

int i = 0, j = 0;

#pragma omp parallel for shared(numThreads, A, B, ans) private(i)

for (i = 0; i < N; i++) {

#pragma omp parallel for

for (j = 0; j < N; j++) {

for (int k = 0; k < N; k++) {

ans[i][j] += A[i][k] \* B[k][j];

}

}

}

}

void multiplyCase3(matrix A, matrix B, matrix ans, int numThreads) {

printf("All loops are parallelized\n");

int i = 0, j = 0, k = 0;

#pragma omp parallel for shared(numThreads, A, B, ans) private(i)

for (i = 0; i < N; i++) {

#pragma omp parallel for

for (j = 0; j < N; j++) {

#pragma omp parallel for

for (k = 0; k < N; k++) {

ans[i][j] += A[i][k] \* B[k][j];

}

}

}

}

void multiplyCase4(matrix A, matrix B, matrix ans, int numThreads) {

printf("All loops are parallelized\n");

int i = 0, j = 0, k = 0;

#pragma omp parallel for shared(numThreads, A, B, ans) private(i) collapse(3)

for (i = 0; i < N; i++) {

for (j = 0; j < N; j++) {

for (k = 0; k < N; k++) {

ans[i][j] += A[i][k] \* B[k][j];

}

}

}

}

void print(matrix A) {

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

printf("%s", i == 0 ? "[\n " : " ");

for (int j = 0; j < N; j++) {

printf(

"%s%d%s",

j == 0 ? "[" : " ",

A[i][j],

j == N - 1 ? "]" : ","

);

}

printf("%s", i == N - 1 ? "\n]\n" : ",\n");

}

}

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

int numThreads = atoi(argv[1]);

int type = atoi(argv[2]);

printf("multiplying 2 %dx%d matrices using %d threads and case %d:\n", N, N, numThreads, type);

matrix A, B, C;

A = (int \*\*)calloc(N, sizeof(int\*));

B = (int \*\*)calloc(N, sizeof(int\*));

C = (int \*\*)calloc(N, sizeof(int\*));

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

A[i] = (int \*) calloc(N, sizeof(int));

B[i] = (int \*) calloc(N, sizeof(int));

C[i] = (int \*) calloc(N, sizeof(int));

for (int j = 0; j < N; j++) {

A[i][j] = rand() % 10;

B[i][j] = rand() % 10;

C[i][j] = 0;

}

}

// print(A);

// print(B);

double t = omp\_get\_wtime();

switch (type)

{

case 0:

multiply(A, B, C);

break;

case 1:

multiplyCase1(A, B, C, numThreads);

break;

case 2:

multiplyCase2(A, B, C, numThreads);

break;

case 3:

multiplyCase3(A, B, C, numThreads);

break;

case 4:

multiplyCase4(A, B, C, numThreads);

break;

default:

}

t = omp\_get\_wtime() - t;

// print(C);

printf("took %f seconds\n", t);

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

free(A[i]);

free(B[i]);

}

free(A);

free(B);

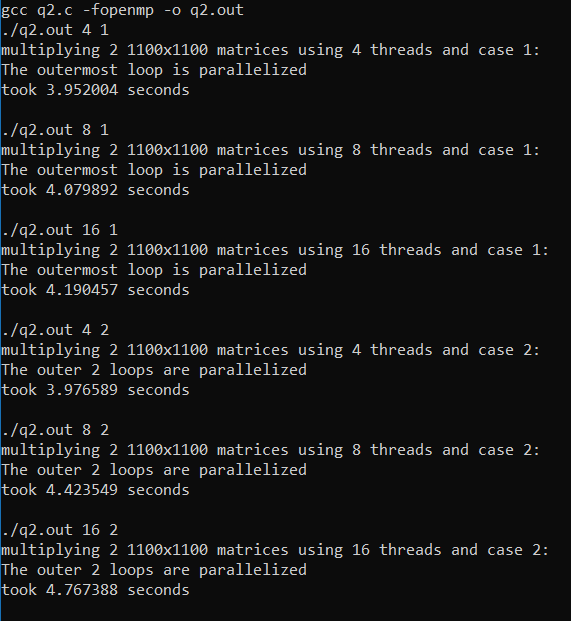
free(C);

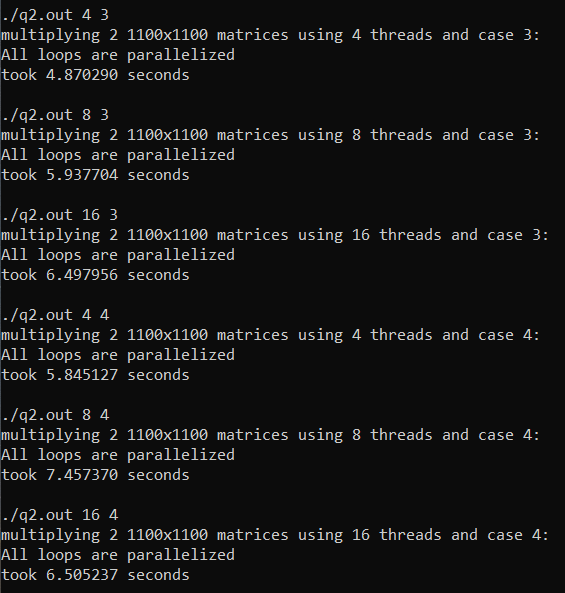
printf("\n");

return 0;

}

**Output:**





**Lab 05: OMP Synchronization Constructs**

**Q1.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 10

#define OMP\_NUM\_THREADS 4

/\*

1. Write your own code snippet to demonstrate the following

a. Barrier

b. Master

c. Single

d. Critical

e. Ordered

\*/

void barrier() {

printf("\na. Barrier:\n");

int a[1000], b[1000], i = 0, sum = 0;

for (i = 0; i < 1000; i++) {

a[i] = rand() % 100;

b[i] = rand() % 10;

}

#pragma omp parallel private(i)

{

for (i = 0; i < 1000; i++) {

a[i] = a[i] - b[i];

}

#pragma omp barrier

#pragma omp for reduction(+ : sum)

for (i = 0; i < 1000; i++)

{

sum += a[i];

}

}

printf("sum: %d\n", sum);

}

void master() {

printf("\nb. Master\nwithout 'master':\n");

#pragma omp parallel

{

printf("hello, from thread %d\n", omp\_get\_thread\_num());

}

printf("\nwith master:\n");

#pragma omp parallel

{

#pragma omp master

{

printf("hello, from thread %d\n", omp\_get\_thread\_num());

}

}

}

void single() {

printf("\nc. Single:\n");

int a=0, b=0;

#pragma omp parallel num\_threads(4)

{

#pragma omp single

a++;

#pragma omp critical

b++;

}

printf("single: %d | critical: %d\nsingle runs once, critical is run once per thread\n", a, b);

}

void critical() {

printf("\nd. Critical:\n");

int i; int max; int a[N];

for (i = 0; i < N; i++) {

a[i] = rand();

printf(

"a[%d] = %d\tthread no %d\n",

i,

a[i],

omp\_get\_num\_threads()

);

}

max = a[0];

#pragma omp parallel for

for (i = 1; i < N; i++) {

if (a[i] > max) {

#pragma omp critical

{

if (a[i] > max) max = a[i];

}

}

}

printf("\nmax = %d\t%d threads\n", max, omp\_get\_num\_threads());

}

void ordered() {

printf("\ne. Ordered:\nwithout ordered:\n");

int i = 0;

int n = 10;

#pragma omp parallel shared(n) private(i)

{

#pragma omp for

for (i = 0; i < n; i++) {

printf("thread %d at index %d\n", omp\_get\_thread\_num(), i);

}

}

printf("\nwith ordered:\n");

#pragma omp parallel shared(n) private(i)

{

#pragma omp for ordered

for (i = 0; i < n; i++) {

#pragma omp ordered

{

printf("thread %d at index %d\n", omp\_get\_thread\_num(), i);

}

}

}

}

int main() {

barrier();

master();

single();

critical();

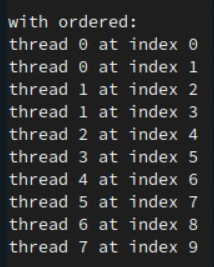
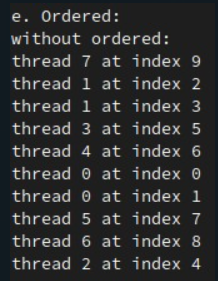
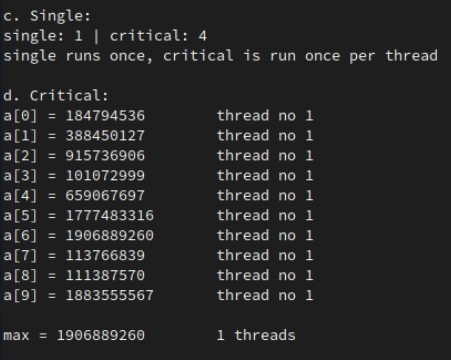
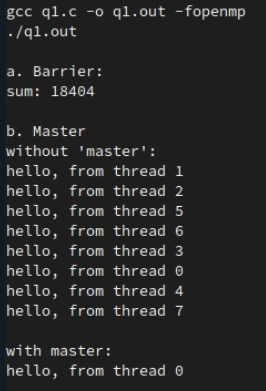
ordered();

printf("\n");

return 0;

}

**Output:**



**Q2**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <unistd.h>

#include <time.h>

#define MAX\_SLEEP 10

omp\_lock\_t lock;

int cr = 0;

void reader(int i) {

#pragma omp critical

{

cr++;

if (cr == 1) {

omp\_set\_lock(&lock);

printf("lock set by reader %d \n", i);

}

}

printf(

"reader %d (on thread %d) is reading\n",

i,

omp\_get\_thread\_num()

);

sleep(rand() % MAX\_SLEEP);

#pragma omp critical

{

cr--;

if (cr == 0) {

omp\_unset\_lock(&lock);

printf("lock unset by reader %d\n", i);

}

}

}

void writer(int i) {

omp\_set\_lock(&lock);

printf("lock set by writer %d\n", i);

printf(

"writer %d (on thread %d) is writing\n",

i,

omp\_get\_thread\_num()

);

sleep(rand() % MAX\_SLEEP);

omp\_unset\_lock(&lock);

printf("lock unset by writer %d\n", i);

}

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

printf("Readers-writers in parallel\n");

srand(clock());

omp\_init\_lock(&lock);

#pragma omp parallel sections num\_threads(8)

{

#pragma omp section

{

writer(0);

}

#pragma omp section

{

reader(0);

}

#pragma omp section

{

reader(1);

}

#pragma omp section

{

reader(2);

}

#pragma omp section

{

writer(1);

}

#pragma omp section

{

writer(2);

}

#pragma omp section

{

reader(3);

}

#pragma omp section

{

reader(4);

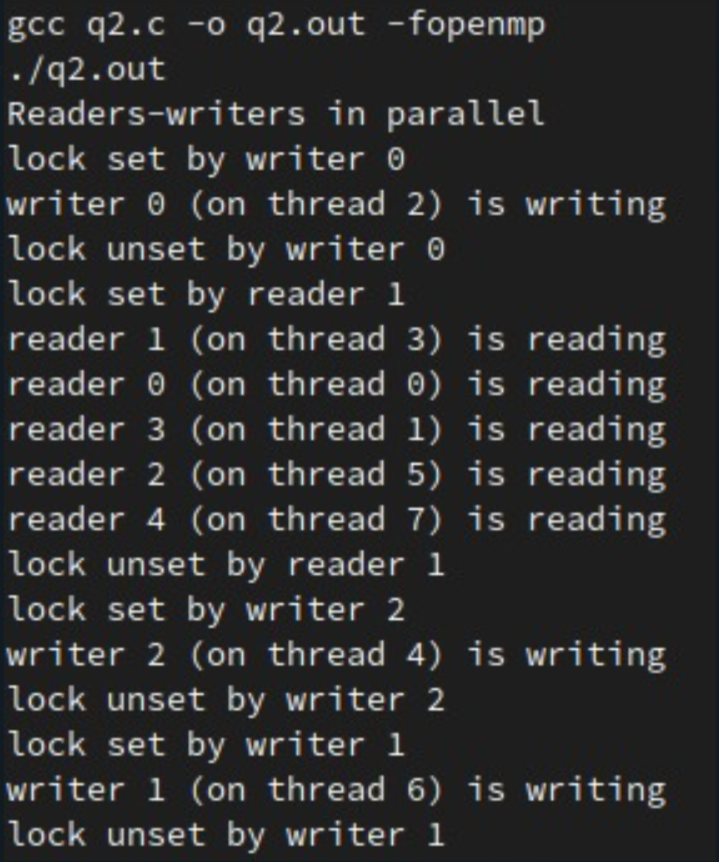
}

}

return 0;

}

**Output:**



**Lab 06: Reductions**

**Q1. Write a program in OpenMP to find out the largest number in an array of 1000000 randomly generated numbers from 1 to 100000 using reduction clause. Compare the versions of serial, parallel for and reduction clause.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <time.h>

#define N 1000000

#define MAX 100000

void findMinSerial(int ar[]) {

int max = 0;

for (int i = 0; i < N; i++) if (ar[i] > max) max = ar[i];

printf("%d is the maximum element\n", max);

}

void findMinParallel(int ar[]) {

int max = 0, i;

#pragma omp parallel for shared(ar, max) private(i)

for (i = 0; i < N; i++) {

if (ar[i] > max) {

#pragma omp critical

{

max = ar[i];

}

}

}

printf("%d is the maximum element\n", max);

}

void findMinReduction(int ar[]) {

int max = 0, i;

#pragma omp parallel for shared(ar) private(i) reduction(max: max)

for (i = 0; i < N; i++) if (ar[i] > max) max = ar[i];

printf("%d is the maximum element\n", max);

}

int main() {

int ar[N];

srand(clock());

for (int i = 0; i < N; i++) ar[i] = rand() % MAX;

double t = omp\_get\_wtime();

findMinSerial(ar);

t = omp\_get\_wtime() - t;

printf("Serial execution took %fs\n", t);

t = omp\_get\_wtime();

findMinParallel(ar);

t = omp\_get\_wtime() - t;

printf("Parallel execution using a critical section took %fs\n", t);

t = omp\_get\_wtime();

findMinReduction(ar);

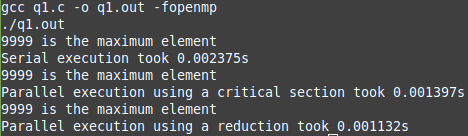
t = omp\_get\_wtime() - t;

printf("Parallel execution using a reduction took %fs\n", t);

return 0;

}

**Output:**

****

**Q2. Write a program in OpenMP to find out the standard deviation of 1000000 randomly generated numbers using reduction clause. Document the development versions of serial, parallel for and reduction clause.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#include <omp.h>

#define N 1000000

#define MAX 100000

void sdSerial(int ar[]) {

double mean = 0;

for (int i = 0; i < N; i++) mean += ar[i];

mean /= N;

double runningSum = 0;

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

runningSum += pow(ar[i] - mean, 2);

}

double sd = sqrt(runningSum / N);

printf("sd = %f\n", sd);

}

void sdParallel(int ar[]) {

double mean = 0;

int i = 0;

#pragma omp parallel for shared(ar, mean) private(i)

for (i = 0; i < N; i++) {

#pragma omp critical

{

mean += ar[i];

}

}

mean /= N;

double runningSum = 0;

#pragma omp parallel for shared(runningSum, mean) private(i)

for (i = 0; i < N; i++) {

#pragma omp critical

{

runningSum += pow(ar[i] - mean, 2);

}

}

double sd = sqrt(runningSum / N);

printf("sd = %f\n", sd);

}

void sdReduction(int ar[]) {

double mean = 0;

int i = 0;

#pragma omp parallel for shared(ar) private(i) reduction(+ : mean)

for (i = 0; i < N; i++) mean += ar[i];

mean /= N;

double runningSum = 0;

#pragma omp parallel for shared(ar, mean) private(i) reduction(+ : runningSum)

for (i = 0; i < N; i++) runningSum += pow(ar[i] - mean, 2);

double sd = sqrt(runningSum / N);

printf("sd = %f\n", sd);

}

int main() {

int ar[N];

srand(clock());

for (int i = 0; i < N; i++) ar[i] = rand() % MAX;

double t = omp\_get\_wtime();

sdSerial(ar);

t = omp\_get\_wtime() - t;

printf("Serial execution took %fs\n\n", t);

t = omp\_get\_wtime();

sdParallel(ar);

t = omp\_get\_wtime() - t;

printf("Parallel execution using a critical section took %fs\n\n", t);

t = omp\_get\_wtime();

sdReduction(ar);

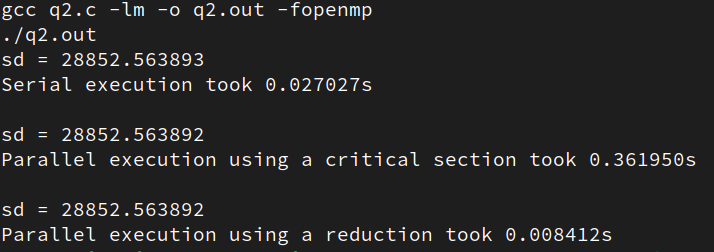
t = omp\_get\_wtime() - t;

printf("Parallel execution using a reduction took %fs\n", t);

return 0;

}

**Output:**

****

**Q3. Write a multithreaded program using OpenMP to implement sequential and parallel version of the Monte Carlo algorithm for approximating Pi. Compare the results of sequential, loop-level parallelism and reduction clause with 10000000 samples.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <math.h>

#include <time.h>

#define N 100000

#define M 100000

void serialMonteCarloPi(double xSamples[], double ySamples[]) {

int counter = 0;

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

double x = xSamples[i];

double y = ySamples[i];

if (x \* x + y \* y < 1) counter++;

}

printf("pi = %f\n", 4.0 \* (double) counter / (double) N);

}

void parallelMonteCarloPi(double xSamples[], double ySamples[]) {

int counter = 0, i;

double x, y;

#pragma omp parallel for shared(xSamples, ySamples, counter) private(i, x, y)

for (i = 0; i < N; i++) {

x = xSamples[i];

y = ySamples[i];

if (x \* x + y \* y < 1) {

#pragma omp critical

{

counter++;

}

}

}

printf("pi = %f\n", 4.0 \* (double) counter / (double) N);

}

void reductionMonteCarloPi(double xSamples[], double ySamples[]) {

int counter = 0, i;

double x, y;

#pragma omp parallel for shared(xSamples, ySamples) private(i, x, y) reduction(+: counter)

for (i = 0; i < N; i++) {

x = xSamples[i];

y = ySamples[i];

if (x \* x + y \* y < 1) counter += 1;

}

printf("pi = %f\n", 4.0 \* (double) counter / (double) N);

}

int main() {

srand(clock());

double xSamples[N], ySamples[N];

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

xSamples[i] = (double)(rand() % M) / M;

ySamples[i] = (double)(rand() % M) / M;

}

double t = omp\_get\_wtime();

serialMonteCarloPi(xSamples, ySamples);

t = omp\_get\_wtime() - t;

printf("Serial execution took %fs\n\n", t);

t = omp\_get\_wtime();

parallelMonteCarloPi(xSamples, ySamples);

t = omp\_get\_wtime() - t;

printf("Parallel execution using a critical section took %fs\n\n", t);

t = omp\_get\_wtime();

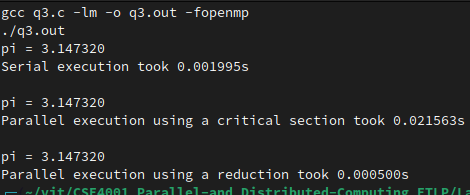
reductionMonteCarloPi(xSamples, ySamples);

t = omp\_get\_wtime() - t;

printf("Parallel execution using a reduction took %fs\n", t);

}

**Output:**



**Lab 07: Profiling**

**Matrix Multiplication**

Nested parallelism is not supported in ompP, so the ‘collapse’ keyword has been used.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

typedef int \*\* matrix;

#define N 1100 // the size of the matrices

void multiply(matrix A, matrix B, matrix ans, int numThreads) {

int i = 0, j = 0, k = 0;

#pragma omp parallel shared(A, B, ans) private(i, j, k) num\_threads(numThreads)

{

#pragma omp for collapse(3)

for (i = 0; i < N; i++) {

for (j = 0; j < N; j++) {

for (k = 0; k < N; k++) {

ans[i][j] += A[i][k] \* B[k][j];

}

}

}

}

}

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

int numThreads = atoi(argv[1]);

matrix A, B, C;

A = (int \*\*)calloc(N, sizeof(int\*));

B = (int \*\*)calloc(N, sizeof(int\*));

C = (int \*\*)calloc(N, sizeof(int\*));

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

A[i] = (int \*) calloc(N, sizeof(int));

B[i] = (int \*) calloc(N, sizeof(int));

C[i] = (int \*) calloc(N, sizeof(int));

for (int j = 0; j < N; j++) {

A[i][j] = rand() % 10;

B[i][j] = rand() % 10;

C[i][j] = 0;

}

}

double t = omp\_get\_wtime();

multiply(A, B, C, numThreads);

t = omp\_get\_wtime() - t;

printf("took %f seconds\n", t);

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

free(A[i]);

free(B[i]);

}

free(A);

free(B);

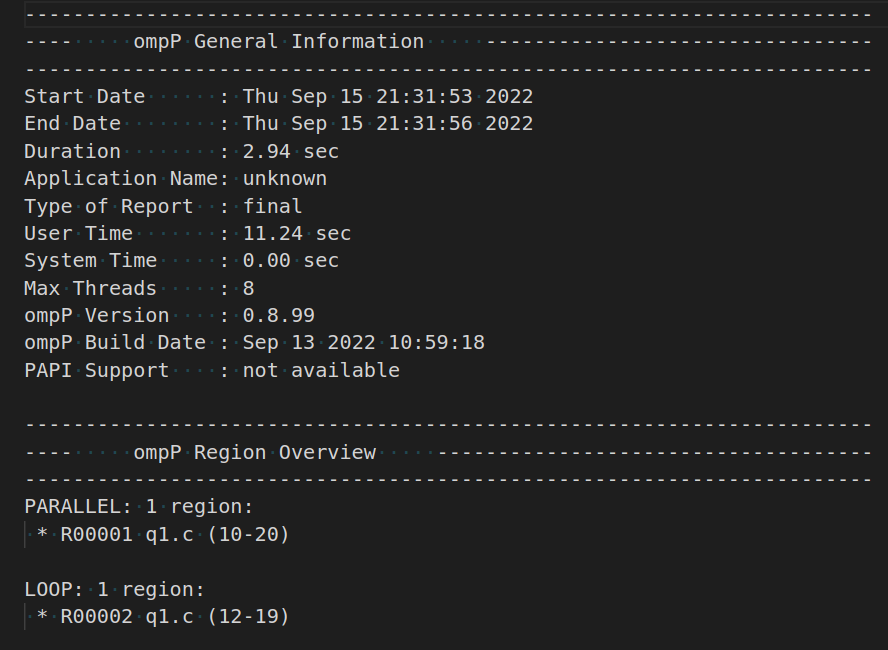
free(C);

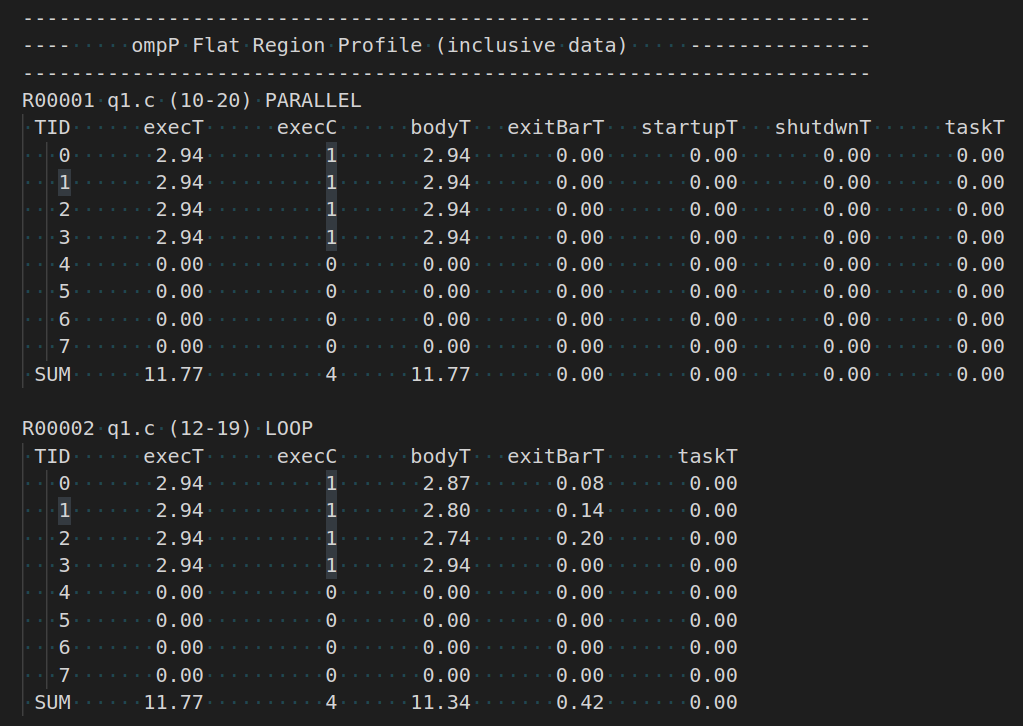
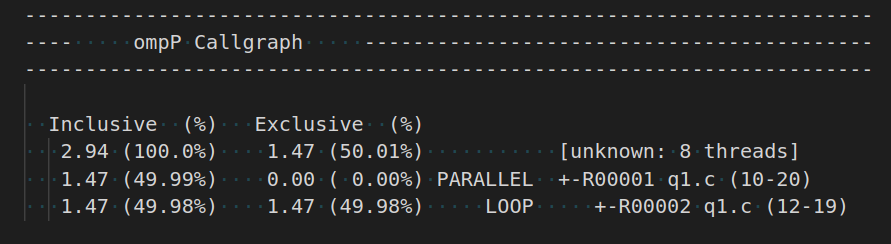
printf("\n");

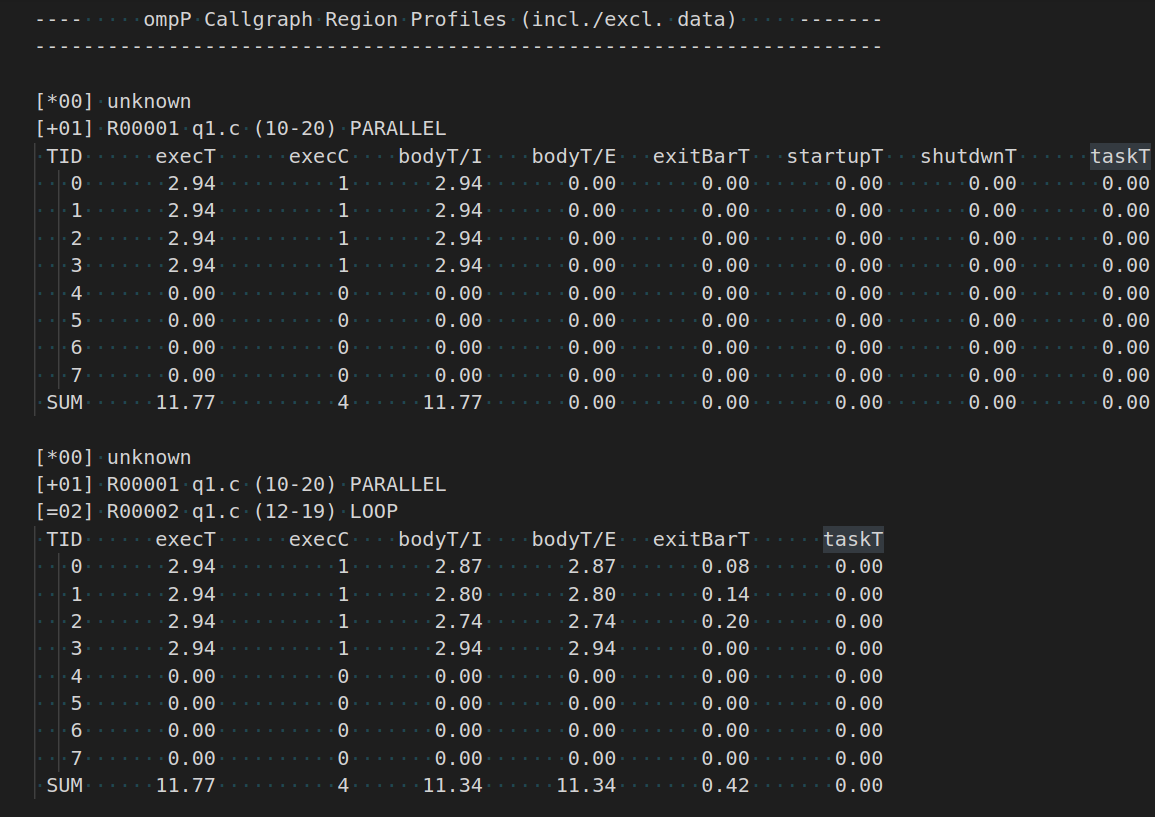
return 0;

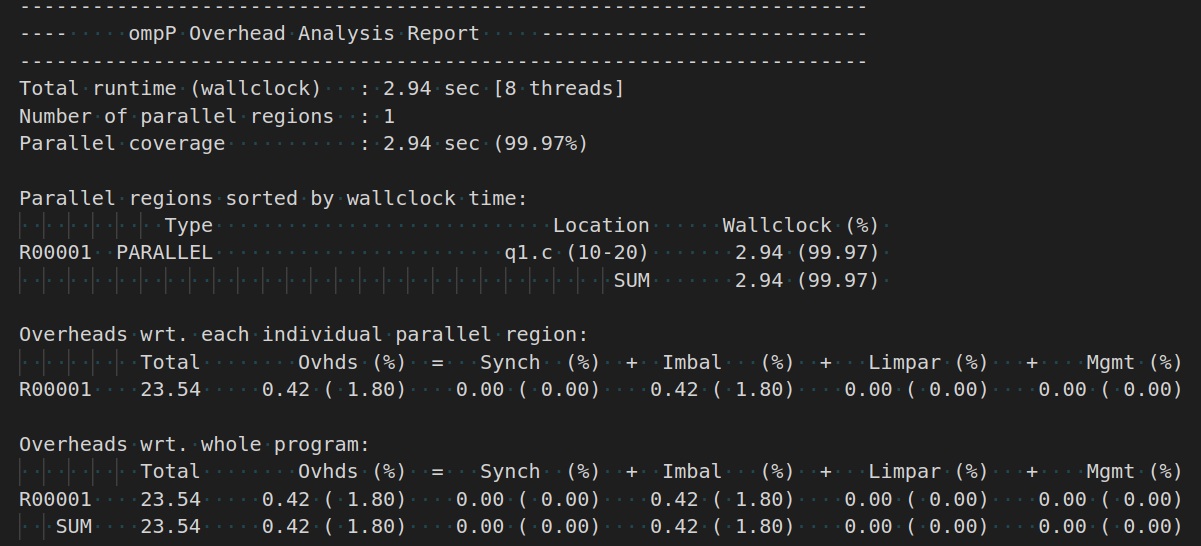
}

**OmpP Profiler Output:**

****

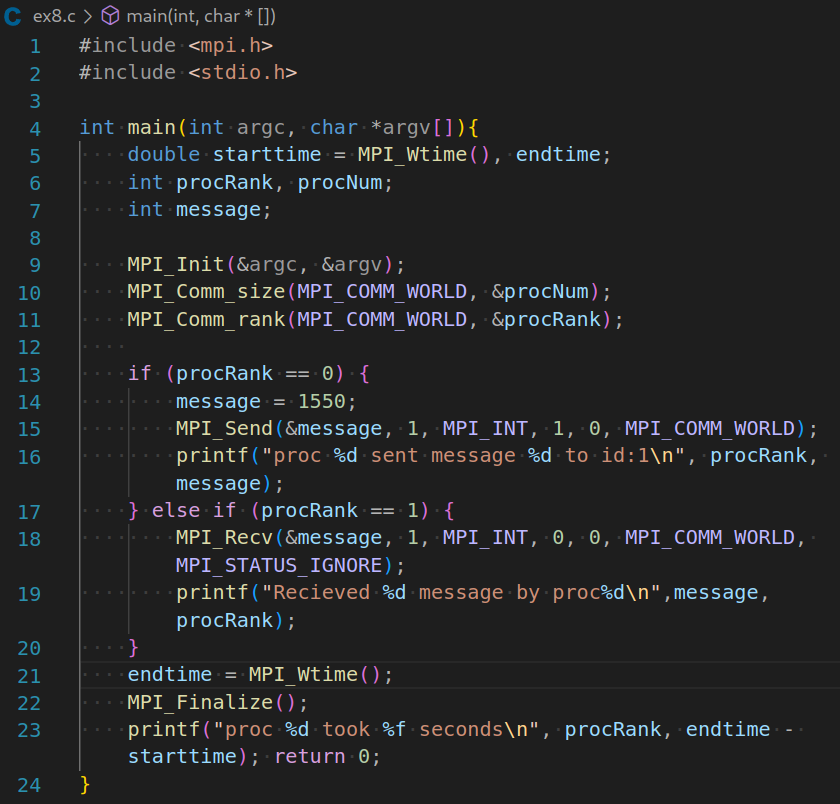
****

****

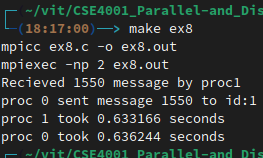


**Lab 8: Simple MPI Program**

**Code:**

****

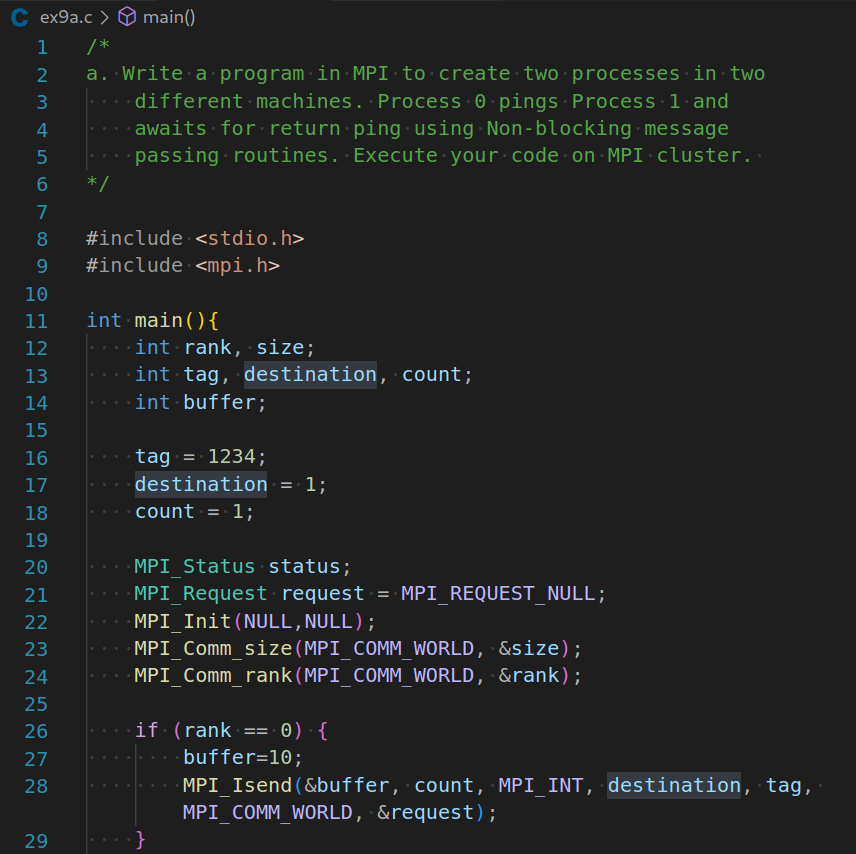
**Output:**

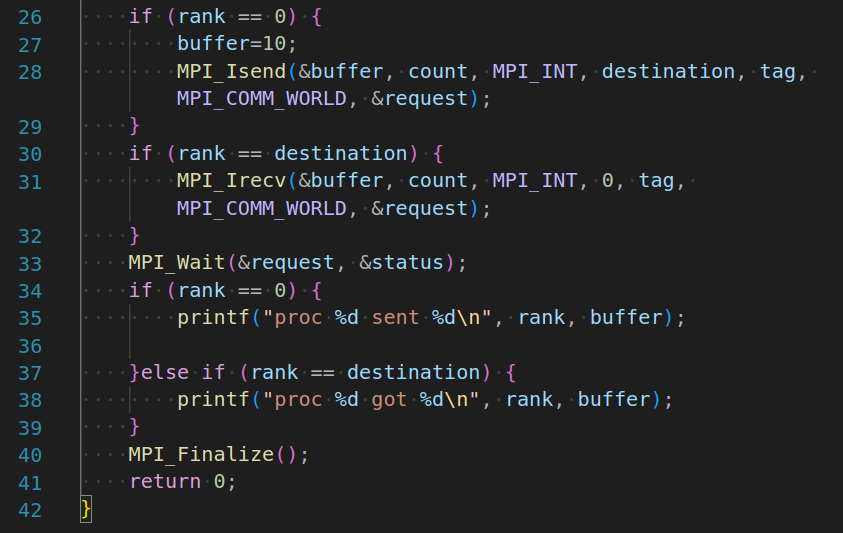


**Lab 9: MPI Point to Point Communication**

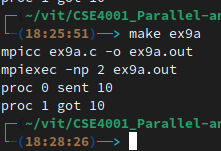
**a:**

**Code:**

****

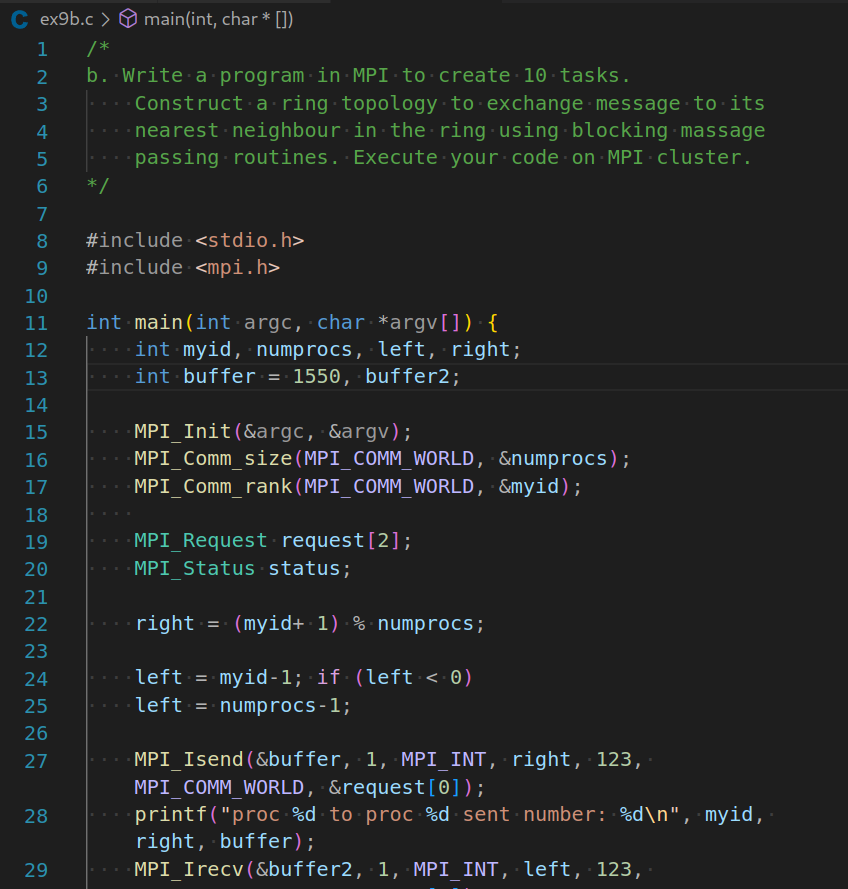


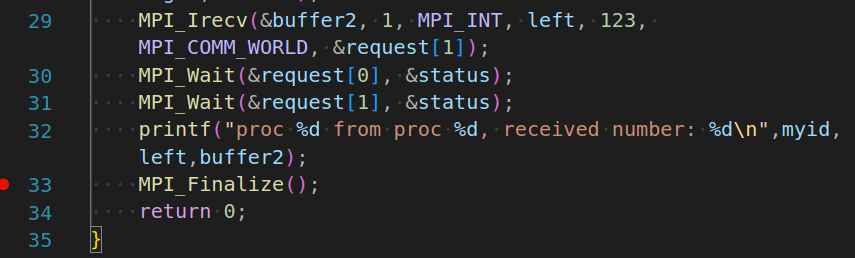
**Output:**

****

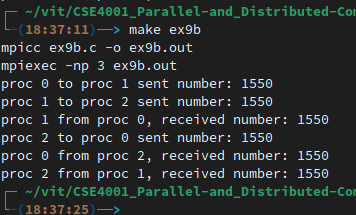
**b:**

**Code:**

****

****

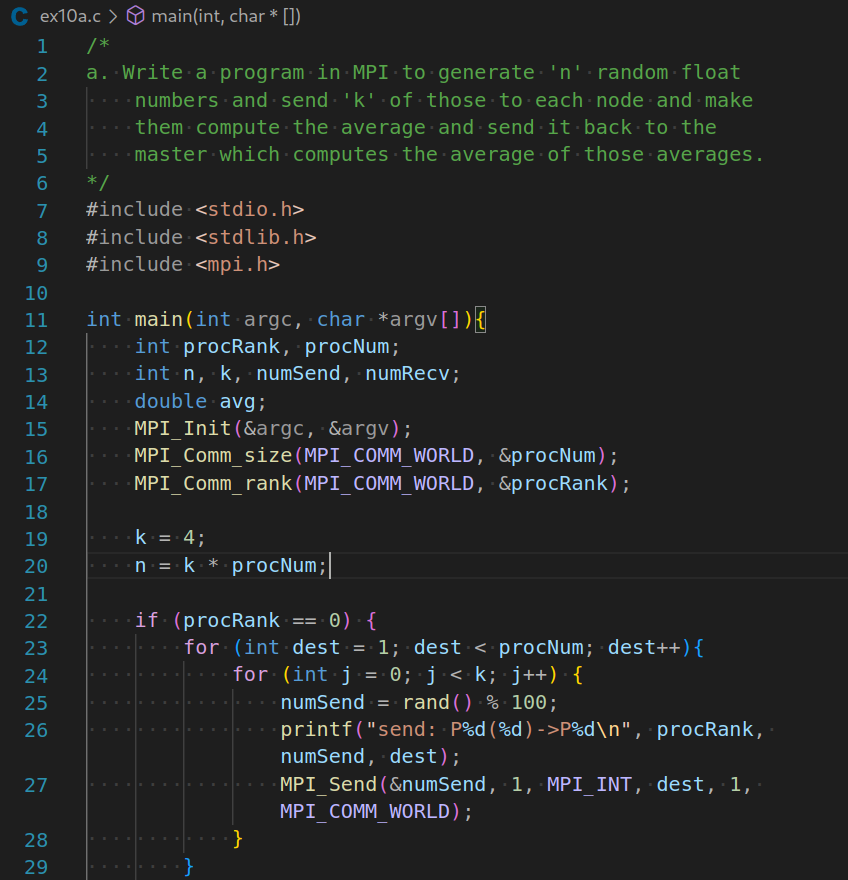
**Output:**

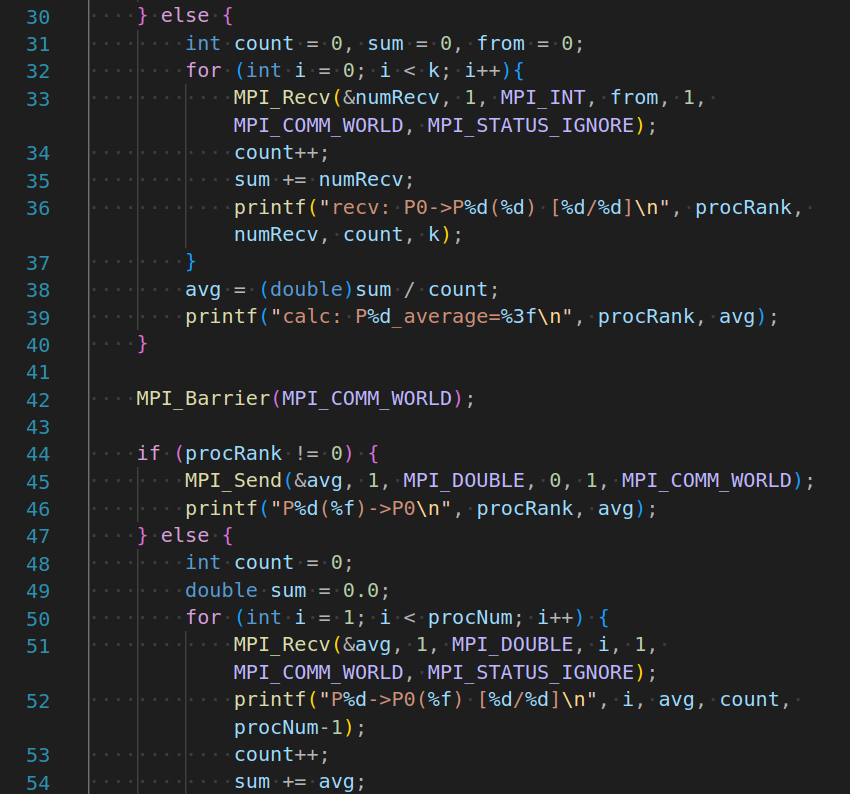


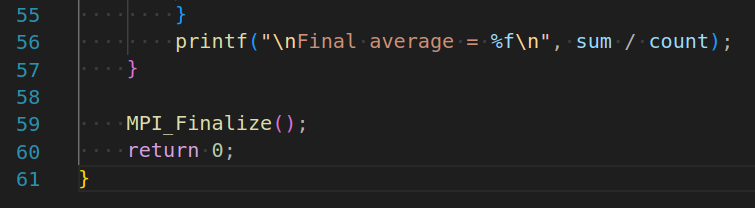
**Lab 10: MPI Collective Communication**

**a:**

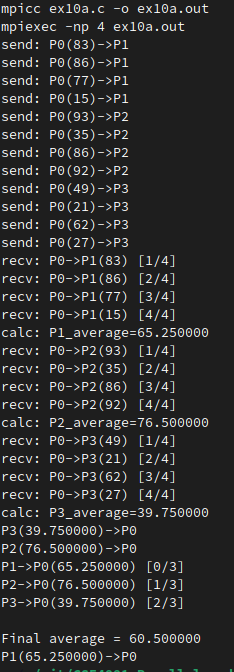
**Code:**

****

****

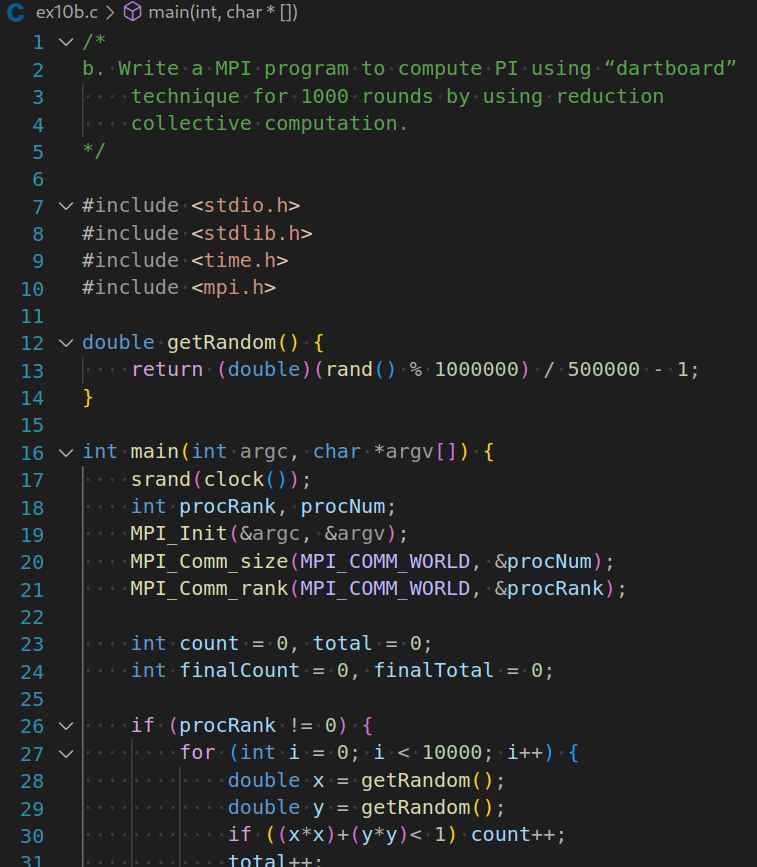
****

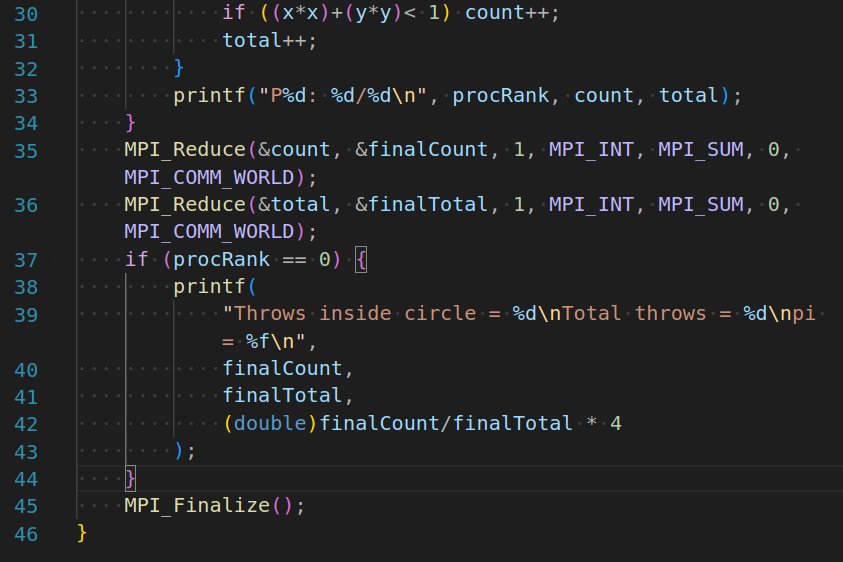
**Output:**

****

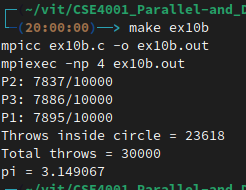
**b:**

**Code:**

****

****

**Output:**

****

**c:**

**Code:**

**/\***

c. Write a MPI program to perform matrix multiplication

(1000x1000) using scatter and gather routines.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <mpi.h>

#include <stdio.h>

#define SIZE 8

int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];

void fill\_matrix(int m[SIZE][SIZE])

{

static int n=0;

int i, j;

for (i=0; i<SIZE; i++)

for (j=0; j<SIZE; j++)

m[i][j] = n++;

}

void print\_matrix(int m[SIZE][SIZE])

{

int i, j = 0;

for (i=0; i<SIZE; i++) {

printf("\n\t| ");

for (j=0; j<SIZE; j++)

printf("%2d ", m[i][j]);

printf("|");

}

}

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

{

int myrank, P, from, to, i, j, k;

int tag = 666; /\* any value will do \*/

MPI\_Status status;

MPI\_Init (&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myrank); /\* who am i \*/

MPI\_Comm\_size(MPI\_COMM\_WORLD, &P); /\* number of processors \*/

/\* Just to use the simple variants of MPI\_Gather and MPI\_Scatter we \*/

/\* impose that SIZE is divisible by P. By using the vector versions, \*/

/\* (MPI\_Gatherv and MPI\_Scatterv) it is easy to drop this restriction. \*/

if (SIZE%P!=0) {

if (myrank==0) printf("Matrix size not divisible by number of processors\n");

MPI\_Finalize();

exit(-1);

}

from = myrank \* SIZE/P;

to = (myrank+1) \* SIZE/P;

/\* Process 0 fills the input matrices and broadcasts them to the rest \*/

/\* (actually, only the relevant stripe of A is sent to each process) \*/

if (myrank==0) {

fill\_matrix(A);

fill\_matrix(B);

}

MPI\_Bcast (B, SIZE\*SIZE, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Scatter (A, SIZE\*SIZE/P, MPI\_INT, A[from], SIZE\*SIZE/P, MPI\_INT, 0, MPI\_COMM\_WORLD);

printf("computing slice %d (from row %d to %d)\n", myrank, from, to-1);

for (i=from; i<to; i++)

for (j=0; j<SIZE; j++) {

C[i][j]=0;

for (k=0; k<SIZE; k++)

C[i][j] += A[i][k]\*B[k][j];

}

MPI\_Gather (C[from], SIZE\*SIZE/P, MPI\_INT, C, SIZE\*SIZE/P, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (myrank==0) {

printf("\n\n");

print\_matrix(A);

printf("\n\n\t \* \n");

print\_matrix(B);

printf("\n\n\t = \n");

print\_matrix(C);

printf("\n\n");

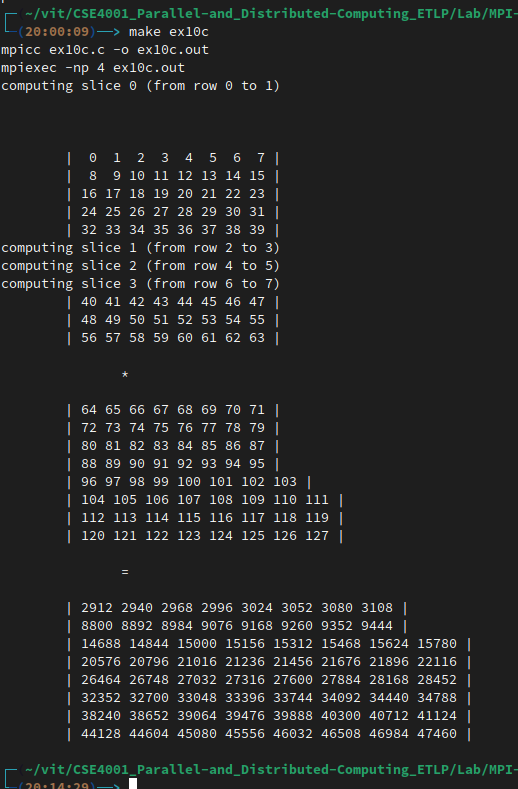
}

MPI\_Finalize();

return 0;

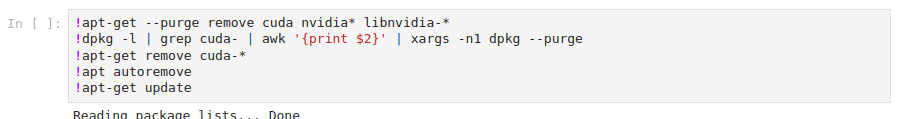
}

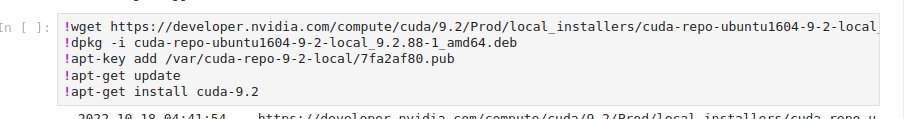
**Output:**



**Lab 11: CUDA Programming**

**Setup:**

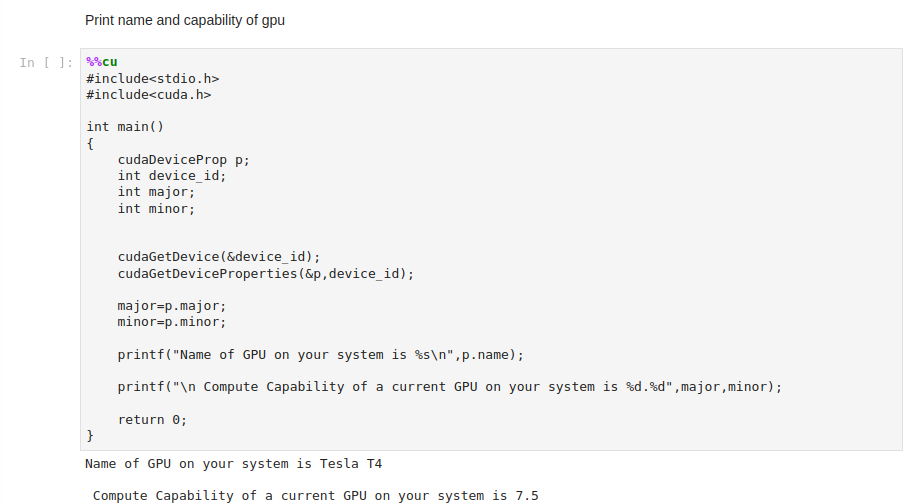


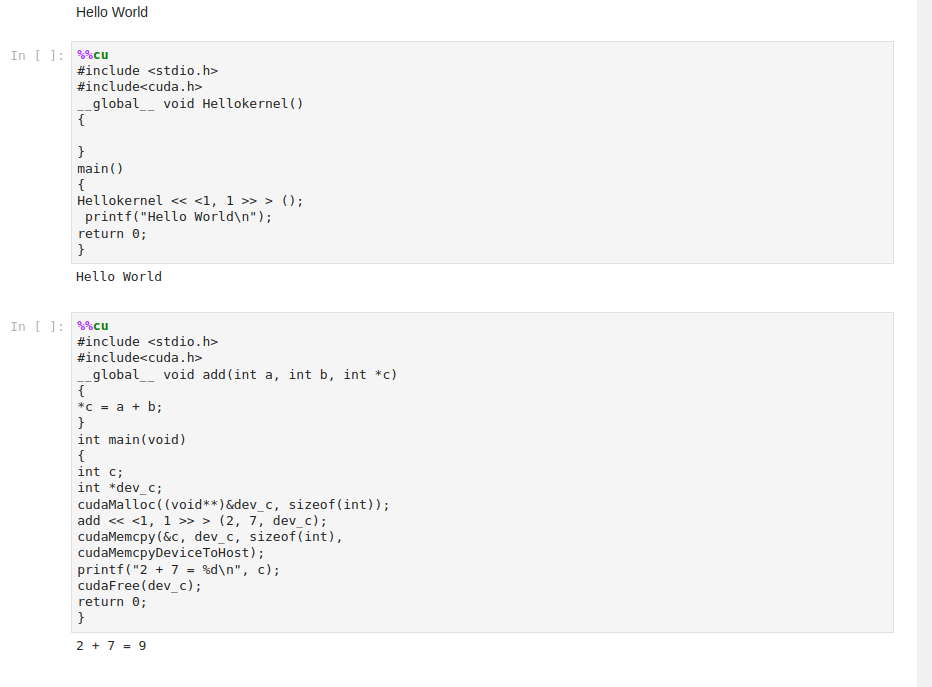


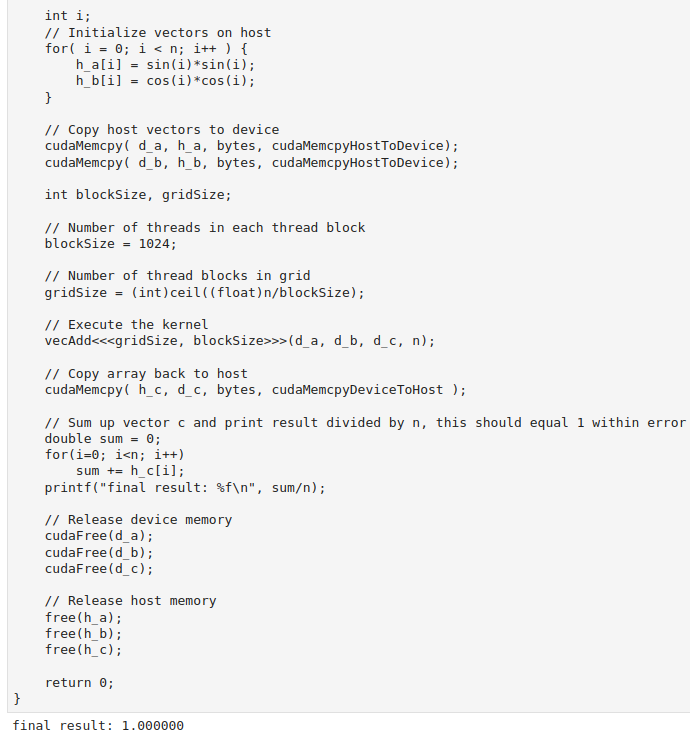
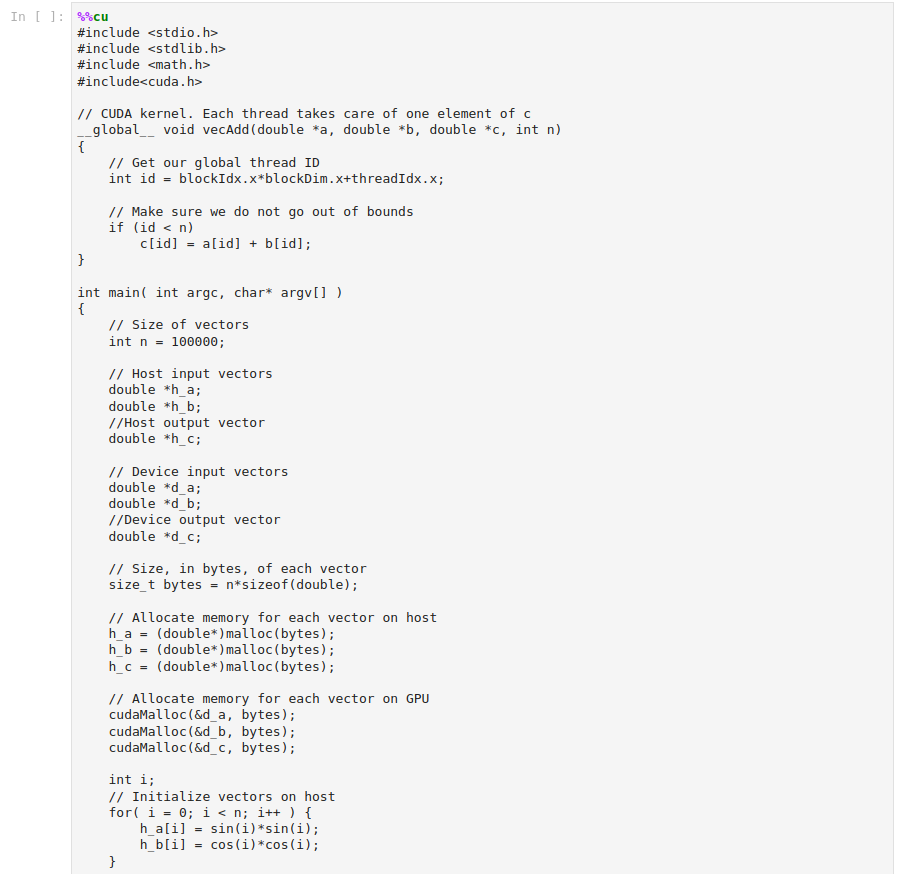


****

**Code:**



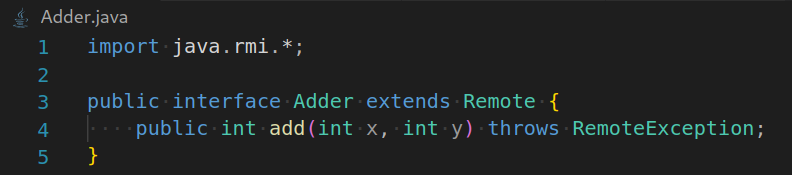




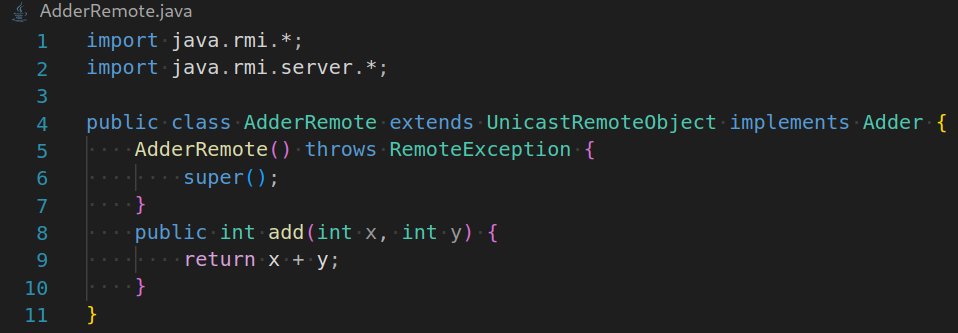
**Lab 12****: Java RMI**

**Code:**

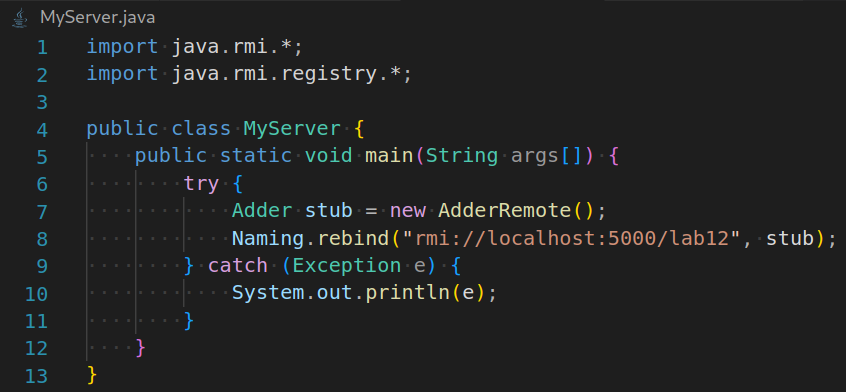
Adder.java



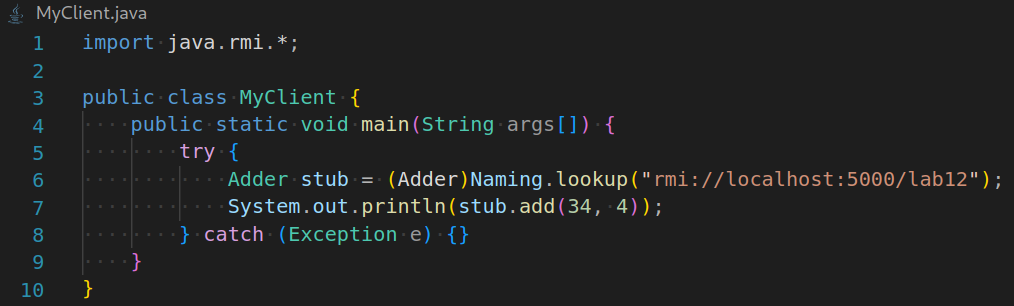
AdderRemote.java



MyServer.java



MyClient.java



Steps:

1. Create all the files and compile them all:



2. Create the stub and skeleton for AdderRemote:



3. Create the RMI Registry:



4. Run MyServer:



5. Run MyClient:

****