|  |  |
| --- | --- |
| Ex-5 | Variable types & Parallel Prefix sum |
| 12/08/2025 |  |

Aim:

To

1. Write a C++ program in which, inside the parallel region, change the datatype of variables private, firstprivate, lastprivate, shared. Explore the default behaviour inside a parallel

region.

1. Write a C++ program with OpenMP directives to find the parallel prefix sum. Compare with the serial version of the code for array sizes changing from

{10,100,1000,10000,100000,1000000}.

1. Find an application with a while loop which can be parallelised using the slave model. Compare the speedup you achieve post-parallelisation.

Algorithm:

1)

1. start
2. declare variables for shared,private,firstprivate and last private
3. open openmp section and modify and access the variables
4. print the changes in the variables 5)stop

2)

1. start
2. implement functions for prefix sum in parallel and prefix sum in serial
3. In main function call both function and measure time 4)stop

3)

1. start
2. implement functions for performing monte carlo estimation of pi using serial and parallel methods
3. In main functions call the functions and compare by measuring time
4. stop

CODE:

1)

#include <iostream> #include <omp.h>

int main() {

int shared\_var=10; int private\_var=20;

int firstprivate\_var=30; int lastprivate\_var=40;

std::cout<<"Before parallel region:"<<std::endl; std::cout<<"shared\_var = "<<shared\_var<<std::endl; std::cout<<"private\_var = "<<private\_var<<std::endl; std::cout<<"firstprivate\_var = "<<firstprivate\_var<<std::endl; std::cout<<"lastprivate\_var = "<<lastprivate\_var<<std::endl;

#pragma omp parallel for \ shared(shared\_var) \ private(private\_var) \ firstprivate(firstprivate\_var) \ lastprivate(lastprivate\_var)

for(int i=0;i<omp\_get\_max\_threads();i++){ int default\_var=100;

shared\_var+=i; private\_var+=i; firstprivate\_var+=i; lastprivate\_var=i; default\_var+=i;

#pragma omp critical

{

std::cout<<"Thread "<<omp\_get\_thread\_num()<<": "

<<"shared\_var = "<<shared\_var<<", "

<<"private\_var = "<<private\_var<<", "

<<"firstprivate\_var = "<<firstprivate\_var<<", "

<<"lastprivate\_var = "<<lastprivate\_var<<", "

<<"default\_var = "<<default\_var<<std::endl;

}

}

std::cout << "\nAfter parallel region:" << std::endl; std::cout << "shared\_var = " << shared\_var << std::endl;

std::cout << "private\_var = " << private\_var << " (unchanged in master thread)" << std::endl; std::cout << "firstprivate\_var = " << firstprivate\_var << " (unchanged in master thread)" <<

std::endl;

std::cout << "lastprivate\_var = " << lastprivate\_var << " (value from last iteration/thread)" <<

std::endl;

return 0;

}

2)

#include <iostream> #include <vector> #include <chrono> #include <omp.h> #include <cassert>

void prefix\_sum\_serial(const std::vector<int>& input, std::vector<int>& output) { output[0]=input[0];

for (size\_t i=1;i<input.size();i++) { output[i]=output[i-1]+input[i];

}

}

void prefix\_sum\_parallel(const std::vector<int>& input, std::vector<int>& output) { int n=input.size();

int num\_threads=1;

#pragma omp parallel

{

#pragma omp single num\_threads=omp\_get\_num\_threads();

}

std::vector<int> partial\_sums(num\_threads+1,0); #pragma omp parallel

{

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

int chunk\_size=(n+num\_threads-1)/num\_threads; int start=tid\*chunk\_size;

int end=std::min(start+chunk\_size,n);

if(start<n){ output[start]=input[start];

for (int i=start+1;i<end;i++) { output[i]=output[i-1]+input[i];

}

partial\_sums[tid+1]=output[end-1]; // store sum of this chunk

}

}

for (int i=1;i<num\_threads+1;i++){ partial\_sums[i]+=partial\_sums[i - 1];

}

#pragma omp parallel

{

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

int chunk\_size=(n+num\_threads-1)/num\_threads; int start=tid\*chunk\_size;

int end=std::min(start+chunk\_size,n);

if(tid>0&&start<n){

int add\_value=partial\_sums[tid]; for(int i=start;i<end;i++){

output[i]+=add\_value;

}

}

}

}

int main() {

std::vector<int> sizes={10,100,1000,10000,100000,1000000};

for (int n : sizes){ std::vector<int> input(n);

std::vector<int> output\_serial(n); std::vector<int> output\_parallel(n);

for (int i=0;i<n;i++) { input[i] = i+1;

}

auto start\_serial=std::chrono::high\_resolution\_clock::now(); prefix\_sum\_serial(input, output\_serial);

auto end\_serial=std::chrono::high\_resolution\_clock::now();

auto start\_parallel=std::chrono::high\_resolution\_clock::now(); prefix\_sum\_parallel(input, output\_parallel);

auto end\_parallel=std::chrono::high\_resolution\_clock::now();

bool correct=true;

for (int i=0;i<n;i++) { if(output\_serial[i]!=output\_parallel[i]) {

correct=false; break;

}

}

auto serial\_time=std::chrono::duration<double, std::milli>(end\_serial - start\_serial).count(); auto parallel\_time=std::chrono::duration<double, std::milli>(end\_parallel -

start\_parallel).count();

std::cout<<"Array size: "<<n<<"\n"; std::cout<<"Serial time: "<<serial\_time<<" ms\n"; std::cout<<"Parallel time: "<<parallel\_time<<" ms\n";

std::cout<<"Results match? "<<(correct ? "Yes" : "No")<<"\n\n";

}

return 0;

}

3)

#include <iostream> #include <random> #include <chrono> #include <omp.h>

double monte\_carlo\_serial(long long num\_points) { std::mt19937\_64 rng(42); std::uniform\_real\_distribution<double> dist(0.0, 1.0); long long inside\_circle=0;

long long i=0;

while(i<num\_points){ double x=dist(rng); double y=dist(rng);

if(x\*x+y\*y<=1.0) inside\_circle++;

i++;

}

return 4.0\*inside\_circle/num\_points;

}

double monte\_carlo\_parallel(long long num\_points, int num\_threads) { long long inside\_circle = 0;

#pragma omp parallel num\_threads(num\_threads)

{

std::mt19937\_64 rng(42 + omp\_get\_thread\_num()); std::uniform\_real\_distribution<double> dist(0.0, 1.0); long long local\_count=0;

#pragma omp for

for (long long i=0;i<num\_points;i++) { double x=dist(rng);

double y=dist(rng);

if (x\*x+y\*y<=1.0) local\_count++;

}

#pragma omp atomic inside\_circle+=local\_count;

}

return 4.0\*inside\_circle/num\_points;

}

int main(){

long long num\_points=1e8; int num\_threads=4;

std::cout<<"Estimating pi using "<<num\_points<<" points\n"; std::cout<<"Using "<<num\_threads<<" threads for parallel version\n\n";

auto start\_serial=std::chrono::high\_resolution\_clock::now(); double pi\_serial=monte\_carlo\_serial(num\_points);

auto end\_serial=std::chrono::high\_resolution\_clock::now(); std::chrono::duration<double> time\_serial=end\_serial-start\_serial;

std::cout<<"Serial pi ≈ "<<pi\_serial<<"\n";

std::cout<<"Serial Time: "<<time\_serial.count()<<" seconds\n\n";

auto start\_parallel=std::chrono::high\_resolution\_clock::now(); double pi\_parallel=monte\_carlo\_parallel(num\_points, num\_threads); auto end\_parallel=std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> time\_parallel=end\_parallel-start\_parallel;

std::cout<<"Parallel pi ≈ "<<pi\_parallel<<"\n";

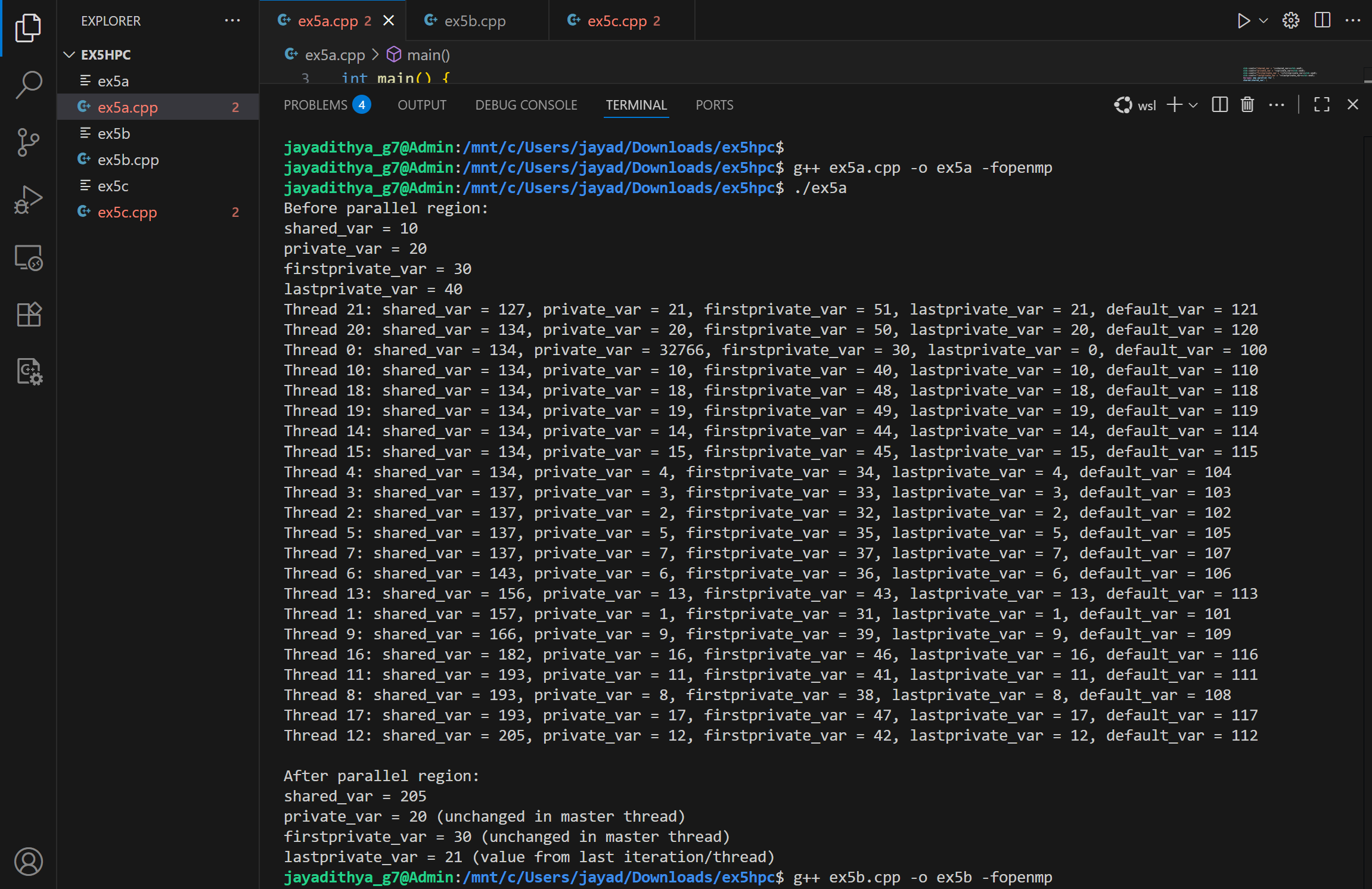
std::cout<<"Parallel Time: "<<time\_parallel.count()<<" seconds\n";

double speedup=time\_serial.count()/time\_parallel.count(); std::cout<<"\nSpeedup: "<<speedup<<"×\n";

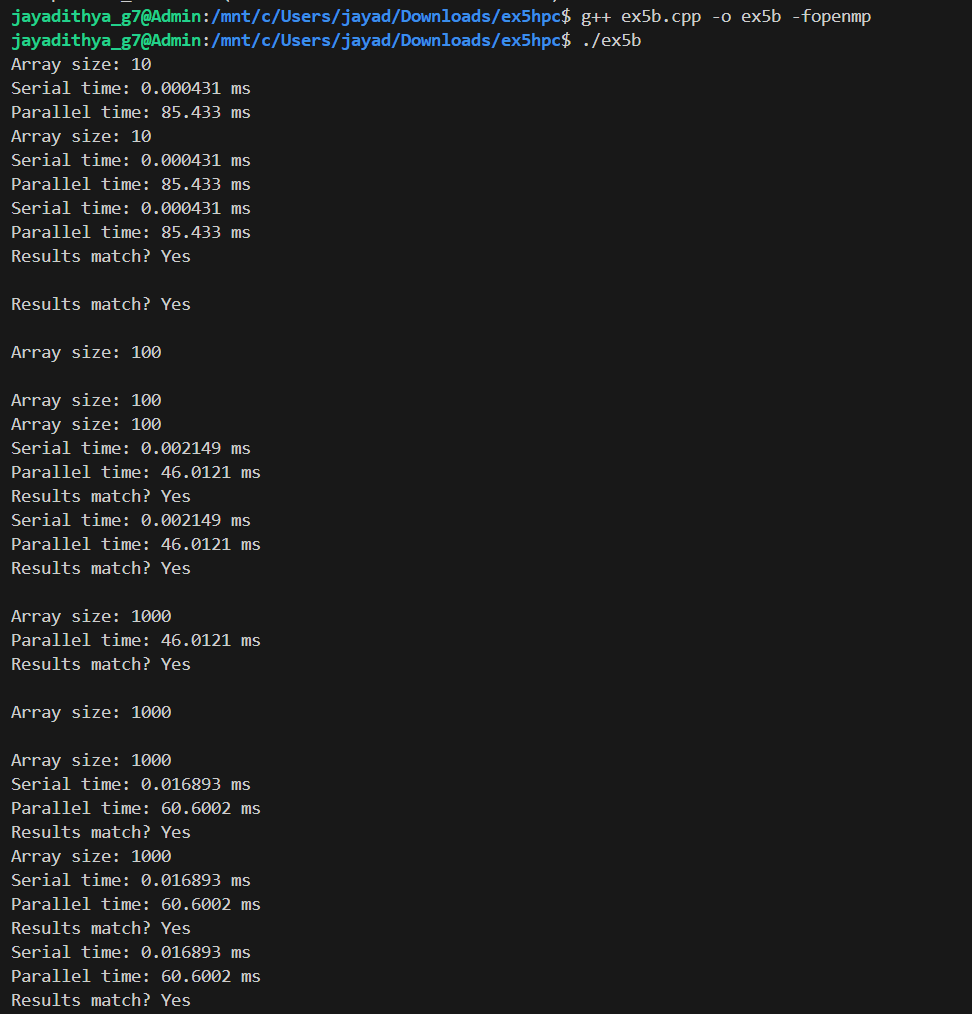
return 0;

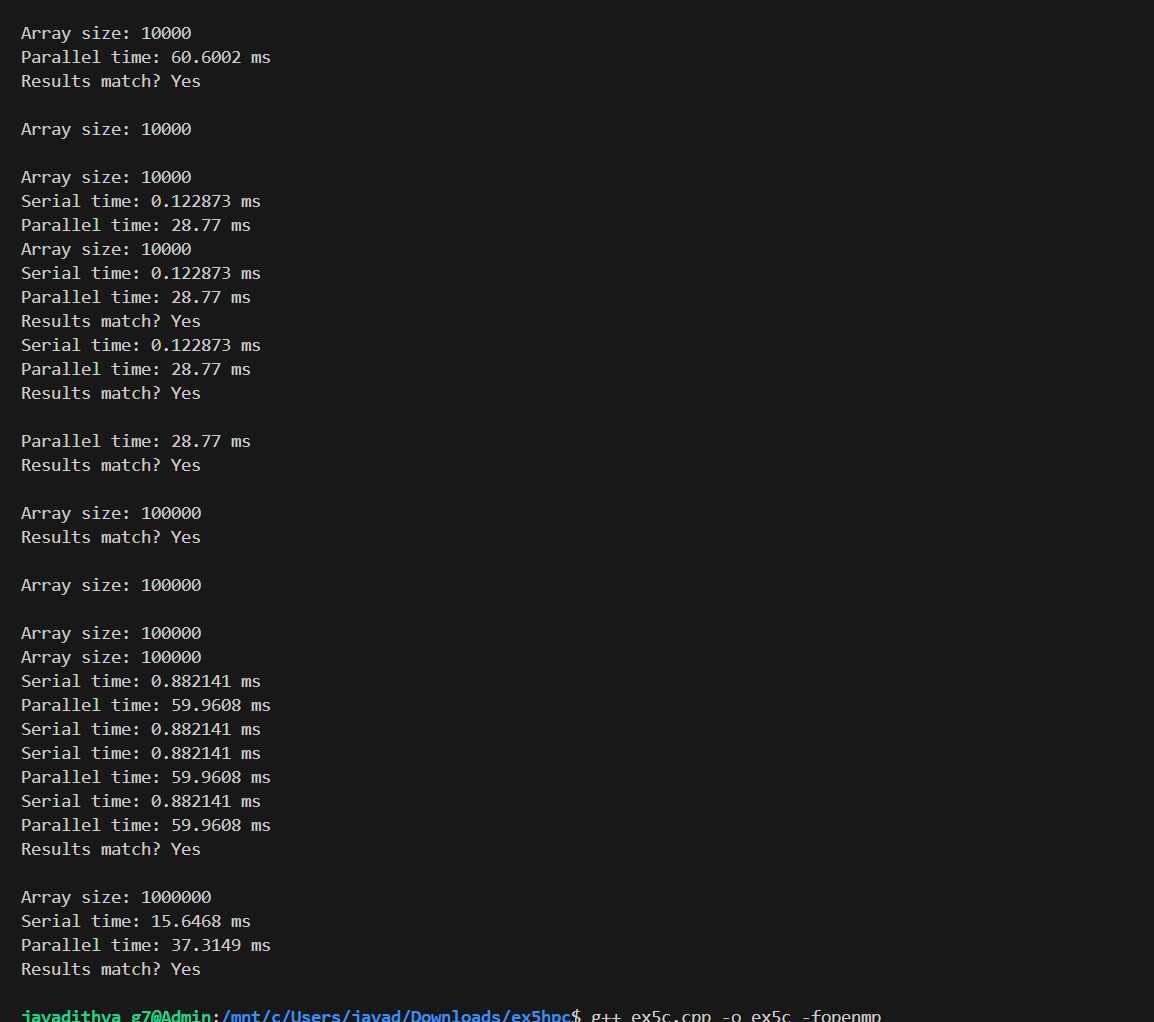
}

OUTPUT: 1)

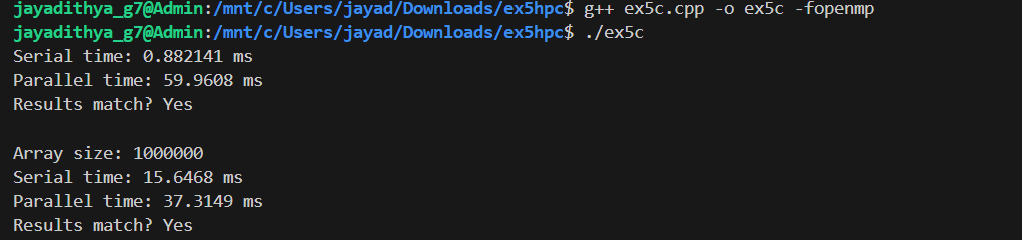


2)





3)



Result:-

Hence the c++ programs are executed successfully and output has been verified.