**Program-3**

**Write a program for Cache unfriendly sieve of Eratosthenes and Cache friendly Sieve of Eratosthenes for enumerating prime numbers upto N and prove the correctness.**

#include<math.h>

#include<string.h>

#include<omp.h>

#include<iostream>

using namespace std;

double t=0.0;

inline long Strike(bool composite[], long i, long stride, long limit) {

for (; i <= limit; i += stride)

composite[i] = true;

return i;

}

long min(long a, long b){

return a > b ? b : a;

}

long CacheUnfriendlySieve(long n)

{

long count = 0;

long m = (long)sqrt((double)n);

bool\* composite = new bool[n + 1];

memset(composite, 0, n);

t = omp\_get\_wtime();

for (long i = 2; i <= m; ++i)

if (!composite[i]) {

++count;

// Strike walks array of size n here.

Strike(composite, 2 \* i, i, n);

}

for (long i = m + 1; i <= n; ++i)

if (!composite[i]) {

++count;

}

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

delete[] composite;

return count;

}

long CacheFriendlySieve(long n)

{

long count = 0;

long m = (long)sqrt((double)n);

bool\* composite = new bool[n + 1];

memset(composite, 0, n);

long\* factor = new long[m];

long\* striker = new long[m];

long n\_factor = 0;

t = omp\_get\_wtime();

for (long i = 2; i <= m; ++i)

if (!composite[i])

{

++count;

striker[n\_factor] = Strike(composite, 2 \* i, i, m);

factor[n\_factor++] = i;

}

// Chops sieve into windows of size sqrt(n)

for (long window = m + 1; window <= n; window += m)

{

long limit = min(window + m - 1, n);

for (long k = 0; k < n\_factor; ++k)

// Strike walks window of size sqrt(n) here.

striker[k] = Strike(composite, striker[k], factor[k], limit);

for (long i = window; i <= limit; ++i)

if (!composite[i])

++count;

}

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

delete[] striker;

delete[] factor;

delete[] composite;

return count;

}

long ParallelSieve(long n){

long count = 0;

long m = (long)sqrt((double)n);

long n\_factor = 0;

long\* factor = new long[m];

t = omp\_get\_wtime();

#pragma omp parallel

{

bool\* composite = new bool[m + 1];

long\* striker = new long[m];

#pragma omp single

{

memset(composite, 0, m);

for (long i = 2; i <= m; ++i)

if (!composite[i])

{

++count;

Strike(composite, 2 \* i, i, m);

factor[n\_factor++] = i;

}

}

long base = -1;

#pragma omp for reduction (+:count)

for (long window = m + 1; window <= n; window += m)

{

memset(composite, 0, m);

if (base != window)

{

// Must compute striker from scratch.

base = window;

for (long k = 0; k < n\_factor; ++k)

striker[k] = (base + factor[k] - 1) / factor[k] \* factor[k] - base;

} long limit = min(window + m - 1, n) - base;

for (long k = 0; k < n\_factor; ++k)

striker[k] = Strike(composite, striker[k], factor[k], limit) - m;

for (long i = 0; i <= limit; ++i)

if (!composite[i])

++count;

base += m;

}

delete[] striker;

delete[] composite;

}

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

delete[] factor;

return count;

}

int main(){

long size = 10000,count;

printf("Size\t\tCache Unfriendly\tCache Friendly\t\tParallel Sieve\n");

for(int i=1; i<=4; i++ ){

size = size\*10;

printf("%ld\t",size);

if(i<3)

printf("\t");

count = CacheUnfriendlySieve(size);

printf("%ld\t%f\t",count,t);

count = CacheFriendlySieve(size);

printf("%ld\t%f\t",count,t);

count = ParallelSieve(size);

printf("%ld\t%f\n",count,t);

}

return 0;

}



