TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN TOÁN ỨNG DỤNG VÀ TIN HỌC

—————o0o—————

BÁO CÁO GIỮA KỲ

MỘT SỐ BÀI TẬP TRÊN OPENMP

Học phần: **Tính toán song song**

Giảng viên hướng dẫn: **TS. ĐOÀN DUY TRUNG**

Sinh viên: **Nguyễn Công Hiếu**

Lớp: **Toán tin 02 - khóa 64**

HÀ NỘI, 11/2021

Lời nói đầu

Trong thời đại của cuộc cách mạng công nghệ số, tốc độ là một yếu tố quan trọng gắn liền với các bài toán lớn của thế giới như dự báo thời tiết, xử lý dữ liệu, nghiên cứu vũ trụ - hàng không, ... Tuy nhiên, phần cứng máy tính đang cho thấy dấu hiệu của sự chững lại. Khi mà, các linh kiện máy tính đã gần đạt tới giạn hạn thu nhỏ của nó. Và trong bối cảnh đó, tính toán hiệu hiệu năng cao (song song) được cho là một trong những giải pháp hiệu quả nhất.

Đáp ứng nhu cầu đó, học phần Tính toán song song đã được thêm vào trong các chương trình học tập của nhiều trường Đại học. Trong bài báo cáo này, em xin trình bày một số bài toán "nhập môn" dựa theo phương pháp song song với bộ nhớ chia sẻ và thư viện OpenMP trên C/C++.

Giới thiệu chung

Phần cứng máy tính:

- CPU Core i7-4900(4910) MQ (4 nhân, 8 luồng)
- Các chương trình được viết bằng ngôn ngữ C/C++ và OpenMP
- Sử dụng hệ điều hành Window 10 và Linux(Ubuntu) chạy trên máy ảo VMware.
- Code được viết và build trên Visual Studio Code.

Mục lục

Chương 1

Nhân vector với ma trận

1.1 Thuật toán tuần tự

```c
#include <stdio.h>
#include <stdlib.h>
#include <random>
#include <conio.h>
#include <windows.h>

double LiToDouble(LARGE_INTEGER x);
double GetTime();

void ProcessInitialization(double* &pMat, double* &pVec, double* &pRes, int &Size);
void ProcessTermination(double* pMat, double* pVec, double* pRes);
void DummyDataInitialization(double* pMat, double* pVec, double* pRes, int Size);
void RandomDataInitialization(double* pMath, double* pVec, int Size);

void PrintMatrix(double* pMat, int RowCount, int ColCount);
void PrintVector(double* pVec, int Size);

void SerialResultCalculation(double* pMat, double* pVec, double* pRes, int Size);

int main()
{
    double* pMatrix;
    double* pVector;
    double* pResult;
    double start, finish, duration;
    int Size;

    printf ("Serial matrix-vector multiplication program\n");
    ProcessInitialization(pMatrix, pVector, pResult, Size);

    printf("\nInitial Matrix:\n");
    // PrintMatrix(pMatrix, Size, Size);
    printf("\nInitial Vector:\n");
    // PrintVector(pVector, Size);

    start = GetTime();
    SerialResultCalculation(pMatrix, pVector, pResult, Size);
    finish = GetTime();
    duration = finish - start;

    printf("\nResult Vector:\n");
    // PrintVector(pResult, Size);
```

```
43      printf("\nTime of execution: %f\n", duration);
44
45      ProcessTermination(pMatrix, pVector, pResult);
46      return 0;
47  }
48
49  double LiToDouble(LARGE_INTEGER x)
50  {
51      double result = ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
52      return result;
53  }
54  double GetTime()
55  {
56      LARGE_INTEGER lpFrequency, lpPerformanceCount;
57      QueryPerformanceFrequency (&lpFrequency);
58      QueryPerformanceCounter (&lpPerformanceCount);
59      return LiToDouble(lpPerformanceCount)/LiToDouble(lpFrequency);
60  }
61
62  void ProcessTermination(double* pMat, double* pVec, double* pRes)
63  {
64      delete [] pMat;
65      delete [] pVec;
66      delete [] pRes;
67  }
68  void ProcessInitialization(double* &pMat, double* &pVec, double* &pRes, int &Size)
69  {
70      do {
71      printf("\nEnter size of the initial objects: ");
72      scanf("%d", &Size);
73      printf("\nChosen objects size = %d", Size);
74      if (Size <= 0) {
75          printf("\nSize of objects must be greater than 0!\n");
76      }
77      } while (Size <= 0);
78
79      pMat = new double[Size*Size];
80      pVec = new double[Size];
81      pRes = new double[Size];
82
83      // DummyDataInitialization(pMat, pVec, pRes, Size);
84      RandomDataInitialization(pMat, pVec, Size);
85  }
86  void DummyDataInitialization(double* pMat, double* pVec, double* pRes, int Size)
87  {
88      int i, j;
89
90      for (i = 0; i < Size; i++) {
91          pVec[i] = 1;
92          for (j = 0; j < Size; j++) {
93              pMat[i*Size+j] = i;
94          }
95      }
96  }
97  void RandomDataInitialization(double* pMath, double* pVec, int Size)
98  {
99      int i, j;
100     srand(unsigned(clock()));
101     for (i = 0; i < Size; i++) {
102         pVec[i] = rand()/double(1000);
103         for (j = 0; j < Size; j++) {
104             pMath[i*Size+j] = rand()/double(1000);
105         }
106     }
107 }
108
109 void PrintMatrix(double* pMat, int RowCount, int ColCount) {
110     int i, j;
```

```
111
112     for (i = 0; i < RowCount; i++) {
113         for (j = 0; j < ColCount; j++) {
114             printf("%7.4f ", pMat[i*ColCount+j]);
115         }
116         printf("\n");
117     }
118 }
119 void PrintVector(double* pVec, int Size) {
120     int i;
121     for (i=0; i<Size; i++) {
122         printf("%7.4f ", pVec[i]);
123     }
124     printf("\n");
125 }
126
127 void SerialResultCalculation(double* pMat, double* pVec, double* pRes, int Size)
128 {
129     int i, j;
130
131     for (i = 0; i < Size; i++) {
132         pRes[i] = 0;
133         for (j = 0; j < Size; j++) {
134             pRes[i] += pMat[j*Size+i]*pVec[j];
135         }
136     }
137 }
```

1.2 Thuật toán song song

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <random>
4  #include <conio.h>
5  #include <windows.h>
6
7  double LiToDouble(LARGE_INTEGER x);
8  double GetTime();
9
10 void ProcessInitialization(double* &pMat, double* &pVec, double* &pRes, int &Size);
11 void ProcessTermination(double* pMat, double* pVec, double* pRes);
12 void DummyDataInitialization(double* pMat, double* pVec, double* pRes, int Size);
13 void RandomDataInitialization(double* pMath, double* pVec, int Size);
14
15 void PrintMatrix(double* pMat, int RowCount, int ColCount);
16 void PrintVector(double* pVec, int Size);
17
18 void SerialResultCalculation(double* pMat, double* pVec, double* pRes, int Size);
19 void ParallelResultCalculation(double* pMat, double* pVec, double* pRes, int Size);
20 void TestResult(double* pMat, double* pVec, double* pRes, int Size);
21
22 int main()
23 {
24     double* pMatrix;
25     double* pVector;
26     double* pResult;
```

```
    double start, finish, duration;
    int Size;

    printf ("Serial matrix-vector multiplication program\n");
    ProcessInitialization(pMatrix, pVector, pResult, Size);

    printf("\nInitial Matrix:\n");
    // PrintMatrix(pMatrix, Size, Size);
    printf("\nInitial Vector:\n");
    // PrintVector(pVector, Size);

    start = GetTime();
    ParallelResultCalculation(pMatrix, pVector, pResult, Size);
    finish = GetTime();
    duration = finish - start;
    printf("\nTime of execution: %f\n", duration);


    start = GetTime();
    ParallelResultCalculation(pMatrix, pVector, pResult, Size);
    finish = GetTime();
    duration = finish - start;

    // TestResult(pMatrix, pVector, pResult, Size);
    printf("\nTime of execution: %f\n", duration);

    // printf("\nResult Vector:\n");
    // PrintVector(pResult, Size);
    // printf("\nTime of execution: %f\n", duration);

    ProcessTermination(pMatrix, pVector, pResult);
    return 0;
}

double LiToDouble(LARGE_INTEGER x)
{
    double result = ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
    return result;
}
double GetTime()
{
    LARGE_INTEGER lpFrequency, lpPerformanceCount;
    QueryPerformanceFrequency (&lpFrequency);
    QueryPerformanceCounter (&lpPerformanceCount);
    return LiToDouble(lpPerformanceCount)/LiToDouble(lpFrequency);
}

void ProcessTermination(double* pMat, double* pVec, double* pRes)
{
    delete [] pMat;
    delete [] pVec;
    delete [] pRes;
}
void ProcessInitialization(double* &pMat, double* &pVec, double* &pRes, int &Size)
{
    do {
    printf("\nEnter size of the initial objects: ");
    scanf("%d", &Size);
    printf("\nChosen objects size = %d", Size);
    if (Size <= 0) {
        printf("\nSize of objects must be greater than 0!\n");
    }
    } while (Size <= 0);

    pMat = new double[Size*Size];
    pVec = new double[Size];
    pRes = new double[Size];

```

```
94
95      // DummyDataInitialization(pMat, pVec, pRes, Size);
96      RandomDataInitialization(pMat, pVec, Size);
97  }
98  void DummyDataInitialization(double* pMat, double* pVec, double* pRes, int Size)
99  {
100     int i, j;
101
102     for (i = 0; i < Size; i++) {
103         pVec[i] = 1;
104         for (j = 0; j < Size; j++) {
105             pMat[i*Size+j] = i;
106         }
107     }
108 }
109 void RandomDataInitialization(double* pMath, double* pVec, int Size)
110 {
111     int i, j;
112     srand(unsigned(clock()));
113     for (i = 0; i < Size; i++) {
114         pVec[i] = rand()/double(1000);
115         for (j = 0; j < Size; j++) {
116             pMath[i*Size+j] = rand()/double(1000);
117         }
118     }
119 }
120
121 void PrintMatrix(double* pMat, int RowCount, int ColCount) {
122     int i, j;
123
124     for (i = 0; i < RowCount; i++) {
125         for (j = 0; j < ColCount; j++) {
126             printf("%7.4f ", pMat[i*ColCount+j]);
127         }
128         printf("\n");
129     }
130 }
131 void PrintVector(double* pVec, int Size) {
132     int i;
133     for (i=0; i<Size; i++) {
134         printf("%7.4f ", pVec[i]);
135     }
136     printf("\n");
137 }
138
139 void SerialResultCalculation(double* pMat, double* pVec, double* pRes, int Size)
140 {
141     int i, j;
142
143     for (i = 0; i < Size; i++) {
144         pRes[i] = 0;
145         for (j = 0; j < Size; j++) {
146             pRes[i] += pMat[i*Size+j]*pVec[j];
147         }
148     }
149 }
150 void ParallelResultCalculation(double* pMat, double* pVec, double* pRes, int Size)
151 {
152     int i, j;
153
154     #pragma omp parallel for private(j)
155     for (i = 0; i < Size; i++) {
156         pRes[i] = 0;
157         for (j = 0; j < Size; j++) {
158             pRes[i] += pMat[i*Size+j]*pVec[j];
159         }
```

```
    }
}
void TestResult(double* pMat, double* pVec, double* pRes, int Size)
{
    double* pSerialResult;
    int equal = 0;
    int i;

    pSerialResult = new double[Size];
    SerialResultCalculation(pMat, pVec, pSerialResult, Size);
    for (i = 0; i < Size; i++) {
        if (pRes[i] != pSerialResult[i]) {
            equal = 1;
        }
    }

    if (equal == 1) {
        printf("\n
The results of serial and parallel algorithms are NOT identical. Check your code!\n");
    }
    else {
        printf("\nThe results of serial and parallel algorithms are identical.");
    }
    delete [] pSerialResult;
}
```

1.3 Đánh giá

Số luồng N = 8

STT	Kích thước ma trận	Thời gian thực hiện tuần tự	Thời gian thực hiện song song	Hiệu suất
Test 01	10	0,000001	0,000819	0,0001526251526
Test 02	100	0,000039	0,001032	0,0047238372093
Test 03	1000	0,005227	0,001954	0,3343781985670
Test 04	2000	0,012303	0,005556	0,2767953563715
Test 05	3000	0,026002	0,011558	0,2812121474304
Test 06	4000	0,049190	0,020526	0,2995590957810
Test 07	5000	0,077830	0,030720	0,3166910807292
Test 08	6000	0,107972	0,039507	0,3416230035184
Test 09	7000	0,142901	0,056750	0,3147599118943
Test 10	8000	0,182762	0,056298	0,4057915023624
Test 11	9000	0,225388	0,083872	0,3359106734071
Test 12	10000	0,281088	0,096444	0,3643150429265

Chương 2

Tìm số nguyên tố

2.1 Thuật toán tuần tự

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <cstring>
#include <math.h>

void ProcessInitialization(char* &pEratos, int &n, int &Size);
void ProcessTermination(char* &pEratos);

int SerialResult(char* pEratos, int n, int Size);

int main()
{
    int N = 0, Size = 0;
    char* pEratosthenes;

    ProcessInitialization(pEratosthenes, N, Size);

    int res = SerialResult(pEratosthenes, N, Size);

    printf("\n%d\n", res);

    ProcessTermination(pEratosthenes);
    return 0;
}

void ProcessInitialization(char* &pEratos, int &n, int &Size);
```

```
27 void ProcessInitialization(char* &pEratos, int &n, int &Size)
28 {
29     do {
30         printf("\nInput N: ");
31         scanf("%d", &n);
32 
33         if (n < 2) {
34             printf("N must be larger than 2!\n");
35         }
36     } while (n < 2);
37 
38     // Only representing odd numbers
39     Size = (n-1)/2;
40 
41     // Size+1 means the array contains the number n
42     pEratos = new char[Size+1];
43     if (!pEratos) {
44         printf("\nError Allocating Memory!\n");
45         exit(0);
46     }
47     memset(pEratos, 1, Size+1);
48 }
49 void ProcessTermination(char* &pEratos)
50 {
51     delete[] pEratos;
52 }
53 
54 int SerialResult(char* pEratos, int n, int Size)
55 {
56     int i, loop = 1, prime;
57     int number_of_primes = 1;
58 
59     for (i = 3; i*i <= n; i += 2)
60     {
61         if (pEratos[i/2]) {
62             for (int j = i*i; j <= n; j += 2*i) {
63                 pEratos[j/2] = 0;
64             }
65         }
66     }
67 
68     for (i = 1; i < Size; i++)
69     {
70         number_of_primes += pEratos[i];
71     }
72 
73     return number_of_primes;
74 }
```

2.2 Thuật toán song song

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <cstring>
#include <math.h>
#include <omp.h>
#include <windows.h>

double LiToDouble(LARGE_INTEGER x);
double GetTime();

void ProcessInitialization(char* &pEratos, int &n, int &Size);
void ProcessTermination(char* &pEratos);

int SerialResult(char* pEratos, int n, int Size);
int ParallelResult(char* pEratos, int n, int Size, bool useOpenMP);

void Test(int n, int Size);

int main()
{
    int N = 0, Size = 0;
    int res;
    char* pEratosthenes;
    double start, end, duration;

    ProcessInitialization(pEratosthenes, N, Size);

    start = GetTime();
    res = SerialResult(pEratosthenes, N, Size);
    end = GetTime();
    duration = end - start;
    printf("\nIn serialism: %d primes was found.\tTime of execution: %f(s)\n", res,
duration);

    start = GetTime();
    res = ParallelResult(pEratosthenes, N, Size, 1);
    end = GetTime();
    duration = end - start;

    printf("\nIn parallelism: %d primes was found.\tTime of execution: %f(s)\n", res
, duration);

    Test(N, Size);

    ProcessTermination(pEratosthenes);
    return 0;
}

double LiToDouble(LARGE_INTEGER x)
{
    double result = ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
    return result;
}
double GetTime()
{
    LARGE_INTEGER lpFrequency, lpPerformanceCount;
    QueryPerformanceFrequency (&lpFrequency);
```

```
    QueryPerformanceCounter (&lpPerformanceCount);
    return LiToDouble(lpPerformanceCount)/LiToDouble(lpFrequency);
}

void ProcessInitialization(char* &pEratos, int &n, int &Size)
{
    do {
        printf("\nInput N: ");
        scanf("%d", &n);

        if (n < 2) {
            printf("N must be larger than 2!\n");
        }
    } while (n < 2);

    // Only representing odd numbers
    Size = (n-1)/2;

    // Size+1 means the array contains the number n
    pEratos = new char[Size+1];
    if (!pEratos) {
        printf("\nError Allocating Memory!\n");
        exit(0);
    }
    memset(pEratos, 1, Size+1);
}
void ProcessTermination(char* &pEratos)
{
    delete[] pEratos;
}

int SerialResult(char* pEratos, int n, int Size)
{
    int i, loop = 1, prime;
    int number_of_primes = 1;

    for (i = 3; i*i <= n; i += 2)
    {
        if (pEratos[i/2]) {
            for (int j = i*i; j <= n; j += 2*i) {
                pEratos[j/2] = 0;
            }
        }
    }

    for (i = 1; i < Size; i++)
    {
        number_of_primes += pEratos[i];
    }

    return number_of_primes;
}
int ParallelResult(char* pEratos, int n, int Size, bool useOpenMP)
{
    omp_set_num_threads(useOpenMP ? omp_get_num_procs() : 1);

    const int Isqrt = (int)sqrt((double)n);
    int number_of_primes = 1;

    #pragma omp parallel for schedule(dynamic)
        for (int i = 3; i <= Isqrt; i += 2) {
            if (pEratos[i/2]) {
                for (int j = i*i; j <= n; j += 2*i) {
                    pEratos[j/2] = 0;
                }
            }
        }
```

```
124     #pragma omp parallel for reduction(+:number_of_primes)
125         for (int i = 1; i <= Size; i++) {
126             number_of_primes += pEratos[i];
127         }
128 
129     return number_of_primes;
130 }
131 
132 void Test(int n, int Size)
133 {
134     char* p1 = new char[Size+1];
135     char* p2 = new char[Size+1];
136     bool flag;
137     memset(p1, 1, Size+1);
138     memset(p2, 1, Size+1);
139 
140     SerialResult(p1, n, Size);
141     ParallelResult(p2, n, Size, 1);
142 
143     for (int i = 0; i < Size; i++) {
144         if (p1[i] == p2[i]) {
145             flag = 1;
146         }
147         else {
148             flag = 0;
149             break;
150         }
151     }
152 
153     if (flag == 1) {
154         printf("True!");
155     }
156     else {
157         printf("False!");
158     }
159 
160     delete [] p1;
161     delete [] p2;
162 }
```

2.3 Đánh giá

Số luồng N = 8

STT	Kích thước đầu vào	Thời gian thực hiện tuần tự	Thời gian thực hiện song song	Hiệu suất
Test 01	100000	0,000337	0,001025	0,04109756098
Test 02	500000	0,002181	0,001178	0,23143039049
Test 03	1000000	0,004866	0,002537	0,23975167521
Test 04	5000000	0,029578	0,005407	0,68378953209
Test 05	10000000	0,066455	0,011312	0,73434184936
Test 06	50000000	0,399280	0,223278	0,22353299474
Test 07	100000000	0,847871	0,446771	0,23722192130
Test 08	200000000	1,781355	0,936822	0,23768589444
Test 09	400000000	3,751899	2,597461	0,18055607957
Test 10	600000000	5,623977	3,320863	0,21169109506
Test 11	800000000	7,619751	4,419313	0,21552419460
Test 12	1000000000	9,651920	6,263810	0,19261280275

Chương 3

Tính số Fibonacci

3.1 Thuật toán tuần tự

```c
#include <math.h>
#include <stdio.h>

#define GT (1.0 + sqrt(5.0))/2.0

void ProcessInitialization(int &n);

long double SerialResult(unsigned int n);

int main()
{
    int n;
    long double res;

    ProcessInitialization(n);

    res = SerialResult(n);
    printf("Fib(%d): %.0Lf.\n", n, res);

    return 0;
}

void ProcessInitialization(int &n)
{
    do {
        printf("Input a number: ");
```

```
        scanf("%d", &n);
        if (n < 3) {
            printf("Number must be larger than 2");
        }
    } while (n < 3);
}

long double SerialResult(unsigned int n)
{
    long double res;
    long double Phi = GT;
    long double oMinusPhi = 1 - GT;
    for (int i = 0; i < n-1; i++) {
        Phi *= GT;
        oMinusPhi *= (1-GT);
    }
    res = (Phi-oMinusPhi)/sqrt(5);
    return res;
}
```

3.2 Thuật toán song song

```
#include <math.h>
#include <stdio.h>
#include <omp.h>
#include <windows.h>

#define GT (1.0 + sqrt(5.0))/2.0

void ProcessInitialization(int &n);

double LiToDouble(LARGE_INTEGER x);
double GetTime();

long double SerialResult(unsigned int n);
long double ParallelResult(unsigned int n);

void Test(unsigned int n);


int main()
{
    int n;
    long double res;
```

```
    double start, end;

    ProcessInitialization(n);

    start = GetTime();
    res = SerialResult(n);
    end = GetTime();
    printf("Fib(%d): .\nTime: %lf\n", n, end - start);

    start = GetTime();
    res = ParallelResult(n);
    end = GetTime();
    printf("\nFib(%d): .\nTime: %lf", n, end - start);

    Test(n);

    return 0;
}

void ProcessInitialization(int &n)
{
    do {
        printf("Input a number: ");
        scanf("%d", &n);
        if (n < 3) {
            printf("Number must be larger than 2");
        }
    } while (n < 3);
}

double LiToDouble(LARGE_INTEGER x)
{
    double result = ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
    return result;
}
double GetTime()
{
    LARGE_INTEGER lpFrequency, lpPerformanceCount;
    QueryPerformanceFrequency (&lpFrequency);
    QueryPerformanceCounter (&lpPerformanceCount);
    return LiToDouble(lpPerformanceCount)/LiToDouble(lpFrequency);
}

long double SerialResult(unsigned int n)
{
    long double res;
    long double Phi = GT;
    long double oMinusPhi = 1 - GT;
    for (int i = 0; i < n-1; i++) {
        Phi *= GT;
        oMinusPhi *= (1-GT);
    }
    res = (Phi-oMinusPhi)/sqrt(5);
    return res;
}
long double ParallelResult(unsigned int n)
{
    long double res;
    long double Phi = GT;
    long double oMinusPhi = 1 - GT;

    #pragma omp parallel for reduction(*: Phi, oMinusPhi)
        for (int i = 0; i < n-1; i++) {
            Phi *= GT;
            oMinusPhi *= (1-GT);
        }
    res = (Phi-oMinusPhi)/sqrt(5);
```

```
90      return res;
91  }
92
93  void Test(unsigned int n)
94  {
95      long double serialRes = SerialResult(n);
96      long double parallelRes = ParallelResult(n);
97
98      if (serialRes == parallelRes) {
99          printf("\nTrue!");
100     }
101     else {
102         printf("\nFalse!");
103     }
104 }
```

3.3 Đánh giá

Số luồng N = 8

STT	Kích thước đầu vào	Thời gian thực hiện tuần tự	Thời gian thực hiện song song	Hiệu suất
Test 01	10000	0,000052	0,000734	0,0088556
Test 02	100000	0,026419	0,001212	2,7247318
Test 03	200000	0,049595	0,001744	3,5546875
Test 04	400000	0,100480	0,009304	1,3499570
Test 05	600000	0,160637	0,017313	1,1598004
Test 06	800000	0,211080	0,027435	0,9617277
Test 07	1000000	0,260486	0,036020	0,9039631
Test 08	2000000	0,517079	0,083116	0,7776466
Test 09	4000000	1,039489	0,170544	0,7618921
Test 10	6000000	1,542630	0,244294	0,7893307
Test 11	8000000	2,067787	0,325183	0,7948551
Test 12	10000000	2,591170	0,414124	0,7821238

Chương 4

Tính số PI

4.1 Thuật toán tuần tự

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <windows.h>

double LiToDouble(LARGE_INTEGER x);
double GetTime();

void ProcessInitialization(int &n, double& dx);

double SerialResult(int n, double& integral, double dx);

int main()
{
    int N = 0;
    double pi, dx, integral = 0.0;

    ProcessInitialization(N, dx);

    pi = SerialResult(N, integral, dx);

    printf("%f\n", pi);

    return 0;
}

double LiToDouble(LARGE_INTEGER x)
{
    double result = ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
    return result;
}
double GetTime()
{
    LARGE_INTEGER lpFrequency, lpPerformanceCount;
    QueryPerformanceFrequency (&lpFrequency);
    QueryPerformanceCounter (&lpPerformanceCount);
    return LiToDouble(lpPerformanceCount)/LiToDouble(lpFrequency);
}

void ProcessInitialization(int &n, double& dx)
{
```

```
    do {
        printf("Input a number: ");
        scanf("%d", &n);
        if (n <= 0) {
            printf("Number must be positive");
        }
    } while (n <= 0);
    dx = 1.0 / n;
}

double SerialResult(int n, double& integral, double dx)
{
    double fx, x, pi;
    for (int i = 0; i < n; i++) {
        x = i * dx;
        fx = sqrt(1.0 - x*x);
        integral += fx * dx;
    }
    pi = 4 * integral;
    return pi;
}
```

4.2 Thuật toán song song

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>
#include <windows.h>

double LiToDouble(LARGE_INTEGER x);
double GetTime();

void ProcessInitialization(int &n, double& dx);

double SerialResult(int n, double& integral, double dx);
double ParallelResult(int n, double& integral, double dx);

int main()
{
    int N = 0;
    double pi, dx, integral = 0.0;
    double start, end, duration;

    ProcessInitialization(N, dx);

    start = GetTime();
    pi = SerialResult(N, integral, dx);
    end = GetTime();
    duration = end - start;
    printf("PI = %f. Time: %f\n", pi, duration);

    integral = 0.0;

    start = GetTime();
    pi = ParallelResult(N, integral, dx);
```

```
    end = GetTime();
    duration = end - start;
    printf("PI = %f. Time: %f", pi, duration);


    return 0;
}

double LiToDouble(LARGE_INTEGER x)
{
    double result = ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
    return result;
}
double GetTime()
{
    LARGE_INTEGER lpFrequency, lpPerformanceCount;
    QueryPerformanceFrequency (&lpFrequency);
    QueryPerformanceCounter (&lpPerformanceCount);
    return LiToDouble(lpPerformanceCount)/LiToDouble(lpFrequency);
}

void ProcessInitialization(int &n, double& dx)
{
    do {
        printf("Input a number: ");
        scanf("%d", &n);
        if (n <= 0) {
            printf("Number must be positive!\n\n");
        }
        fflush(stdin);
    } while (n <= 0);
    dx = 1.0 / n;
}

double SerialResult(int n, double& integral, double dx)
{
    double fx, x, pi;
    for (int i = 0; i < n; i++) {
        x = i * dx;
        fx = sqrt(1.0 - x*x);
        integral += fx * dx;
    }
    pi = 4 * integral;
    return pi;
}
double ParallelResult(int n, double& integral, double dx)
{
    double fx, x, pi;
    #pragma omp parallel for private(fx, x) reduction(+:integral)
        for (int i = 0; i < n; i++) {
            x = i * dx;
            fx = sqrt(1.0 - x * x);
            integral += fx * dx;
        }
    pi = 4 * integral;
    return pi;
}


```

4.3 Đánh giá

4.3.1 Trên Windows

Số luồng N = 8				
STT	Kích thước đầu vào	Thời gian thực hiện tuần tự	Thời gian thực hiện song song	Hiệu suất
Test 01	1000	0,000017	0,001046	0,002031549
Test 02	10000	0,000085	0,000760	0,013980263
Test 03	50000	0,000625	0,000930	0,084005376
Test 04	100000	0,001299	0,001108	0,146547834
Test 05	500000	0,004823	0,002349	0,256651767
Test 06	1000000	0,009754	0,005217	0,233707111
Test 07	5000000	0,045611	0,012687	0,449387168
Test 08	10000000	0,082219	0,018399	0,558583347
Test 09	50000000	0,373817	0,105756	0,441838997
Test 10	100000000	0,751970	0,201214	0,467145676
Test 11	500000000	3,819332	0,977041	0,488635073
Test 12	1000000000	7,459011	1,959673	0,475781610

4.3.2 Trên Linux

Số luồng N = 8

STT	Kích thước đầu vào	Thời gian thực hiện tuần tự	Thời gian thực hiện song song	Hiệu suất
Test 01	1000	0,000021	0,000648	0,004050926
Test 02	10000	0,000215	0,000750	0,035833333
Test 03	50000	0,001014	0,000811	0,156288533
Test 04	100000	0,002112	0,000862	0,306264501
Test 05	500000	0,008222	0,001654	0,621372430
Test 06	1000000	0,017015	0,002612	0,814270674
Test 07	5000000	0,088780	0,011876	0,934447625
Test 08	10000000	0,166154	0,022882	0,907667599
Test 09	50000000	0,797847	0,114091	0,874134463
Test 10	100000000	1,578868	0,217466	0,907537270
Test 11	500000000	7,876455	1,062698	0,926469114
Test 12	1000000000	15,660792	2,558728	0,765067252

Kết luận

Do năng lực còn hạn chế cũng như thời gian có hạn, nên bài báo cáo còn xuất hiện nhiều sai sót, code còn sơ sài, một số bài toán kết quả chỉ mang tính chất minh họa do không tìm được cấu trúc dữ liệu phù hợp với bài toán (số lớn - Fibonacci, số lượng số sau dấu phẩy - PI). Ở phần đánh giá trên Linux, em sử dụng hệ điều hành ubuntu trên máy ảo nên kết quả đánh giá có thể không được chính xác khi so sánh với những máy thật sự chạy trên Linux.
Rất mong nhận được những lời nhận xét của thầy cũng như những người đọc được bài báo cáo này.

Tài liệu tham khảo

[1] Bulić Patricio, Robič Borut, Slivnik Boštjan, Trobec Roman, *Introduction to Parallel Computing From Algorithms to Programming on State-of-the-Art Platforms*, 2018.

[2] Tim Mattson, *Introduction to OpenMP*, Youtube.

[3] https://hpc.llnl.gov/training/tutorials.

[4] https://create.stephan-brumme.com/eratosthenes.