Tính toán song song

Báo cáo giữa kì lập trình OpenMP

Giảng viên hướng dẫn: Thầy Đoàn Duy Trung

Sinh viên thực hiện: Nguyễn Thị Duyên 20195866

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

VIỆN TOÁN ỨNG DỤNG VÀ TIN HỌC
School of Applied Mathematics and Informatics

Mục lục

Lời nói đầu

Tính toán song song là một học phần quan trọng trong ngành công nghệ thông tin. Công nghệ và kĩ thuật ngày càng phát triển, dữ liệu ngày càng lớn đòi hỏi cần phải có một kĩ thuật tính toán giúp cho việc đưa ra kết quả được nhanh hơn, có ý nghĩa về mặt thời gian, tận dụng được tài nguyên phi cục bộ, tiết kiệm chi phí và vượt ra ngoài được giới hạn bộ nhớ của máy tính, ... Từ đó đã dẫn đến sự ra đời của kĩ thuật tính toán song song.

Chúng ta có thể lập trình tính toán song song trên nhiều nền tảng khác nhau. Và trong phạm vi của bài báo cáo này, em đã chọn sử dụng lập trình OpenMp trên nền ngôn ngữ C/C++. Nội dung của bài báo cáo bao gồm kết quả chạy chương trình và mã nguồn của các chương trình nhân ma trận với vector, tìm số Fibonacci, tìm các số nguyên tố, tính tích phân xác định; em đã thực hiện chạy chương trình trên cả Windows và phần mềm giả lập Linux.

Để có thể hoàn thành bài báo cáo này, em xin được gửi lời cảm ơn chân thành và sâu sắc đến thầy **TS. Đoàn Duy Trung**, thầy đã tận tình giảng dạy và hướng dẫn em trong suốt quá trình học tập và làm bài báo cáo.

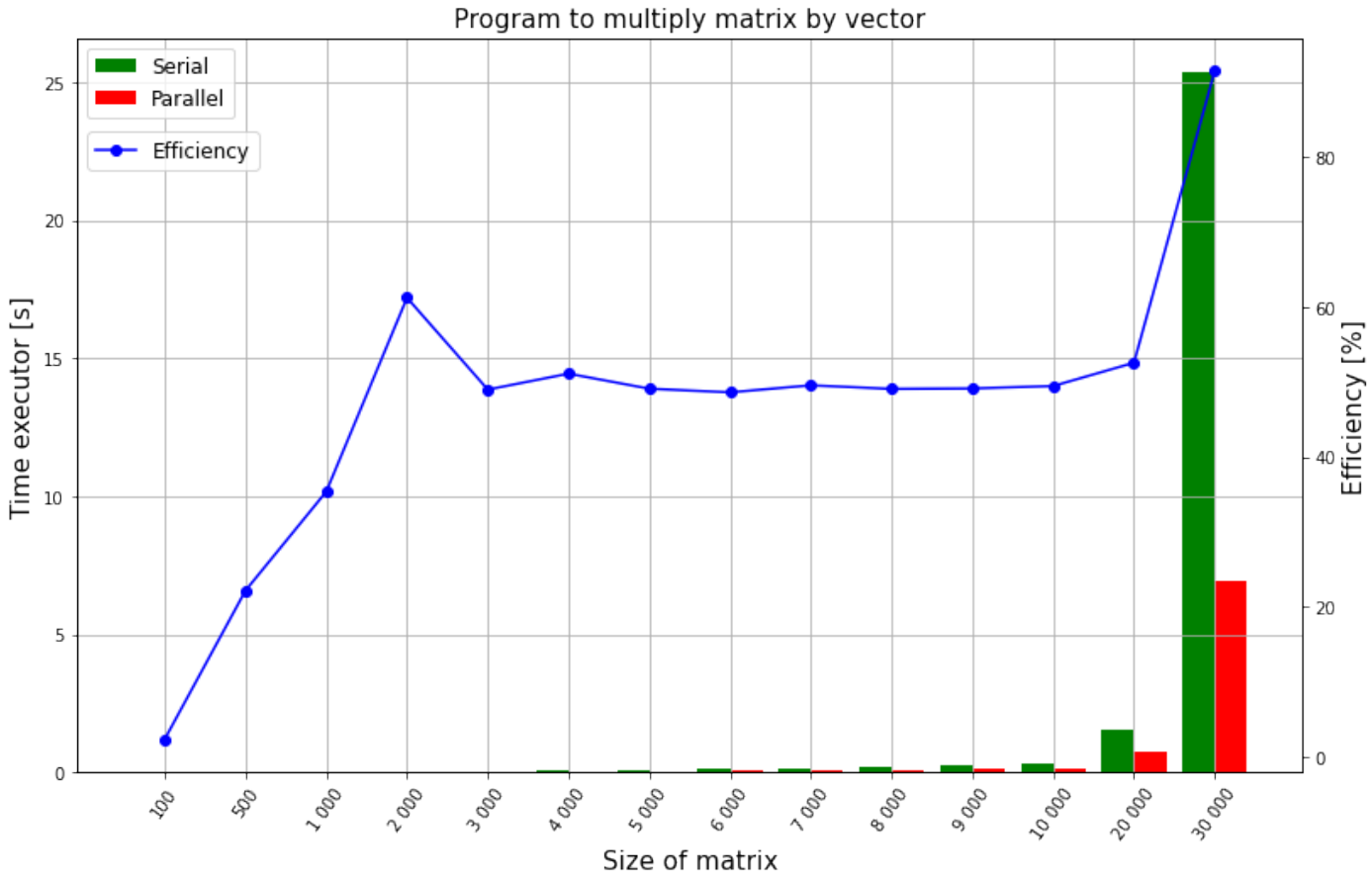
I Bảng kết quả

1 Bài tập chung

1.1 Nhân ma trận với vector

Bảng 1: Kết quả chạy chương trình nhân ma trận với vector

STT	Kích thước ma trận	Thời gian thực hiện tuần tự (s)	Thời gian thực hiện song song (s)	Hiệu suất (%)
Test No1	100	0.000058	0.000627	2.306937
Test No2	500	0.000845	0.000954	22.1398
Test No3	1 000	0.003454	0.002443	35.34425
Test No4	2 000	0.021042	0.008593	61.21695
Test No5	3 000	0.031627	0.016148	48.9653
Test No6	4 000	0.057078	0.027929	51.09215
Test No7	5 000	0.082590	0.042068	49.08135
Test No8	6 000	0.118604	0.061013	48.59835
Test No9	7 000	0.160506	0.081002	49.5376
Test No10	8 000	0.209203	0.106611	49.0576
Test No11	9 000	0.266426	0.135605	49.11805
Test No12	10 000	0.330038	0.166890	49.4394
Test No13	20 000	1.569242	0.746575	52.548
Test No14	30 000	25.356385	6.928511	91.4929

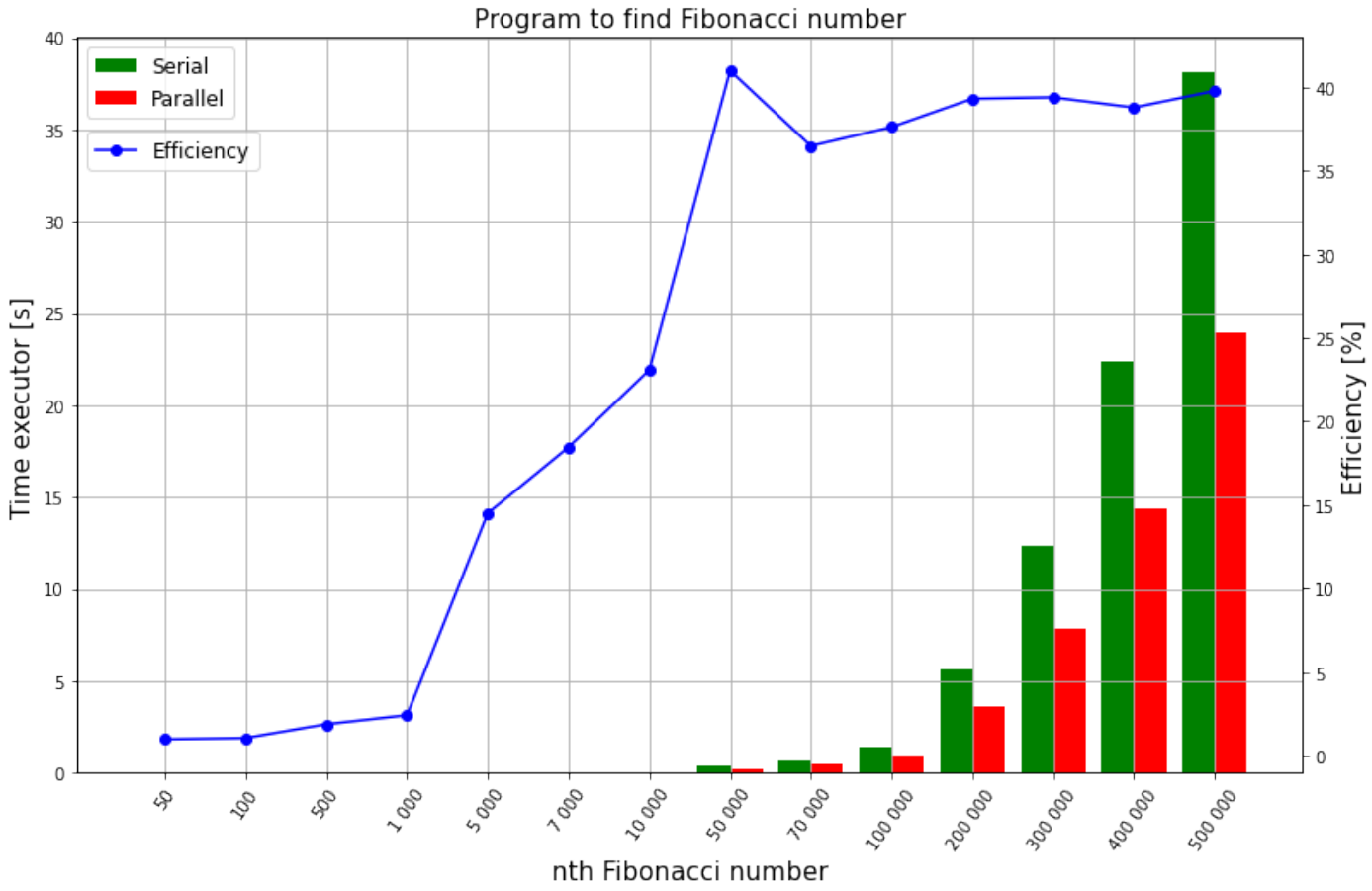


Hình 1: Biểu đồ kết quả chạy chương trình nhân ma trận với vector

1.2 Tìm số Fibonacci

Bảng 2: Kết quả chạy chương trình tìm số Fibonacci

STT	Số tự nhiên n	Thời gian thực hiện tuần tự (s)	Thời gian thực hiện song song (s)	Hiệu suất (%)
Test No1	50	9.64103e-05	0.00247426	0.974135
Test No2	100	0.000114872	0.00273313	1.050735
Test No3	500	0.00024041	0.00320492	1.87532
Test No4	1 000	0.000361846	0.00374441	2.41591
Test No5	5 000	0.00395159	0.0068119	14.50255
Test No6	7 000	0.00765169	0.0103639	18.45755
Test No7	10 000	0.014928	0.0161723	23.0765
Test No8	50 000	0.357307	0.217853	41.00325
Test No9	70 000	0.654374	0.448311	36.49105
Test No10	100 000	1.40937	0.936477	37.6242
Test No11	200 000	5.60525	3.5639	39.31965
Test No12	300 000	12.3127	7.81257	39.40015
Test No13	400 000	22.3397	14.3973	38.79155
Test No14	500 000	38.1209	23.9539	39.7857

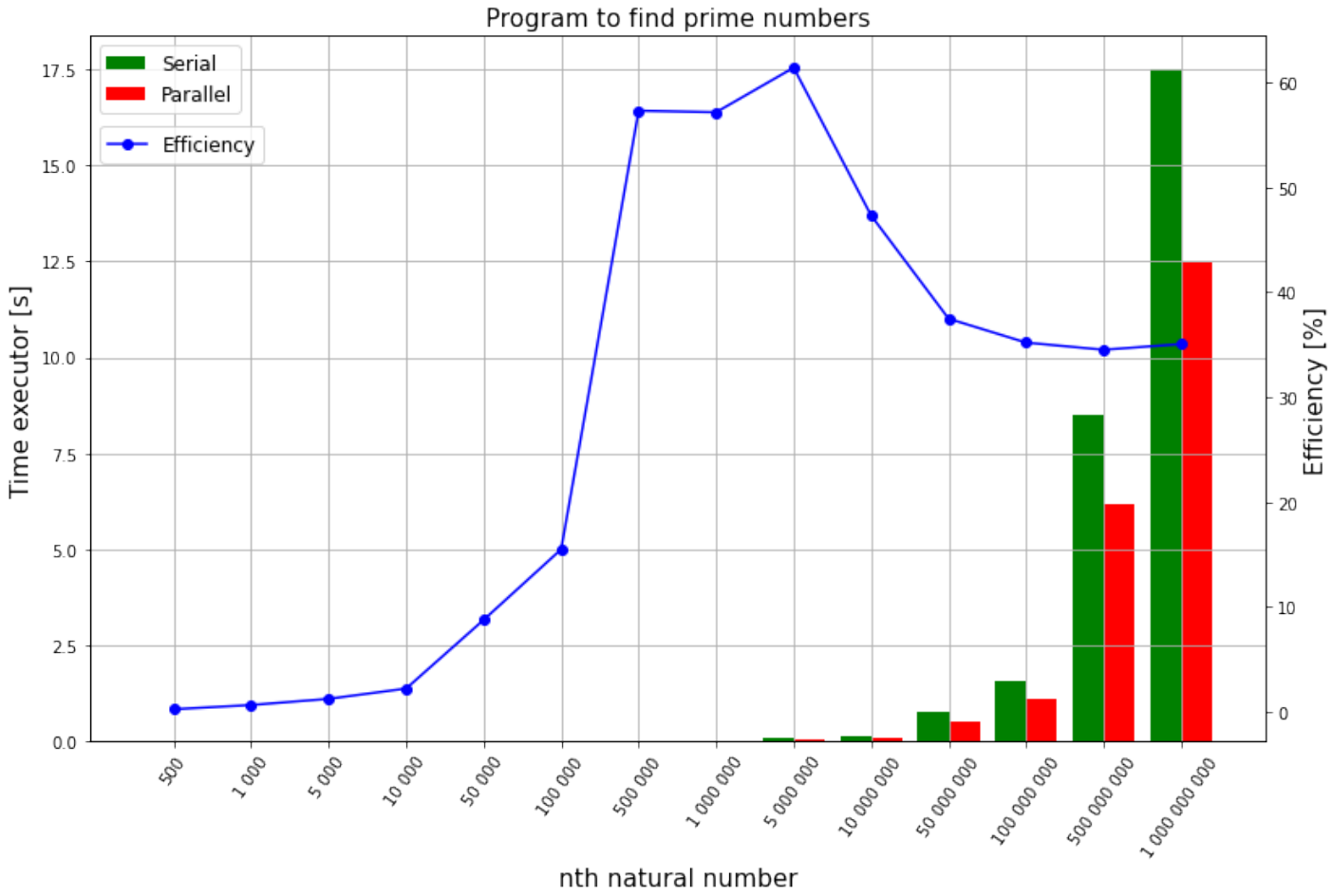


Hình 2: Biểu đồ kết quả chạy chương trình tìm số Fibonacci

1.3 Tìm các số nguyên tố

Bảng 3: Kết quả chạy chương trình tìm các số nguyên tố

STT	Số tự nhiên n	Thời gian thực hiện tuần tự (s)	Thời gian thực hiện song song (s)	Hiệu suất (%)
Test No1	500	8.20514e-06	0.000836104	0.2453385
Test No2	1 000	9.84617e-06	0.000381539	0.64516
Test No3	5 000	3.81539e-05	0.000770462	1.23802
Test No4	10 000	6.52308e-05	0.00073559	2.216955
Test No5	50 000	0.000320821	0.000915283	8.7629
Test No6	100 000	0.000667898	0.00107816	15.48705
Test No7	500 000	0.00495016	0.00216	57.2935
Test No8	1 000 000	0.00820186	0.0035877	57.1525
Test No9	5 000 000	0.0731915	0.0298019	61.3985
Test No10	10 000 000	0.138147	0.0730077	47.30565
Test No11	50 000 000	0.757597	0.505453	37.47115
Test No12	100 000 000	1.57996	1.12236	35.193
Test No13	500 000 000	8.5053	6.16161	34.50925
Test No14	1 000 000 000	17.5059	12.4851	35.0536



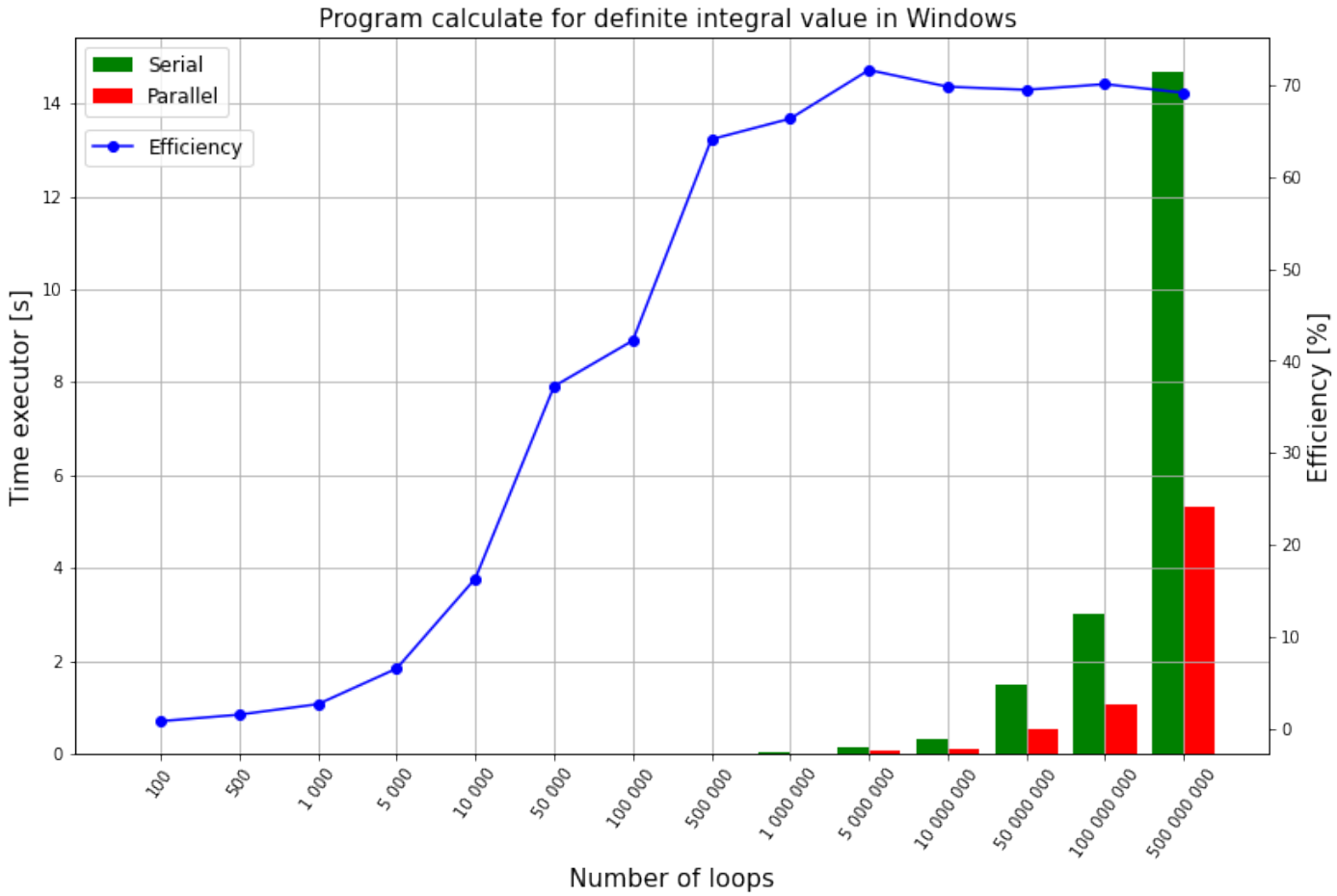
Hình 3: Biểu đồ kết quả chạy chương trình tìm số nguyên tố

2 Bài tập cá nhân - Tính tích phân xác định

2.1 Kết quả chạy chương trình trên Windows

Bảng 4: Kết quả chạy chương trình tính tích phân trên Windows

STT	Số vòng lặp	Thời gian thực hiện tuần tự (s)	Thời gian thực hiện song song (s)	Hiệu suất (%)
Test No1	100	1.35385e-05	0.000412924	0.81967
Test No2	500	2.78975e-05	0.000455795	1.530155
Test No3	1 000	5.2359e-05	0.00049046	2.66885
Test No4	5 000	0.000173539	0.000663385	6.5399
Test No5	10 000	0.000323282	0.00049477	16.335
Test No6	50 000	0.00152369	0.00102318	37.22935
Test No7	100 000	0.00309621	0.00183426	42.19975
Test No8	500 000	0.0148817	0.00580514	64.0885
Test No9	1 000 000	0.0343734	0.0129559	66.3275
Test No10	5 000 000	0.15481	0.0540399	71.6185
Test No11	10 000 000	0.300373	0.107582	69.801
Test No12	50 000 000	1.4773	0.531662	69.466
Test No13	100 000 000	3.00166	1.07074	70.0835
Test No14	500 000 000	14.6859	5.3103	69.139

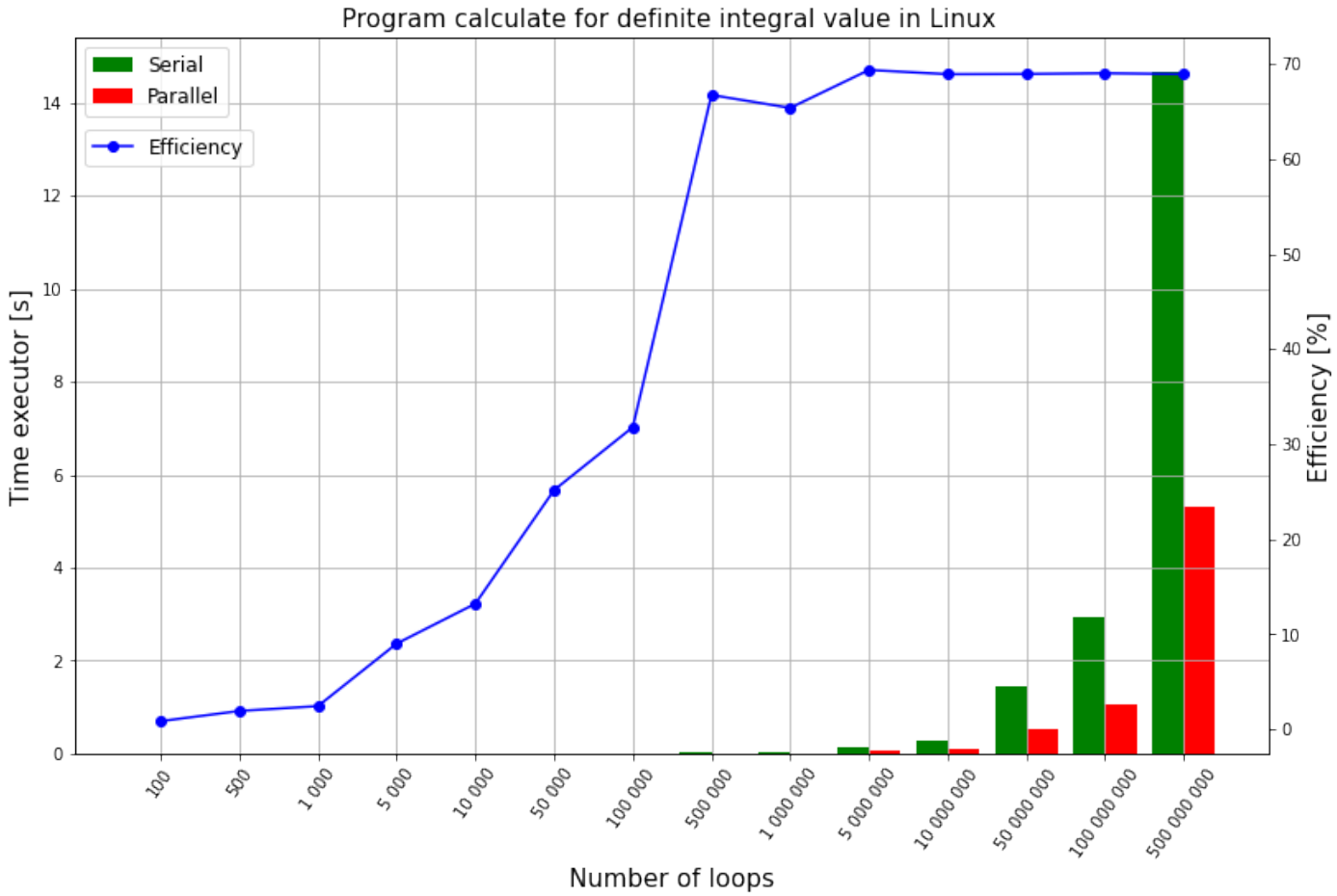


Hình 4: Biểu đồ kết quả chạy chương trình tính tích phân xác định trên Windows

2.2 Kết quả chạy chương trình trên Linux

Bảng 5: Kết quả chạy chương trình tính tích phân trên Linux

STT	Số vòng lặp	Thời gian thực hiện tuần tự (s)	Thời gian thực hiện song song (s)	Hiệu suất (%)
Test No1	100	1.39847e-05	0.00039877	0.874485
Test No2	500	2.8718e-05	0.000368821	1.946605
Test No3	1 000	4.75898e-05	0.000484513	2.455545
Test No4	5 000	0.000177641	0.000491898	9.02835
Test No5	10 000	0.000339639	0.000642052	13.22685
Test No6	50 000	0.00164472	0.00163364	25.1695
Test No7	100 000	0.00321641	0.00252964	31.7872
Test No8	500 000	0.0161391	0.0060476	66.717
Test No9	1 000 000	0.0304632	0.0116521	65.36
Test No10	5 000 000	0.148619	0.0535775	69.348
Test No11	10 000 000	0.295433	0.107188	68.905
Test No12	50 000 000	1.46818	0.532511	68.927
Test No13	100 000 000	2.93727	1.06419	69.003
Test No14	500 000 000	14.6728	5.32313	68.911



Hình 5: Biểu đồ kết quả chạy chương trình tính tích phân xác định trên Linux

II

Mã nguồn

1 Tìm số Fibonacci

```
#include <windows.h>
#include <omp.h>
#include <bits/stdc++.h>
using namespace std;

//struct big integer
typedef vector<int> bigint;
const int BASE = 10000;
void fix(bigint &a)
{
    a.push_back(0);
    for (int i = 0; i < a.size() - 1; ++i)
    {
        a[i + 1] += a[i] / BASE;
        a[i] %= BASE;
        if (a[i] < 0)
        {
            a[i] += BASE;
            a[i + 1]--;
        }
    }
    while (a.size() >= 2 && a.back() == 0)
        a.pop_back();
}
bigint operator*(const bigint &a, const bigint &b) {
    bigint c(a.size() + b.size() + 1);
    for (int i = 0; i < a.size(); ++i)
        for (int j = 0; j < b.size(); ++j)
        {
            c[i + j] += a[i] * b[j];
            c[i + j + 1] += c[i + j] / BASE;
            c[i + j] %= BASE;
        }
    fix(c);
    return c;
}
bigint operator*(bigint a, int x)
{
    for (int i = 0; i < a.size(); ++i)
        a[i] *= x;
    fix(a);
    return  a;
}
bigint operator+(bigint a, const bigint &b) {
    a.resize(max(a.size(), b.size()));
    for (int i = 0; i < b.size(); ++i)
        a[i] += b[i];
    fix(a);
    return a;
```

```
}
bigint operator-(bigint a, const bigint &b)
{
    for (int i = 0; i < b.size(); ++i)
        a[i] -= b[i];
    fix(a);
    return a;
}
ostream &operator<<(ostream &cout, const bigint &a)
{
    printf("%d", a.back());
    for (int i = (int)a.size() - 2; i >= 0; i--)
        printf("%04d", a[i]);
    return cout;
}


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

bigint SerialFibonacci(long long int n);
void SerialMultip(bigint x[], bigint y[]);

void ParallelMultip(bigint x[], bigint y[]);
bigint ParallelFibonacci(long long int n);
void TestResult(long long int n, bigint fibonacci);

int main()
{
    double Start, Finish, Duration_Serial, Duration_Parallel, Efficiency;
    long long int n;
    int p;
    bigint Fibonacci_Serial, Fibonacci_Parallel;
    bigint result;

    cout << "Parallel program to find Fibonacci number \n";
    cout << "Enter the natural number n: ";
    cin >> n;

    Start = GetTime();
    Fibonacci_Serial = SerialFibonacci(n);
    Finish = GetTime();
    Duration_Serial = Finish - Start;

    Start = GetTime();
    Fibonacci_Parallel = ParallelFibonacci(n);
    Finish = GetTime();
    Duration_Parallel = Finish - Start;

    #pragma omp parallel
        p = omp_get_num_threads();
    Efficiency = Duration_Serial / (Duration_Parallel * p);
    TestResult(n, Fibonacci_Parallel);

    cout << "\n\nnth Fibonacci number is: " << Fibonacci_Serial;
    cout << "\n\nTime of execution Serial: " << Duration_Serial;
    cout << "\n\nTime of execution Parallel: " << Duration_Parallel;
    cout << "\n\nEfficiency of execution: " << Efficiency * 100 << "%";
```

```
108     return 0;
109 }
110 
111 void SerialMultip(bigint x[], bigint y[])
112 {
113     bigint *c = new bigint[4];
114     c[0] = x[0] * y[0] + x[1] * y[2];
115     c[1] = x[0] * y[1] + x[1] * y[3];
116     c[2] = x[2] * y[0] + x[3] * y[2];
117     c[3] = x[2] * y[1] + x[3] * y[3];
118 
119     for (int i = 0; i < 4; i++)
120         y[i] = c[i];
121     delete c;
122 }
123 
124 bigint SerialFibonacci(long long int n)
125 {
126     if (n <= 2)
127         return bigint(1, 1);
128     else
129     {
130         n = n - 1;
131         bigint x[4] = {bigint(1, 1), bigint(1, 1), bigint(1, 1), bigint(1, 1) * 0};
132         bigint y[4] = {bigint(1, 1), bigint(1, 1) * 0, bigint(1, 1) * 0, bigint(1, 1)};
133         while (n >= 2)
134         {
135             if (n % 2 == 0)
136             {
137                 SerialMultip(x, x);
138                 n = n / 2;
139             }
140             else
141             {
142                 SerialMultip(x, y);
143                 SerialMultip(x, x);
144                 n = (n - 1)/2;
145             }
146         }
147         SerialMultip(x, y);
148         return y[0];
149     }
150 }
151 
152 void ParallelMultip(bigint x[], bigint y[])
153 {
154     bigint *c = new bigint[4];
155     #pragma omp parallel sections
156     {
157         #pragma omp section
158             c[0] = x[0] * y[0] + x[1] * y[2];
159         #pragma omp section
160             c[1] = x[0] * y[1] + x[1] * y[3];
161         #pragma omp section
162         {
163             c[2] = x[2] * y[0] + x[3] * y[2];
164             c[3] = x[2] * y[1] + x[3] * y[3];
165         }
```

```
166     }
167     #pragma omp parallel for
168         for (int i = 0; i < 4; i++)
169             y[i] = c[i];
170     delete c;
171 }
172
173 bigint ParallelFibonacci(long long int n)
174 {
175     if (n <= 2)
176         return bigint(1, 1);
177     else
178     {
179         n = n - 1;
180         bigint x[4] = {bigint(1, 1), bigint(1, 1), bigint(1, 1), bigint(1, 1) * 0};
181         bigint y[4] = {bigint(1, 1), bigint(1, 1) * 0, bigint(1, 1) * 0, bigint(1, 1)};
182         while (n >= 2)
183         {
184             if (n % 2 == 0)
185             {
186                 ParallelMultip(x, x);
187                 n = n / 2;
188             }
189             else
190             {
191                 ParallelMultip(x, y);
192                 ParallelMultip(x, x);
193                 n = (n - 1)/2;
194             }
195         }
196         ParallelMultip(x, y);
197         return y[0];
198     }
199 }
200
201 void TestResult(long long int n, bigint fibonacci)
202 {
203     bigint fibonacciSerial;
204     fibonacciSerial = SerialFibonacci(n);
205     if (fibonacciSerial != fibonacci)
206         cout << "\nThe results of serial and parallel algorithms "
207                 "are NOT identical. Check your code.";
208     else
209         cout << "\nThe results of serial and parallel algorithms are "
210                 "identical.";
211 }
212
213 double LiToDouble(LARGE_INTEGER x)
214 {
215     double result = ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
216     return result;
217 }
218 double GetTime()
219 {
220     LARGE_INTEGER lpFrequency, lpPerfomanceCount;
221     QueryPerformanceFrequency(&lpFrequency);
222     QueryPerformanceCounter(&lpPerfomanceCount);
223     return LiToDouble(lpPerfomanceCount) / LiToDouble(lpFrequency);
```

```
224 }
```

2 Tìm các số nguyên tố

```
 1 #include <iostream>
 2 #include <math.h>
 3 #include <stdio.h>
 4 #include <windows.h>
 5 #include <omp.h>
 6 #include <time.h>
 7 #include <fstream>
 8 using namespace std;
 9 double LiToDouble(LARGE_INTEGER x);
10 double GetTime();
11 void ParallelFindPrimes(long long int n, bool *a);
12 void SerialFindPrimes(long long int n, bool *a);
13 void TestResult(long long int n, bool *a);
14 void PrintPrimeInFile(long long int n, bool *a, string FileName);
15
16 main()
17 {
18     long long int n;
19     int p;
20     double Start, Finish, Duration_Serial, Duration_Parallel, Efficiency;
21     bool *Result_Serial;
22     bool *Result_Parallel;
23     cout << "Parallel program to find primes less than or equal to n\n";
24     cout << "Enter the natural number n: ";
25     cin >> n;
26     Result_Serial = new bool[n + 1];
27     Result_Parallel = new bool[n + 1];
28
29     Start = GetTime();
30     SerialFindPrimes(n, Result_Serial);
31     Finish = GetTime();
32     Duration_Serial = Finish - Start;
33
34     Start = GetTime();
35     ParallelFindPrimes(n, Result_Parallel);
36     Finish = GetTime();
37     Duration_Parallel = Finish - Start;
38
39     #pragma omp parallel
40         p = omp_get_num_threads();
41     Efficiency = Duration_Serial / (Duration_Parallel * p);
42
43     TestResult(n, Result_Parallel);
44     PrintPrimeInFile(n, Result_Serial, "output.txt");
45
46     cout << "\n\nTime of execution serial: " << Duration_Serial;
47     cout << "\n\nTime of execution parallel: " << Duration_Parallel;
48     cout << "\n\nEfficiency of execution: " << Efficiency * 100 <<"%";
49     delete[] Result_Serial;
50     delete[] Result_Parallel;
51     return 0;
```

```
52 }
53
54 void ParallelFindPrimes(long long int n, bool *a)
55 {
56     long long int m;
57     #pragma omp parallel for
58         for (long long int i = 2; i <= n; i++)
59             a[i] = true;
60
61     m = sqrt(n);
62     #pragma omp parallel for schedule (dynamic, 1) shared (a)
63         for (long long int i = 2; i <= m; i++)
64         {
65             if (a[i] == true)
66                 for (long long int j = i * i; j <= n; j = j + i)
67                     a[j] = false;
68         }
69 }
70
71 void SerialFindPrimes(long long int n, bool *a)
72 {
73     double m;
74     for (long long int i = 2; i <= n; i++)
75         a[i] = true;
76
77     m = sqrt(n);
78     for (long long int i = 2; i <= m; i++)
79         if (a[i] == true)
80             for (int j = i * i; j <= n; j = j + i)
81                 a[j] = false;
82 }
83
84 void TestResult(long long int n, bool *a)
85 {
86     bool *b;
87     b = new bool[n + 1];
88     bool check = true;
89     SerialFindPrimes(n, b);
90     for (long long int i = 2; i <= n; i++)
91     {
92         if (a[i] != b[i])
93         {
94             check = false;
95             break;
96         }
97     }
98     if (check == false)
99         cout << "\nThe results of serial and parallel algorithms "
100                "are NOT identical. Check your code.";
101    else
102        cout << "\nThe results of serial and parallel algorithms are "
103                "identical.";
104    delete[] b;
105 }
106
107 void PrintPrimeInFile(long long int n, bool *a, string FileName)
108 {
109     long long int count = 1, i;
```

```
    fstream out;
    out.open(FileName, ios::out);
    out << "\nPrime numbers less than or equal " << n << ": \n";
    #pragma omp parallel for ordered
        for (i = 2; i <= n; i++)
        {
            #pragma omp ordered
                if (a[i] == true)
                {
                    if (count % 10 == 0)
                    {
                        out << "\n";
                    }
                    count ++;
                    out << i << "\t";
                }
        }
    out.close();
}

double LiToDouble(LARGE_INTEGER x)
{
    double result = ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
    return result;
}

double GetTime()
{
    LARGE_INTEGER lpFrequency, lpPerfomanceCount;
    QueryPerformanceFrequency(&lpFrequency);
    QueryPerformanceCounter(&lpPerfomanceCount);
    return LiToDouble(lpPerfomanceCount) / LiToDouble(lpFrequency);
}
```

3 Tính tích phân xác định

```
#include <iostream>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <omp.h>
#include <time.h>
using namespace std;
double LiToDouble(LARGE_INTEGER x);
double GetTime();

double Function(double x);
double SerialIntegralValue(double a, double b, long long int N);
double ParallelIntegralValue(double a, double b, long long int N);
void TestResult(double a, double b, long long int N, double Result_Parallel);

int main()
{
    double a, b, Start, Finish, Duration_Serial, Duration_Parallel;
```

```
20      double Result_Serial, Result_Parallel, Efficiency;
21      long long int N;
22      int p;
23      cout << "\nParallel program calculate for definite integral value\n";
24      cout << "\nEnter the range (a, b):";
25      cout << "\na = ";
26      cin >> a;
27      cout << "\nb = ";
28      cin >> b;
29      cout << "\nEnter the number of loop: ";
30      cin >> N;
31
32      Start = GetTime();
33      Result_Serial = SerialIntegralValue(a, b, N);
34      Finish = GetTime();
35      Duration_Serial = Finish - Start;
36
37      Start = GetTime();
38      Result_Parallel = ParallelIntegralValue(a, b, N);
39      Finish = GetTime();
40      Duration_Parallel = Finish - Start;
41
42      #pragma omp parallel
43          p = omp_get_num_threads();
44      Efficiency = Duration_Serial / (Duration_Parallel * p);
45      TestResult(a, b, N, Result_Parallel);
46
47      cout << "\n\nIntegral value of execution serial is: " << Result_Serial;
48      cout << "\n\nIntegral value of execution parallel is: " << Result_Parallel;
49      cout << "\n\nTime of execution Serial: " << Duration_Serial;
50      cout << "\n\nTime of execution Parallel: " << Duration_Parallel;
51      cout << "\n\nEfficiency of execution: " << Efficiency * 100 << "%";
52      return 0;
53  }
54
55  double Function(double x)
56  {
57      return sin(2 * x + x * x) + x;
58  }
59
60  double SerialIntegralValue(double a, double b, long long int N)
61  {
62      double h = (b - a)/N;
63      double sum = 0;
64      for (int i = 0; i < N; i++)
65      {
66          sum += Function(i * h + a);
67      }
68      return sum * h;
69  }
70
71  double ParallelIntegralValue(double a, double b, long long int N)
72  {
73      double h = (b - a)/N;
74      double sum = 0;
75      #pragma omp parallel for reduction(+:sum)
76          for (int i = 0; i < N; i++)
77          {
```

```
78              sum += Function(i * h + a);
79          }
80      return sum * h;
81  }
82
83  void TestResult(double a, double b, long long int N, double Result_Parallel)
84  {
85      if (Result_Parallel - SerialIntegralValue(a, b, N) >= 1e-10)
86          cout << "\nThe results of serial and parallel algorithms "
87                  "are NOT identical. Check your code.";
88      else
89          cout << "\nThe results of serial and parallel algorithms are "
90                  "identical.";
91  }
92
93  double LiToDouble(LARGE_INTEGER x)
94  {
95      double result = ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
96      return result;
97  }
98  double GetTime()
99  {
100     LARGE_INTEGER lpFrequency, lpPerfomanceCount;
101     QueryPerformanceFrequency(&lpFrequency);
102     QueryPerformanceCounter(&lpPerfomanceCount);
103     return LiToDouble(lpPerfomanceCount) / LiToDouble(lpFrequency);
104 }
```

III

Nhận xét

Từ bảng số liệu và biểu đồ kết quả chạy các chương trình trên em có một số nhận xét về thời gian thực thi và hiệu suất của tính toán song song so với tính toán tuần tự cũng như là kết quả chạy chương trình trên Windows và Linux.

1. Tất cả các bảng số liệu trên đều cho thấy với số lặp ít thì tính toán tuần tự có thời gian chạy ít hơn tính toán song song, đồng nghĩa với đó là hiệu suất thấp. Và ngược lại khi số vòng lặp là rất lớn.

2. Hầu hết hiệu suất tính toán được tăng lên khi số vòng lặp tăng lên, chỉ riêng với chương trình tìm số nguyên tố thì thấy hiệu suất đang đạt cao nhất ở mức độ giữa (số vòng lặp khoảng 500 000 $\rightarrow$ 5 000 000) và hiệu suất thấp hơn trong các trường hợp khác.

3. Hiệu suất của chương trình tìm số Fibonacci là không được cao lắm.

4. Thời gian thực thi và hiệu suất khi chạy chương trình trên Windows và Linux là tương đương nhau, mức độ chênh lệch là rất nhỏ.

Tài liệu tham khảo

[1] Tim Mattson, *A Hands-on Introduction to OpenMP*

[2] **https://openmp.org/**

[3] **https://www.geeksforgeeks.org/**

[4] **https://stackoverflow.com/**

[5] Đoàn Duy Trung, *Slide bài giảng Tính toán song song*