

BÀI THỰC HÀNH NHÂN MA TRẬN VỚI VEC TƠ TRÊN OPENMP

Nội dung bài thực hành

Thực hiện song song hóa bài toán ma trận nhân véc tơ khi sử dụng mô hình song song đa bộ vi xử lý với bộ nhớ dùng chung để tính toán. Bao gồm 4 phần:

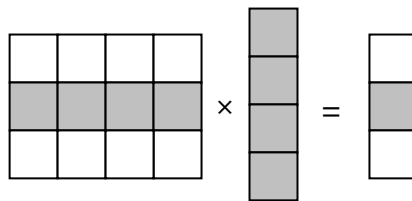
- Phần 1: Giới thiệu bài toán ma trận nhân véc tơ
- Phần 2: Thực hiện bài toán tuần tự ma trận nhân véc tơ
- Phần 3: Phát triển thuật toán song song ma trận nhân véc tơ
- Phần 4: Thực hiện bài toán song song ma trận nhân véc tơ

Phần 1: Giới thiệu bài toán ma trận nhân véc tơ

Bài toán nhân ma trận A cỡ $m \times n$ và véc tơ b cỡ $n \times 1$ kết quả nhận được là 1 véc tơ cỡ $m \times 1$, tức là

$$A_{m \times n} \cdot b_{n \times 1} = c_{m \times 1}$$

$$c_i = (a_i, b) = \sum_{j=1}^n a_{ij} b_j, 1 \leq i \leq m. \quad (1.1)$$



Hình 1 – Kết quả nhân ma trận với véc tơ

Ví dụ nhân ma trận $A_{3 \times 4}$ và véc tơ $b_{4 \times 1}$ ta được véc tơ $c_{3 \times 1}$ như sau

$$\begin{pmatrix} 3 & 2 & 0 & -1 \\ 5 & -2 & 1 & 1 \\ 1 & 0 & -1 & -1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 8 \\ -6 \end{pmatrix}$$

Hình 2 – Nhân ma trận và véc tơ

Nhận được kết quả của véc tơ c bằng cách lặp đi lặp lại m lần quá trình tính toán của biểu thức 1.1 được mô tả bên trên. Mỗi quá trình tính toán 1.1 sẽ nhận được 1 phần tử của véc tơ c.

Code mô tả thuật toán ma trận nhân véc tơ có thể được thực hiện như sau:

```
// Serial algorithm of matrix-vector multiplication
for (i = 0; i < m; i++){
    c[i] = 0;
    for (j = 0; j < n; j++){
        c[i] += A[i][j]*b[j]
    }
}
```

Phần 2 – Thực hiện bài toán tuần tự ma trận nhân véc tơ

Bước 1 – Đặt biến

Thực hiện bài toán tuần tự ma trận nhân véc tơ, đặt tên project là *SerialMatrixVectorMult*. Trong hàm main của project đặt các biến là pMatrix và pVector – đại diện cho ma trận và véc tơ khi thực hiện tính toán bài toán nhân ma trận và véc tơ.

Biến thứ ba là biến `pResult` – là véc tơ kết quả được nhận được từ bài toán nhân ma trận và véc tơ. Biến `Size` định nghĩa kích thước ma trận (Giả thiết bài toán ở trường hợp ma trận là ma trận vuông kích cỡ $pMatrix_{Size \times Size}$ và nhân với véc tơ bao gồm `Size` phần tử)

Cách đặt biến trong hàm `main`

```
double* pMatrix; // Initial matrix
double* pVector; // Initial vector
double* pResult; // Result vector for matrix-vector multiplication
int Size; // Sizes of initial matrix and vector
```

Nhận thấy rằng khi lưu ma trận `pMatrix` sử dụng mảng một chiều, ở đó ma trận được lưu giữ theo dòng. Do đó phần tử được phân bố tại hàng thứ i cột thứ j của ma trận, trong mảng một chiều có chỉ số là $i * Size + j$

Bước 2 – nhập vào kích thước cho các đối tượng

Đặt hàm `ProcessInitialization` là hàm số nhập dữ liệu cho thuật toán tuần tự nhân ma trận với véc tơ. Hàm này cho phép thiết lập kích thước ma trận và véc tơ, phân chia bộ nhớ cho các đối tượng tham gia thực hiện nhân `pMatrix` và `pVector`. Hàm số được mô tả như sau

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int &Size);
```

Thiết lập kích thước của đối tượng (đưa vào giá trị của biến `Size`). Trong hàm số `ProcessInitialization` thêm vào đoạn code sau:

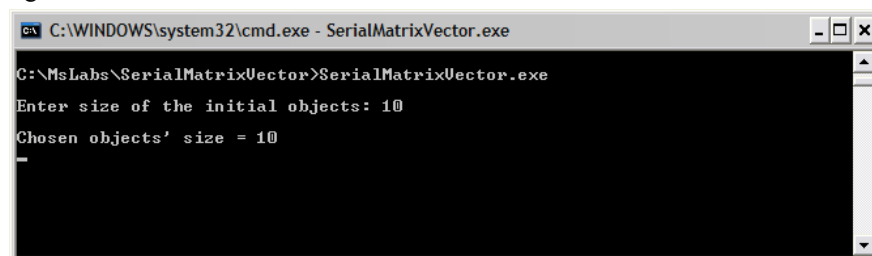
```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int &Size) {
    // Setting the size of initial matrix and vector
    printf("\nEnter size of the initial objects: ");
    scanf("%d", &Size);
    printf("\nChosen objects size = %d", Size);
}
```

Sau khi thực hiện xong nhập giá trị trong hàm `ProcessInitialization`, trong hàm `main` gọi vào hàm như sau:

```
void main() {
    double* pMatrix; // Initial matrix
    double* pVector; // Initial vector
    double* pResult; // Result vector for matrix-vector multiplication
    int Size; // Sizes of initial matrix and vector
    time_t start, finish;
    double duration;

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

Project sau khi debug sẽ hiển thị như sau:



Hình 3 – Đưa vào kích thước các đối tượng

Thêm đoạn code kiểm tra việc nhập số n vào có đúng hay không?

```
// Setting the size of initial matrix and vector
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);
```

Bước 3 – Nhập vào dữ liệu

Hàm ProcessInitialization cần phải phân chia bộ nhớ để tiến hành lưu các đối tượng, do đó thêm vào hàm ProcessInitialization đoạn code như sau:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int Size) {
    // Setting the size of initial matrix and vector
    do {
        <...>
    }
    while (Size <= 0);

    // Memory allocation
    pMatrix = new double [Size*Size];
    pVector = new double [Size];
    pResult = new double [Size];
}
```

Tiếp theo cần phải đưa vào giá trị toàn bộ phần tử của ma trận pMatrix và véc tơ pVector. Để thực hiện việc này tiến hành thêm hàm DummyDataInitialization. Việc mô tả hàm được thực hiện như sau:

```
// Function for simple initialization of matrix and vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
    int i, j; // Loop variables

    for (i=0; i<Size; i++) {
        pVector[i] = 1;
        for (j=0; j<Size; j++)
            pMatrix[i*Size+j] = i;
    }
}
```

Hàm này cho phép nhập vào giá trị các phần tử của pMatrix và pVector như sau: đối với pVector tất cả các phần tử đều bằng 1, còn giá trị các phần tử của pMatrix trùng với giá trị của chỉ số hàng đang xét đến

$$pMatrix = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{pmatrix}, pVector = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Gọi hàm DummyDataInitialization cần thực hiện sau khi phân chia bộ nhớ bên trong hàm ProcessInitialization

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int Size) {
    // Setting the size of initial matrix and vector
    do {
        <...>
    }
    while (Size <= 0);

    // Memory allocation
    <...>
```

```
// Initialization of matrix and vector elements
DummyDataInitialization(pMatrix, pVector, Size);
}
```

Cần thêm 2 hàm nữa để kiểm tra dữ liệu nhập vào, đó là 2 hàm đưa ra dữ liệu của ma trận và véc tơ: PrintMatrix và PrintVector. Các tham số trong hàm số PrintMatrix bao gồm RowCount cho số hàng và ColCount cho số cột

```
// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
    int i, j; // Loop variables
    for (i=0; i<RowCount; i++) {
        for (j=0; j<ColCount; j++)
            printf("%7.4f ", pMatrix[i*ColCount+j]);
        printf("\n");
    }
}

// Function for formatted vector output
void PrintVector (double* pVector, int Size) {
    int i;
    for (i=0; i<Size; i++)
        printf("%7.4f ", pVector[i]);
    printf("\n");
}
```

Thêm việc gọi hàm xuất kết quả ra màn hình trong hàm chính

```
// Memory allocation and data initialization
ProcessInitialization(pMatrix, pVector, pResult, Size);

// Matrix and vector output
printf ("Initial Matrix: \n");
PrintMatrix (pMatrix, Size, Size);
printf ("Initial Vector: \n");
PrintVector (pVector, Size);
```

Thử nghiệm việc nhập dữ liệu và hiển thị kết quả trên màn hình

```
C:\WINDOWS\system32\cmd.exe - SerialMatrixVector.exe
C:\MsLabs\SerialMatrixVector>SerialMatrixVector.exe
Enter size of the initial objects: 4
Chosen objects' size = 4
Initial Matrix
0.0000 0.0000 0.0000 0.0000
1.0000 1.0000 1.0000 1.0000
2.0000 2.0000 2.0000 2.0000
3.0000 3.0000 3.0000 3.0000
Initial Vector
1.0000 1.0000 1.0000 1.0000 _
```

Hình 4 – Kết quả sau khi hoàn thành nội dung thứ 3

Bước 4 – Hoàn thành quá trình tính toán

Trước khi thực hiện nhân ma trận và véc tơ cần phải thực hiện hàm hoàn thành quá trình tính toán, đó là việc giải phóng bộ nhớ đã được phân chia động ở phần trước. Tạo ra hàm ProcessTermination, để giải phóng bộ nhớ được dành cho pMatrix và pVector, và pResult.

```
// Function for computational process termination
void ProcessTermination (double* pMatrix, double* pVector, double* pResult) {
    delete [] pMatrix;
    delete [] pVector;
    delete [] pResult;
}
```

Gọi hàm `ProcessTermination` cần phải thực hiện trước khi hoàn thành phần chương trình thực hiện nhân ma trận và véc tơ.

```
// Memory allocation and data initialization
ProcessInitialization(pMatrix, pVector, pResult, Size);

// Matrix and vector output
printf ("Initial Matrix: \n");
PrintMatrix (pMatrix, Size, Size);
printf ("Initial Vector: \n");
PrintVector (pVector, Size);

// Computational process termination
ProcessTermination(pMatrix, pVector, pResult);
```

Bước 5 – Thực hiện nhân ma trận và véc tơ

Nội dung này tiến hành thực hiện phần tính toán quan trọng nhất của chương trình. Để thực hiện quá trình tính toán nhân ma trận và véc tơ ta tiến hành xây dựng hàm `SerialResultCalculation`.

Sử dụng thuật toán được mô tả tại phần 1, đoạn code được thực hiện như sau

```
// Function for matrix-vector multiplication
void SerialResultCalculation(double* pMatrix, double* pVector, double* pResult,
    int Size) {
    int i, j; // Loop variables
    for (i=0; i<Size; i++) {
        pResult[i] = 0;
        for (j=0; j<Size; j++)
            pResult[i] += pMatrix[i*Size+j]*pVector[j];
    }
}
```

Thực hiện gọi hàm tính toán nhân ma trận và véc tơ từ hàm chính của chương trình. Để kiểm tra tính đúng đắn của chương trình được viết ta tiến hành như sau:

```
// Memory allocation and data initialization
ProcessInitialization(pMatrix, pVector, pResult, Size);

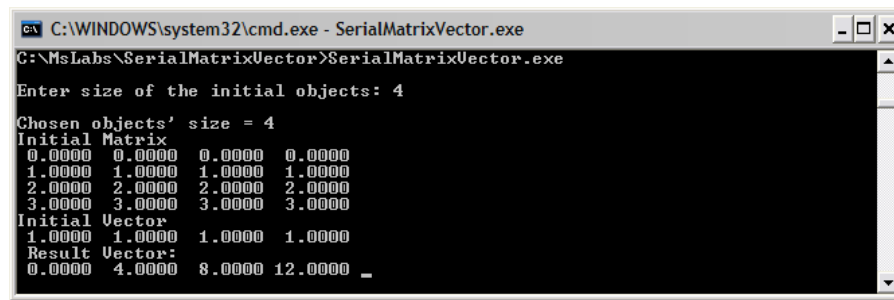
// Matrix and vector output
printf ("Initial Matrix: \n");
PrintMatrix (pMatrix, Size, Size);
printf ("Initial Vector: \n");
PrintVector (pVector, Size);

// Matrix-vector multiplication
SerialResultCalculation(pMatrix, pVector, pResult, Size);

// Printing the result vector
printf ("\n Result Vector: \n");
PrintVector(pResult, Size);

// Computational process termination
ProcessTermination(pMatrix, pVector, pResult);
```

Debug ứng dụng. Xem xét lại kết quả tính toán theo thuật toán nhân ma trận và véc tơ. Nếu thuật toán đúng kết quả khi `Size = 4` của véc tơ `pResult` là `pResult=(0,4,8,12)`. Thực hiện lại vài lần để kiểm định kết quả thu được



```
C:\WINDOWS\system32\cmd.exe - SerialMatrixVector.exe
C:\MsLabs\SerialMatrixVector>SerialMatrixVector.exe
Enter size of the initial objects: 4
Chosen objects' size = 4
Initial Matrix
0.0000 0.0000 0.0000 0.0000
1.0000 1.0000 1.0000 1.0000
2.0000 2.0000 2.0000 2.0000
3.0000 3.0000 3.0000 3.0000
Initial Vector
1.0000 1.0000 1.0000 1.0000
Result Vector:
0.0000 4.0000 8.0000 12.0000 _
```

Hình 5 – Kết quả hiển thị lên màn hình

Bước 6 – Thí nghiệm quá trình tính toán

Để kiểm tra hiệu quả công việc tính toán song song cần phải đưa vào các thí nghiệm khác nhau để hiển thị thời gian của việc tính toán. Phân tích thời gian thực hiện của thuật toán cần phải dẫn vào các đối tượng với dữ liệu lớn. Dữ liệu của lớn của ma trận và véc tơ được đưa vào qua các phân tử ngẫu nhiên. Thực hiện thêm vào hàm `RandomDataInitialization`.

```
// Function for random initialization of objects' elements
void RandomDataInitialization (double* pMatrix, double* pVector, int Size) {
    int i, j; // Loop variables
    srand(unsigned(clock()));
    for (i=0; i<Size; i++) {
        pVector[i] = rand()/double(1000);
        for (j=0; j<Size; j++)
            pMatrix[i*Size+j] = rand()/double(1000);
    }
}
```

Gọi hàm này thay thế cho hàm `DummyDataInitialization` mà thực hiện trước đó

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int Size) {
    // Size of initial matrix and vector definition
    <...>

    // Memory allocation
    <...>

    // Random data initialization
    RandomDataInitialization(pMatrix, pVector, Size);
}
```

Build và Debug ứng dụng. Dễ dàng nhận thấy dữ liệu được đưa vào một cách ngẫu nhiên

Để xác định thời gian tính toán có thể sử dụng hàm gọi thời gian chuẩn của ngôn ngữ C

```
time_t clock(void);
```

Tuy nhiên sử dụng hàm này với sự chính xác khá thấp, bởi vì nó không chính xác khi đo thời gian của quá trình tính toán ít. Do đó tiến hành xây dựng hàm `GetTime()`, sử dụng để đo chính thời gian tính toán, sử dụng hàm số của thư viện WinAPI:

```
// Function that converts numbers form LongInt type to double type
double LiToDouble (LARGE_INTEGER x) {
    double result =
        ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
    return result;
}

// Function that gets the timestamp in seconds
double GetTime() {
    LARGE_INTEGER lpFrequency, lpPerfomanceCount;
    QueryPerformanceFrequency (&lpFrequency);
```

```

QueryPerformanceCounter (&lpPerfomanceCount);
return LiToDouble(lpPerfomanceCount)/LiToDouble(lpFrequency);
}

```

Hàm GetTime() trả về giá trị thời gian hiện tại, mà được đọc từ thời điểm trước đó theo đơn vị giây. Tiến hành gọi hàm này 2 lần – trước và sau mỗi lần thực hiện để đo thời gian của công việc. Ví dụ việc tính thời gian của hàm f() như sau:

```

double t1, t2;
t1 = GetTime();
f();
t2 = GetTime();
double duration = (t2-t1);

```

Thêm vào trong chương trình đoạn code tính toán và hiển thị thời gian thực hiện nhân ma trận và véc tơ, để thực hiện nó tiến hành đo thời gian trước và sau khi gọi hàm Serial ResultCalculation

```

// Matrix-vector multiplication
Start = clock();
SerialResultCalculation(pMatrix, pVector, pResult, Size);
Finish = clock();
Duration = (Finish-Start);

// Printing the result vector
printf ("\n Result Vector: \n");
PrintVector(pResult, Size);
// Printing the time spent by matrix-vector multiplication
printf("\n Time of serial execution: %f", Duration);

```

Build và debug ứng dụng. Đưa vào các thí nghiệm với kích thước ma trận khác nhau

STT	Kích thước ma trận
Test № 1	10
Test № 2	100
Test № 3	1000
Test № 4	2000
Test № 5	3000
Test № 6	4000
Test № 7	5000
Test № 8	6000
Test № 9	7000
Test № 10	8000
Test № 11	9000
Test № 12	10 000

Thời gian tính toán trên lý thuyết được mô tả bởi công thức sau:

$$T_1 = n(2n-1) \cdot \tau + 8n^2 / \beta, \quad (1.2)$$

Ở đó n là kích thước ma trận, τ thời gian thực hiện của 1 toán tử, β truy cập đến bộ nhớ

Phần 3 – Thực hiện thuật toán song song

Xác định sự phân chia các thành phần nhỏ

Các phương pháp tính toán ma trận được đặc trưng bởi sự lặp đi lặp quá trình tính toán các phần tử khác nhau của ma trận. Trong trường hợp này bài toán được đặc trưng bởi sự song song dữ liệu khi thực hiện tính toán ma trận, sự phân bố song song các hoạt động của ma trận được phân bổ vào các processor trong hệ thống tính toán. Lựa chọn phương pháp phân

chia ma trận được thực hiện bởi các thuật toán song song tính toán ma trận. Có rất nhiều phương pháp phân chia ma trận để tính toán, tuy nhiên 2 phương pháp phổ biến và sử dụng rộng rãi hơn cả đó là phương pháp phân chia dữ liệu theo dải băng (theo chiều ngang hoặc chiều dọc) và phân đoạn theo hình vuông (khối)

1. Block-striped. Khi thực hiện theo phương pháp này mỗi processor sẽ được phân chia theo nhóm các hàng (theo hàng ngang) hoặc nhóm các cột (theo cột dọc) của ma trận. Phân chia hàng và cột theo dải băng trong hầu hết các trường hợp được thực hiện một cách liên tục (tuần tự). Dưới đây mô tả cách phân chia theo chiều ngang bởi các hàng của ma trận, ví dụ ma trận A được mô tả như sau:

$$A = (A_0, A_1, \dots, A_{p-1})^T, A_i = (a_{i_0}, a_{i_1}, \dots, a_{i_{k-1}})^T, i_j = ik + j, 0 \leq j < k, k = m / p,$$

Ở đó $a_i = (a_{i1}, a_{i2}, \dots, a_{in})$, $0 \leq i < m$, được gọi là hàng thứ i của ma trận A (Giả thiết rằng số lượng hàng của ma trận là bội số của số processor do đó $m = k.p$).

2. Checkerboard block . Khi thực hiện phân chia ma trận theo khối, trong nhiều trường hợp sử dụng sự phân chia một cách liên tục. Giả sử số lượng processor là p được tính theo công thức $p = s.q$, số lượng hàng của ma trận là s chia hết cho s, còn số lượng cột là số chia hết cho q, do đó $m = k.s$ và $n = l.q$.

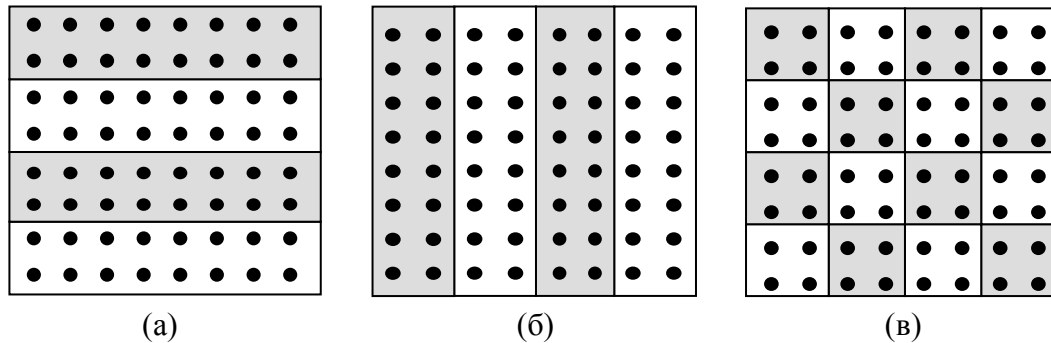
Cho ma trận A dưới dạng tập hợp các khối hình chữ nhật như sau

$$A = \begin{pmatrix} A_{00} & A_{02} & \dots & A_{0q-1} \\ & & \dots & \\ A_{s-11} & A_{s-12} & \dots & A_{s-1q-1} \end{pmatrix},$$

Ở đó A_{ij} - ma trận khối, bao gồm các phần tử như sau

$$A_{ij} = \begin{pmatrix} a_{i_0j_0} & a_{i_0j_1} & \dots & a_{i_0j_{l-1}} \\ & & \dots & \\ a_{i_{k-1}j_0} & a_{i_{k-1}j_1} & \dots & a_{i_{k-1}j_{l-1}} \end{pmatrix}, i_v = ik + v, 0 \leq v < k, k = m / s, j_u = jl + u, 0 \leq u < l, l = n / q.$$

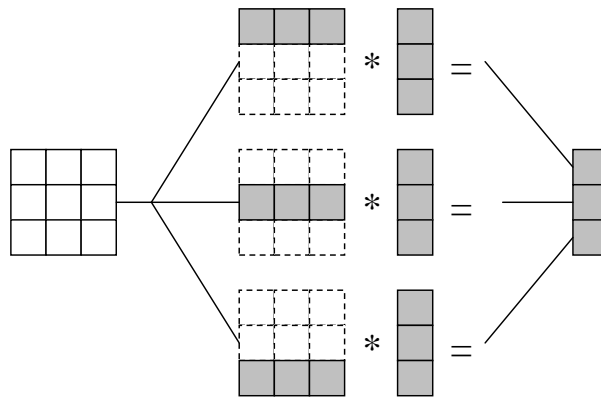
Cách phân chia được mô tả theo hình dưới đây



Các phương pháp phân chia phần tử của ma trận giữa các processor của hệ thống tính toán

Phân chia thông tin phụ thuộc

Sự phân bố dữ liệu của ma trận được mô tả theo sơ đồ dưới đây



Tổ chức các phép tính toán khi thực hiện song song thuật toán nhân ma trận với véc tơ, phân chia ma trận theo hàng

Phần 4 – Thực hiện song song thuật toán

Phần này thực hiện song song thuật toán ma trận nhân véc tơ trên OpenMP với mô hình bộ nhớ chia sẻ

Bước 1 – Thực hiện song song thuật toán

Thực hiện chương trình song song nhân ma trận và véc tơ với kiểu phân chia dữ liệu theo mô hình dải băng theo từng hàng. Chương trình được thực hiện trên mô hình bộ nhớ chia sẻ sử dụng kỹ thuật OpenMP, trong trường hợp thực hiện mô hình song song này thì chỉ cần thêm parallel for vào hàm SerialResultCalculation ở bài toán trước, hàm mới với tên gọi là ParallelResultCalculation

```
// Function for calculating matrix-vector multiplication
void ParallelResultCalculation (double* pMatrix, double* pVector,
    double* pResult, int Size) {
    int i, j; // Loop variables
    #pragma omp parallel for private (j)
    for (i=0; i<Size; i++) {
        for (j=0; j<Size; j++)
            pResult[i] += pMatrix[i*Size+j]*pVector[j];
    }
}
```

Hàm này cho phép nhân dãy các hàng của ma trận với véc tơ sử dụng một vài luồng song song. Mỗi luồng sẽ thực hiện nhân từng nhóm các hàng của ma trận pMatrix với véc tơ pVector và đưa các khối phần tử kết quả vào véc tơ pResult.

Song song theo vùng ở hàm này được thực hiện bởi parallel for, sử dụng tham số private cho biến chạy j. Nhận thấy rằng ma trận pMatrix và véc tơ pVector và pResult là các biến chia sẻ cho tất cả các luồng. Nhưng phần tử của ma trận pMatrix và véc tơ pVector chỉ sử dụng để đọc.

Bước 2 – Kiểm tra tính đúng đắn của thuật toán song song

Để kiểm tra tính đúng đắn của chương trình song song ta tiến hành xây dựng hàm số TestResult, để so sánh kết quả giữa thuật toán song song và tuần tự. Sử dụng véc tơ pSerialResult để lưu kết quả của phép tính tuần tự

```
// Testing the result of parallel matrix-vector multiplication
void TestResult(double* pMatrix, double* pVector, double* pResult,
    int Size) {
    // Buffer for storing the result of serial matrix-vector multiplication
    double* pSerialResult;
    int equal = 0; // Flag, that shows wheather the vectors are identical
    int i; // Loop variable

    pSerialResult = new double [Size];
    SerialResultCalculation(pMatrix, pVector, pSerialResult, Size);
    for (i=0; i<Size; i++) {
        if (pResult[i] != pSerialResult[i])
            equal = 1;
    }
}
```

```

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

```

Bước 3 – Tiến hành thí nghiệm kết quả tính toán

Vấn đề quan trọng nhất khi thực hiện thuật toán song song đối với các bài toán phức tạp là việc đảm bảo hiệu suất tốt hơn việc tính toán tuần tự. Về nguyên tắc thời gian thực hiện thuật toán song song phải được giảm thiểu so với thời gian thực hiện thuật toán tuần tự. Để xác định thời gian thực hiện song song tiến hành thêm vào trong code chương trình hàm GetTime(). Thêm vào trong hàm main

```

ProcessInitialization(pMatrix, pVector, pResult, pProcRows, pProcResult,
                    Size, RowNum);

double Start, Finish, Duration;

Start = GetTime();
ParallelResultCalculation(pMatrix, pVector, pResult, Size);
Finish = GetTime();
Duration = Finish-Start;
TestResult(pMatrix, pVector, pResult, Size);
printf("Time of parallel execution = %f\n", Duration);

```

Code tuần tự

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <windows.h>

// Function that converts numbers form LongInt type to double type
double LiToDouble (LARGE_INTEGER x) {
    double result =
        ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
    return result;
}

// Function that gets the timestamp in seconds
double GetTime() {
    LARGE_INTEGER lpFrequency, lpPerfomanceCount;
    QueryPerformanceFrequency (&lpFrequency);
    QueryPerformanceCounter (&lpPerfomanceCount);
    return LiToDouble(lpPerfomanceCount)/LiToDouble(lpFrequency);
}

// Function for simple definition of matrix and vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
    int i, j; // Loop variables

    for (i=0; i<Size; i++) {
        pVector[i] = 1;
        for (j=0; j<Size; j++)
            pMatrix[i*Size+j] = i;
    }
}

```

```

}

// Function for random definition of matrix and vector elements
void RandomDataInitialization(double* pMatrix, double* pVector, int Size) {
    int i, j; // Loop variables
    srand(unsigned(clock()));
    for (i=0; i<Size; i++) {
        pVector[i] = rand()/double(1000);
        for (j=0; j<Size; j++)
            pMatrix[i*Size+j] = rand()/double(1000);
    }
}

// Function for memory allocation and definition of object's elements
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int &Size) {
    // Size of initial matrix and vector definition
    do {
        printf("\nEnter size of the initial objects: ");
        scanf("%d", &Size);
        printf("\nChosen objects size = %d\n", Size);
        if (Size <= 0)
            printf("\nSize of objects must be greater than 0!\n");
    }
    while (Size <= 0);
    // Memory allocation
    pMatrix = new double [Size*Size];
    pVector = new double [Size];
    pResult = new double [Size];
    // Definition of matrix and vector elements
    DummyDataInitialization(pMatrix, pVector, Size);
}

// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
    int i, j; // Loop variables
    for (i=0; i<RowCount; i++) {
        for (j=0; j<ColCount; j++)
            printf("%7.4f ", pMatrix[i*RowCount+j]);
        printf("\n");
    }
}

// Function for formatted vector output
void PrintVector (double* pVector, int Size) {
    int i;
    for (i=0; i<Size; i++)
        printf("%7.4f ", pVector[i]);
}

// Function for matrix-vector multiplication
void ResultCalculation(double* pMatrix, double* pVector, double* pResult,
    int Size) {
    int i, j; // Loop variables
    for (i=0; i<Size; i++) {
        pResult[i] = 0;
        for (j=0; j<Size; j++)
            pResult[i] += pMatrix[i*Size+j]*pVector[j];
    }
}

```

```

}

// Function for computational process termination
void ProcessTermination(double* pMatrix, double* pVector, double* pResult) {
    delete [] pMatrix;
    delete [] pVector;
    delete [] pResult;
}

void main() {
    double* pMatrix; // The first argument - initial matrix
    double* pVector; // The second argument - initial vector
    double* pResult; // Result vector for matrix-vector multiplication
    int Size; // Sizes of initial matrix and vector
    Double Start, Finish, Duration;

    printf("Serial matrix-vector multiplication program\n");
    // Memory allocation and definition of objects' elements
    ProcessInitialization(pMatrix, pVector, pResult, Size);

    // Matrix and vector output
    printf ("Initial Matrix \n");
    PrintMatrix(pMatrix, Size, Size);
    printf("Initial Vector \n");
    PrintVector(pVector, Size);

    // Matrix-vector multiplication
    Start = GetTime();
    ResultCalculation(pMatrix, pVector, pResult, Size);
    Finish = GetTime();
    Duration = Finish-Start;

    // Printing the result vector
    printf ("\n Result Vector: \n");
    PrintVector(pResult, Size);

    // Printing the time spent by matrix-vector multiplication
    printf("\n Time of execution: %f\n", Duration);

    // Computational process termination
    ProcessTermination(pMatrix, pVector, pResult);
}

```

Code song song

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <windows.h>
#include <openmp.h>

// Function that converts numbers form LongInt type to double type
double LiToDouble (LARGE_INTEGER x) {
    double result =
        ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
    return result;
}

// Function that gets the timestamp in seconds
double GetTime() {

```

```

    LARGE_INTEGER lpFrequency, lpPerformanceCount;
    QueryPerformanceFrequency (&lpFrequency);
    QueryPerformanceCounter (&lpPerformanceCount);
    return LiToDouble(lpPerformanceCount)/LiToDouble(lpFrequency);
}

// Function for simple definition of matrix and vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
    int i, j; // Loop variables

    for (i=0; i<Size; i++) {
        pVector[i] = 1;
        for (j=0; j<Size; j++)
            pMatrix[i*Size+j] = i;
    }
}

// Function for random definition of matrix and vector elements
void RandomDataInitialization(double* pMatrix, double* pVector, int Size) {
    int i, j; // Loop variables
    srand(unsigned(clock()));
    for (i=0; i<Size; i++) {
        pVector[i] = rand()/double(1000);
        for (j=0; j<Size; j++)
            pMatrix[i*Size+j] = rand()/double(1000);
    }
}

// Function for memory allocation and definition of object's elements
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int &Size) {
    // Size of initial matrix and vector definition
    do {
        printf("\nEnter size of the initial objects: ");
        scanf("%d", &Size);
        printf("\nChosen objects size = %d\n", Size);
        if (Size <= 0)
            printf("\nSize of objects must be greater than 0!\n");
    }
    while (Size <= 0);
    // Memory allocation
    pMatrix = new double [Size*Size];
    pVector = new double [Size];
    pResult = new double [Size];
    // Definition of matrix and vector elements
    DummyDataInitialization(pMatrix, pVector, Size);
}

// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
    int i, j; // Loop variables
    for (i=0; i<RowCount; i++) {
        for (j=0; j<ColCount; j++)
            printf("%7.4f ", pMatrix[i*RowCount+j]);
        printf("\n");
    }
}

// Function for formatted vector output

```

```

void PrintVector (double* pVector, int Size) {
    int i;
    for (i=0; i<Size; i++)
        printf("%7.4f ", pVector[i]);
}

// Function for serial matrix-vector multiplication
void SerialResultCalculation(double* pMatrix, double* pVector,
    double* pResult, int Size) {
    int i, j; // Loop variables
    for (i=0; i<Size; i++) {
        pResult[i] = 0;
        for (j=0; j<Size; j++)
            pResult[i] += pMatrix[i*Size+j]*pVector[j];
    }
}

// Function for parallel matrix-vector multiplication
void ParallelResultCalculation (double* pMatrix, double* pVector,
    double* pResult, int Size) {
    int i, j; // Loop variables
    #pragma omp parallel for private (j)
    for (i=0; i<Size; i++) {
        for (j=0; j<Size; j++)
            pResult[i] += pMatrix[i*Size+j]*pVector[j];
    }
}

// Testing the result of parallel matrix-vector multiplication
void TestResult(double* pMatrix, double* pVector, double* pResult,
    int Size) {
    // Buffer for storing the result of serial matrix-vector multiplication
    double* pSerialResult;
    int equal = 0; // Flag, that shows wheather the vectors are identical
    int i; // Loop variable

    pSerialResult = new double [Size];
    SerialResultCalculation(pMatrix, pVector, pSerialResult, Size);
    for (i=0; i<Size; i++) {
        if (pResult[i] != pSerialResult[i])
            equal = 1;
    }
    if (equal == 1)
        printf("The results of serial and parallel algorithms "
            "are NOT identical. Check your code.");
    else
        printf("The results of serial and parallel algorithms are "
            "identical.");
    delete [] pSerialResult;
}

// Function for computational process termination
void ProcessTermination(double* pMatrix, double* pVector, double* pResult) {
    delete [] pMatrix;
    delete [] pVector;
    delete [] pResult;
}

void main() {

```

```

double* pMatrix; // The first argument - initial matrix
double* pVector; // The second argument - initial vector
double* pResult; // Result vector for matrix-vector multiplication
int Size; // Sizes of initial matrix and vector
double Start, Finish, Duration;

printf("Parallel matrix-vector multiplication program\n");
// Memory allocation and definition of objects' elements
ProcessInitialization(pMatrix, pVector, pResult, Size);

// Matrix and vector output
printf ("Initial Matrix \n");
PrintMatrix(pMatrix, Size, Size);
printf("Initial Vector \n");
PrintVector(pVector, Size);

// Matrix-vector multiplication
Start = GetTime();
ParallelResultCalculation(pMatrix, pVector, pResult, Size);
Finish = GetTime();
Duration = Finish-Start;
TestResult(pMatrix, pVector, pResult, Size);

// Printing the result vector
printf ("\n Result Vector: \n");
PrintVector(pResult, Size);

// Printing the time spent by matrix-vector multiplication
printf("\n Time of execution: %f\n", Duration);

// Computational process termination
ProcessTermination(pMatrix, pVector, pResult);
}

```

BÀI TẬP:

1) Thực hiện chương trình tuần tự và song song, đưa ra kết quả theo bảng dưới đây:

STT	Kích thước ma trận	Thời gian thực hiện tuần tự	Thời gian thực hiện song song	Hiệu suất
Test № 1	10			
Test № 2	100			
Test № 3	1000			
Test № 4	2000			
Test № 5	3000			
Test № 6	4000			
Test № 7	5000			
Test № 8	6000			
Test № 9	7000			
Test № 10	8000			
Test № 11	9000			
Test № 12	10 000			

2) Viết chương trình OpenMP tính toán số Fibonacci

3) Viết chương trình tìm số nguyên tố từ 1 đến n (n nhập vào từ bàn phím) bằng OpenMP