

CSE703032
Lập trình song song

Bài giảng 7
**LẬP TRÌNH SONG SONG
VỚI BỘ NHỚ CHIA SẺ**

Khoa Công nghệ thông tin
Đại học Phenikaa

OpenMP: chia sẻ công việc (work sharing)

Từng phần (sections):

- Sections: các thread chia một số phần cố định cho nhau.
- Vòng lặp song song là một ví dụ về các đơn vị công việc độc lập được đánh số. Nếu ta có số lượng đơn vị công việc độc lập được xác định trước thì các *section* sẽ phù hợp hơn.

```
#pragma omp sections
{
    #pragma omp section
        // one calculation
    #pragma omp section
        // another calculation
}
```

OpenMP: chia sẻ công việc (work sharing)

Từng phần (sections):

- Sections có thể được sử dụng để phân chia các khối công việc độc lập lớn như ở đoạn code sau đây:

```
float f(float x) { return 1; }
float g(float x) { return 2; }
float h(float x) { return 3; }
int main(int argc,char **argv) {
    // section 1:
    float fx,gx,hx;
    #pragma omp parallel sections
    {
        #pragma omp section
        fx = f(1.);
        #pragma omp section
        gx = g(1.);
        #pragma omp section
        hx = h(1.);
    }
    float s = fx+gx+hx;
```

OpenMP: chia sẻ công việc (work sharing)

Từng phần (sections):

- Tuy nhiên, giải pháp tốt nhất nên dùng **reduction clause** trong chỉ thị các phần song song. Sau đó ta có thể viết lại code như sau:

```
s=0;  
#pragma omp parallel sections reduction(+:s)  
{  
    #pragma omp section  
    s += f(1.);  
    #pragma omp section  
    s += g(1.);  
    #pragma omp section  
    s += h(1.);  
}  
printf("sum: %5.2f\n",s);
```

OpenMP: chia sẻ công việc (work sharing)

Thực thi luồng đơn:

- OpenMP có hai cơ chế cho phép một phần code chỉ được thực thi bởi một luồng duy nhất: chỉ thị **single** và **master/masked**.
- Chỉ thị **single** sẽ được sử dụng cho các phần là một phần của luồng điều khiển vì nó có **barrier** kết thúc ngầm.
- Chỉ thị **master/masked** là tương tự nhau, nhưng chỉ định việc thực thi cho luồng chính và không có **barrier** khi kết thúc.

OpenMP: chia sẻ công việc (work sharing)

Thực thi luồng đơn:

- *Single pragma* giới hạn việc thực thi một khối trong một luồng đơn. Ví dụ, điều này có thể được sử dụng để in thông tin theo dõi hoặc thực hiện các thao tác vào/ra (I/O).

```
#pragma omp parallel
{
    #pragma omp single
        printf("We are starting this section!\n");
        // parallel stuff
}
```

- Cũng có thể dùng *single* để khởi tạo các biến riêng tư. Trong trường hợp đó ta thêm mệnh đề *copyprivate*. Đây là một giải pháp tốt nếu việc đặt biến cần I/O.

OpenMP: chia sẻ công việc (work sharing)

Thực thi luồng đơn:

- Các chỉ thị **master/masked** cũng thực thi trên một luồng duy nhất, cụ thể là luồng chính của nhóm.
- Đây không phải là cấu trúc chia sẻ công việc và do đó không có sự đồng bộ hóa thông qua **barier** ngầm.

```
1 #include <stdio.h>
2 #include <omp.h>
3 int main()
4 {
5     #pragma omp parallel
6     {
7         int num=omp_get_thread_num();
8         #pragma omp master
9         {
10            printf("Thread %d is executing master construct\n", num);
11        }
12        printf("Thread %d is executing non-master construct\n", num);
13    }
14    return 0;
15 }
```

OpenMP: Kiểm soát dữ liệu luồng (Controlling thread data)

- Trong một vùng song song có hai loại dữ liệu: **riêng tư** và **chia sẻ**.
- Ở phần này, ta sẽ thấy nhiều cách khác nhau để có thể kiểm soát danh mục dữ liệu của mình.
- Đối với các mục dữ liệu **riêng tư**, sẽ thảo luận về việc giá trị của dữ liệu riêng tư này liên quan đến dữ liệu chia sẻ như thế nào.

OpenMP: Kiểm soát dữ liệu luồng (Controlling thread data)

Dữ liệu chia sẻ:

- Trong một vùng song song, mọi dữ liệu được khai báo bên ngoài nó sẽ được chia sẻ: bất kỳ luồng nào sử dụng biến **x** sẽ truy cập vào cùng một vị trí bộ nhớ được liên kết với biến đó.

```
1 #include <stdio.h>
2 #include <omp.h>
3 int main()
4 {
5     int x=1;
6     #pragma omp parallel
7     {
8         int num=omp_get_thread_num();
9         x+=1;
10        printf("shared: x= %d at thread:%d\n",x,num);
11    }
12    printf("final: x= %d\n",x);
13    return 0;
14 }
```

OpenMP: Kiểm soát dữ liệu luồng (Controlling thread data)

Dữ liệu riêng tư:

- Các vùng và lệnh song song OpenMP: bất kỳ biến nào được khai báo trong một khối (block) theo lệnh OpenMP sẽ là biến cục bộ của luồng thực thi.

```
1 #include <stdio.h>
2 #include <omp.h>
3 int main()
4 {
5     int x=0;
6     #pragma omp parallel
7     {
8         int x=0;
9         int num=omp_get_thread_num();
10        x+=num;
11        printf("private: x= %d at thread:%d\n",x,num);
12    }
13    printf("final: x= %d\n",x);
14    return 0;
15 }
```

```
private: x= 3 at thread:3
private: x= 2 at thread:2
private: x= 0 at thread:0
private: x= 1 at thread:1
private: x= 4 at thread:4
private: x= 5 at thread:5
private: x= 6 at thread:6
private: x= 7 at thread:7
final: x= 0
```

```
Process exited after 0.03938
Press any key to continue .
```

OpenMP: Kiểm soát dữ liệu luồng (Controlling thread data)

Mặc định (default):

- Có các quy tắc mặc định cho việc dữ liệu trong cấu trúc OpenMP là riêng tư hay chia sẻ và bạn có thể kiểm soát điều này một cách rõ ràng.
- Đầu tiên là hành vi mặc định:
 - ✓ Các biến được khai báo bên ngoài vùng song song được chia sẻ như mô tả ở trên;
 - ✓ Các biến vòng lặp trong omp for là biến riêng tư;
 - ✓ Các biến cục bộ trong vùng song song là biến riêng tư.

OpenMP: Kiểm soát dữ liệu luồng (Controlling thread data)

Mặc định (default):

- Chúng ta có thể thay đổi hành vi mặc định này bằng mệnh đề mặc định:

```
#pragma omp parallel default(shared) private(x)
{ ... }
#pragma omp parallel default(private) shared(matrix)
{ ... }
```

- và nếu chúng ta muốn thực hiện nó an toàn:

```
#pragma omp parallel default(none) private(x) shared(matrix)
{ ... }
```

- Mệnh đề chia sẻ có nghĩa là tất cả các biến từ phạm vi bên ngoài được chia sẻ trong vùng song song; bất kỳ biến riêng tư nào cũng cần phải được khai báo rõ ràng. Đây là hành vi mặc định.

- Mệnh đề riêng có nghĩa là tất cả các biến bên ngoài trở thành riêng tư trong vùng song song.

OpenMP: Kiểm soát dữ liệu luồng (Controlling thread data)

Riêng tư đầu tiên và cuối cùng (First and last private):

- Các biến riêng tư được tạo với giá trị không xác định. Ta có thể buộc khởi tạo chúng bằng ***firstprivate***

```
1 #include <stdio.h>
2 #include <omp.h>
3 int main()
4 {
5     int x=0;
6     #pragma omp parallel firstprivate(x)
7     {
8         int num=omp_get_thread_num();
9         x+=num;
10        printf("private: x= %d at thread:%d\n",x,num);
11    }
12    printf("final: x= %d\n",x);
13    return 0;
14 }
```

```
private: x= 2 at thread:2
private: x= 3 at thread:3
private: x= 6 at thread:6
private: x= 5 at thread:5
private: x= 7 at thread:7
private: x= 1 at thread:1
private: x= 0 at thread:0
private: x= 4 at thread:4
final: x= 0
```

```
-----  
Process exited after 0.042  
Press any key to continue
```

OpenMP: Kiểm soát dữ liệu luồng (Controlling thread data)

Riêng tư đầu tiên và cuối cùng (First and last private):

- Ta có thể muốn một giá trị riêng được bảo toàn cho môi trường bên ngoài khu vực song song. Điều này thực sự chỉ có ý nghĩa trong một trường hợp, ta bảo toàn một biến riêng tư từ lần lặp cuối cùng của vòng lặp song song.
- Việc này được thực hiện bằng **Lastprivate**:

```
1 #include <stdio.h>
2 #include <omp.h>
3 int main()
4 {
5     int x=0;
6     #pragma omp parallel for lastprivate(x)
7     for(int i=0;i<=10;i++){
8         x=i;
9         printf("private: x= %d at thread:%d\n",x,omp_get_thread_num());
10    }
11    printf("final: x= %d\n",x);
12 }
```

OpenMP: Kiểm soát dữ liệu luồng (Controlling thread data)

Dữ liệu mảng:

- Các quy tắc cho mảng hơi khác so với các quy tắc cho dữ liệu vô hướng:
 - ✓ Dữ liệu được phân bổ tĩnh có thể được chia sẻ hoặc riêng tư, tùy thuộc vào mệnh đề ta sử dụng.
 - ✓ Dữ liệu được phân bổ động, nghĩa là được tạo bằng **malloc** hoặc **allocate**, chỉ có thể được chia sẻ.

```
int array[nthreads];
for (int i=0; i<nthreads; i++)
    array[i] = 0;
```

```
#pragma omp parallel firstprivate(array)
{
    int t = omp_get_thread_num();
    array[t] = t+1;
}
```

```
int *array =
    (int*) malloc(nthreads*sizeof(int));
for (int i=0; i<nthreads; i++)
    array[i] = 0;
```

```
#pragma omp parallel firstprivate(array)
{
    int t = omp_get_thread_num();
    // ptr arith: needs private array
    array += t;
    array[0] = t;
}
```

OpenMP: Kiểm soát dữ liệu luồng (Controlling thread data)

Dữ liệu mảng:

- Xem ví dụ về mảng tĩnh sau:

```
1 #include <stdio.h>
2 #include <omp.h>
3 int main(){
4     int nthreads;
5     #pragma omp parallel
6     #pragma omp master
7         nthreads = omp_get_num_threads();
8         int array[nthreads];
9         for(int i=0;i<nthreads;i++) array[i] = 0;
10        #pragma omp parallel firstprivate(array)
11        {
12            int t = omp_get_thread_num();
13            array[t] = t+1;
14        }
15        printf("Private\n");
16        printf("Array result:\n");
17        for (int i=0; i<nthreads; i++)printf(" arr[%d]=%4d,",i,array[i]);
18        return 0;
19 }
```

OpenMP: Kiểm soát dữ liệu luồng (Controlling thread data)

Dữ liệu mảng:

- Xem ví dụ về mảng cấp phát bộ nhớ động:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4 int main(){
5     int nthreads;
6     #pragma omp parallel
7     #pragma omp master
8         nthreads = omp_get_num_threads();
9         int *array =(int*) malloc(nthreads * sizeof(int));
10        for(int i=0;i<nthreads;i++) array[i] = 0;
11        #pragma omp parallel firstprivate(array)
12    {
13        int t = omp_get_thread_num();
14        array[t] = t+1;
15    }
16    printf("Private\n");
17    for (int i=0; i<nthreads; i++)printf(" arr[%d]=%4d,",i,array[i]);
18    return 0;
19 }
```

OpenMP: Kiểm soát dữ liệu luồng (Controlling thread data)

Dữ liệu liên tục thông qua threadprivate:

- Hầu hết dữ liệu trong các vùng song song OpenMP đều được kế thừa từ luồng chính và do đó được chia sẻ hoặc tạm thời trong phạm vi của vùng và hoàn toàn riêng tư.
- Ngoài ra còn có một cơ chế cho dữ liệu riêng tư theo luồng, không bị giới hạn trong suốt thời gian tồn tại ở một vùng song song.
- **#pragma threadprivate** được sử dụng để khai báo rằng mỗi luồng phải có một bản sao riêng của một biến:

```
#pragma omp threadprivate(var)
```

OpenMP: Kiểm soát dữ liệu luồng (Controlling thread data)

Dữ liệu liên tục thông qua threadprivate:

- Ví dụ:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <omp.h>
4 static int tp;
5 #pragma omp threadprivate(tp)
6
7 int main(int argc,char **argv) {
8     #pragma omp parallel num_threads(4)
9     tp = omp_get_thread_num();
10    #pragma omp parallel num_threads(8)
11    printf("Thread %d has %d\n",omp_get_thread_num(),tp);
12    return 0;
13 }
```

```
D:\parallel\program\l
Thread 2 has 2
Thread 3 has 3
Thread 1 has 1
Thread 6 has 0
Thread 5 has 0
Thread 4 has 0
Thread 0 has 0
Thread 7 has 0
-
Process exited
Press any key t
```

References

1. Đỗ Thanh Nghị, Nguyễn Văn Hòa, Đỗ Hiệp Thuận (2014), *Giáo trình lập trình song song*. NXB Trường ĐHCT. ISBN: 978-604-919-065-0
2. A. Grama, G. Karypis, V. Kumar and A. Gupta (2003), *Introduction to Parallel Computing*. Addison Wesley. ISBN: 978-0201648652