



Lập trình song song với OpenMP

Nội dung

- Lập trình đa luồng
- OpenMP là gì?
- Chương trình OpenMP đầu tiên
- Các thành phần của OpenMP
- OpenMP được dịch như thế nào?

Lập trình đa luồng (MultiThreaded Programming)

Tiến trình (Process)

- Tiến trình: một thực thể thực thi của chương trình đã bắt đầu nhưng chưa kết thúc
- Tiến trình là đơn vị nhỏ nhất cho cấp phát tài nguyên
- Tiến trình được tạo qua lời gọi hệ thống, vd. `fork()` trong UNIX
- Hệ thống quản lý tiến trình qua khối điều khiển tiến trình (PCB)
- Liên lạc giữa các tiến trình thông qua các giao thức IPC

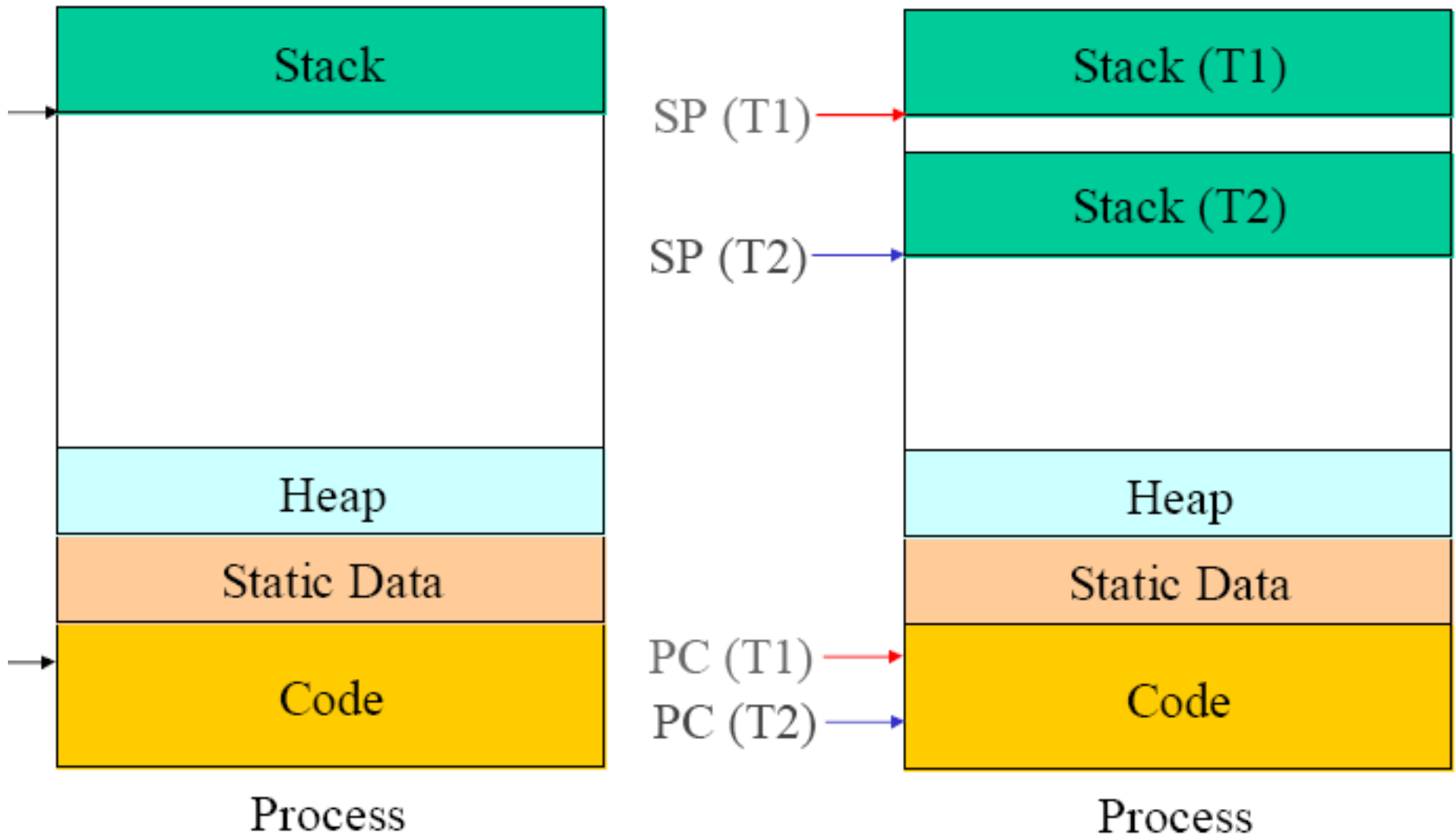
Tiến trình

- Theo quan điểm hệ thống
 - Tiến trình là đơn vị chiếm dụng tài nguyên: CPU, bộ nhớ, thanh ghi, thẻ tập...
 - Các tiến trình là riêng biệt: tiến trình không thể truy cập trực tiếp đến tài nguyên của tiến trình khác
 - Liên lạc giữa các tiến trình rất tốn chi phí
- Tiến trình có thể được nhìn theo 2 góc độ: chiếm dụng tài nguyên và thực thi lệnh → theo góc độ 2 thì tiến trình là tập hợp các luồng

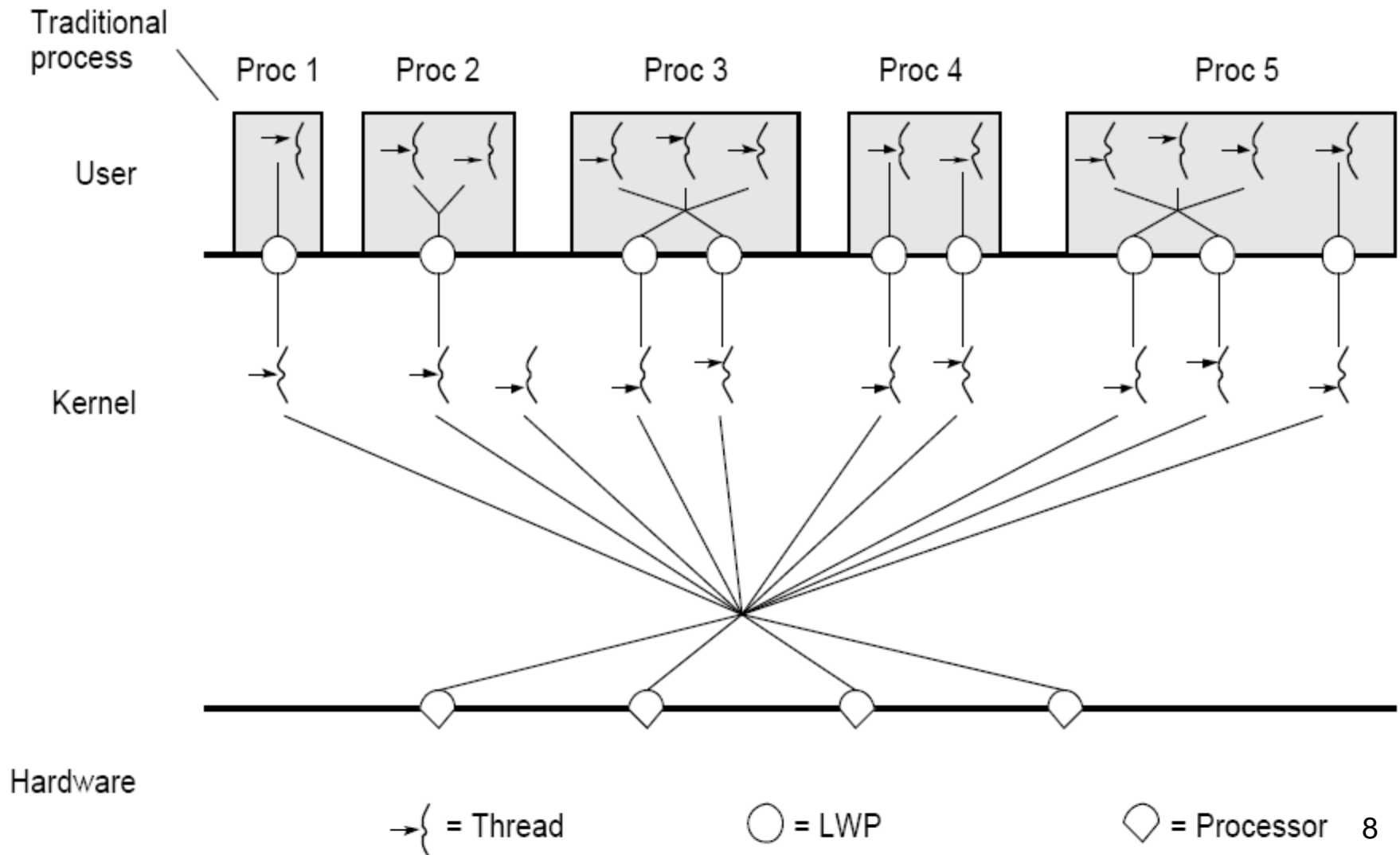
Luồng (Thread)

- Luồng là đơn vị thực thi của tiến trình
- Một tiến trình bao gồm một hoặc nhiều luồng, mỗi luồng thì thuộc về một tiến trình
- Luồng có vùng nhớ ngăn xếp riêng, con trỏ lệnh riêng, các thanh ghi riêng
- Các luồng trong tiến trình chia sẻ các tài nguyên khác của tiến trình, vd. bộ nhớ
- Liên lạc giữa các luồng thông qua vùng nhớ của tiến trình

Tiến trình và luồng



Kiến trúc đa luồng

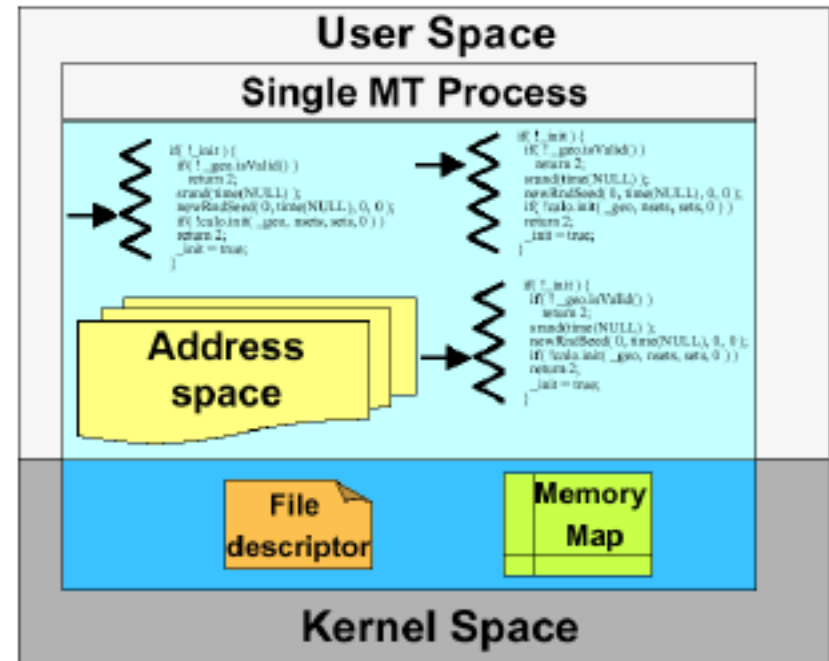
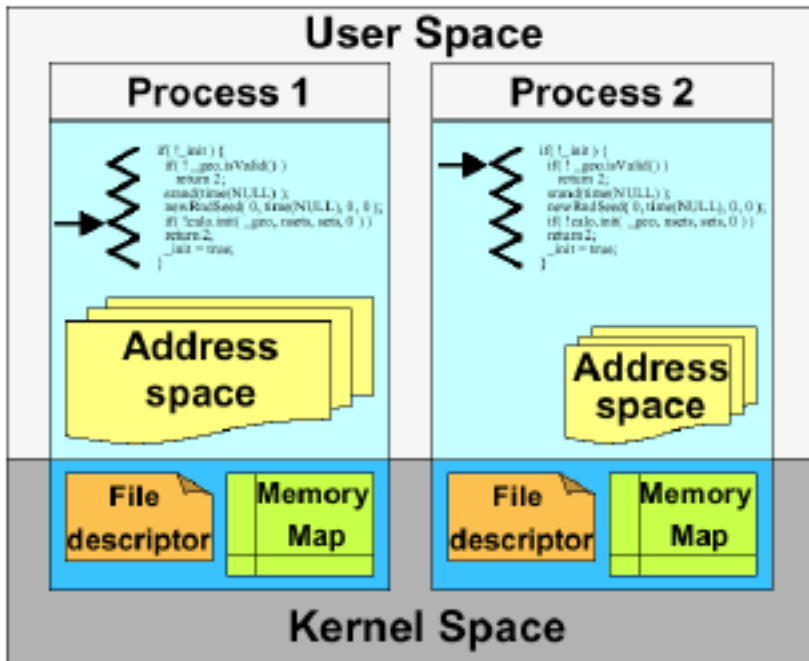


Lập trình đa luồng

- Theo quan điểm lập trình
 - Luồng là dòng điều khiển độc lập, tức là hàm
 - Tham số của hàm là dữ liệu của luồng
 - Mỗi hàm thực hiện một công việc cụ thể → một tiến trình có thể thực hiện nhiều công việc một lúc bằng cách chia nó thành các luồng → lập trình đa luồng
- Phân biệt với lập trình đa tiến trình
 - Chi phí khởi tạo, quản lý, kết thúc công việc
 - Chi phí trao đổi dữ liệu giữa các công việc
 - Hệ thống máy tính đem triển khai

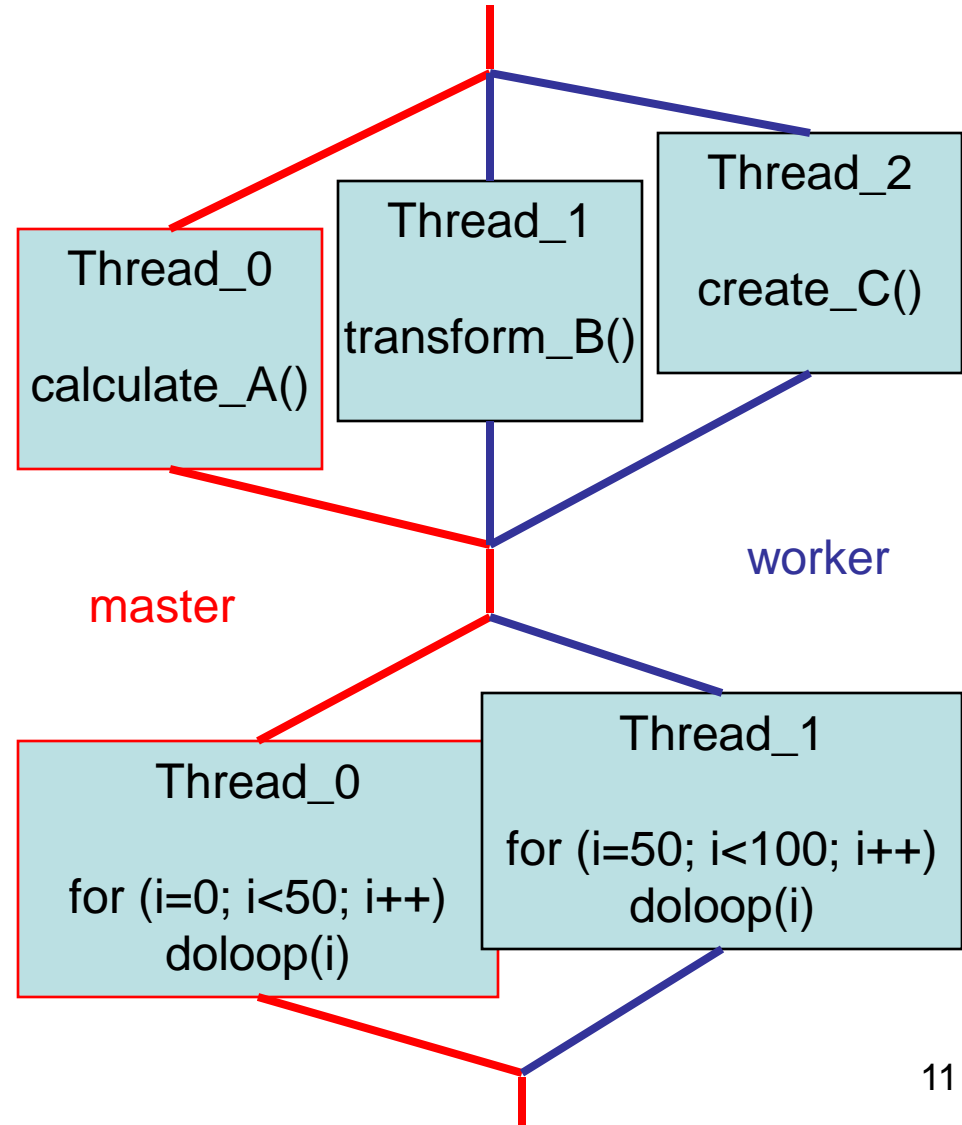
Lập trình đa luồng

- Mô hình lập trình áp dụng tốt cho hệ thống song song SMP (Symmetric Multi-Processing)



Lập trình đa luồng

```
01 main()
02 {
03     ...
04     calculate_A();
05     transform_B();
06     create_C();
07     ...
08     for (i=0; i<100; i++)
09         doloop(i);
10     ...
11 }
12
13
14
15
16
17 }
```



Ưu điểm của lập trình đa luồng

- Khai thác tối đa tính song song của hệ thống đa xử lý đối xứng (SMP)
- Sử dụng tối đa khả năng của bộ xử lý
- Tăng hiệu suất của chương trình ngay cả với máy đơn xử lý
- Tăng khả năng đáp ứng của chương trình
- Đưa ra cơ chế liên lạc giữa các công việc nhanh và hiệu quả hơn

Vấn đề trong lập trình đa luồng

- Đồng bộ hóa các công việc
- An toàn và toàn vẹn với dữ liệu chia sẻ
- Xử lý điều kiện đua tranh
- Dò lỗi chương trình

OpenMP là gì?

OpenMP: *Open specifications for Multi Processing*

- OpenMP là
 - Application Programming Interface (API),
 - đem lại mô hình lập trình linh động cho những nhà phát triển ứng dụng song song chia sẻ bộ nhớ
- OpenMP được hợp thành bởi
 - Chỉ thị chương trình dịch (*compiler directives*)
 - Thư viện hàm thời gian chạy (*library runtime routines*)
 - Các biến môi trường (*environment variables*)
- Có thể dùng được trên hầu hết các máy với kiến trúc một không gian nhớ (*single memory space*)

OpenMP không phải

- Một ngôn ngữ lập trình mới
 - Thực ra OpenMP hoạt động trên sự liên kết chặt chẽ với ngôn ngữ lập trình làm cơ sở, vd. Fortran, C/C++
- Tự động song song hóa chương trình
 - Người lập trình phải tự ý thức về tính song song của công việc
 - OpenMP cung cấp cơ chế để chỉ định việc thực hiện song song
- Phương tiện lập trình cho hệ thống có bộ nhớ phân tán

Ưu điểm của OpenMP

- Một chuẩn hoàn chỉnh và được công nhận trên thực tế
- Hiệu suất và khả năng mở rộng tốt
 - Nếu chương trình được thiết kế đúng!
- Tính khả chuyển cao
 - Chương trình viết ra có thể dịch bởi nhiều chương trình dịch khác nhau
- Dễ sử dụng nhờ sự đơn giản và số lượng ít các chỉ thị
- Cho phép song song hóa tăng dần chương trình tuần tự

OpenMP trong kiến trúc chia sẻ bộ nhớ

User Layer

End User

Application

OpenMP API

Directives,
Compiler

OpenMP library

Environment
variables

Runtime library

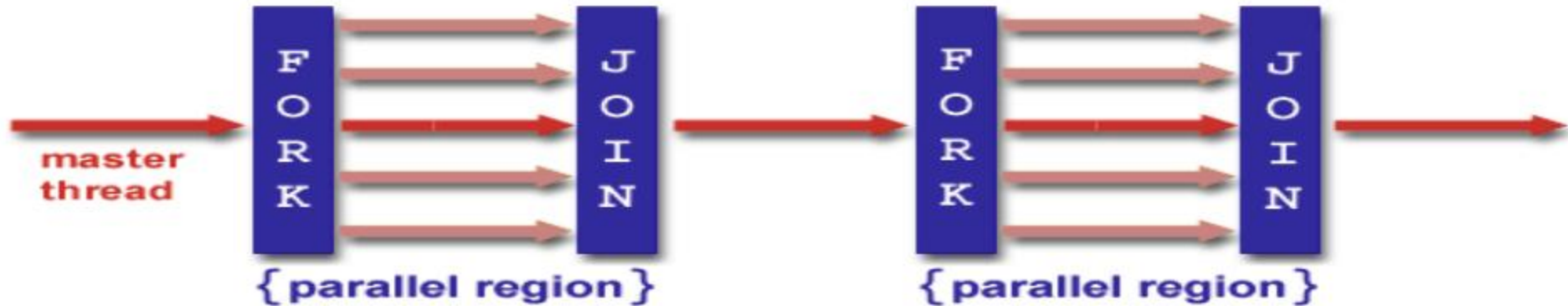
OS/system support for shared memory.

System Layer

Mô hình song song OpenMP

- OpenMP cung cấp mô hình lập trình đa luồng cấp cao, xây dựng trên thư viện lập trình đa luồng của hệ thống, vd. POSIX Threads
- Thực hiện theo mô hình Fork-Join
 - Chương trình OpenMP bắt đầu việc thực hiện như một luồng chủ duy nhất, master thread
 - Luồng chủ thực hiện tuần tự cho đến vùng song song đầu tiên
 - Luồng chủ tạo nhóm các luồng để chia sẻ thực hiện các công việc song song

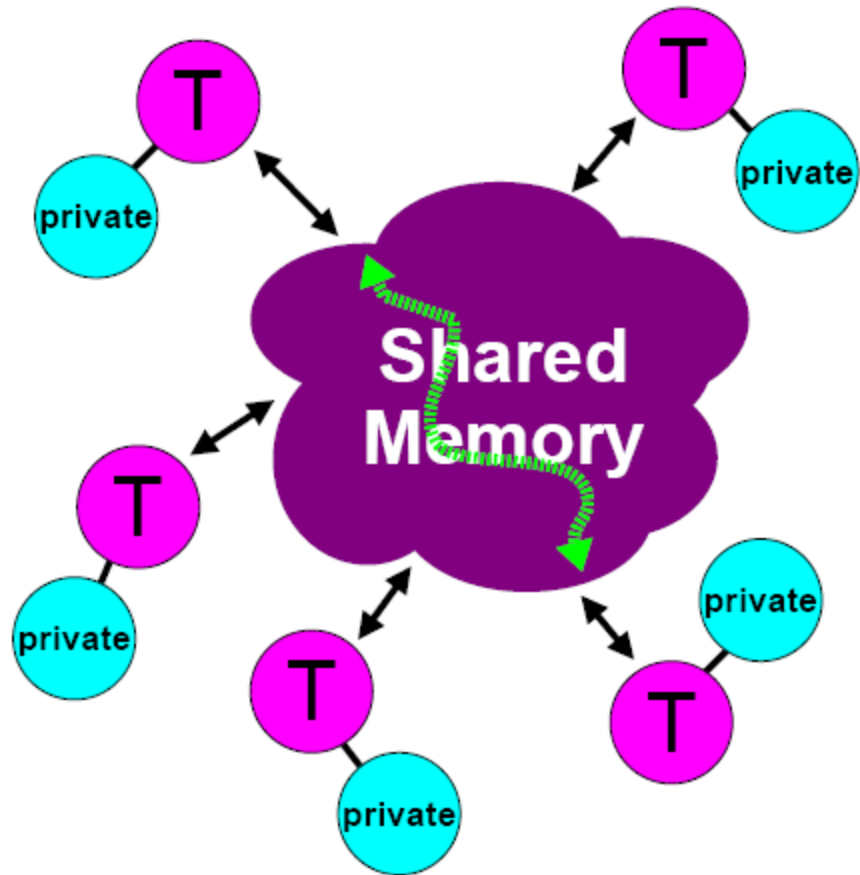
Mô hình Fork-Join



- Song song đa luồng
- Song song có khai báo
- Song song động

Mô hình bộ nhớ OpenMP

- Mọi luồng có quyền truy cập đến vùng nhớ chung toàn cục
- Dữ liệu hoặc là chia sẻ hoặc là riêng tư
- Việc truyền dữ liệu là trong suốt với người lập trình
- Cần thiết phải đồng bộ hóa nhưng hầu như được thực hiện ngầm



Tính năng chính của OpenMP

- Tạo nhóm các luồng cho thực hiện song song
- Chỉ rõ cách các chia sẻ công việc giữa các luồng thành viên của nhóm
- Khai báo dữ liệu chia sẻ và riêng tư
- Đồng bộ các luồng và cho phép các luồng thực hiện thực hiện công việc một cách độc quyền
- Cung cấp hàm thời gian chạy
- Quản lý số lượng luồng

Viết chương trình song song

- Chia tách bài toán thành các công việc
 - Lý tưởng nhất khi các công việc là hoàn toàn độc lập
- Gán công việc cho các luồng thực thi
- Viết mã trên môi trường lập trình song song
- Thiết kế chương trình phụ thuộc vào
 - Nền tảng phần cứng
 - Cấp độ song song
 - Bản chất của bài toán

Phong cách lập trình OpenMP

- Song song theo dữ liệu
 - Khuyến khích lập trình song song có cấu trúc dựa trên phân chia công việc trong vòng lặp
 - `#pragma omp parallel for`
- Song song theo công việc
 - Hỗ trợ việc gán các công việc cụ thể cho các luồng thông qua chỉ số của luồng
 - `#pragma omp parallel sections`


```
/* first.c */
```

```
#include <omp.h>
```

OpenMP Runtime Routines

```
int main()
```

```
{
```

```
    double a[1000], b[1000], c[1000];
```

```
    #pragma omp parallel for
```

OpenMP Compiler Directives

```
    for (int i=0; i < 1000; i++)
```

```
        c[i] = a[i] + b[i];
```

```
    return 0;
```

```
}
```

Thread 0

Thread 1

Thread 2

Thread 3

Thread 4

i=0-199

i=200-399

i=400-599

i=600-799

i=800-999

a[i]

a[i]

a[i]

a[i]

a[i]

```
$ gcc -fopenmp first.c -o first
```

```
$ export OMP_NUM_THREADS=5
```

```
$ ./first
```

OpenMP Environment Variables

+

+

b[i]

b[i]

=

=

=

=

=

c[i]

c[i]

c[i]

c[i]

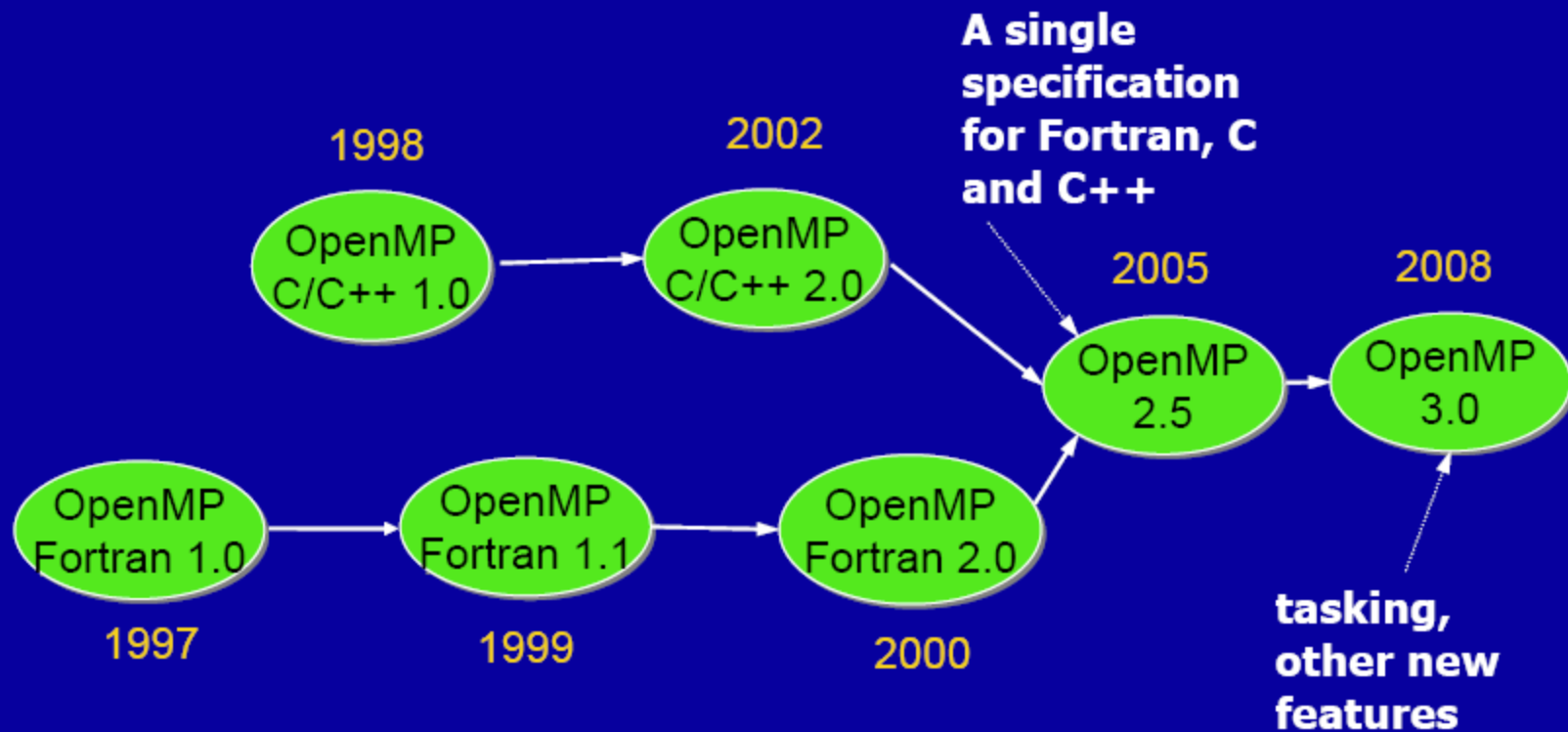
c[i]

```
/* second.c */  
#include <omp.h>  
#define N 1000  
int main()  
{  
    double a[N], b[N], c[N], d[N];  
    #pragma omp parallel sections nowait  
    {  
        #pragma omp section  
        for (int i=0; i < N; i++)  
            c[i] = a[i] + b[i];  
  
        #pragma omp section  
        for (int i=0; i < N; i++)  
            d[i] = a[i] & b[i];  
    }  
    return 0;  
}
```

Dịch chương trình OpenMP

- GNU GCC
 - Version GCC 4.3.2, OpenMP 3.0
 - `#include <omp.h>`
 - `gcc -fopenmp`
- Microsoft Visual C++
 - Version VC++ 2005, 2008, OpenMP 2.0
 - Win 32 Console Project → Empty Project
 - project properties → configuration properties → C.C++/language
 - Activate OpenMP option
 - Build project
 - Run “without debug”

OpenMP Release History



Tài liệu tham khảo

- <http://www.openmp.org>
- <http://www.compunity.org>
- An Introduction to OpenMP, tutorials, by Ruud van der Pas
- Parallel Programming in OpenMP, by Rohit Chandra et al.
- Using OpenMP, by Chapman, Jost, and Van Der Pas



Chương trình OpenMP đầu tiên

Nhân ma trận với vector

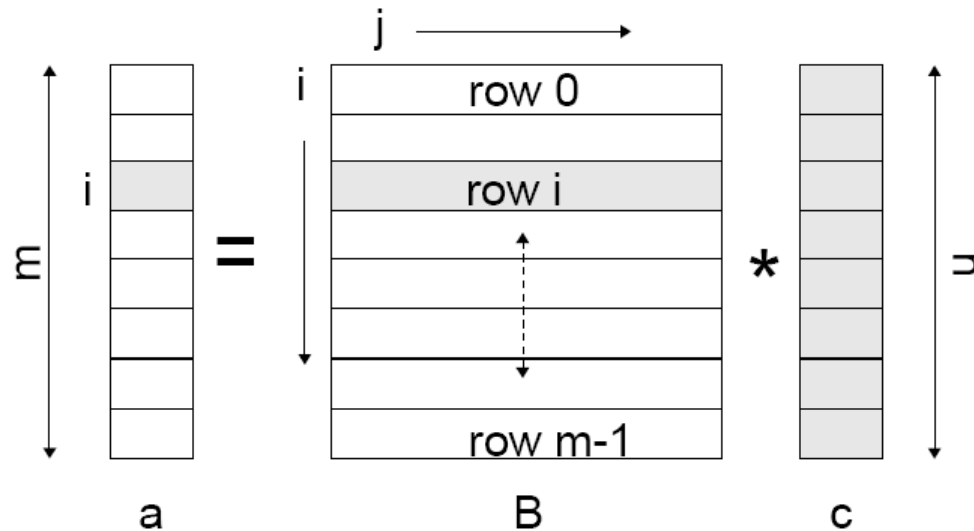
- Cho ma trận $B_{m \times n}$ và vector $c_{n \times 1}$
- Tính $a_{m \times 1} = B \times c$
- a_i được tính bằng phép lấy tích vô hướng

$$a_i = \sum_{j=1}^n B_{i,j} * c_j \quad i = 1, \dots, m$$

Chương trình nhân ma trận với vector “mxv.c”

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 void mxv(int m, int n, double* a, double* b, double* c);
05
06 int main()
07 {
08     double *a,*b,*c;
09     int i, j, m, n;
10     printf("Please give m and n: ");
11     scanf("%d %d",&m,&n);
12     a = (double*) malloc(m*sizeof(double));
13     b = (double*) malloc(m*n*sizeof(double));
14     c = (double*) malloc(n*sizeof(double));
15     printf("Initializing matrix B and vector c\n");
16     for (j=0; j<n; j++)
17         c[j] = 2.0;
18     for (i=0; i<m; i++)
19         for (j=0; j<n; j++)
20             b[i*n+j] = i;
21     printf("Executing mxv function for m = %d n = %d\n",m,n);
22     mxv(m, n, a, b, c);
23     free(a);free(b);free(c);
24     return(0);
25 }
```


Thủ tục nhân ma trận với vector



```
01 void mxv(int m, int n, double* a, double* b, double* c)
02 {
03     int i, j;
04     for (i=0; i<m; i++)
05     {
06         a[i] = 0.0;
07         for (j=0; j<n; j++)
08             a[i] += b[i*n+j]*c[j];
09     }
10 }
```

Song song hóa hàm `mxv`

```
01 void mxv(int m, int n, double* a, double* b, double* c)
02 {
03     int i, j;
04     #pragma omp parallel for default(none) shared(m,n,a,b,c) private(i,j)
05     for (i=0; i<m; i++)
06     {
07         a[i] = 0.0;
08         for (j=0; j<n; j++)
09             a[i] += b[i*n+j]*c[j];
10     } /*-- End of omp parallel for --*/
11 }
```

- Dấu hiệu bắt đầu chỉ thị: `#pragma omp`
- Chỉ thị: `parallel for`
- Mệnh đề: `default, shared, private`

Chỉ thị OpenMP

- OpenMP sử dụng chỉ thị chương trình dịch để điều khiển sự song song
- Chỉ thị của OpenMP luôn bắt đầu bởi **#pragma omp**
- Dạng tổng quát cho một chỉ thị OpenMP

#pragma omp <i>directive-name</i> [<i>clause</i> [[,] <i>clause</i>]....] <i>new-line</i>
--

- Chương trình ví dụ sử dụng chỉ thị **parallel for** và các mệnh đề **default**, **shared**, **private**

Chỉ thị kết hợp parallel for

- Sử dụng trước câu lệnh lặp for
- Tạo vùng thực hiện song song và chỉ định rằng vòng lặp được phân phối cho các luồng
- Các mệnh đề tiếp theo xác định tính chất của dữ liệu: chia sẻ hay riêng tư
- `default(none)`: không sử dụng các quy tắc dựng sẵn của OpenMP cho tính chất của dữ liệu
- → người lập trình phải tự khai báo tính chất cho dữ liệu

Khai báo tính chất dữ liệu

- `shared(list)` các biến trong *list* là chung cho các luồng
- `private(list)` các biến là riêng cho luồng → mỗi luồng có một bản copy riêng của biến
- Trong chương trình ví dụ
 - Các biến *a, b, c, m, n* cần được truy cập bởi các luồng nên các biến này là chia sẻ: `shared(a,b,c,m,n)`
 - Các biến chạy *i, j* là riêng cho các luồng?
`private(i, j)`

Ghép mã nguồn tuần tự và song song

- OpenMP cho phép viết chương trình song song trong khi vẫn giữ nguyên mã nguồn của chương trình tuần tự
- Dịch mã nguồn thành chương trình tuần tự

```
$ gcc -fopenmp mvx.c -o mvx
```



- không thiết lập lựa chọn OpenMP hoặc,
- sử dụng các chương trình dịch không hỗ trợ OpenMP
- → các chỉ thị chương trình dịch bị bỏ qua khi dịch
- chương trình bỏ qua các biến môi trường khi chạy

Bỏ qua hàm của OpenMP

- Chương trình OpenMP có thể có thêm các hàm thư viện được khai báo trong `<omp.h>`
- Mã nguồn chương trình tuần tự cần bỏ qua các hàm này
- Có thể sử dụng macro `_OPENMP`, được định nghĩa khi dịch chương trình với lựa chọn OpenMP

```
#ifdef _OPENMP
#include <omp.h>
#endif

int main()
{
    int TID;


    ...

#ifdef _OPENMP
    TID = omp_get_thread_num();
#else
    TID = 0;
#endif

    ...

}
```

trả về chỉ số của luồng



Các thành phần của OpenMP

Thuật ngữ

- Thread Teams: Master + Workers
- Khối song song (vùng song song – parallel region)
 - Khối mã được thực hiện đồng thời bởi tất cả các luồng
 - Thường được đóng trong { }
- Khối chia sẻ công việc (work-sharing construct)
 - Việc thực hiện đoạn mã của khối được chia cho các luồng

Thành phần của OpenMP

Directives

- ◆ *Parallel regions*
- ◆ *Work sharing*
- ◆ *Synchronization*
- ◆ *Data scope attributes*
 - ☞ *private*
 - ☞ *firstprivate*
 - ☞ *lastprivate*
 - ☞ *shared*
 - ☞ *reduction*
- ◆ *Orphaning*

Environment variables

- ◆ *Number of threads*
- ◆ *Scheduling type*
- ◆ *Dynamic thread adjustment*
- ◆ *Nested parallelism*

Runtime environment

- ◆ *Number of threads*
- ◆ *Thread ID*
- ◆ *Dynamic thread adjustment*
- ◆ *Nested parallelism*
- ◆ *Timers*
- ◆ *API for locking*

Khối mã của OpenMP

- Khối mã của OpenMP là chỉ thị thực thi OpenMP kèm theo đoạn mã gồm câu lệnh, vòng lặp, ký tự bao đóng { }
- Khối mã của OpenMP được chia làm
 - Khối song song (vùng song song)
 - Khối chia sẻ công việc
 - Khối đồng bộ
 - Hàm thời gian chạy và biến môi trường

Quan trọng nhất

Khối song song

- Chỉ thị `#pragma omp parallel`
- Đoạn mã của khối được thực hiện bởi tất cả các luồng
- Kết thúc khối có điểm đồng bộ `barrier` ngầm định
- Sử dụng mệnh đề `nowait` để bỏ `barrier`
- Luồng gặp khối song song sẽ trở thành luồng chủ
- Luồng chủ tạo nhóm các luồng thực hiện công việc
- Đến cuối khối, luồng chủ đợi các luồng trong nhóm kết thúc rồi mới thoát ra ngoài
- Bên ngoài khối, chỉ có luồng chủ làm việc → thực hiện tuần tự

Cú pháp khối song song

```
#pragma omp parallel [clause[[,] clause] ...]  
{  
    "this will be executed in parallel"  
} (implied barrier)
```

Cú pháp khai báo khối song song

```
#pragma omp parallel if (n>limit) default(none) \  
    shared(n,a,b,c,x,y,z) private(f,i,scale)  
{
```

Khối song song với mệnh đề ‘ if ’

- Nếu mệnh đề ‘ if ’ không thỏa mãn thì nội dung của khối được thực hiện tuần tự

Khởi song song đơn giản

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

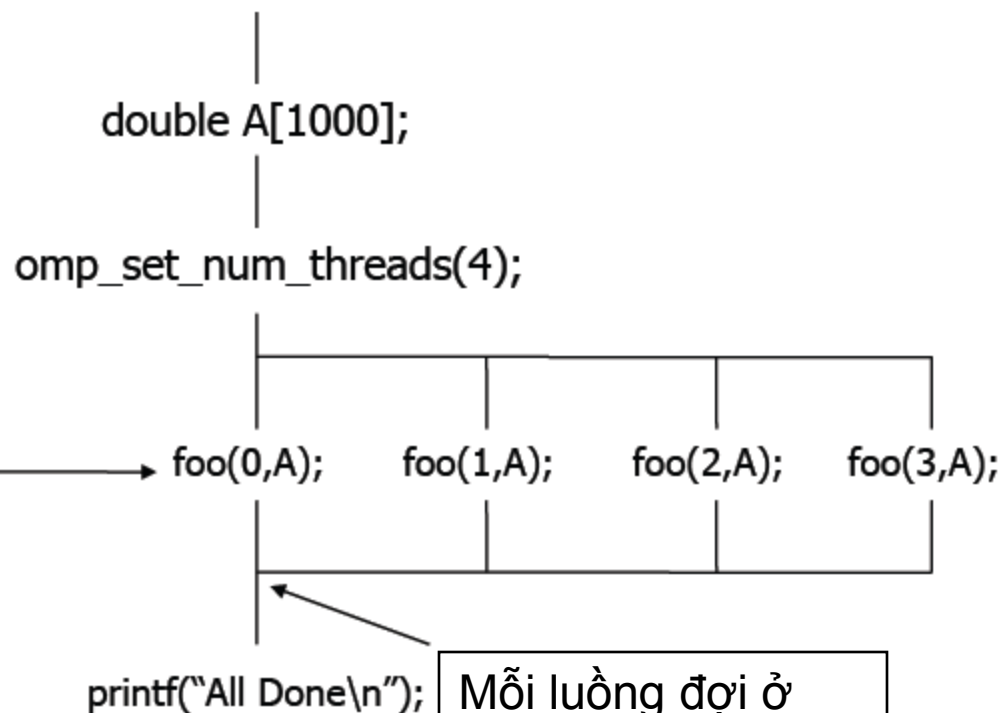
int main ( ) {
    printf("Starting off in the sequential world.\n");
    #pragma omp parallel
    {

        printf("Hello from thread number %d\n", omp_get_thread_num() );
    }
    printf("Back to the sequential world.\n");
}
```

Hình ảnh thực hiện khối song song

```
double A[1000];  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    int ID =omp_get_thread_num();  
    foo(ID,A);  
}  
printf("All Done\n");
```

Chỉ có một biến A duy nhất được chia sẻ giữa các luồng



Mỗi luồng đợi ở đây cho đến khi tất cả các luồng kết thúc

Các mệnh đề dùng với khối song song

if (<i>scalar-expression</i>)	(C/C++)
if (<i>scalar-logical-expression</i>)	(Fortran)
num_threads (<i>integer-expression</i>)	(C/C++)
num_threads (<i>scalar-integer-expression</i>)	(Fortran)
private (<i>list</i>)	
firstprivate (<i>list</i>)	
shared (<i>list</i>)	
default (<i>none</i> <i>shared</i>)	(C/C++)
default (<i>none</i> <i>shared</i> <i>private</i>)	(Fortran)
copyin (<i>list</i>)	
reduction (<i>operator:list</i>)	(C/C++)
reduction (<i>{ operator intrinsic_procedure_name } :list</i>)	(Fortran)

Xác định số luồng

- Số lượng luồng được khởi tạo trong vùng song song được xác định theo các cách:
 - Mệnh đề `num_threads()`
 - Hàm thư viện `omp_set_num_threads()`
 - Biến môi trường `OMP_NUM_THREADS`
 - Giá trị ngầm định đặt bởi trình dịch
- Số luồng thực tế có thể nhỏ hơn số chỉ định do thiếu tài nguyên → kiểm tra cụ thể số luồng khi chạy chương trình

Khối chia sẻ công việc

- Chỉ thị
 - `#pragma omp for`
 - `#pragma omp sections`
 - `#pragma omp single`
- Công việc trong khối được phân cho các luồng
- Phải được đặt trong khối song song
- Phải được tiếp cận bởi tất cả các luồng hoặc không một luồng nào
- Khối chia sẻ không tạo thêm luồng mới

Minh họa 3 khối chia sẻ

```
#pragma omp for  
{  
    ....  
}
```

```
#pragma omp sections  
{  
    ....  
}
```

```
#pragma omp single  
{  
    ....  
}
```

- Điểm đồng bộ ngầm định được đặt ở cuối khối
- Sử dụng mệnh đề **nowait** để bỏ điểm đồng bộ

Khối chia sẻ for

- Chỉ thị `#pragma omp for`
- Nội dung của khối là câu lệnh lặp for
- Song song dữ liệu: mỗi đoạn của lệnh lặp for được thực hiện bởi một luồng
- Khối chia sẻ công việc chuẩn và được dùng nhiều nhất
- Chú ý:
 - Số bước lặp trong lệnh lặp for phải đếm được
 - Biến chạy phải là biến riêng của luồng
 - Biến kích thước vòng lặp là biến chia sẻ

Cú pháp và mệnh đề

```
#pragma omp for [clause[[,] clause] ...]  
    <original for-loop>
```

for (*init-expr*; *var logical-op b*; *incr-expr*)

Câu lệnh for phải có dạng chính tắc để có thể xác định trước số bước lặp

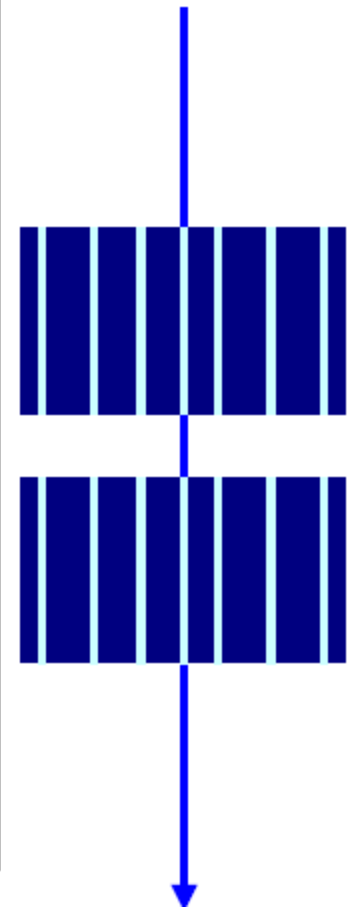
```
private (list)  
firstprivate (list)  
lastprivate (list)  
reduction (operator: list) (C/C++)  
reduction ({ operator | intrinsic_procedure_name } : list) (Fortran)  
ordered  
schedule (kind [, chunk_size])  
nowait
```

Ví dụ khối lặp for

```
#pragma omp parallel default(none)\
    shared(n,a,b,c,d) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;

    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];

} /*-- End of parallel region --*/
    (implied barrier)
```



Khối chia sẻ section

- Chỉ thị: `#pragma omp sections`
- Nội dung gồm các khối con, bắt đầu bởi chỉ thị `#pragma omp section`
- Song song công việc: mỗi khối section con được thực hiện bởi một luồng
- Chú ý:
 - Số section lớn hơn số luồng: có luồng thực hiện nhiều section
 - Số luồng lớn hơn số section: có luồng được để ở trạng thái nghỉ

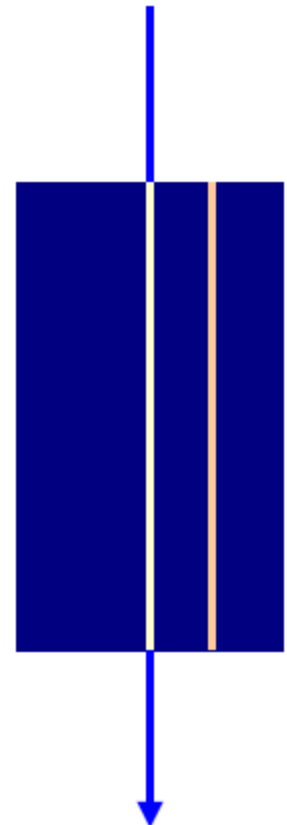
Cú pháp và mệnh đề

```
#pragma omp sections [clause(s)]  
{  
  #pragma omp section  
    <code block1>  
  #pragma omp section  
    <code block2>  
  #pragma omp section  
    :  
}
```

```
private(list)  
firstprivate(list)  
lastprivate(list)  
reduction(operator:list) (C/C++)  
reduction({operator | intrinsic_procedure_name}:list) (Fortran)  
nowait
```


Ví dụ khối section

```
#pragma omp parallel default(none) \  
    shared(n,a,b,c,d) private(i)  
{  
    #pragma omp sections nowait  
    {  
        #pragma omp section  
        for (i=0; i<n-1; i++)  
            b[i] = (a[i] + a[i+1])/2;  
  
        #pragma omp section  
        for (i=0; i<n; i++)  
            d[i] = 1.0/c[i];  
  
    } /*-- End of sections --*/  
  
} /*-- End of parallel region --*/
```



Cú pháp rút gọn

```
#pragma omp parallel  
{  
    #pragma omp for  
    for-loop  
}
```

```
#pragma omp parallel for  
for-loop
```

```
#pragma omp parallel  
#pragma omp for  
for (...)
```



```
#pragma omp parallel for  
for (...)
```

Single PARALLEL loop

```
#pragma omp parallel  
{  
    #pragma omp sections  
{  
        [#pragma omp section ]  
        structured block  
        [#pragma omp section  
        structured block ]  
    }  
}
```

```
#pragma omp parallel sections  
{  
    [#pragma omp section ]  
    structured block  
    [#pragma omp section  
    structured block ]  
    ...  
}
```

```
#pragma omp parallel  
#pragma omp sections  
{ ... }
```



```
#pragma omp parallel sections  
{ ... }
```

Single PARALLEL sections

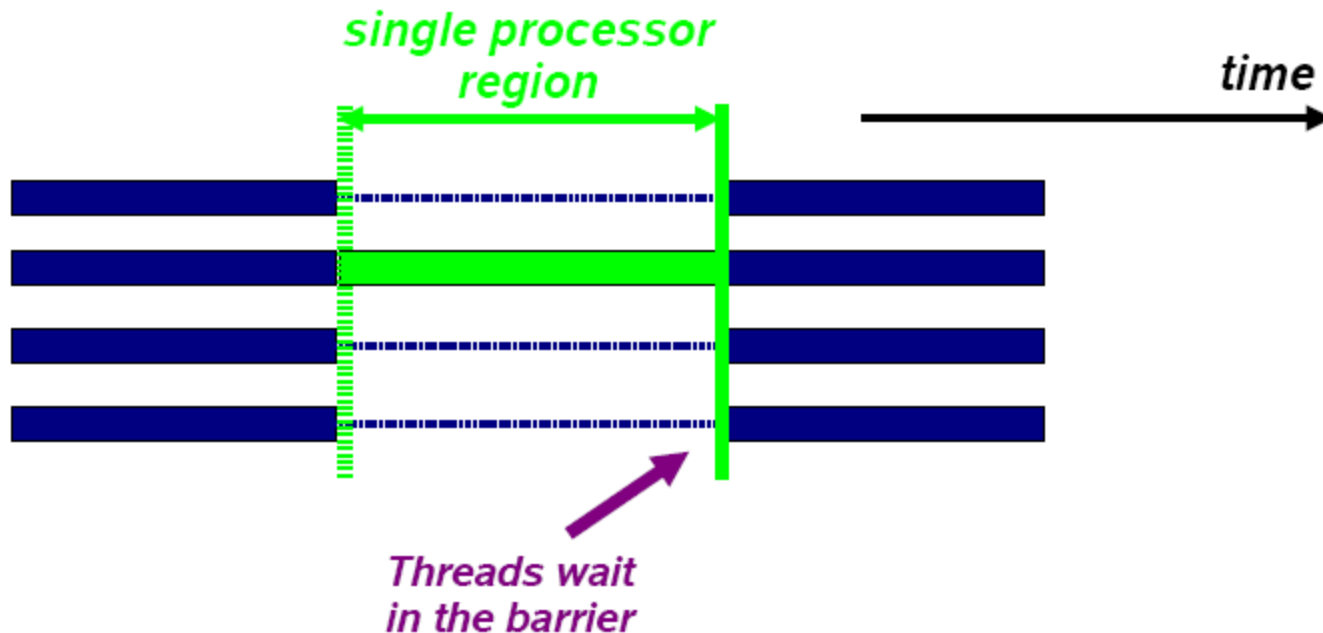
Khối single

```
#pragma omp single [clause[[,] clause] ...]  
{  
    <code-block>  
}
```

- Chỉ thị: `#pragma omp single`
- Nội dụng là đoạn mã trong khối { }
- Chỉ được thực hiện bởi một luồng
- Luồng thực hiện có thể được chọn ngẫu nhiên trong nhóm
- Kết thúc khối có điểm đồng bộ ngầm định

Ví dụ khối single

```
#pragma omp parallel private (tmp)
{
    do_many_things();
    #pragma omp single
    { exchange_boundaries(); }
    do_many_other_things();
}
```



Một số mệnh đề điều khiển

- shared,
- private,
- lastprivate,
- firstprivate,
- default,
- num_threads,
- nowait,
- reduction,
- schedule

xác định phạm vi truy cập
của biến đã khai báo

shared và private

- **private:**
 - Chỉ định các biến là cục bộ của luồng
 - Giá trị không xác định tại điểm vào và điểm ra luồng
 - Biến đã khai báo gọi là đối tượng tham chiếu của biến cục bộ
- **shared:**
 - Chỉ định các biến là toàn cục, có thể được truy cập bởi mọi luồng
 - Mọi luồng đều truy cập đến cùng địa chỉ trên không gian nhớ

private (list)

shared (list)

```
#pragma omp parallel if (n > threshold) \  
    shared(n,x,y) private(i)  
{  
    #pragma omp for  
    for (i=0; i<n; i++)  
        x[i] += y[i];  
} /*-- End of parallel region --*/
```

lastprivate và firstprivate

firstprivate (list)

- firstprivate:
 - biến cục bộ được khởi tạo bằng giá trị của đối tượng tham chiếu trước khi vào khối song song

lastprivate (list)

- lastprivate:
 - giá trị của biến cục bộ tại đoạn lặp cuối (for) hoặc tại khối section cuối được gán cho đối tượng tham chiếu trước khi kết thúc khối song song

Ví dụ firstprivate và lastprivate

```
main()
{
    A = 10;

    #pragma omp parallel
    {
        #pragma omp for private(i) firstprivate(A) lastprivate(B)...
        for (i=0; i<n; i++)
        {
            ....
            B = A + i;
            ....
        }

        C = B;

    } /*-- End of OpenMP parallel region --*/
}
```

/*-- A undefined, unless declared firstprivate --*/

/*-- B undefined, unless declared lastprivate --*/

default và nowait

- **default:** `default (none | shared)`
 - **none**: bỏ thiết lập ngầm định đối với phạm vi truy xuất của biến → người lập trình phải tự xác định
 - **shared**: các biến đều là biến chia sẻ giữa các luồng, trừ khi sử dụng mệnh đề **private**
- **nowait:**
 - Bỏ các điểm đồng bộ (barrier) ngầm định

```
#pragma omp for nowait
{
    :
}
```

num_threads

- Chỉ định số luồng cho khối song song

```
#include <omp.h>

int main()
{
    int TID;
    int NoT;
    #pragma omp parallel num_threads(5)
    {
        TID = omp_get_thread_num();
        NoT = omp_get_num_threads();
        printf("I'm the number %d in %d", TID, NoT);
    }
    return 0;
}
```

reduction

reduction (operator : list)

Operator	Initialization value
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0

```
x = x op expr
x binop= expr
x = expr op x (except for subtraction)
x++
++x
x--
--x
```

Ví dụ reduction

```
sum = 0;
for (i=0; i<n; i++)
    sum += a[i];
```

```
/*-- Executed by thread 0 --*/      /*-- Executed by thread 1 --*/

sumLocal = 0;                        sumLocal = 0;
for (i=0; i<n/2; i++)                for (i=n/2-1; i<n; i++)
    sumLocal += a[i];                sumLocal += a[i];

sum += sumLocal;                     
                     sum += sumLocal;
```

```
#pragma omp parallel for default(none) shared(n,a) \
                        reduction(+:sum)
    for (i=0; i<n; i++)
        sum += a[i];
/*-- End of parallel reduction --*/
printf("Value of sum after parallel region: %d\n",sum);
```

*Variable SUM is a
shared variable*

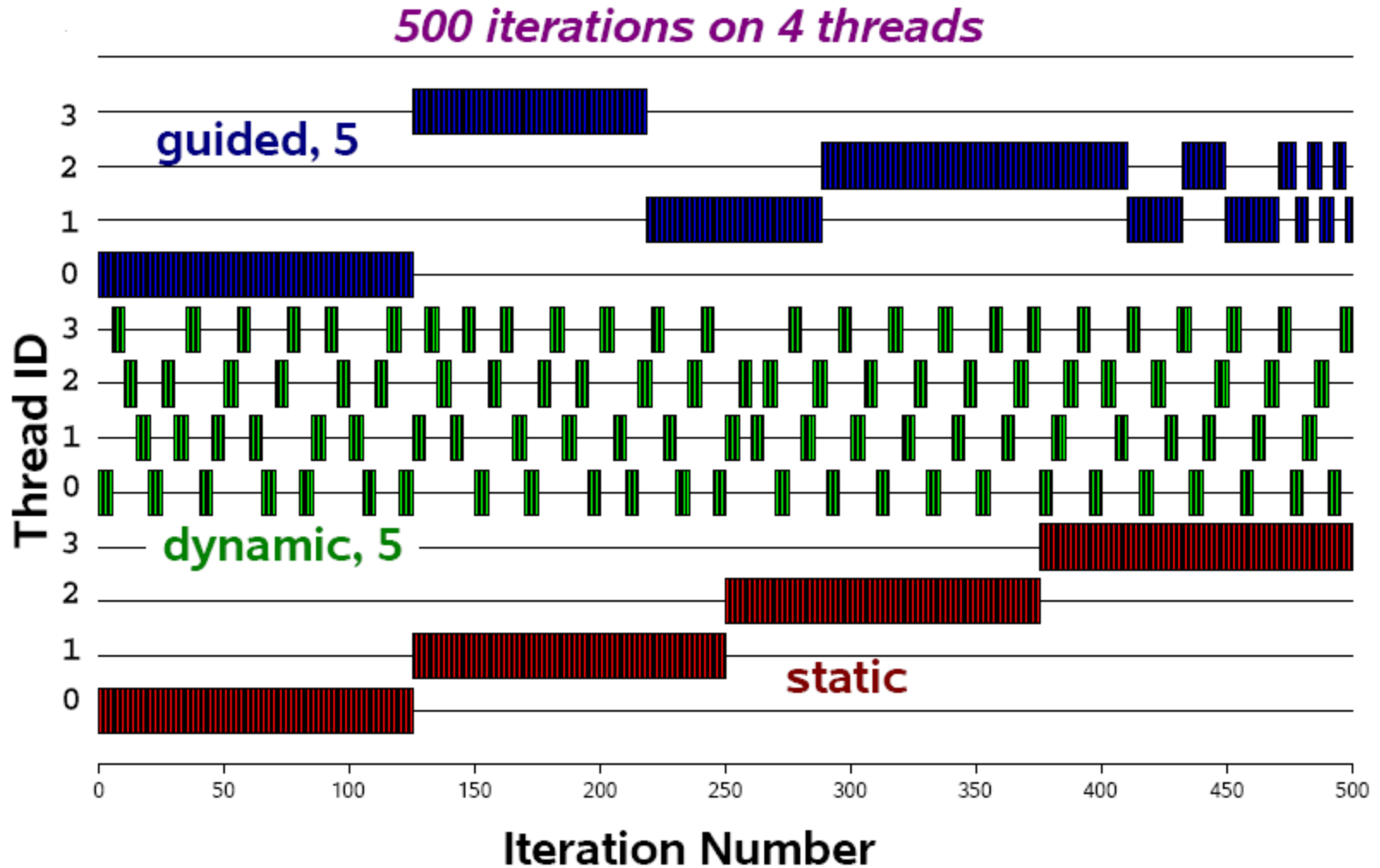


schedule

```
schedule ( static | dynamic | guided [, chunk] )  
schedule (runtime)
```

- Phân chia công việc cho các luồng
 - **chunk**: kích thước mỗi đoạn lặp
 - **static**: chia cố định các đoạn lặp cho các luồng
 - **dynamic**: có hàng đợi chung chứa các đoạn lặp, khi luồng kết thúc một đoạn lặp, nó lấy đoạn lặp tiếp theo từ hàng đợi
 - **guide**: như **dynamic**, nhưng kích thước các đoạn lặp giảm theo hàm mũ, mỗi đoạn lặp có kích thước không nhỏ hơn **chunk**
 - **runtime**: cách phân chia được xác định tại thời điểm chạy, thông qua biến môi trường **OMP_SCHEDULE**

Minh họa các cách chia



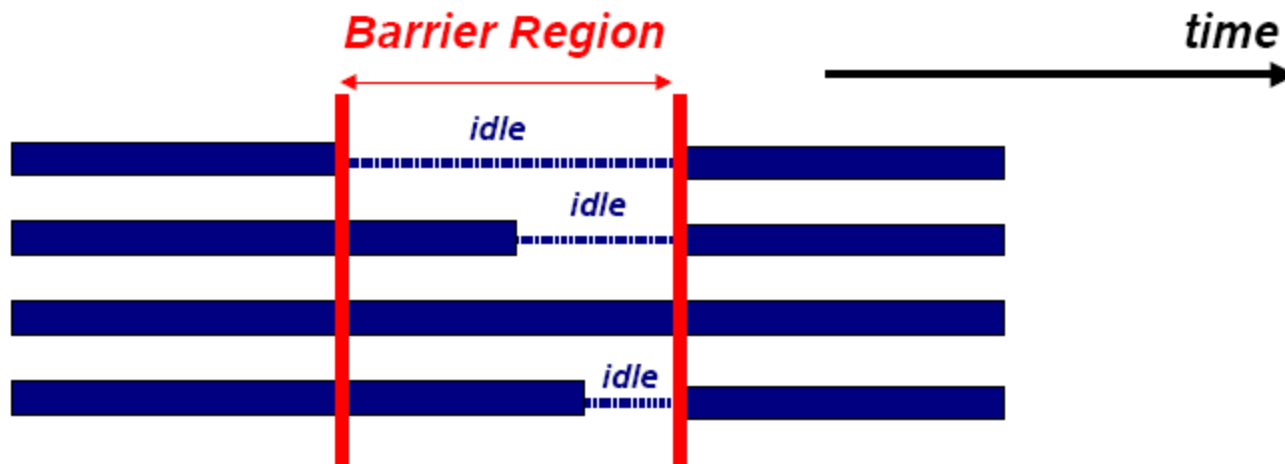
Các khối tạo sự đồng bộ

- barrier,
- critical,
- atomic,
- master,
- ordered,
- flush

Xem thêm tài liệu tham khảo

barrier

```
#pragma omp barrier
```



- Tạo điểm đồng bộ trong chương trình
- Không luồng nào được phép vượt qua barrier trước khi mọi luồng đều tiếp cận barrier
- Buộc các luồng phải đợi nhau
- **barrier** ngầm định được đặt cuối các khối song song
- Sử dụng **barrier** tốn chi phí → cân nhắc !

Ví dụ sử dụng barrier

Tạo thời gian chênh lệch giữa các luồng

```
#pragma omp parallel private(TID)
{
    TID = omp_get_thread_num();
    if (TID < omp_get_num_threads()/2 ) system("sleep 3");
    (void) print_time(TID,"before");
```

```
#pragma omp barrier
```

Đặt điểm đồng bộ

```
(void) print_time(TID,"after ");
} /*-- End of parallel region --*/
```

critical

```
#pragma omp critical [(name)]  
{<code-block>}
```

- Không cho phép nhiều luồng đồng thời cập nhật dữ liệu chia sẻ
- Tên **name** có thể được sử dụng, tên là đối tượng toàn cục nên mỗi khối critical phải có tên riêng
- Luồng phải đợi đến khi không có luồng nào thực hiện đoạn găng cùng tên để vào đoạn găng với tên đó
- Ví dụ với sum là biến chia sẻ

```
for (i=0; i < N; i++){  
    ..... one at a time can proceed  
    sum += a[i];  
    ..... next in line, please  
}
```

Ví dụ sử dụng critical

```
sum = 0;
#pragma omp parallel shared(n,a,sum) private(TID,sumLocal)
{
    TID = omp_get_thread_num();
    sumLocal = 0;
    #pragma omp for
    for (i=0; i<n; i++)
        sumLocal += a[i];
    #pragma omp critical (update_sum)
    {
        sum += sumLocal;
        printf("TID=%d: sumLocal=%d sum = %d\n",TID,sumLocal,sum);
    }
} /*-- End of parallel region --*/
printf("Value of sum after parallel region: %d\n",sum);
```

atomic

```
#pragma omp atomic  
<statement>
```

- Hoạt động như khối critical, tuy nhiên:
 - Nội dung của khối chỉ là một câu lệnh đơn
 - Câu lệnh phải là một trong các dạng sau:

$x \text{ binop} = \text{expr} \mid x++ \mid x-- \mid ++x \mid --x$

Trong đó:

- x có kiểu vô hướng
- expr là biểu thức không tham chiếu đến x và trả về giá trị vô hướng
- binop là một trong các toán tử: +, -, *, /, %, ^, <<, >>

Ví dụ sử dụng atomic

```
int ic, i, n;  
ic = 0;  
#pragma omp parallel shared(n,ic) private(i)  
  for (i=0; i++, i<n)  
  {  
    #pragma omp atomic  
    ic += bigfunc();  
  }  
printf("counter = %d\n", ic);
```

Không hạn chế việc các luồng
thực hiện bigfunc() đồng thời

master và ordered

```
#pragma omp master  
{<code-block>}
```

- master

- Khối mã chỉ được thực hiện bởi luồng chủ
- Không đặt barrier ở cuối khối

```
#pragma omp ordered  
{<code-block>}
```

- ordered

- đoạn mã lặp trong khối được thực hiện theo thứ tự của thực hiện tuần tự lệnh lặp
- Chỉ xuất hiện trong chỉ thị **for** hoặc **parallel for** có kèm mệnh đề **ordered**

Ví dụ sử dụng master

```
#pragma omp parallel shared(a,b) private(i)
{
    #pragma omp master
    {
        a = 10;
        printf("Master construct is executed by thread %d\n",
               omp_get_thread_num());
    }

    #pragma omp barrier ← Đặt barrier công khai

    #pragma omp for
    for (i=0; i<n; i++)
        b[i] = a;

} /*-- End of parallel region --*/

printf("After the parallel region:\n");
for (i=0; i<n; i++)
    printf("b[%d] = %d\n",i,b[i]);
```

Chỉ thị “mồ côi”

- orphaned directive
- OpenMP không giới hạn chỉ thị chia sẻ và đồng bộ phải nằm trong đoạn mã của vùng chỉ thị song song

```
(void) dowork(); !- Sequential FOR  
  
#pragma omp parallel  
{  
    (void) dowork(); !- Parallel FOR  
}
```

```
void dowork()  
{  
    #pragma for  
        for (i=0;....)  
        {  
            :  
        }  
}
```


Tương tác với biến môi trường

- Biến môi trường có thể được sử dụng để điều khiển chương trình OpenMP khi chạy
- Ví dụ `OMP_NUM_THREADS` để chỉ định số luồng cho vùng song song
- Tương tác thông qua biến điều khiển nội tại của chương trình OpenMP
 - Quản lý bởi trình dịch OpenMP
 - Không thể truy cập hay thay đổi trực tiếp
 - Bị thay đổi qua hàm OpenMP hoặc biến môi trường

Ví dụ điều khiển số luồng

- Biến nội tại *nthreads_var* của thư viện OpenMP điều khiển số lượng luồng làm việc trong vùng song song
- Tại dòng lệnh hệ thống, biến môi trường **OMP_NUM_THREADS** được thiết lập
- Giá trị đó được gán cho biến *nthreads_var*

Một số biến môi trường

OpenMP environment variable

OMP_NUM_THREADS n

OMP_SCHEDULE “schedule,[chunk]”

OMP_DYNAMIC { TRUE | FALSE }

OMP_NESTED { TRUE | FALSE }

- **OMP_STACKSIZE** (for non-master threads)
 - **OMP_WAIT_POLICY** (ACTIVE or PASSIVE)
 - **OMP_MAX_ACTIVE_LEVELS**
—integer value for maximum # nested parallel regions
 - **OMP_THREAD_LIMIT** (# threads for entire program)
- OpenMP 3.0

Các hàm OpenMP

- OpenMP cung cấp các hàm giúp người dùng
 - Điều khiển và truy vấn môi trường song song
 - Thủ tục semaphore/lock đa mục đích...
- Hàm có độ ưu tiên cao hơn các biến môi trường tương ứng
- Chương trình C/C++ cần thêm khai báo
`#include <omp.h>`
- Nên sử dụng kèm macro
`#ifdef _OPENMP`

Một số hàm OpenMP

- Đếm số BXL

```
int omp_get_num_procs(); /* # PE currently available */  
int omp_in_parallel(); /* determine whether running in parallel */
```

- Đếm và xác định luồng

```
/* max # threads for next parallel region. only call in serial region */  
void omp_set_num_threads(int num_threads);  
  
int omp_get_num_threads(); /*# threads currently active */
```

Một số hàm OpenMP

- Điều khiển và giám sát khởi tạo luồng

```
void omp_set_dynamic (int dynamic_threads);  
int omp_get_dynamic ();  
void omp_set_nested (int nested);  
int omp_get_nested ();
```

- Loại trừ lẫn nhau

```
void omp_init_lock(omp_lock_t *lock);  
void omp_destroy_lock(omp_lock_t *lock);  
  
void omp_set_lock(omp_lock_t *lock);  
void omp_unset_lock(omp_lock_t *lock);  
int omp_test_lock(omp_lock_t *lock);
```

OpenMP 3.0

- Mục tiêu: hỗ trợ song song hóa cho những bài toán lặp không chính quy
 - Lặp với số bước không xác định,
 - Đệ quy,
 - producer/consumer
- Đưa vào chỉ thị nhiệm vụ **task**

#pragma omp task [clause list]

- nhiệm vụ được thực hiện bởi luồng
- nhiệm vụ gắn chặt với luồng

Ví dụ duyệt danh sách

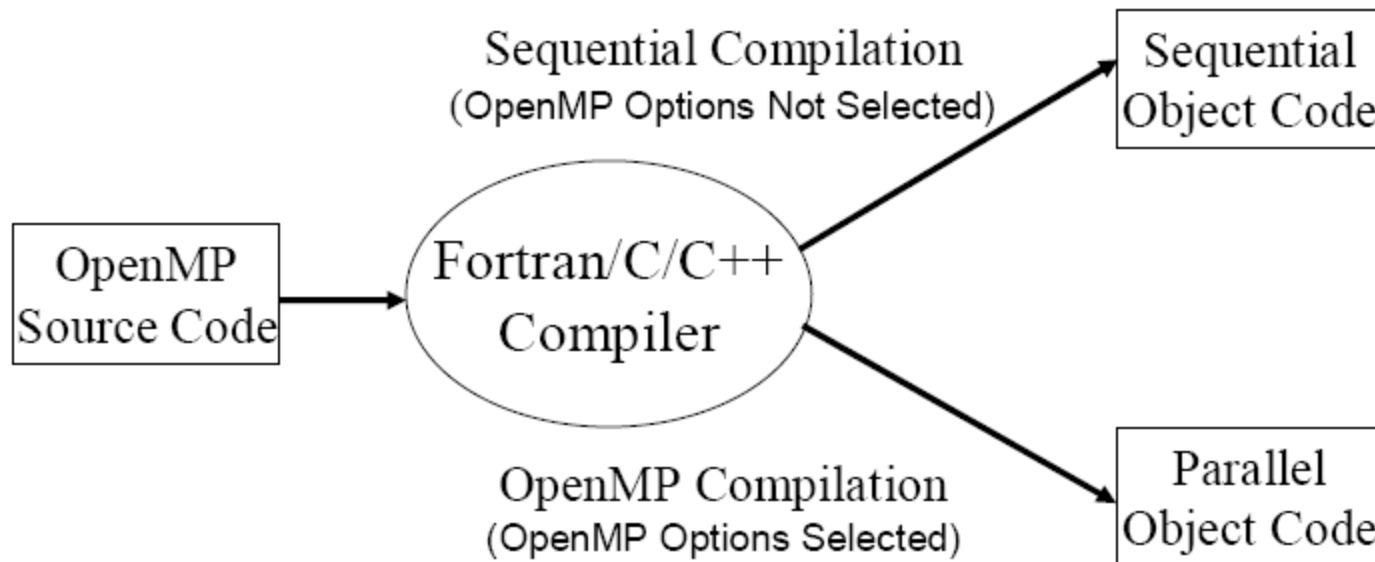
- Nhiệm vụ (duyệt phần tử) được tạo khi một luồng gặp chỉ thị **task**
- Nhiệm vụ này được gán cho một luồng nào đó
- → Duyệt song song danh sách liên kết !!

```
Element first, e;  
#pragma omp parallel  
#pragma omp single  
{  
    for (e = first; e; e = e->next)  
#pragma omp task firstprivate(e)  
        process(e);  
}
```


OpenMP được dịch
như thế nào?

Trình dịch phải hỗ trợ OpenMP

- Chuẩn OpenMP được triển khai trên nền tảng của chương trình dịch C/C++ hoặc Fortran
- Chương trình được dịch bằng cách thiết lập lựa chọn OpenMP trong tham số của lệnh dịch



Thư viện lập trình đa luồng

- Các chỉ thị OpenMP được dịch thành lời gọi các hàm quản lý và gán công việc cho luồng
- Chương trình dịch sử dụng thư viện lập trình đa luồng để dịch ra chương trình đích
- Các chương trình dịch khác nhau sử dụng thư viện lập trình đa luồng khác nhau
 - POSIX threads
 - Solaris Threads
 - QuickThreads

Chuyển chỉ thị OpenMP về các hàm PThreads

```
int a, b;
main() {
    [ // serial segment
      #pragma omp parallel num_threads (8) private (a) shared (b)
      { [ // parallel segment
        ]
      } [ // rest of serial segment
    ]
}
```

Sample OpenMP program

Code inserted by the OpenMP compiler

```
int a, b;
main() {
    [ // serial segment
      for (i = 0; i < 8; i++)
        pthread_create (....., internal_thread_fn_name, ...);
      for (i = 0; i < 8; i++)
        pthread_join (.....);
    ] [ // rest of serial segment
  ]
  void *internal_thread_fn_name (void *packaged_argument) [
    int a;
  ] [ // parallel segment
  ]
}
```

Corresponding Pthreads translation

Dịch chương trình OpenMP

- OpenMP tách người lập trình với thư viện lập trình đa luồng
- Chương trình dịch xử lý các chỉ thị OpenMP và dùng chúng để tạo các đoạn mã đa luồng
- Các mã này sẽ gọi hàm thời gian chạy của thư viện đa luồng
 - Những hàm của OpenMP cũng được thực hiện qua các lời gọi đến các hàm thư viện trên

Pha đầu

- Đọc, phân tích chỉ thị OpenMP
- Kiểm tra lỗi
- Chuyển về dạng biểu diễn trung gian với các chú giải của OpenMP

Pha giữa

- Tiền xử lý các khối OpenMP
 - Chuẩn hóa khối
 - Chuyển về khối tương đương
 - Công khai các điểm đồng bộ ngầm định
 - Tiếp tục kiểm tra lỗi
- Áp dụng một số phép tối ưu hóa
 - Hợp các vùng song song kề nhau
 - Hợp các điểm đồng bộ kề nhau

Chuẩn hóa các khối OpenMP

```
#pragma omp sections
{
    #pragma omp section
    section1();
    #pragma omp section
    section2();
    #pragma omp section
    section3();
}
```



```
#pragma omp for
for(omp_section0 =0; omp_section0 <= 2; omp_section0 ++ )
{
    switch( omp_section0)    {
        case 0 : section1(); break;
        case 1 : section2();  break;
        case 2 : section3();  break;
    }
}
```

- Chuyển về dạng chuẩn
- Chuyển về khối tương đương

Pha cuối

- Dịch khối OpenMP thành mã đa luồng
- Đơn giản: Thay thế khối bởi lời gọi hàm
- Phức tạp hơn:
 - Chuyển đoạn mã trong vùng song song thành hàm
 - Thêm các phép đồng bộ hóa qua các hàm thư viện
 - Dịch các khối song song, khối chia sẻ công việc...
- Thiết lập tính chất dữ liệu, vị trí trên vùng nhớ
 - Sử dụng vùng nhớ stack của luồng để lưu biến riêng
 - Thêm biến để thể hiện dữ liệu riêng, kết quả quy giản
 - Thêm lệnh để thực hiện firstprivate, lastprivate

Ví dụ chuyển mã khối song song thành hàm

```
/* Outlined function has an extra argument for passing addresses*/
static void __ompc_func_0(void **__ompc_args){
    int *_pp_b, *_pp_a, _p_c;

    /* need to dereference addresses to get shared variables a and b*/
    _pp_b=(int *)(*__ompc_args);
    _pp_a=(int *)(*(__ompc_args+1));
```

```
/*substitute accesses for all variables*/
do_something (*_pp_a,*_pp_b,_p_c);
}
```

```
int _ompc_main(void){
    int a,b,c;
    void *__ompc_argv[2];
```

```
/*wrap addresses of shared variables*/
*(__ompc_argv)=(void *)&b;
*(__ompc_argv+1)=(void *)&a;
```

```
/*OpenMP runtime call must pass the addresses of shared variables*/
_ompc_do_parallel(__ompc_func_0, __ompc_argv);
. . .
}
```

```
int main(void)
{
    int a,b,c;
    #pragma omp parallel private(c)
    do_something(a,b,c);
    return 0;
}
```

Các cách chia sẻ công việc

Number of loop iterations $n = 18$
Chunk size: $c=2$ if specified

t=3 Threads: 0 1 2

Legend:



schedule (static)



schedule(static,2)



schedule(dynamic,2)



schedule(guided,2)



Chia cố định công việc

```
static double a[1000];  
int i;  
#pragma omp for  
for (i=0;i<1000;i++)  
    a[i]=(double)i/2.0;
```



```
mytid = __ompc_get_thread_num(); /* get threadid */  
. . . . .  
/* invoke static scheduler */  
__ompc_static_init(mytid, mylower, myupper,  
                    mystride );  
  
/* execute loop body using assigned iteration space */  
for(myloci = mylower;  
    myloci<= myupper);myloci=myloci+1)  
{    a[myloci] = myloci * 2;  
}
```



```
__ompc_barrier(); /* Implicit barrier after worksharing */
```

Thanks!



Khái quát về chương trình dịch



- Pha đầu: đọc mã nguồn, kiểm tra lỗi và chuyển về dạng mã trung gian
- Pha giữa: phân tích và tối ưu chương trình, chuyển mã trung gian về dạng gần giống mã máy
- Pha cuối: sắp đặt vị trí cho dữ liệu chương trình trên bộ nhớ, sinh và tối ưu mã máy

