Parallel Computing (IV) Shared-Memory Programming model: openMP

Cheng-Hung Lin

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

```
#pragma omp parallel
```

Create as many threads as there are cores

```
#pragma omp parallel for
for(i=0; i<N; i++) {
   c[i] = a[i] + b[i];
}
```

#pragma omp parallel

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

int main(){
        omp_set_num_threads(16);

        #pragma omp parallel
        {
                printf("Hello world!\n");
        }


        return 0;
}
```

```
brucelin@brucelin-VirtualBox:~/OS/ch04$ gcc omp1.c -o omp1 -fopenmp
brucelin@brucelin-VirtualBox:~/OS/ch04$ ./omp1
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

#pragma omp parallel for

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

int main(){
        int i;
        omp_set_num_threads(16);

        #pragma omp parallel for
        for(i=0; i<16; i++){
                printf("%d ", i);
        }
        printf("\n");


        return 0;
}
```

```
brucelin@brucelin-VirtualBox:~/OS/ch04$ gcc omp2.c -o omp2 -fopenmp
brucelin@brucelin-VirtualBox:~/OS/ch04$ ./omp2
3 4 13 8 7 6 5 2 9 1 15 10 0 12 11 14
```

Omp 用法

- #pragma omp directive [clause]
  - Parallel
  - For

```
#pragma omp parallel for
for( int i = 0; i < 10; ++ i )
  Test( i );
```

```
#pragma omp parallel
{
  #pragma omp for
  for( int i = 0; i < 10; ++ i
)
   Test( i );
}
```

directive

directive	function
parallel	代表接下來的程式區塊將被平行化。
for	用在 for 迴圈之前，會將迴圈平行化處理。（註：迴圈的 index 只能是 int）
master	指定由主執行緒來執行接下來的程式。
ordered	指定接下來被程式，在被平行化的 for 迴圈將依序的執行。
atomic	這個指令的目的在於避免變數被同時修改而造成計算結果錯誤。
barrier	等待，直到所有的執行緒都執行到 barrier。用來同步化。

directive

directive	function
Sections	將接下來的 section 平行化處理。
Single	之後的程式將只會在一個執行緒執行，不會被平行化。
threadprivate	定義一個變數是一個線程私有
critical	強制接下來的程式區塊一次只會被一個執行緒執行。
flush	Specifies that all threads have the same view of memory for all shared objects.

clause

clause	**functions**
copyin	讓 threadprivate 的變數的值和主執行緒的值相同。
copyprivate	將不同執行緒中的變數共用。
default	設定平行化時對變數處理方式的預設值。
firstprivate	讓每個執行緒中, 都有一份變數的複本, 以免互相干擾;而起始值則會是開始平行化之前的變數值。

clause

clause	functions
if	判斷條件，可以用來決定是否要平行化。
lastprivate	讓每個執行緒中，都有一份變數的複本，以免互相干擾；而在所有平行化的執行緒都結束後，會把最後的值，寫回主執行緒。
nowait	忽略 barrier（等待）。
num_threads	設定平行化時執行緒的數量。

clause

clause	functions
ordered	使用於 for，可以在將迴圈平行化的時候，將程式中有標記 directive ordered 的部份依序執行。
private	定義變數為私有變數，讓每個執行緒中，都有一份變數的複本，以免互相干擾。
reduction	對各執行緒的變數，直行指定的運算元來合併寫回主執行緒。
schedule	設定 for 迴圈的平行化方法；有 dynamic、guided、runtime、static 四種方法。
shared	將變數設定為各執行緒共用。

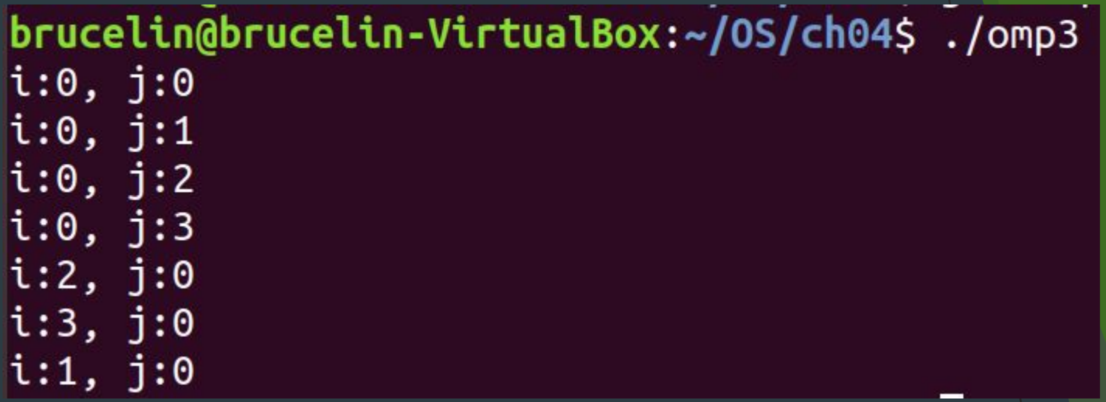
Find the error

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#define N 16
int main(){
	int i;
	int temp;
	int A[N], B[N], AA[N], BB[N];

	for(i=0; i<N; i++){
		A[i] = rand() % 256;
		B[i] = rand() % 256;
		AA[i] = A[i];
		BB[i] = B[i];
	}
	for(i=0; i<N; i++){
		temp = A[i];
		A[i] = B[i];
		B[i] = temp;
	}

	#pragma omp parallel for
	for(i=0; i<N; i++){
		temp = AA[i];
		AA[i] = BB[i];
		BB[i] = temp;
	}

	for(i=0; i<N; i++){
		if(A[i] != AA[i] || B[i]!=BB[i])
			break;
	}

	if(i==N)
		printf("Test pass!!!\n");
	else
		printf("Test failure\n");
	return 0;
}
```

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#define N 16
int main(){
        int i;
        int temp;
        int A[N], B[N], AA[N], BB[N];

        for(i=0; i<N; i++){
                A[i] = rand() % 256;
                B[i] = rand() % 256;
                AA[i] = A[i];
                BB[i] = B[i];
        }

        for(i=0; i<N; i++){
                temp = A[i];
                A[i] = B[i];
                B[i] = temp;
        }

        #pragma omp parallel for private(temp)
        for(i=0; i<N; i++){
                temp = AA[i];
                AA[i] = BB[i];
                BB[i] = temp;
        }

        for(i=0; i<N; i++){
                if(A[i] != AA[i] || B[i]!=BB[i])
                        break;
        }

        if(i==N)
                printf("Test pass!!!\n");
        else
                printf("Test failure\n");
        return 0;
}
```

Find the error




```
#include <stdio.h>
#include <omp.h>
#define N 4

int main(){
	int i, j;
	#pragma omp parallel for
	for(i=0; i<N; i++){
		for(j=0; j<N; j++){
			printf("i:%d, j:%d\n", i, j);
		}
	}
	return 0;
}
```

Solution

```
#include <stdio.h>
#include <omp.h>
#define N 4

int main(){
	int i, j;
	#pragma omp parallel for private( j)
	for(i=0; i<N; i++){
		for(j=0; j<N; j++){
			printf("i:%d, j:%d\n", i, j);
		}
	}
	return 0;
}
```

parallelize only the outer loop

```
i:1, j:0
i:1, j:1
i:1, j:2
i:1, j:3
i:3, j:0
i:3, j:1
i:3, j:2
i:3, j:3
i:0, j:0
i:0, j:1
i:0, j:2
i:0, j:3
i:2, j:0
i:2, j:1
i:2, j:2
i:2, j:3
```

平行內外迴圈

```
#include <stdio.h>
#include <omp.h>
#define N 4

int main(){
	int i, j;
	#pragma omp parallel for collapse(2)
	for(i=0; i<N; i++){
		for(j=0; j<N; j++){
			printf("i:%d, j:%d\n", i, j);
		}
	}
	return 0;
}
```

```
i:1, j:0
i:1, j:1
i:1, j:2
i:1, j:3
i:3, j:0
i:3, j:1
i:3, j:2
i:3, j:3
i:0, j:0
i:0, j:1
i:0, j:2
i:0, j:3
i:2, j:0
i:2, j:1
i:2, j:2
i:2, j:3
```

Find the bug

```c
#include <stdio.h>
#include <omp.h>
#define N 1000

int main(){
	int i, sum = 0;

	#pragma omp parallel for
	for(i=1; i<=N; i++){
		sum += i;
	}

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

	return 0;
}
```

```
brucelin@brucelin-VirtualBox:~/OS/ch04$ ./race
55833
brucelin@brucelin-VirtualBox:~/OS/ch04$ ./race
257503
brucelin@brucelin-VirtualBox:~/OS/ch04$ ./race
124327
brucelin@brucelin-VirtualBox:~/OS/ch04$ ./race
461431
```

Solution

```
#include <stdio.h>
#include <omp.h>
#define N 1000

int main(){
    int i, sum = 0;

    #pragma omp parallel for
    for(i=1; i<=N; i++){
        #pragma omp atomic
        sum += i;
    }

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

    return 0;
}
```

Solution II

```
#include <stdio.h>
#include <omp.h>
#define N 1000

int main(){
      int i, sum = 0;

      #pragma omp parallel for reduction(+:sum)
      for(i=1; i<=N; i++){
            sum += i;
      }

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

      return 0;
}
```

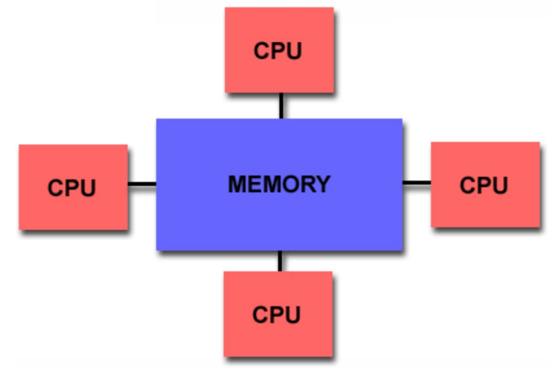
Compare the performance of atomic and reduction

- Please compare the performance of atomic add and reduction Add.
- int a[1000000]

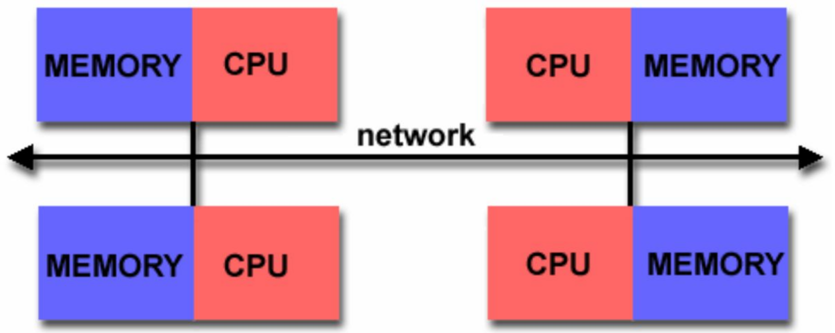
```
for(i=0; i<1000000; i++)
    a[i] = i;

for(i=0; i<1000000; i++)
    sum = sum + a[i];
```

記憶體架構

- 記憶體架構分成兩種共享式記憶體與分散式記憶體
- 共享式記憶體:記憶體是由多個CPU來共同存取的，所以通常CPU和記憶體會放在同一台電腦裡，優點是可以很快存取記憶體，缺點是擴充性不佳。
- 分散式記憶體:利用網路串連多台機器，優點是可以CPU就可以很快存取到記憶體，缺點是機器間的資料交換不易



共享式記憶體



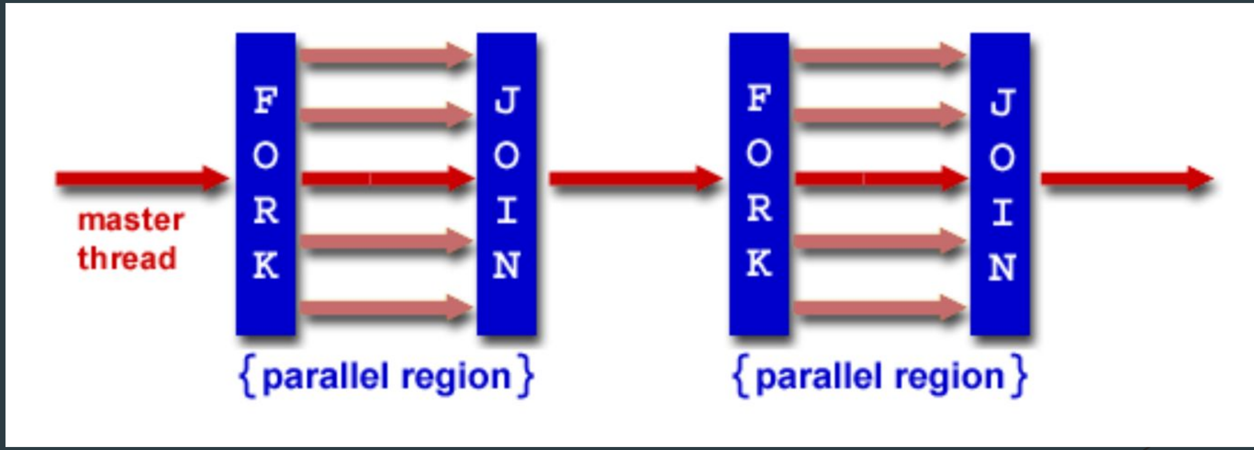
分散式記憶體

OpenMP Outline

- Parallel Region Construct
  - Parallel Directive
- Working-Sharing Construct
  - DO/for Directive
  - SECTIONS Directive
  - SINGLE Directive
- Synchronization Construct
- Date Scope Attribute Clauses
- Run-Time Library Routines

What is OpenMp?

- OpenMp(Open Multi-Processing)是一種利用thread進行平行化處理，進而加快程式處理的速度的函式庫，可跨平台使用。
- 程式語言:C,C++,Fortran
- OpenMp會在進入parallel region將master thread複製好幾份放到記憶體內同時執行(從parallel region開始的地方執行)，最後離開parallel region的時候會等待所有thread執行完畢後再繼續執行master thread的程式



OpenMp教學

- 首先設定預設的thread數量，當程式碼中沒有指定thread數量時則會使用預設或是以Logical CPU當作預設值，這邊先預設thread數量為2

$ export OMP_NUM_THREADS=2

- OpenMp使用語法 : #pragma omp <directive> [clause[[,] clause] …]
- OpenMp基本Function
  - omp_get_thread_num() 取得目前thread的id
  - omp_set_num_threads(n) 在程式中設定thread的數量
  - omp_get _num_threads()取得使用中thread的數量
  - omp_set_schedule()設定schedule的方法

Example1:利用parallel進行程式的平行化

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

int main(int argc,char* argv[]){
   #pragma omp parallel
   {
   printf("Thread %d Hello World \n",omp_get_thread_num());
   }
}
```

因為設定兩個thread所以出現0和1，thread 0 為 master thread

```
Thread 1 Hello World
Thread 0 Hello World
```

- 編譯與執行:-fopenmp 用來載入libgomp這個動態函式庫

```
$ gcc -fopenmp example.c
$ ./a.out
```

Example2:for迴圈平行化

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

int main(){
  #pragma omp parallel
  {
    #pragma omp for
    for(int i=0;i<5;i++)
    {
      printf("thread %d : loop %d\n",omp_get_thread_num(),i);
    }
  }
  return 0;
}
```

因為是平行化處理所以跑
出來的結果不一定照順序

```
thread 1 : loop 3
thread 1 : loop 4
thread 0 : loop 0
thread 0 : loop 1
thread 0 : loop 2
```

Example3:sections
平行化每個section個別平行運算

```
#include<omp.h>
#include<stdio.h>
#include<stdlib.h>
void Test(int);int main()
{
 #pragma omp parallel sections
  {

    #pragma omp section
    {
      for(int i=0;i<100000;++i)
      {}
      printf("thread %d , first section\n",omp_get_thread_num());
    }
    #pragma omp section
    {
      printf("thread %d , second section\n",omp_get_thread_num());
    }
    #pragma omp section
    {
      printf("thread %d , third section\n",omp_get_thread_num());
    }
    #pragma omp section
    {
        printf("thread %d , fourth section\n",omp_get_thread_num());
    }
  }
}
```

因為第1個section跑比較久所以最後才顯示出來

```
thread 1 , second section
thread 1 , third section
thread 1 , fourth section
thread 0 , first section
```

Example4: single 只跑一次, master 只讓master thread跑

```
#include <stdio.h>
#include <omp.h>
int main(){
	int i, j;
	#pragma omp parallel num_threads(4)
	{
		#pragma omp for
		for(i=0; i<4; i++){
			for(j=0; j<100000; j++){}
			printf("thread %d:%d\n", omp_get_thread_num(), i);
		}
		printf("thread %d: four times\n", omp_get_thread_num());

		#pragma omp single			{
		printf("thread %d, one times\n", omp_get_thread_num());		}
		#pragma omp master			{
		printf("thread %d, master\n", omp_get_thread_num());			}
		}
	return 0;
}
```

被single包含的程式只執行一次，被master包含的程式只會讓master執行

```
thread 3:3
thread 0:0
thread 1:1
thread 2:2
thread 2: four times
thread 2, one times
thread 0: four times
thread 3: four times
thread 1: four times
thread 0, master
```

Example5: private

- 被private包含的變數再跑平行運算時，每個thread會自己複製一份不會共用同一份變數

```
#include<omp.h>
#include<stdio.h>
#include<stdlib.h>
int main(){
        int i,j;
        #pragma omp parallel for
                for(i=0;i<5;i++)
                        for(j=0;j<5;j++){
                            printf("thread %d : %d loop\n",omp_get_thread_num(),i*5+j);
                        }
        printf("---------------------------------------------------------\n");
        #pragma omp parallel for private(j)
                for(i=0;i<5;i++)
                        for(j=0;j<5;j++){
                            printf("thread %d : %d loop\n",omp_get_thread_num(),i*5+j);
                        }
}
```

左邊因為共用變數j而導致迴圈沒跑滿25圈，右邊因為把j複製多份所以沒這問題

```
thread 2 : 10 loop
thread 2 : 11 loop
thread 2 : 12 loop
thread 2 : 13 loop
thread 2 : 14 loop
thread 0 : 0 loop
thread 0 : 1 loop
thread 0 : 2 loop
thread 0 : 3 loop
thread 0 : 4 loop
thread 3 : 15 loop
thread 1 : 5 loop
thread 4 : 20 loop
thread 4 : 21 loop
thread 4 : 22 loop
thread 4 : 23 loop
thread 4 : 24 loop
```

```
thread 0 : 0 loop
thread 0 : 1 loop
thread 0 : 2 loop
thread 0 : 3 loop
thread 0 : 4 loop
thread 4 : 20 loop
thread 4 : 21 loop
thread 4 : 22 loop
thread 4 : 23 loop
thread 4 : 24 loop
thread 2 : 10 loop
thread 2 : 11 loop
thread 2 : 12 loop
thread 2 : 13 loop
thread 2 : 14 loop
thread 3 : 15 loop
thread 3 : 16 loop
thread 3 : 17 loop
thread 3 : 18 loop
thread 3 : 19 loop
thread 1 : 5 loop
thread 1 : 6 loop
thread 1 : 7 loop
thread 1 : 8 loop
thread 1 : 9 loop
```

Example6:firstprivate 和lastprivate

- firstprivate 和private差不多只是在複製時也會複製初始值，lastprivate則是會在最後將複製出來的值丟回到本尊

```
#include <stdio.h>
#include <omp.h>
int main(){
	int i;
	int count = -5;
	#pragma omp parallel for firstprivate(count) lastprivate(count) num_threads(2)
for(i=0; i<5; i++){
		count++;
		printf("thread %d : count %d\n",omp_get_thread_num(),count);
	}

	printf("Final count: %d\n", count);
	return 0;
}
```

```
thread 0 : count -4
thread 0 : count -3
thread 0 : count -2
thread 1 : count -4
thread 1 : count -3
Final count: -3
```

Example7: atomic

- atomic是為了保證變數在做計算時不被其他thread跟改到而導致計算出的東西有錯誤(race condition)
- 如果沒有加atomic跑出來的數字會是低於5,000,000, 加了atomic可以保證變數做運算時不會被其他thread給更改到數字
- 另外j必須設成private

```
#include <stdio.h>
#include <omp.h>
int main(){
	int sum = 0;
	int i, j;
	#pragma omp parallel for private(j)
	for(i=0; i<1000; i++){
		for(j=0; j<5000; j++){
			#pragma omp atomic
			sum += 1;
		}
	}
	printf("sum: %d\n", sum);
	return 0;
}
```

Example8:reduction

- Reduction目的和上面很像, 他是將每個sum依照thread各別複製一份出來後最後join時將所有sum相加就不會導致錯誤發生
- 但是只可以接受+、*、-、&...等運算符號

```
#include<omp.h>
#include<stdio.h>
#include<stdlib.h>
int main(){
    int sum = 0;
    double start = omp_get_wtime();
    #pragma omp parallel for reduction(+:sum) private(j)
    for(int i=0;i<1000;i++){
        for(int j=0;j<5000;j++){
            sum+=1;
        }
    }
    printf("sum %d : time %4g second\n",sum,omp_get_wtime()-start);
}
```

如果沒有加reduction出來的數字會因為race condition而有錯誤

```
sum 3249186 : time 0.0313631 second
sum 5000000 : time 0.00826513 second
```

Please compare atomic add and reduction add

- 請比較一下example 7 與 example 8的效能

Example9:schedule

- schedule分成4種static, dynamic, guided, runtime, auto
- static:將迴圈每n個分一組，依照thread順序輪流給每個thread執行，當跑過一輪後再從第一個thread開始輪流跑

```
schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****
```

- dynamic:將迴圈每n個分一組，隨機分配給thread執行

```
schedule(dynamic, 4):
            ****                    ****                    ****
****            ****    ****            ****        ****
    ****            ****    ****            ****        ****
        ****                    ****            ****
```

Example9:schedule

- guided:剛開始會依照thread數量下去切，如果迴圈有64個，thread有4個，那一開始第一組的數量則是64/4=16，依序往後每組數量會遞減，收縮到n個一組，如剩下的數量不夠n個則剩下的全部變成1組

```
schedule(guided, 4):
                                     *******
                ************                     ****    ****
                            *********
****************                            *****    ****    ***
```

- runtime:先不指定方法等到要執行時會依照系統變數OMP_SCHEDULE或omp_set_schedule做設定

schedule(runtime):

- auto:由系統幫忙處理

schedule(auto):

schedule(static,4)範例

- 將迴圈每4個一組下去跑，每次跑的thread都會照順序，thread0先跑在換thread1，依此類推

```c
#include<omp.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(){
    #pragma omp parallel for schedule(static,4) num_threads(2) ordered
    for(int i=0;i<16;i++){
        #pragma omp ordered
        printf("Thread %d has completed iteration %d\n",omp_get_thread_num(),i);
    }
    printf("All done!\n");
    return 0;
}
```

```
Thread 0 has completed iteration 0
Thread 0 has completed iteration 1
Thread 0 has completed iteration 2
Thread 0 has completed iteration 3
Thread 1 has completed iteration 4
Thread 1 has completed iteration 5
Thread 1 has completed iteration 6
Thread 1 has completed iteration 7
Thread 0 has completed iteration 8
Thread 0 has completed iteration 9
Thread 0 has completed iteration 10
Thread 0 has completed iteration 11
Thread 1 has completed iteration 12
Thread 1 has completed iteration 13
Thread 1 has completed iteration 14
Thread 1 has completed iteration 15
All done!
```

Static schedule example (I)

- 總共有8個threads
- Total 16的iterations平分給8個threads

```
#include <stdio.h>
#include <omp.h>
int main(){
	int i;
	#pragma omp parallel
	{
		#pragma omp for schedule(static)
		for(i=0; i<16; i++){
			printf("Thread %d: loop %d\n", omp_get_thread_num(), i);
		}
	}
	return 0;
}
```

```
Thread 7: loop 14
Thread 7: loop 15
Thread 5: loop 10
Thread 0: loop 0
Thread 0: loop 1
Thread 2: loop 4
Thread 2: loop 5
Thread 1: loop 2
Thread 1: loop 3
Thread 5: loop 11
Thread 4: loop 8
Thread 4: loop 9
Thread 6: loop 12
Thread 6: loop 13
Thread 3: loop 6
Thread 3: loop 7
```

Static schedule example (II)

- 每個thread負責4個iterations
- Total 16的iterations平分給4個threads

```
#include <stdio.h>
#include <omp.h>
int main(){
	int i;
	#pragma omp parallel
	{
		#pragma omp for schedule(static, 4)
		for(i=0; i<16; i++){
			printf("Thread %d: loop %d\n", omp_get_thread_num(), i);
		}
	}
	return 0;
}
```

```
Thread 3: loop 12
Thread 3: loop 13
Thread 3: loop 14
Thread 3: loop 15
Thread 1: loop 4
Thread 1: loop 5
Thread 1: loop 6
Thread 1: loop 7
Thread 0: loop 0
Thread 0: loop 1
Thread 0: loop 2
Thread 0: loop 3
Thread 2: loop 8
Thread 2: loop 9
Thread 2: loop 10
Thread 2: loop 11
```

Dynamic schedule example (I)

- 動態分派iterations 給沒事的thread

```
#include <stdio.h>
#include <omp.h>
int main(){
	int i;
	#pragma omp parallel
	{
		#pragma omp for schedule(dynamic)
		for(i=0; i<16; i++){
			printf("Thread %d: loop %d\n", omp_get_thread_num(), i);
		}
	}
	return 0;
}
```

```
Thread 2: loop 0
Thread 2: loop 8
Thread 2: loop 9
Thread 2: loop 10
Thread 2: loop 11
Thread 2: loop 12
Thread 2: loop 13
Thread 2: loop 14
Thread 2: loop 15
Thread 6: loop 1
Thread 5: loop 2
Thread 1: loop 5
Thread 7: loop 3
Thread 0: loop 4
Thread 3: loop 6
Thread 4: loop 7
```

Dynamic schedule example (II)

- 動態分派iterations 給沒事的thread，每個thread負責4個iterations.

```
#include <stdio.h>
#include <omp.h>
int main(){
	int i;
	#pragma omp parallel
	{
		#pragma omp for schedule(dynamic, 4)
		for(i=0; i<16; i++){
			printf("Thread %d: loop %d\n", omp_get_thread_num(), i);
		}
	}
	return 0;
}
```

```
Thread 3: loop 0
Thread 3: loop 1
Thread 3: loop 2
Thread 3: loop 3
Thread 7: loop 8
Thread 7: loop 9
Thread 7: loop 10
Thread 7: loop 11
Thread 1: loop 4
Thread 1: loop 5
Thread 1: loop 6
Thread 1: loop 7
Thread 5: loop 12
Thread 5: loop 13
Thread 5: loop 14
Thread 5: loop 15
```

Guided example

- guided 的 chunk 切割方法和 static、dynamic 不一樣;他會以「遞減」的數目，來分割出 chunk。而 chunk 的分配方式，則是和 dynamic 一樣是動態的分配。而遞減的方式，大約會以指數的方式遞減到指定的 chunk_size。

```
#include <stdio.h>
#include <omp.h>
int main(){
	int i;
	#pragma omp parallel  num_threads(8)
	{
		#pragma omp for schedule(guided)
		for(i=0; i<64; i++){
			printf("Thread %d: loop %d\n", omp_get_thread_num(), i);
		}
	}
	return 0;
}
```

```
Thread 1: loop 0
Thread 1: loop 1
Thread 1: loop 2
Thread 0: loop 22
Thread 0: loop 23
Thread 0: loop 24
Thread 0: loop 25
Thread 0: loop 26
Thread 0: loop 27
Thread 0: loop 44
Thread 0: loop 45
Thread 0: loop 46
Thread 0: loop 47
Thread 0: loop 48
Thread 0: loop 49
Thread 0: loop 50
Thread 0: loop 51
Thread 0: loop 52
Thread 0: loop 53
Thread 0: loop 54
Thread 0: loop 55
Thread 0: loop 56
Thread 0: loop 57
Thread 0: loop 58
Thread 0: loop 59
Thread 0: loop 60
Thread 0: loop 61
Thread 0: loop 62
Thread 0: loop 63
Thread 3: loop 15
Thread 3: loop 16
Thread 3: loop 17
Thread 3: loop 18
Thread 3: loop 19
Thread 3: loop 20
Thread 3: loop 21
Thread 6: loop 33
Thread 6: loop 34
Thread 6: loop 35
Thread 6: loop 36
Thread 1: loop 3
Thread 1: loop 4
Thread 1: loop 5
Thread 1: loop 6
Thread 1: loop 7
Thread 7: loop 8
Thread 7: loop 9
Thread 7: loop 10
Thread 4: loop 28
Thread 4: loop 29
Thread 4: loop 30
Thread 4: loop 31
Thread 4: loop 32
Thread 7: loop 11
Thread 7: loop 12
Thread 7: loop 13
Thread 7: loop 14
Thread 2: loop 41
Thread 2: loop 42
Thread 2: loop 43
Thread 5: loop 37
Thread 5: loop 38
Thread 5: loop 39
Thread 5: loop 40
```

Parallel Region Constructs --- Parallel Directive

- Limitations:
  - A parallel region must be a structured block that does not span multiple routines or code files
  - It is illegal to branch (goto) into or out of a parallel region, but you could call other functions within a parallel region

Nested Parallel Region

- check if nested parallel regions are enabled
  - omp_get_nested ()
- To disable/enable nested parallel regions:
  - omp_set_nested (bool)
  - Setting of the OMP_NESTED environment variable
- If nested is not supported or enabled:
  - Only one thread is created for the nested parallel region code

```
// A total of 6 “hello world!” is printed
#pragma omp parallel num_threads(2)
{
  #pragma omp parallel num_threads(3)
  {
          printf(“hello world!”);
  }
}
```

Example 9: nested parallel region

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

int main(){

	if(!omp_get_nested()){
		omp_set_nested(1);
	}
	#pragma omp parallel num_threads(2)
	{
	    #pragma omp parallel num_threads(3)
	    {
		printf("Thread %d: hello world!\n", omp_get_thread_num());
	    }
	}
  return 0;
}
```

```
Thread 2: hello world!
Thread 0: hello world!
Thread 2: hello world!
Thread 0: hello world!
Thread 1: hello world!
Thread 1: hello world!
```

OpenMP Outline

- Synchronization Construct
- Date Scope Attribute Clauses
- Run-Time Library Routines

Synchronization Constructs

- For synchronization purpose among threads

```
#pragma omp [synchronization_directive] [clause ......]
structured_block
```

- Synchronization Directives
  - master: only executed by the master thread
    - No implicit barrier at the end
    - More efficient than SINGLE directive
  - critical: must be executed by only one thread at a time
    - Threads will be blocked until the critical section is clear
  - barrier: blocked until all threads reach the call
  - atomic: memory location must be updated atomically provide a mini-critical section

Example 10: critical

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(){
	int sum = 0;
	int i, j;
	double start;

	start = omp_get_wtime();
	#pragma omp parallel for private(j)
	for(i=0; i<10000; i++){
		for(j=0; j<50000; j++){
			#pragma omp critical
			{
			sum += 1;
			}
		}
	}

	printf("reduction sum: %d: time %4g second\n", sum, omp_get_wtime()-start);
	return 0;
}
```

Compare reduction, atomic, and critical

- Compare the results of example 7, 8, and 10

LOCK OpenMP Routine

- void omp_init_lock(omp_lock_t *lock)
  - Initializes a lock associated with the lock variable
- void omp_destroy_lock(omp_lock_t *lock)
  - Disassociates the given lock variable from any locks
- void omp_set_lock(omp_lock_t *lock)
  - Force the thread to wait until the specified lock is available
- void omp_unset_lock(omp_lock_t *lock)
  - Releases the lock from the executing subroutine
- int omp_test_lock(omp_lock_t *lock)
  - Attempts to set a lock, but does NOT block if unavailable

Example 11: lock vs critical

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(){
	int sum = 0;
	omp_lock_t lock;
	omp_init_lock(&lock);

	#pragma omp parallel
	{
			omp_set_lock(&lock);
			sum += 1;
			omp_unset_lock(&lock);

	}
	omp_destroy_lock(&lock);
	printf("reduction sum: %d\n", sum);
	return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(){
	int sum = 0;

	#pragma omp parallel
	{
		#pragma omp critical
		sum += 1;
	}
	printf("critcal sum: %d\n", sum);

	return 0;
}
```

Example & Comparison

- Advantage of using critical over lock:
  - no need to declare, initialize and destroy a lock
  - you always have explicit control over where your critical section ends
  - Less overhead with compiler assist

```
#pragma omp parallel
	{
		#pragma omp critical
		sum += 1;
	}
```

```
omp_lock_t lock;
omp_init_lock(&lock);

#pragma omp parallel
{
		omp_set_lock(&lock);
		sum += 1;
		omp_unset_lock(&lock);

}
omp_destroy_lock(&lock);
```

OpenMP Outline

- Parallel Region Construct
  - Parallel Directive
- Working-Sharing Construct
  - DO/for Directive
  - SECTIONS Directive
  - SINGLE Directive
- Synchronization Construct
- Date Scope Attribute Clauses
- Run-Time Library Routines

OpenMP Date Scope

- OpenMP is based on shared memory programming model
- Most variables are shared by default
- Global shared variables:
  - File scope variables, static
- Private non-shared variables:
  - Loop index variables 迴圈索引的變數
  - Stack variables in subroutines called from parallel regions
- Data scope can be explicitly defined by clauses…
  - PRIVATE , SHARED, FIRSTPRIVATE, LASTPRIVATE
  - DEFAULT, REDUCTION, COPYIN

Date Scope Attribute Clauses

- PRIVATE (var_list):
  - Declares variables in its list to be private to each thread; variable value is NOT initialized & will not be maintained outside the parallel region
- SHARED (var_list):
  - Declares variables in its list to be shared among all threads
  - By default, all variables in the work sharing region are shared except the loop iteration counter.
- FIRSTPRIVATE (var_list):
  - Same as PRIVATE clause, but the variable is INITIALIZED according to the value of their original objects prior to entry into the parallel region
- LASTPRIVATE (var_list)
  - Same as PRIVATE clause, with a copy from the LAST loop iteration or section to the original variable object

Examples

- firstprivate (var_list)

```
int var1 = 10;
#pragma omp parallel firstprivate (var1)
{
        printf("var1:%d" var1);
}
```

- lastprivate (var_list)

```
int var1 = 10;
#pragma omp parallel lastprivate (var1) num_thread(10)
{
        int id = omp_get_thread_num();
        sleep(id);
        var1=id;
}
printf("var1:%d", var1);
```

Date Scope Attribute Clauses

- DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
  - Allows the user to specify a default scope for ALL variables in the parallel region
- COPYIN (var_list)
  - Assigning the same variable value based on the instance from the master thread
- COPYPRIVATE (var_list)
  - Broadcast values acquired by a single thread directly to all instances in the other thread
  - Associated with the SINGLE directive
- REDUCTION (operator: var_list)
  - **A private copy** for each list variable is created for each thread
  - Performs a reduction on all variable instances
  - Write the final result to the **global shared copy**

Reduction Clause Example

- Reduction operators: +, *, &, |, ^, &&, ||

```
#include <omp.h>
main () {
   int i, n, chunk, a[100], b[100], result;
   n = 10; chunk = 2; result = 0;
   for (i=0; i < n; i++) a[i] = b[i] = I;

   #pragma omp parallel for default(shared) private(i) \
                      schedule(static,chunk) reduction(+:result)
    {
        for (i=0; i < n; i++) result = result + (a[i] * b[i]);
    }
    printf("Final result= %f\n",result);
}
```

OpenMP Clause Summary

- Synchronization Directives DO NOT accept clauses

Clause	Directive			
	PARALLEL	DO/for	SECTIONS	SINGLE
IF	V			
PRIVATE	V	V	V	V
SHARED	V	V		
DEFAULT	V			
FIRSTPRIVATE	V	V	V	V
LASTPRIVATE		V	V	
REDUCTION	V	V	V	
COPYIN	V			
COPYPRIVATE				V
SCHEDULE		V		
ORDERED		V		
NOWAIT		V	V	