Introduction to OpenMP*

Francesc Giné, Josep L. Lérida
(Group of Distributed Computing), University of Lleida
Email: francesc.gine@udl.cat, joseplluis.lerida@udl.cat

ESCOLA POLITÈCNICA SUPERIOR
UNIVERSITAT DE LLEIDA

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Memory Model
- Data environment

OpenMP* Overview:

`C$OMP FLUSH`

`#pragma omp critical`

`C$OMP THREADPRIVATE(/ABC/)`

`CALL OMP_SET_NUM_THREADS(10)`

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice

* The name "OpenMP" is the property of the OpenMP Architecture Review Board.

OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.
- A compiler directive in C is called *pragma*. A pragma has this syntax:

  #pragma omp *construct [clause [clause]…]*

  – Example

  #pragma omp parallel num_threads(4)
- Function prototypes and types in the file:

  #include <omp.h>
- Most OpenMP constructs apply to a "structured block".
  – Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  – It's OK to have an exit() within the structured block.

Exercise 1, Part A: Hello world

Verify that your environment works

- Write a program that prints “hello world”.

```
int main()
{




    int ID = 0;

    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);

}
```

Exercise 1, Part B: Hello world

Verify that your OpenMP environment works

- Write a multithreaded program that prints "hello world".

```
#include "omp.h"
int main()
{
 #pragma omp parallel
 {

    int ID = 0;

    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
 }
}
```

Switches for compiling and linking

```
gcc -fopenmp   hello.c –o hello
```

Exercise 1: Solution

A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

```
#include “omp.h”
void main()
{

#pragma omp parallel
 {

    int ID = omp_get_thread_num();
    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
  }
}
```

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Sample Output:

hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

world(2)

Questions:

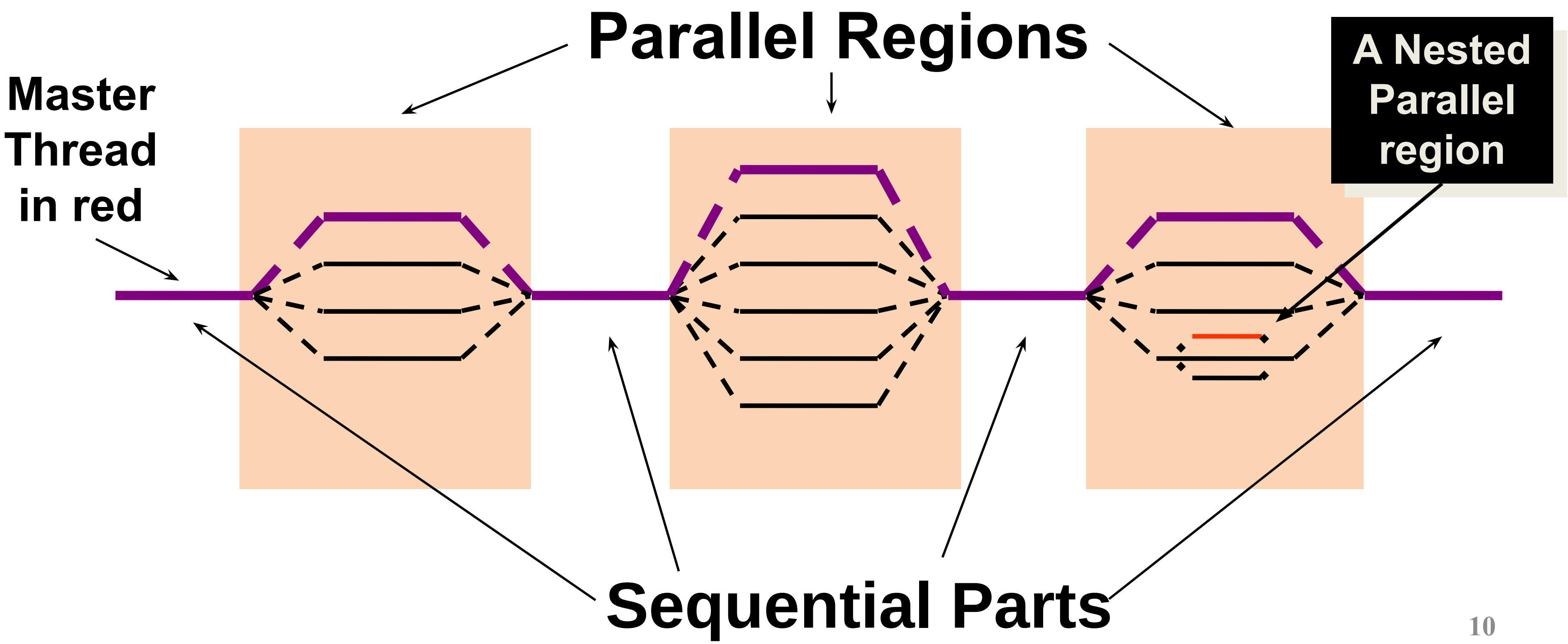
- How many threads are created?
- Are the results deterministics?

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment

OpenMP Programming Model:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.



Thread Creation: Parallel Regions

- You create threads in OpenMP* with the parallel construct.
- For example, to create a 4 thread Parallel region:

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

Runtime function to request a certain number of threads

Runtime function returning a thread ID

- **Each thread calls pooh(ID,A) for `ID` = 0 to 3**

* The name “OpenMP” is the property of the OpenMP Architecture Review Board

Thread Creation: Parallel Regions

- You create threads in OpenMP* with the parallel construct.
- For example, To create a 4 thread Parallel region:

```
double A[1000];

#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

Each thread executes a copy of the code within the structured block

clause to request a certain number of threads

Runtime function returning a thread ID

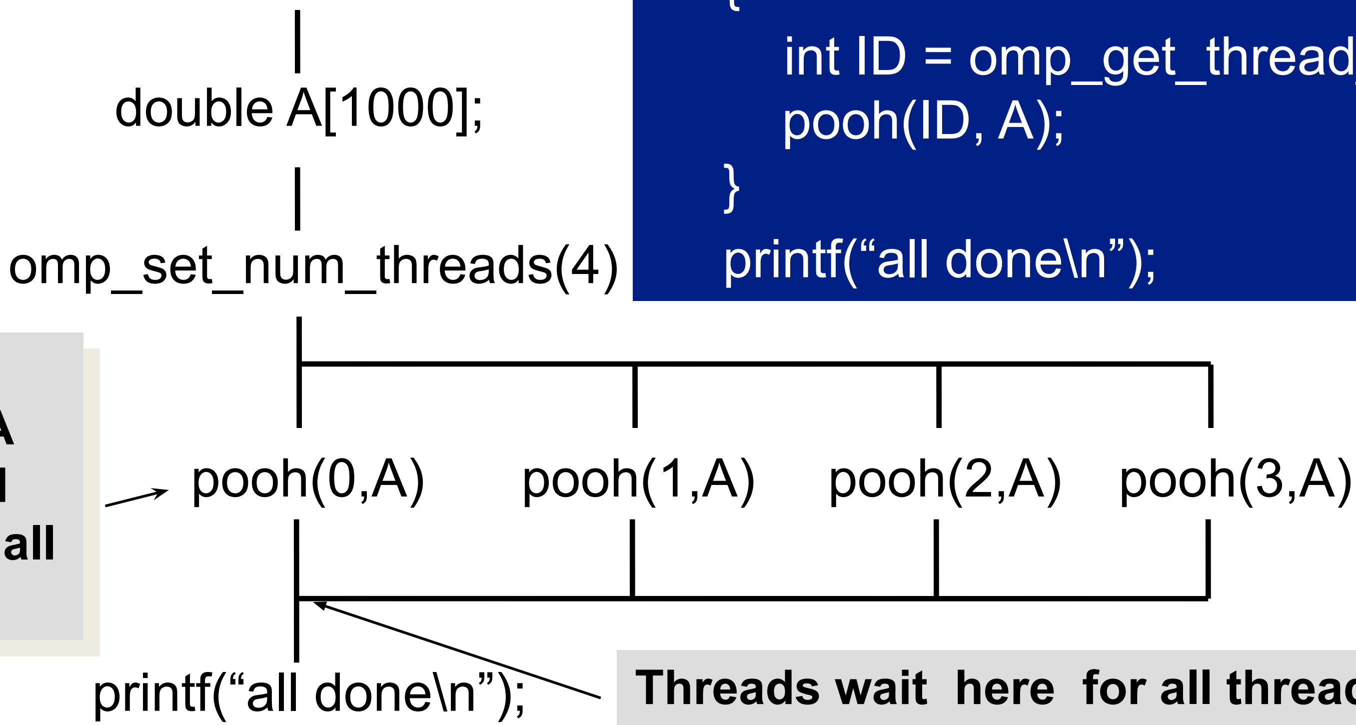
- **Each thread calls pooh(ID,A) for `ID` = 0 to 3**

* The name "OpenMP" is the property of the OpenMP Architecture Review Board

Thread Creation: Parallel Regions example

- Each thread executes the same code redundantly.

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



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

* The name "OpenMP" is the property of the OpenMP Architecture Review Board

Internal Control Variables (ICV)

- An OpenMP implementation must act as if there are internal control variables that control the behavior of an OpenMP program. These ICVs store information such as the number of threads to use for future parallel regions, the schedule to use for worksharing loops or whether nested parallelism is enabled or not.
- Some of them are:
  - *dyn-var*: controls whether dynamic adjustment of the number of threads is enabled for parallel regions.
  - *nest-var*: controls whether nested parallelism is enabled for parallel regions.
  - *nthreads-var*: controls the number of threads requested for parallel regions.
  - *thread-limit-var*: controls the maximum number of threads to use in the whole OpenMP program.
  - *Max-active-levels*: controls the maximum number of nested active parallel regions.

Internal Control Variables (ICV)

- The following Table shows the method for modifying and retrieving the values of ICVs through OpenMP API routines

Variable	Ways to Modify	Ways to Retrieve
dyn-var	omp_set_dynamic()	omp_get_dynamic()
nest-var	omp_set_nested()	omp_get_nested()
nthreads-var	omp_set_num_threads()	omp_get_max_threads()
thread-limit-var	thread_limit clause	omp_get_thread_limit()
run-sched-var	omp_set_schedule()	omp_get_schedule()
Max-active-levels-var	omp_set_max_active_levels()	Omp_get_max_active_levels()

How do threads interact?

- OpenMP is a multi-threading, shared address model.
  - Threads communicate with each other through ordinary reads and writes to shared variables.
- Every thread has its own execution context:
  - an address space containing all of the variables the thread may access.
- Variables may either be shared or private:
  - A Shared variable has the same address space in the execution context of every thread.
  - A private variable has a different address in the execution context of every thread.
- To coordinate access to these shared variable across multiple threads:
  - Explicit coordination between these multiple threads. It means synchronization.

Data Scoping

- **Any variable that existed before** a parallel region still exist inside and is **shared** by default between all threads.
- Each thread can have its own **private variables**. There are three ways to make private variables:
  - A variable that exists before entry to a parallel construct can be privatized by **a PRIVATE clause** to the OMP PARALLEL directive.
  - **The index variable** of a worksharing loop is automatically made private.
  - **Local variables in a subrutine** called from a parallel region are private to each calling thread.

Data Scoping

Example 1:A simple loop that adds two arrays

example1.cpp

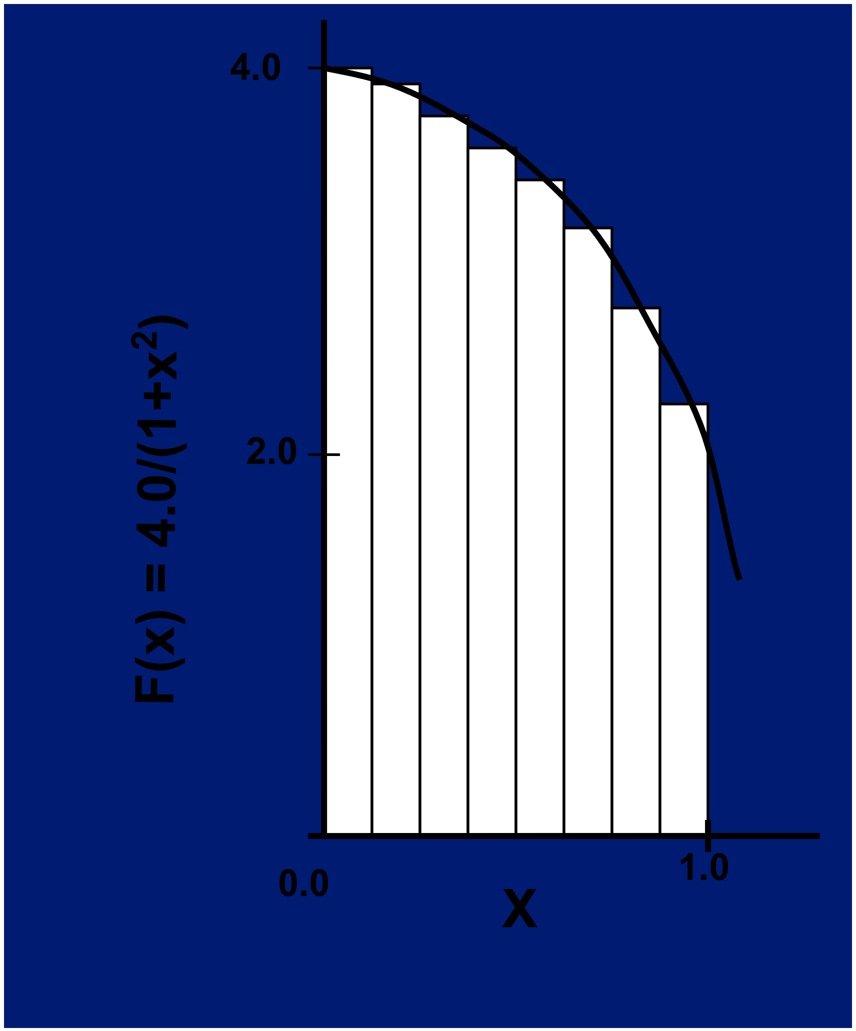
```
#include <omp.h>
#include <stdio.h>
#define  N 100000
void main()
{

  double start_time,run_time;
  int i;
  int a[N],b[N],c[N];
  start_time=omp_get_wtime();

  #pragma omp parallel num_threads(4)
  {
          int bstart, bend, blen, numth, tid, i;
          numth=omp_get_num_threads();
          tid=omp_get_thread_num();
          blen=N/numth;

          if(tid<N%numth) {
                  blen++;
                  bstart=blen*tid; }
          else  bstart=blen*tid+N%numth;


          bend=bstart+blen-1;
          for(i=bstart;i<=bend;i++)
          {
                   b[i]=i;
                  c[i]=i;
           }
          for(i=bstart;i<=bend;i++) a[i]=b[i]+c[i];
  }
  run_time=omp_get_wtime()-start_time;

  printf("Execution Time=%lf\n",run_time);
  printf("Value of Dynamic %d\n",omp_get_dynamic());
}
```

Shared variables

Private variables

Exercises 2 to 4:

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Exercise 2: Serial PI Program

```
static long num_steps = 100000;
double step;
void main ()
{     int i;   double x, pi, sum = 0.0;

      step = 1.0/(double) num_steps;

      for (i=0;i< num_steps; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
      }
      pi = step * sum;
}
```

Exercise 2

- Create a parallel version of the pi program using a parallel construct.
- Pay close attention to shared versus private variables.
- Define a program that compute the PI value with different number of threads. Modify the number of threads in computational time.
- In addition to a parallel construct, you will need the runtime library routines
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();
  - omp_set_num_threads(j);

Returns Number of threads in the team

Thread ID or rank

Time in Seconds since a fixed point in the past

Set the number of threads in j

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Memory Model
- Data environment

Synchronization

Synchronization is used to impose order constraints and to protect access to shared data

- High level synchronization:
    - critical
    - atomic
    - barrier
    - ordered
- Low level synchronization
    - flush
    - locks (both simple and nested)

Discussed later

Synchronization: critical

- Mutual exclusion: Only one thread at a time can enter a critical region.

```
float res;
#pragma omp parallel
{     float B;   int i, id, nthrds;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
       for(i=id;i<niters;i+nthrds){
       B =  big_job(i);
#pragma omp critical
                consume (B, res);
      }
}
```

Threads wait their turn – only one at a time calls consume()

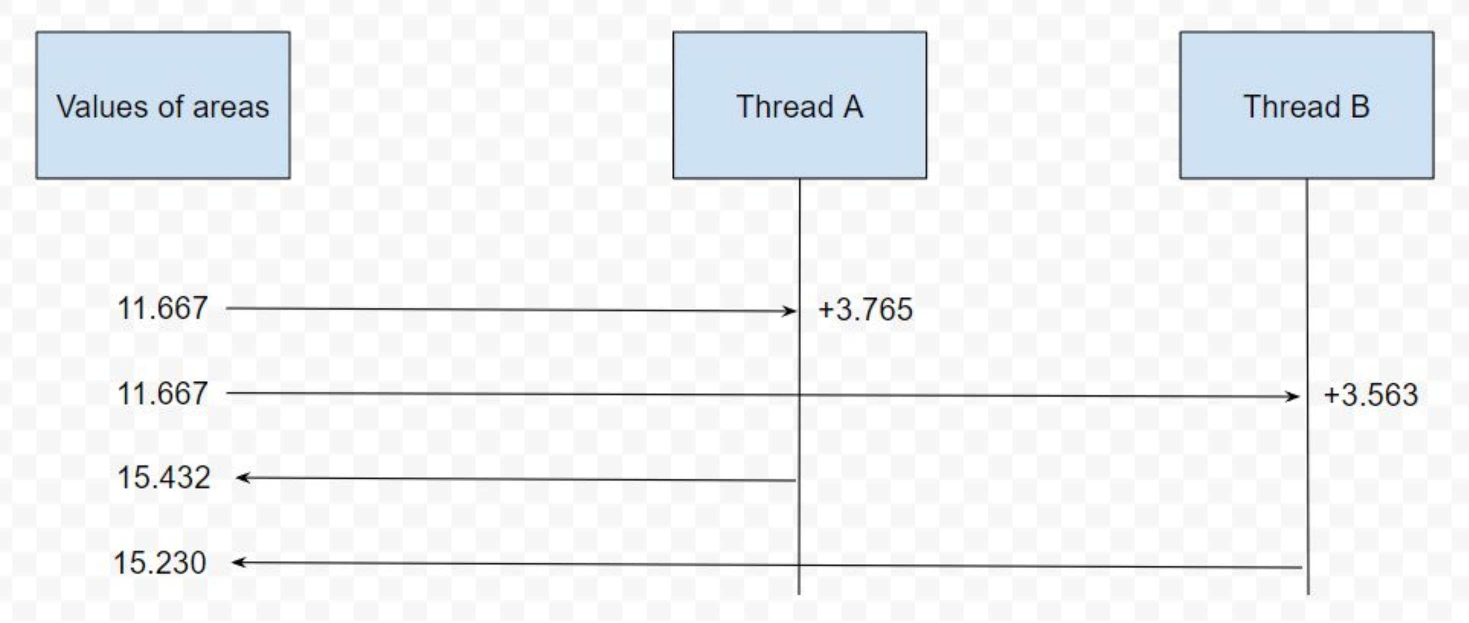
Synchronization: critical

- **Example of the PI_program**

```
double area, pi, x;
int I,n;
…..
area=0.0;
#pragma omp parallel for private(x)
{
for(i=0;i<n;i++){
x=(i+0.5)/n;
area+=4.0/(1.0+x*x); //race condition
}
pi=area/n;
```

Synchronization: critical

- **Problem of the example of the PI_program**: Thread B retrieves the original value of area before thread A can write the new value.



The assignment statement that reads and updates area must be put in a **critical section** (a portion of code that only one thread at a time may execute).

Synchronization: Atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example). The atomic directive can be applied only if a critical section consist of a single assignment statement that updates a scalar variable.

```
#pragma omp parallel
{
        double tmp, B;
        B =  DOIT();
        tmp = big_ugly(B);
 #pragma omp atomic
     X +=  tmp;
}
```

Atomic only protects the read/update of X

Exercise 3

- In exercise 2, you probably used an array to create space for each thread to store its partial sum.
- If array elements happen to share a cache line, this leads to false sharing.
    - Non-shared data in the same cache line so each update invalidates the cache line … in essence “sloshing independent data” back and forth between threads.
- Define a critical section to compute the global addition from the local addition obtained by each thread.
- Modify your “pi program” from exercise 2 to avoid false sharing due to the sum array.
- See PI_spmd.c and PI_spmdv2.c from “*/resources/Chapter3.-OPenMP/Examples*” of CV.

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Memory Model
- Data environment

SPMD vs. worksharing

- A parallel construct by itself creates an SPMD or "Single Program Multiple Data" program … i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team? This is called worksharing
- Worksharing costructs automates the task to divide the iterations of a/multiple parallel loops between the threads:
  - Loop construct
  - Sections/section constructs
  - Single construct
  - Task construct …. Available in OpenMP 3.0

Discussed later

The loop worksharing Constructs

- The for loop worksharing construct splits up loop iterations among the threads in a team. It does not specify parallelism or create a team of parallel threads. Rather, within an existing team of parallel threads, it divides the iterations across the parallel team.
- Syntax: #pragma omp for [clause[clause]……]

```
#pragma omp parallel

{
#pragma omp for
    for (I=0;I<N;I++){
        NEAT_STUFF(I);
    }
}
```

The variable I is made "private" to each thread by default. You could do this explicitly with a "private(I)" clause

Loop worksharing Constructs

A motivating example

Case A: Sequential code

```
for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

Case B: OpenMP parallel region

```
#pragma omp parallel
{
      int id, i, Nthrds, istart, iend;
      id = omp_get_thread_num();
      Nthrds = omp_get_num_threads();
      istart = id * N / Nthrds;
      iend = (id+1) * N / Nthrds;
      if (id == Nthrds-1)iend = N;
for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i];}
}
```

Case C: OpenMP parallel region and a worksharing for construct

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

loop worksharing constructs: The schedule clause

- In some loops, the time needed to execute different loop iteration varies considerably. Consider the following doubly nested loop that initializes an upper triangular matrix:

```
For (i=0;i<n; i++)
      for(j=i;j<n;j++)
            a[i][j]=i*j-4(i+1);
```

- The first iteration of the outmost loop (when *i* equals 0) requires *n* times more work than the last iteration (when *i* equals *n-1*). Suppose these *n* iterations are being executed on *t* threads. If each thread is assigned a contiguous block of [n/t] iterations, the parallel loop execution will have poor efficiency , because some threads will complete much faster than the others.
- The schedule clause allows to specify how the iterations of a loop should be scheduled to threads.

loop worksharing constructs: The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
  - schedule(static [,chunk])
    - All iterations are allocated before they execute any loop iterations. Deal-out blocks of iterations of size "chunk" to each thread. By default, this is taken by the *parallel for pragma*.
  - schedule(dynamic[,chunk])
    - Each thread grabs "chunk" iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library … for OpenMP 3.0).

loop work-sharing constructs:
The schedule clause

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

Discussion size chunk:

- Increasing the chunk size can reduce the overhead and increase the cache hit rate.
- Reducing the chunk size can allow finer balancing of workloads.

The scheduling Example

- Execute this code varying the scheduling clause

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>
#define THREADS 4
#define N 16
int main ( ) {
   int i;
   double start_time;
   start_time = omp_get_wtime();
#pragma omp parallel for schedule(dynamic,2) num_threads(THREADS)
   for (i = 0; i < N; i++) {
      /* wait for i seconds */
      sleep(i);
     printf("Thread %d has completed iteration %d.\n", omp_get_thread_num( ), i);
   }
printf("All done!\n");
 printf("Execution_time: %f s\n",omp_get_wtime()-start_time);
   return 0;
}
```

Exercise: The scheduling example

- Test and discuss the effect of the different schedule clauses:
  - Static
  - Dynamic with chunk=2 and 4.
  - Guided with chunk=2 and 4.
- Is There any significant difference between the different clauses?
- Discussion: See *http://forum.openmp.org/forum/viewtopic.php?f=3&t=83*

Results* Execution: Example_scheduling_clause

	Static			Dynamic			Guided		
Iterations	1	2	4	1	2	4	1	2	4
16	36s	42s	54s	36s	42s	54s	36s	44s	54s
32	136s	148s	172s	136s	148s	172s	139s	143s	177s
32 (without printf)	136s	148s	172s	136s	148s	172s	139s	143s	177s

* Results obtained in the front-end of the cluster “moore.udl.cat”

Discussion:

1. In this case, it is better chunk=1. The worst policy is guided.
2. Taking into account that small chunk improves the load balancing but increase trade-off cache. So, in this case:

- It is better a good load balancing policy that to minimize the overhead due to the high trade-off with cache.

General conclusions about scheduling policy

Conclusions:

- **OpenMP automatically splits for loop iterations** for us. Depending on our program, the default behavior may not be ideal.
- For **loops where each iteration takes roughly equal time, static schedules work best**, as they have little overhead.
- For **loops where each iteration can take very different amounts of time, dynamic scheduling works better** as the work will be split more evenly across threads.
- **Specifying chunks**, or using a guided schedule **provide a trade-off between the cache-memory.**
- Choosing **the best schedule depends on understanding your loop**.

Combined parallel/worksharing construct

- OpenMP shortcut: Put the "parallel" and the worksharing directive on the same line

```
double  res[MAX];  int i;
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
}
```

```
double  res[MAX];  int i;
#pragma omp parallel for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
```

These are equivalent

Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent . So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```
int i, j, A[MAX];
j = 5;
for (i=0;i< MAX; i++) {
    j +=2;
    A[i] = big(j);
}
```

Note: loop index "i" is private by default

```
int i,  A[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++) {
    int j = 5 + 2*(i+1);
    A[i] = big(j);
}
```

Remove loop carried dependence

Reduction

- **How do we handle this case?**

```
double  ave=0.0, A[MAX];    int i;
for (i=0;i< MAX; i++) {
    ave + = A[i];
}
ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … There is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation … it is called a "reduction".
- Support for reduction operations is included in most parallel programming environments.

Reduction

- OpenMP reduction clause:

  reduction (op : list)

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

```
double  ave=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:ave)
for (i=0;i< MAX; i++) {
    ave + = A[i];
}
ave = ave/MAX;
```

OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operator	Initial value
+	0
*	1
-	0

C/C++ only		
Operator	**Meaning**	**Initial value**
&	Bitwise and	All bits 1
|	Bitwise or	0
^	Bitwise exclusive	0
&&	Logical and	1
||	Logical or	0

Exercise 4: Pi with loops

- Go back to the serial pi program and parallelize it with a loop construct
- Your goal is to minimize the number of changes made to the serial program.

Questions:

- Compare the two implementations of the PI_program: using critical section (Exercise 3) and using reduction clause (Exercise 4).
- Which of both do you obtain better execution time?
- Why?

Exec. Time	1 thread	2 Thread	3 Thread	4 Thread
PI_spmdv2	1.52	0.77	0.51	0.40
PI_loop	1.53	0.77	0.51	0.40

* Results obtained on the cluster “moore.udl.cat”

Exercise 5: Optimizing loops

- Parallelize the matrix multiplication program in the file matmul.c
- Can you optimize the program by playing with how the loops are scheduled?

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Memory Model
- Data environment

Synchronization: Barrier

- Barrier: Each thread waits until all threads arrive.

```
#pragma omp parallel shared (A, B, C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
    for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
#pragma omp for nowait
    for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
    A[id] = big_calc4(id);
}
```

implicit barrier at the end of a for worksharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

The nowait Clause

- If there are multiple independent loops within a parallel region you can use the *nowait clause* to avoid the implicit barrier at the end of the loop construct.

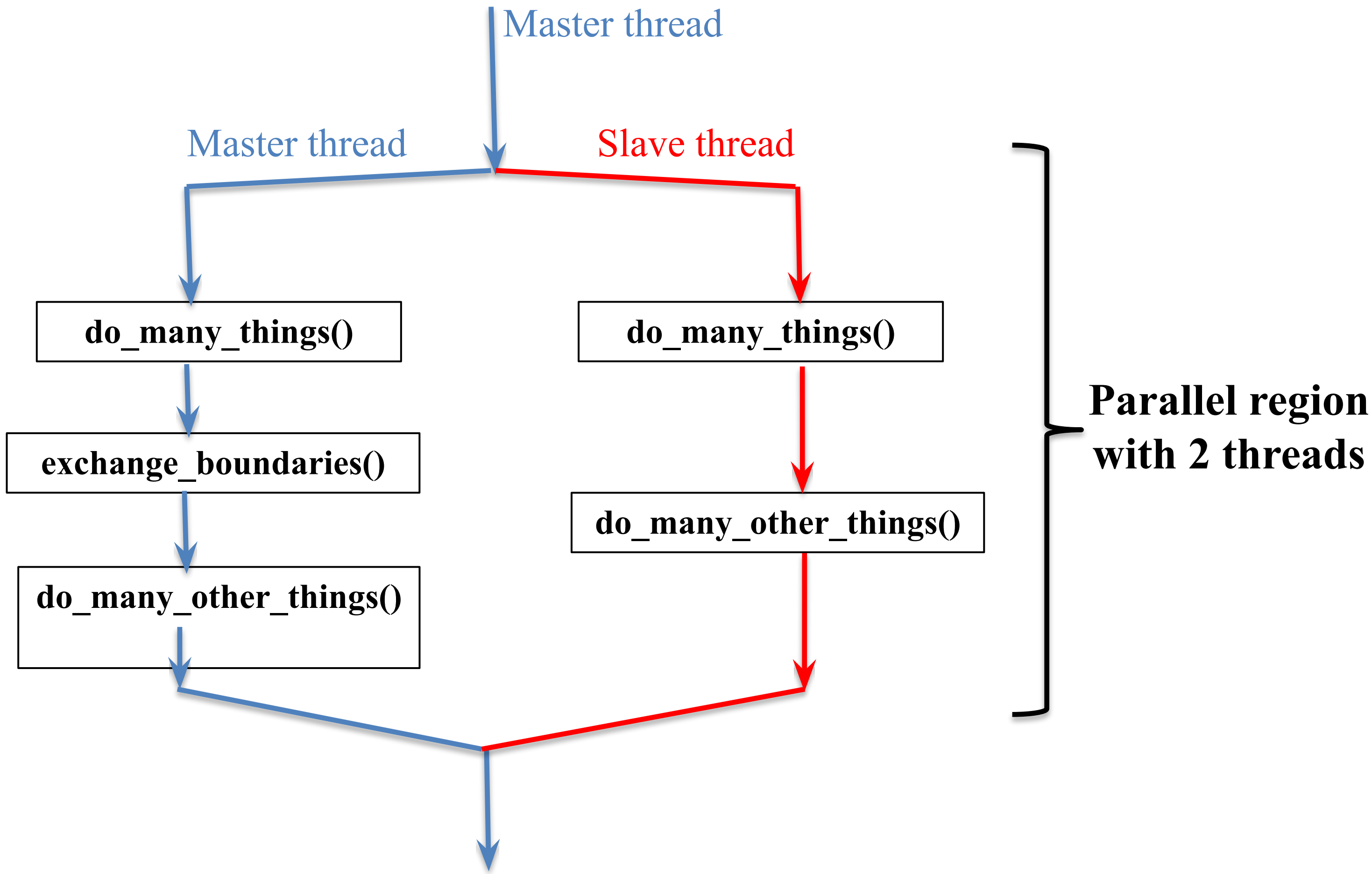
```
#include <math.h>
Void nowait_example(int n,int m, float *a, float *b, float *y, float *z)
{
Int i;
#pragma omp parallel
{

#pragma omp for nowait
for(i=1;i<n;i++)
      {     b[i]=(a[î]+a[i-1])/2.0;   }
#pragma omp  for nowait
for(i=0;i<m;i++)
      {     y[i]=sqrt(z[i]);}
}
}
```

Master Construct

- The master construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

```
#pragma omp parallel
{
    do_many_things();
#pragma omp master
    {    exchange_boundaries();   }
#pragma omp  barrier
    do_many_other_things();
}
```

Master Construct

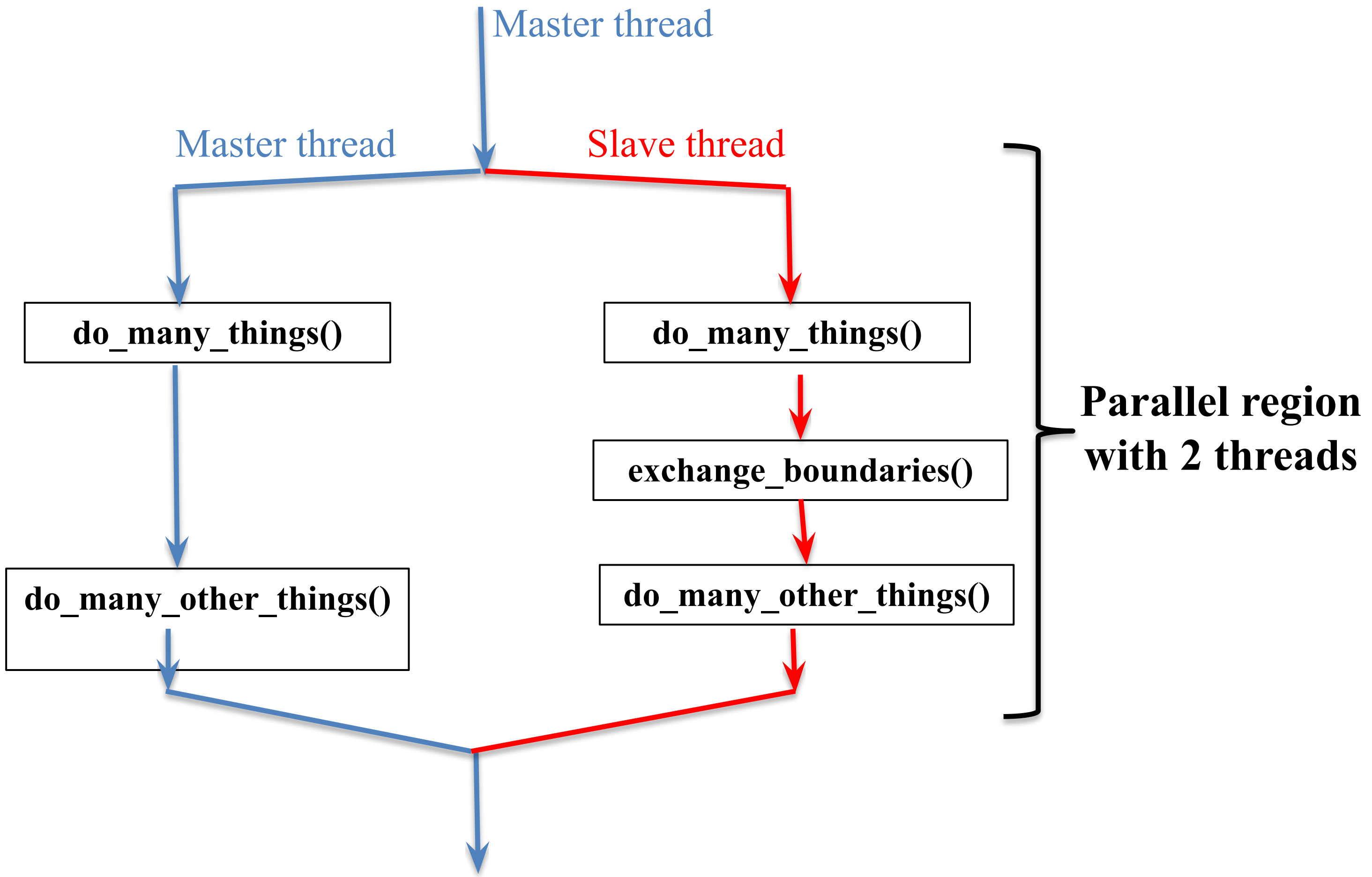


Single worksharing Construct

- The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause).

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

Single Construct



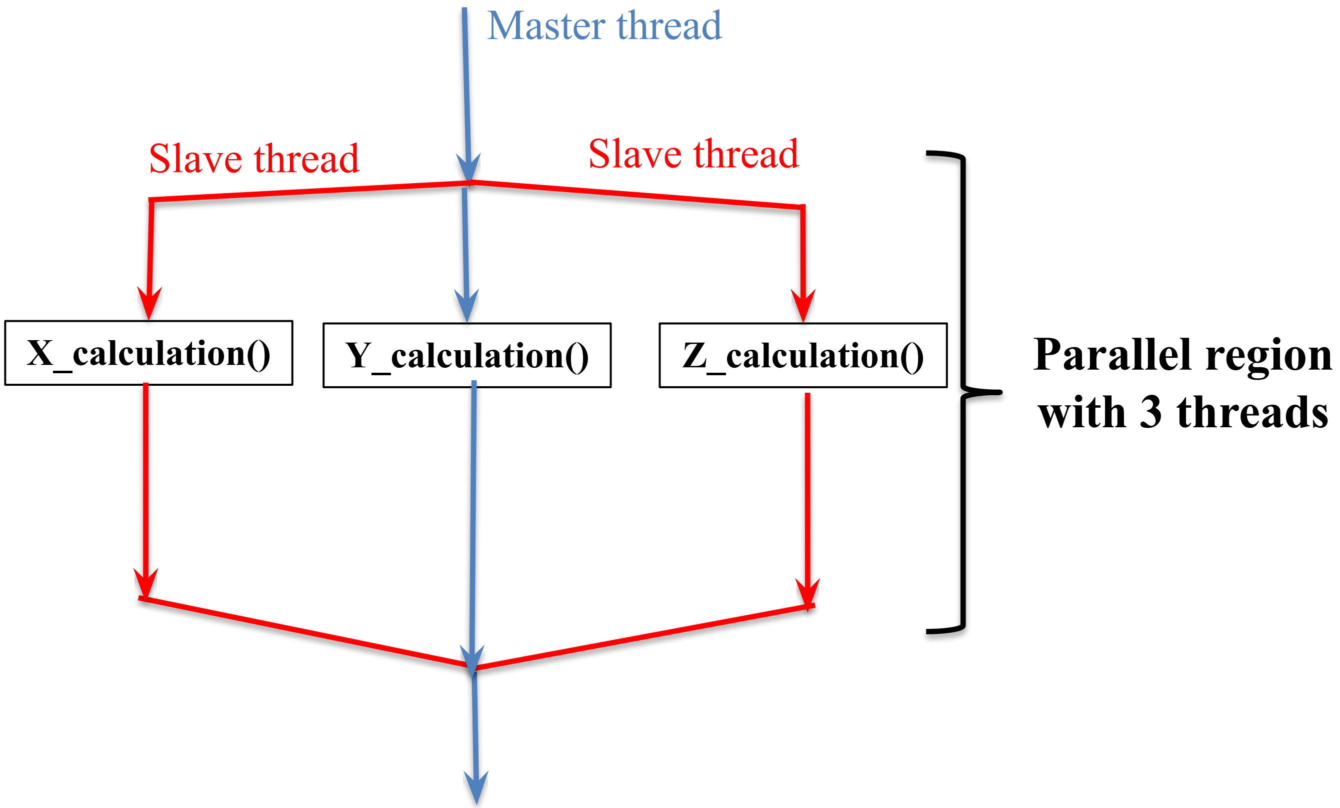
Sections worksharing Construct

- The *Sections* worksharing construct gives a different structured block to each thread.

```
#pragma omp parallel
{

  #pragma omp sections
  {
  #pragma omp section
        X_calculation();
  #pragma omp section
     y_calculation();
  #pragma omp section
     z_calculation();
  }

}
```

By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.

Sections Worksharing Construct

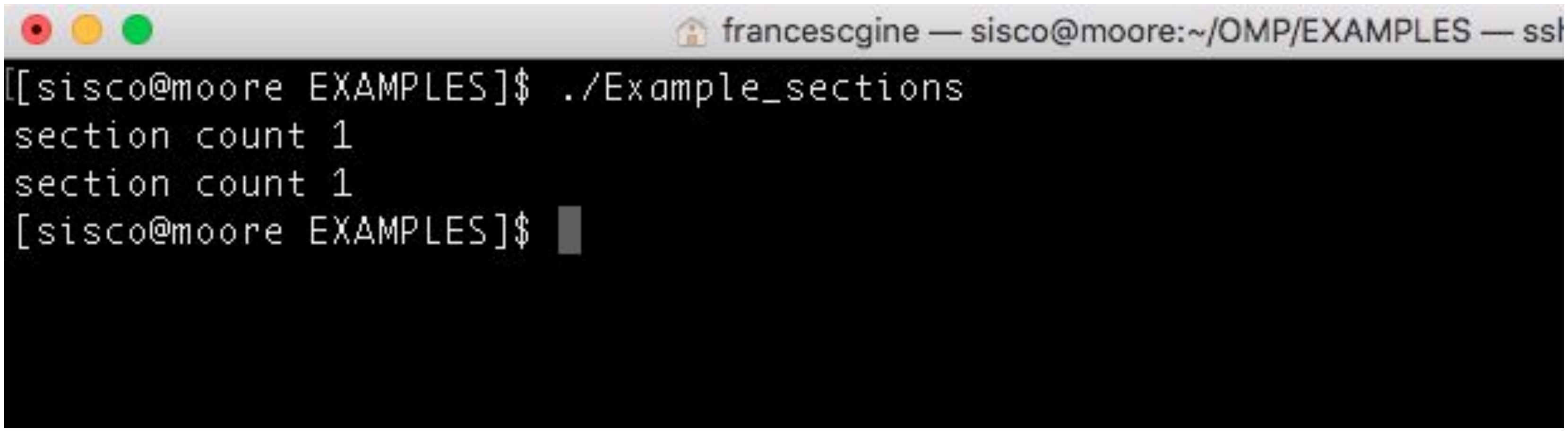


Example of Section Construct

- Guess the output of this code before its execution?

```
#include <omp.h>
#include <stdio.h>
#define NT 4
int main(void){
   int section_count=0;
   omp_set_dynamic(0);
   omp_set_num_threads(NT);
#pragma omp parallel firstprivate(section_count)
#pragma omp sections
   {
#pragma omp section
      {  section_count++;/*may print 1 or 2*/
         printf("section count %d\n", section_count); }
#pragma omp section
      {
         section_count++;/*may print 1 or 2*/
         printf("section count %d\n", section_count); }  }
   return 0;}
```

Output Example of Section Construct



Synchronization: ordered

- Computation across multiple iterations is fully overlapped, but before entering the ordered section each thread waits for the **ordered section** from the previous iteration of the loop to be completed. This ensures that the output is maintained in the original order.

```
#pragma omp parallel private(iam, np,i)
   {
      np = omp_get_num_threads();
      iam = omp_get_thread_num();
#pragma omp for ordered
      for(i=0;i<5;i++) {
         printf("Soy el thread %d, antes del ordered en la iteracion %d\n",iam,i);
#pragma omp ordered {
            printf("Soy el thread %d, actuando en la iteracion %d\n",iam,i);
                sleep(1);   } }
                }//parallel
   return 0;
}
```

Output previous Example

```
francescgine — sisco@moore:~/OMP/EXAMPLES —
[sisco@moore EXAMPLES]$ ./Example_ordered
Soy el thread 0, antes del ordered en la iteracion 0
Soy el thread 0, actuando en la iteracion 0
Soy el thread 2, antes del ordered en la iteracion 3
Soy el thread 3, antes del ordered en la iteracion 4
Soy el thread 1, antes del ordered en la iteracion 2
Soy el thread 0, antes del ordered en la iteracion 1
Soy el thread 0, actuando en la iteracion 1
Soy el thread 1, actuando en la iteracion 2
Soy el thread 2, actuando en la iteracion 3
Soy el thread 3, actuando en la iteracion 4
[sisco@moore EXAMPLES]$ £
```

Synchronization: Lock routines

A lock implies a memory fence (a "flush") of all thread visible variables

- Simple Lock routines:
  - A simple lock is available if it is unset.
    - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock() omp_destroy_lock()
- Nested Locks
  - A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
    - omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()

Note: a thread always accesses the most recent copy of the lock, so you don't need to use a flush on the lock variable.

Synchronization: Simple Locks

- Protect resources with locks.

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
    id = omp_get_thread_num();
    tmp = do_lots_of_work(id);
    omp_set_lock(&lck);
      printf("%d %d", id, tmp);
    omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

Wait here for your turn.

Release the lock so the next thread gets a turn.

Free-up storage when done.

Runtime Library routines

- Runtime environment routines:
  - Modify/Check the number of threads
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - Are we in an active parallel region?
    - omp_in_parallel()
  - Do you want the system to dynamically vary the number of threads from one parallel construct to another?
    - omp_set_dynamic, omp_get_dynamic();
  - How many processors in the system?
    - omp_num_procs()

…plus a few less commonly used routines.

Runtime Library routines

- To use a known, fixed number of threads in a program, (1) tell the system that you don't want dynamic adjustment of the number of threads, (2) set the number of threads, then (3) save the number you got.

Disable dynamic adjustment of the number of threads.

Request as many threads as you have processors.

Protect this op since Memory stores are not atomic

```
#include <omp.h>
void main()
{   int num_threads;
      omp_set_dynamic( 0 );
      omp_set_num_threads( omp_get_num_procs() );
#pragma omp parallel
   {    int id=omp_get_thread_num();
#pragma omp single
            num_threads = omp_get_num_threads();
          do_lots_of_stuff(id);
    }
}
```

Even in this case, the system may give you fewer threads than requested. If the precise # of threads matters, test for it and respond accordingly.

Environment Variables

- Set the default number of threads to use.
  - OMP_NUM_THREADS *int_literal*
- Control how “omp for schedule(RUNTIME)” loop iterations are scheduled.
  - OMP_SCHEDULE “schedule[, chunk_size]”

… Plus several less commonly used environment variables.

Controlling Number of Threads on Multiple Nesting Levels

The following example demonstrate how to use the OMP_NUM_THREADS environment variable to control the number of threads on multiple nesting levels:

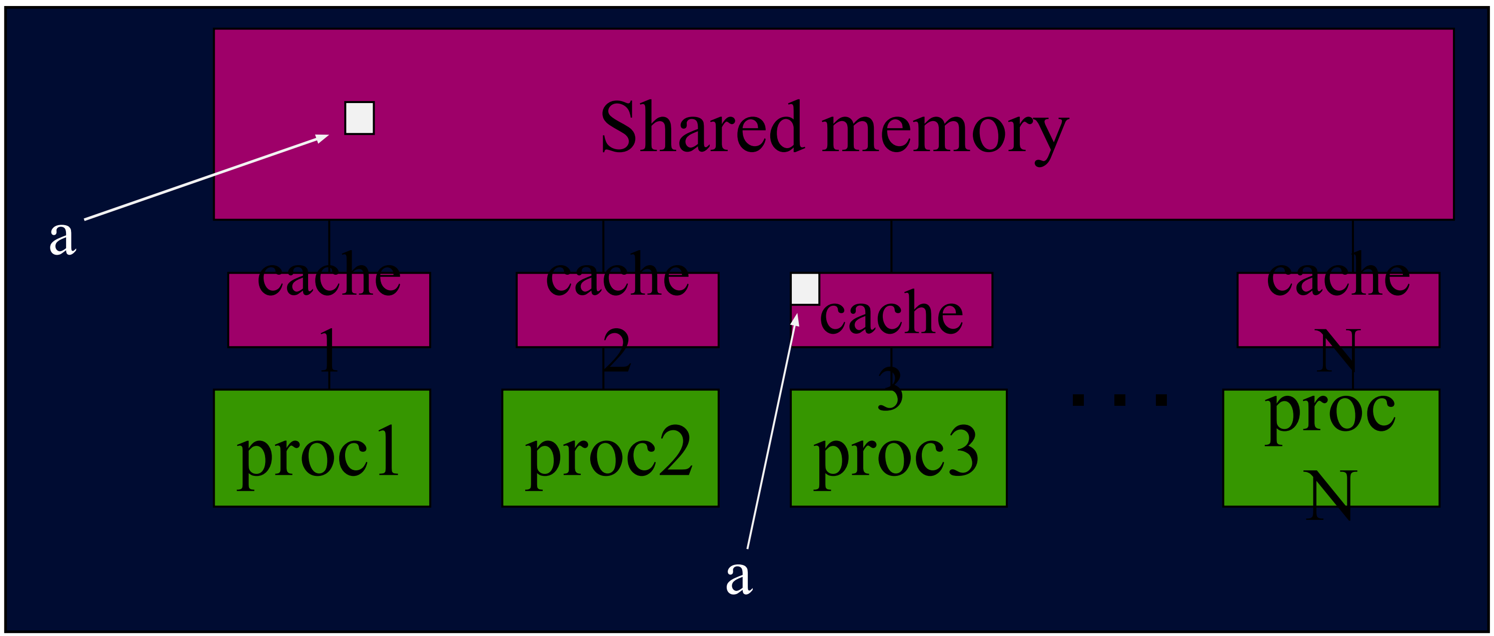
```
#include <omp.h>
#include <stdio.h>
int main(void)
{
   omp_set_nested(1);
   omp_set_dynamic(0);
#pragma omp parallel
   {
#pragma omp parallel
   {
#pragma omp single
         printf("Inner: num_threads=%d\n",omp_get_num_threads());
   }
#pragma omp barrier
   omp_set_nested(0);
#pragma omp parallel
   {
#pragma omp single
         /* EVEN IF OMP_NUM_THREADS=4 was set, the following should be print because nesting is disable: Inner:
num_threads=1*/
         printf("Inner: num_threads=%d\n",omp_get_num_threads());
   }
#pragma omp barrier
#pragma omp single
       printf("Outer: num_threads=%d\n",omp_get_num_threads());
}
return 0;
}
```

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Memory Model
- Data environment

OpenMP memory model

- **OpenMP supports a shared memory model.**
- **All threads share an address space, but it can get a local one.**



- **Multiple copies of data may be present in various levels of cache, or in registers.**

OpenMP and Relaxed Consistency

- OpenMP supports a relaxed-consistency shared memory model.
  - Threads can maintain a temporary view of shared memory which is not consistent with that of other threads.
  - These temporary views are made consistent only at certain points in the program.
  - The operation which enforces consistency is called the flush operation

OPenMP Memory Model (Example_barrier_flush.c)

In the following example, at Print 1, the value of *x* could be either 2 or 5, depending on the timing of the threads and the implementation of the assignment to x. The barrier after Print 1 contains implicit flushes on all the threads, so the programmer is guaranteed that the value 5 will be printed by both Prints.

```
#include <omp.h>
#include <stdio.h>
int main()
{
   int x;
   x=2;
#pragma omp parallel num_threads(4) shared(x)
   {
      if(omp_get_thread_num()==0)
         x=5;
      else
         printf("1: Threads %d: x= %d before barrier\n",omp_get_thread_num(),x);
#pragma omp barrier
      if(omp_get_thread_num()==0)
         printf("2: Threads %d: x= %d after barrier\n",omp_get_thread_num(),x);
      else
         printf("3: Threads %d: x= %d after barrier\n",omp_get_thread_num(),x);
   }
   return 0;
}
```

Output Example_barrier_flush.c

francescgine — sisco@moore:~/OMP/EXAMPLES — ssh sisco@moore.u

```
[sisco@moore EXAMPLES]$ ./Example_barrier_flush
Thread 3 x=5 before barrier
Thread 1 x=5 before barrier
Thread 4 x=5 before barrier
Thread 2 x=5 before barrier
Thread 2 x=5 after barrier
Thread 1 x=5 after barrier
Thread 0 x=5 after barrier
Thread 4 x=5 after barrier
Thread 3 x=5 after barrier
[sisco@moore EXAMPLES]$ ./Example_barrier_flush
Thread 4 x=2 before barrier
Thread 1 x=2 before barrier
Thread 3 x=5 before barrier
Thread 2 x=5 before barrier
Thread 0 x=5 after barrier
Thread 2 x=5 after barrier
Thread 3 x=5 after barrier
Thread 1 x=5 after barrier
Thread 4 x=5 after barrier
[sisco@moore EXAMPLES]$
```

Flush operation

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory
  – All previous read/writes by this thread have completed and are visible to other threads
  – No subsequent read/writes by this thread have occurred
  – A flush operation is analogous to a fence in other shared memory API's

Flush and synchronization

- A flush operation is implied by OpenMP synchronizations, e.g.
  - at entry/exit of parallel regions
  - at implicit and explicit barriers
  - at entry/exit of critical regions
  - whenever a lock is set or unset

  ….

  (but not at entry to worksharing regions or entry/exit of master regions)

Example Flush

```
x=0;
#pragma omp parallel private(iam, np)
{
np = omp_get_num_threads();
iam = omp_get_thread_num();

#pragma omp master
{
x=999;
printf(“thread %d, updating x=999 \n\n",iam);
}
printf(”thread %d, before flush, with x=%d
\n",iam,x);
#pragma omp flush(x)
printf("\t\t I’m thread %d, after flush, with x=%d
\n",iam,x);
}//parallel
```

Output Example Flush

francescgine — sisco@moore:~/OMP/EXAMPLES — ssh

```
[sisco@moore EXAMPLES]$ ./Example_flush2
Thread 0, updating x=999

Thread 0, before flush, with x=999
                I'm thread 0, after flush, with x=999
Thread 3, before flush, with x=0
                I'm thread 3, after flush, with x=999
Thread 2, before flush, with x=999
                I'm thread 2, after flush, with x=999
Thread 1, before flush, with x=999
                I'm thread 1, after flush, with x=999
[sisco@moore EXAMPLES]$
```

Example: producer-consumer pattern

Thread 0	Thread 1
`a = foo();` `flag = 1;`	`while (!flag);` `b = a;`

- This is **incorrect code**
- The compiler and/or hardware may re-order the reads/writes to `a` and `flag`, or `flag` may be held in a register.
- OpenMP has a `#pragma omp flush` directive which specifies an explicit flush operation
  - can be used to make the above example work
  - … but it's use is difficult and prone to subtle bugs

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Memory Model
- Data environment

Data environment:
Default storage attributes

- Shared Memory programming model:
    - Most variables are shared by default
- Global variables are SHARED among threads
    - Fortran: COMMON blocks, SAVE variables, MODULE variables
    - C: File scope variables, static
    - Both: dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
    - Stack variables in subprograms(Fortran) or functions(C) called from parallel regions are PRIVATE
    - Automatic variables within a statement block are PRIVATE.

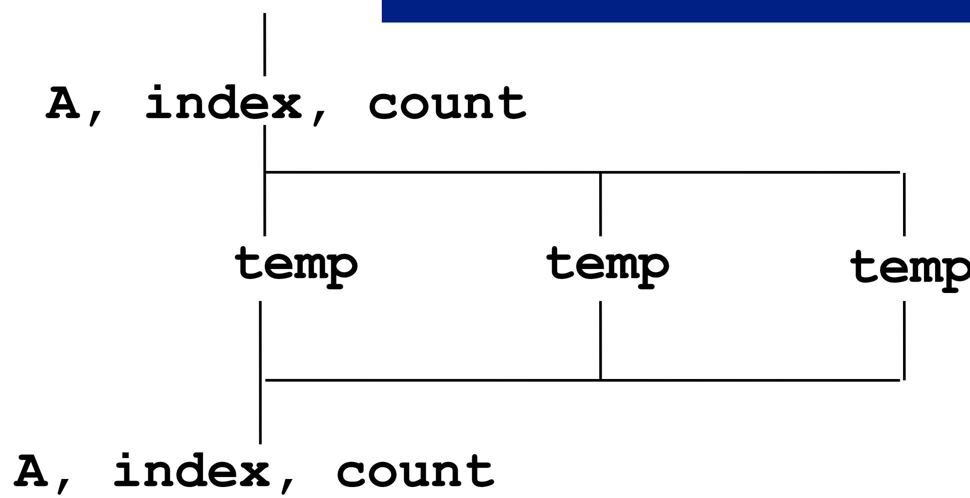
Data sharing: Examples

```
double A[10];
int main() {
int index[10];
#pragma omp parallel
     work(index);
printf("%d\n", index[0]);
}
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```

A, index and count are shared by all threads.

temp is local to each thread



Data sharing:
Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses*
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE

All the clauses on this page apply to the OpenMP construct NOT to the entire region.

- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  - LASTPRIVATE
- The default attributes can be overridden with:
  - DEFAULT (PRIVATE | SHARED | NONE)

DEFAULT(PRIVATE) ***is Fortran only***

All data clauses apply to parallel constructs and worksharing constructs except "shared" which only applies to parallel constructs.

Data Sharing: Private Clause

- private(var) creates a new local copy of var for each thread.
  - The value is uninitialized
  - In OpenMP 2.5 the value of the shared variable is undefined after the region

```
void wrong() {
    int tmp = 0;
#pragma omp for private(tmp)
    for (int j = 0; j < 1000; ++j)
        tmp += j;
    printf("%d\n", tmp);
}
```

tmp was not initialized

tmp: 0 in 3.0, unspecified in 2.5

Data Sharing: Firstprivate Clause

- Firstprivate is a special case of private.
    - Initializes each private copy with the corresponding value from the master thread.

```
void useless() {
    int tmp = 0;
#pragma omp for firstprivate(tmp)
    for (int j = 0; j < 1000; ++j)
      tmp += j;
    printf("%d\n", tmp);
 }
```

Each thread gets its own tmp with an initial value of 0

tmp: 0 in 3.0, unspecified in 2.5

Firstprivate Clause and Section Construct

```
#include <omp.h>
#include <stdio.h>
#define NT 4
int main(void)
{
    int section_count=0;
    omp_set_dynamic(0);
    omp_set_num_threads(NT);
#pragma omp parallel firstprivate(section_count)
#pragma omp sections
    {
#pragma omp section
        {
            section_count++;/*may print 1 or 2*/
            printf("section count %d\n", section_count);
        }
#pragma omp section
        {
            section_count++;/*may print 1 or 2*/
            printf("section count %d\n", section_count);
        }
    }
    return 0;
}
```

Questions:

- Execute the previous example with and without the firstprivate clause:
- Which is the difference?
- Why the value of the *section_count* var is different in both implementations?

francescgine — sisco@moore:~/OMP/EXAMPLES — ssh sisco@moore.udl.cat —

```
[sisco@moore EXAMPLES]$ ./sections_with_private
section count 1
section count 1
[sisco@moore EXAMPLES]$ ./sections_without_private
section count 1
section count 2
[sisco@moore EXAMPLES]$ 
```

Data sharing: Lastprivate Clause

- Lastprivate passes the value of a private from the last iteration to a global variable.

```
void closer() {
    int tmp = 0;
#pragma omp parallel for firstprivate(tmp) \
   lastprivate(tmp)
   for (int j = 0; j < 1000; ++j)
      tmp += j;
   printf("%d\n", tmp);
}
```

Each thread gets its own tmp with an initial value of 0

tmp is defined as its value at the "last sequential" iteration (i.e., for j=999)

Data Sharing:

A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

```
    variables A,B, and C = 1
#pragma omp parallel private(B)  firstprivate(C)
```

- Are A,B,C local to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

- **A" is shared by all threads; equals 1**
- **"B" and "C" are local to each thread.**
  - **B's initial value is undefined**
  - **C's initial value equals 1**

Outside this parallel region ...

- **The values of "B" and "C" are unspecified in OpenMP 2.5, and in OpenMP 3.0 if referenced in the region but outside the construct**

Acknowledgements: OPenMP Slides are Derived from

- Tim Watson from Intel Corporation.
- Wrinn Michael from Intel Corporation.
- Mark Bull from the University of Edinburgh.