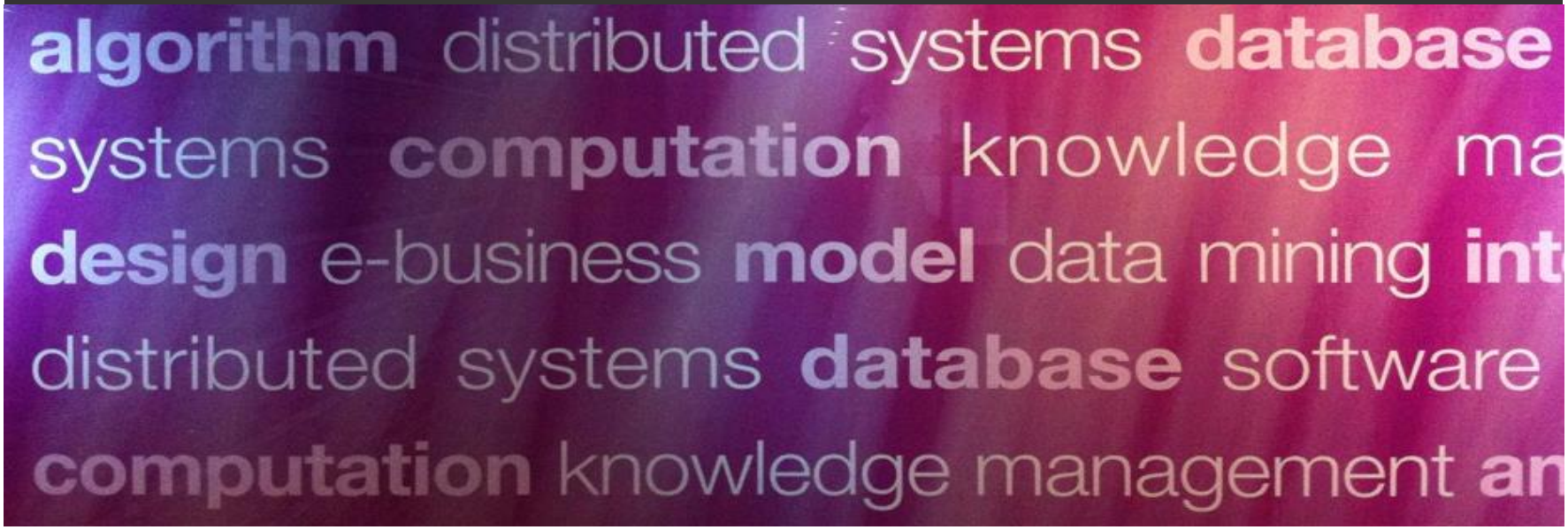
MONASH University

Information Technology

FIT3143 - LECTURE WEEK 3

PARALLEL COMPUTING ON SHARED MEMORY WITH OPEN MP



Overview

1. OpenMP for shared memory parallel programming

Associated learning outcomes

- Explain the fundamental principles of parallel computing architectures and algorithms (LO1)
- Design and develop parallel algorithms for various parallel computing architectures (LO3)

1) Programming with OpenMP

- Thread serial code with basic OpenMP pragmas
- Use OpenMP synchronization pragmas to coordinate thread execution and memory access

OpenMP

What Is OpenMP (or Open Multi-Processing)?

- Compiler directives for multithreaded programming
- Easy to create threaded Fortran and C/C++ codes
- Supports data parallelism model
- Incremental parallelism
  - Combines serial and parallel code in single source

- **Incremental parallelism is when you can modify only a portion of the code to test for better performance in parallel, then move on to another position in the code to test. Explicit threading models require a change to all parts of the code affected by the threading.**
- **Serial code can be "retrieved" by not compiling with the OpenMP options turned on. Assuming that there were no drastic changes required to get the code into a state that could be parallelized.**

http://www.openmp.org

OpenMP* Architecture

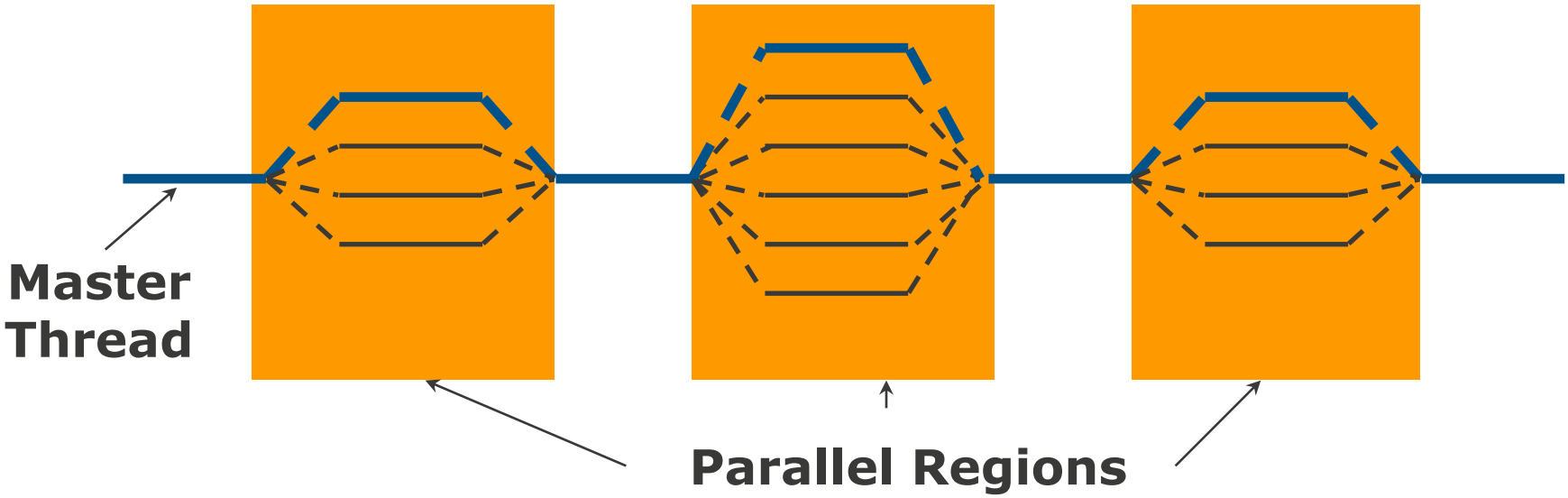
- Fork-join model // execution model of thread in OpenMP parallel regions
- Work-sharing constructs // for the pragmas and directives
- Data environment constructs // for the pragmas and directives
- Synchronization constructs // for the pragmas and directives
- Extensive Application Program Interface (API) for finer control
  // for the pragmas and directives

Programming Model

Fork-join parallelism:

- **Master thread spawns a team of threads as needed**
- **Parallelism is added incrementally: the sequential program evolves into a parallel program**

Applications starts execution in serial with Master Thread. At each parallel region encountered, threads are forked off, execute concurrently, and then join together at the end of the region.



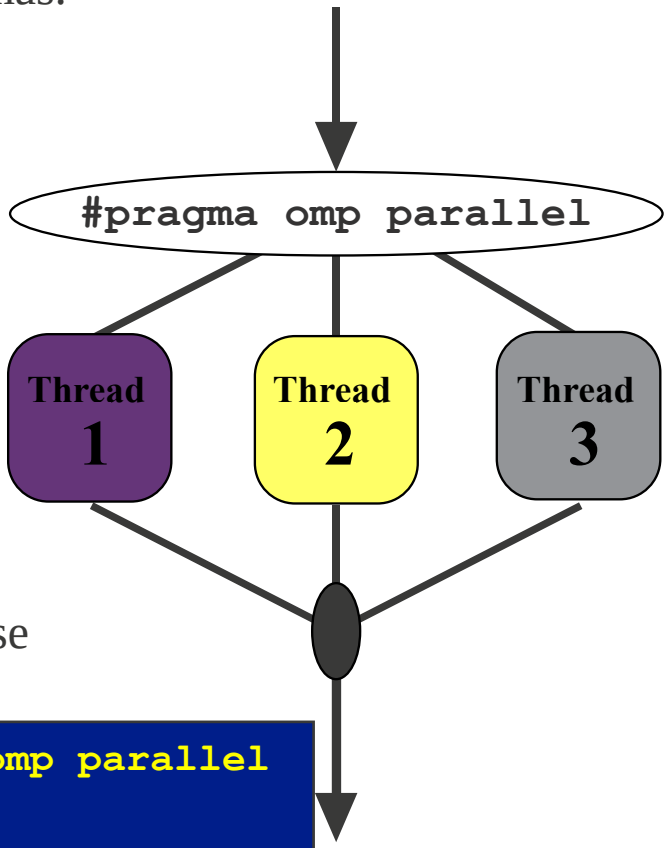
OpenMP* Pragma Syntax

Most constructs in OpenMP* are compiler directives or pragmas.
For C and C++, the pragmas take the form:

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

Parallel Regions

- Defines parallel region over structured block of code
- The '**parallel**' pragma that defines a parallel region.
- Threads are created as '**parallel**' pragma is crossed
- Threads block at end of region
- Data is shared among threads unless specified otherwise



- **Pragma will operate over a single statement or block of statements enclosed within curly braces.**
- **Variables accessed within the parallel region are all *shared* by default.**

C/C++ :

```
#pragma omp parallel
    {
        block
    }
```

How Many Threads?

Set environment variable for number of threads

```
set OMP_NUM_THREADS=4
```

There is no standard default for this variable

- Many systems:
  - # of threads = # of processors
  - Intel compilers use this default

The order in which the system will try to determine the number threads is
1. default
2. environment variable
3. API call
Each successive method (if present) will override the previous.

Example I

Serial Code

```
#include <stdio.h>

int main() {
        int i;
        printf("Hello World\n");

        for(i=0;i<6;i++)
                printf("Iter:%d\n",i);

        printf("GoodBye World\n");
}
```

Parallel Code

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

int main() {
#pragma omp parallel
{
        int i;
        printf("Hello World\n");

#pragma omp for
        for(i=0;i<6;i++)
                printf("Iter:%d\n",i);

}
        printf("GoodBye World\n");
}
```

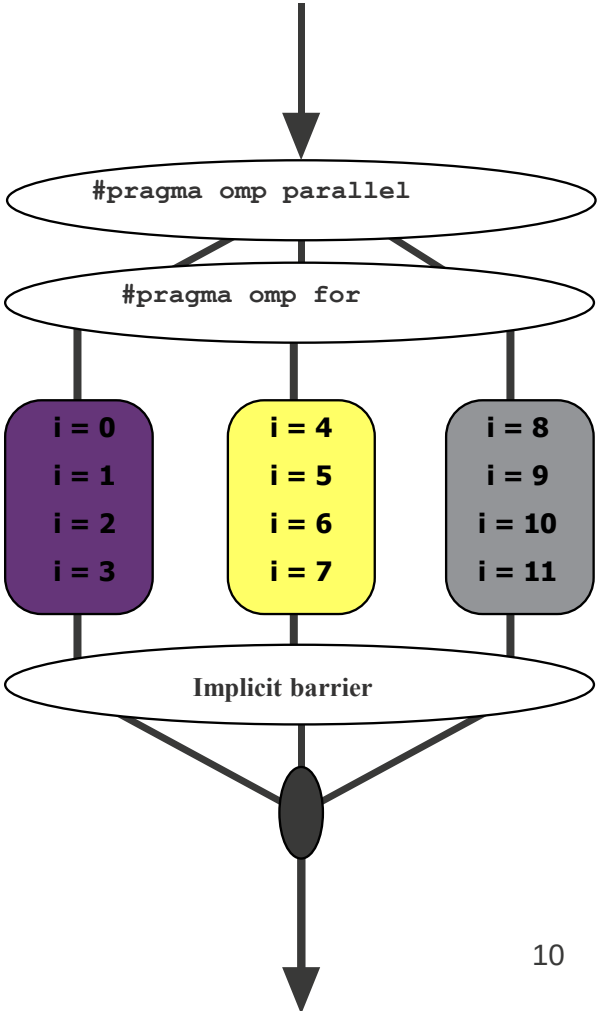
Work-sharing Construct - “for” work-sharing pragma

```
#pragma omp parallel
#pragma omp for
        for (i=0; i<N; i++){
                Do_Work(i);
        }
```

```
#pragma omp parallel
#pragma omp for
   for(i = 0; i < 12; i++)
      c[i] = a[i] + b[i]
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct

- Splits loop iterations into threads
- Must be in the parallel region
- Must precede the loop



Work-sharing Construct - “for” work-sharing pragma

```
#pragma omp parallel
#pragma omp for
   for(i = 0; i < 12; i++)
      c[i] = a[i] + b[i]
```

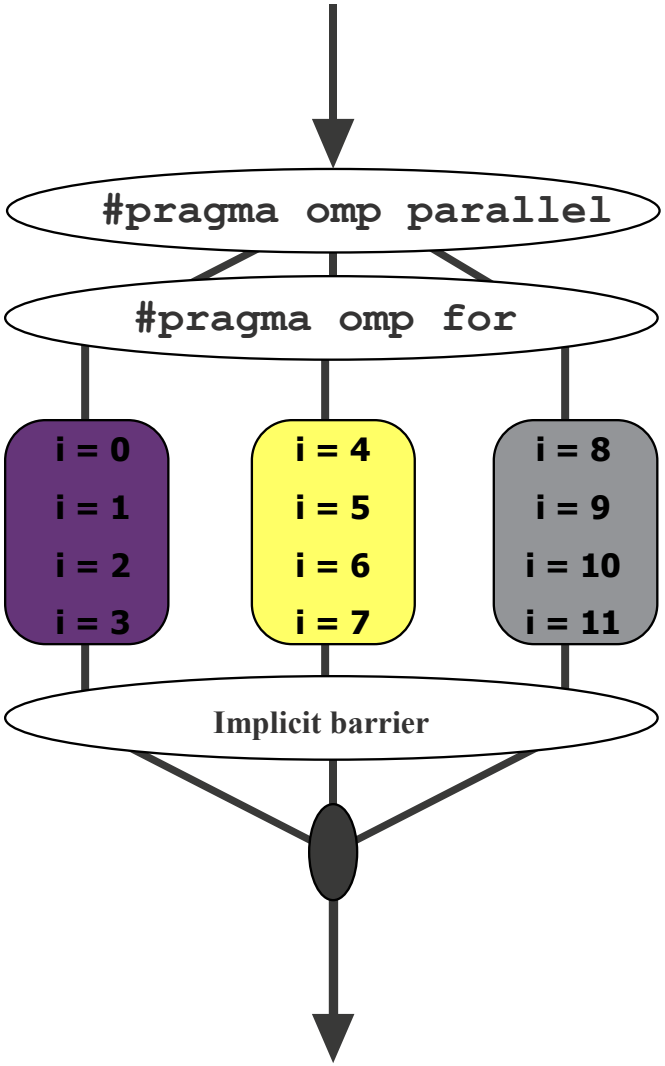


Diagram shows a static division of iterations based on the number of threads.

Note the implicit barrier at end of construct.

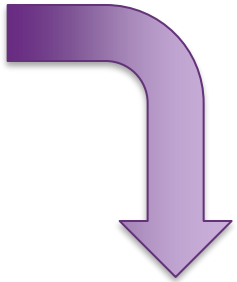
Q: Why is there a barrier at the end of the work-sharing construct?

A: Code following the for-loop may rely on the results of the computations within the for-loop. In serial code, the for-loop completes before proceeding on to the next computation. Thus, to remain serially consistent, the barrier at the end of the construct is enforced.

Combining pragmas

These two code segments are equivalent

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



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

Data Environment

OpenMP uses a shared-memory programming model

- Most variables are shared by default.
- Global variables are shared among threads
  - C/C++: File scope variables, static

But not everything is shared,

- Stack variables in functions called from parallel regions are PRIVATE
- Automatic variables within a statement block are PRIVATE
- Loop index variables are private (with exceptions)
  - C/C+: The first loop index variable in nested loops following a **#pragma omp for**

Data Scope Attributes

The default status can be modified with

Scoping attribute clauses

shared(varname,…)

private(varname,…)

default (shared | none)

All data clauses apply to parallel regions and work-sharing constructs except “shared,” which only applies to parallel regions.

The Private Clause

Reproduces the variable for each thread

- Variables are un-initialized; C++ object is default constructed
- Any value external to the parallel region is undefined

```
void* work(float* c, int N) {
  float x, y; int i;
 #pragma omp parallel for private(x,y)
    for(i=0; i<N; i++) {
            x = a[i]; y = b[i];
            c[i] = x + y;
    }
}
```

- **For-loop iteration variable is SHARED by default.**
- **Emphasize that private variables are uninitialized when entering the region. Thus, the assignment to x and y is done before reading the values.**
- **The private variables are destroyed at the end of the construct to which they are scoped.**

Example II: Dot Product

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

- **Standard dot product algorithm. Multiply corresponding elements from 1-D vectors and add all the products to a single value.**
- **The "shared" clause is superfluous since this is the default. It is used here to emphasize the problem.**
- **We can't make sum a private because we need to have the value after the parallel region.**

- **Q: What is wrong with this code?**
- **A: The shared variable "sum" is being read and updated by multiple threads and the answer is likely to be incorrect because of this data race.**

Example II: Dot Product - Protect Shared Data

Must protect access to shared, modifiable data

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
#pragma omp critical
     sum += a[i] * b[i];
   }
  return sum;
}
```

Demonstrating one method of enforcing mutual exclusion in OpenMP.

The "critical" pragma allows only one thread at a time to execute the update of sum.

OpenMP* Critical Construct

#pragma omp critical [(*lock_name)*]

Defines a critical region on a structured block

Threads wait their turn –at a time, only one calls consum() thereby protecting R1 and R2 from race conditions.

Naming the critical constructs is optional but may increase performance.

Build points reveal that the critical constructs can be named. Without names, each construct has the same name and only one thread will be allowed to execute within each different named region. Thus, in the example, since R1 and R2 are unrelated and will never be aliased, naming the constructs allows one thread to be in each at the same time.

```
float R1, R2;
#pragma omp parallel
{ float A, B;
#pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical
    consum (B, &R1);          //R1_lock
    A = bigger_job(i);
#pragma omp critical
    consum (A, &R2);          //R2_lock
  }
}
```

OpenMP* Reduction Clause

reduction (*op* : *list*)

The variables in "*list*" must be shared in the enclosing parallel region

Inside parallel or work-sharing construct:

- A PRIVATE copy of each list variable is created and initialized depending on the "op"
- These copies are updated locally by threads
- At end of construct, local copies are combined through "op" into a single value and combined with the value in the original SHARED variable

- **The operation must be an associative operation. Different operations are defined for C and Fortran, depending on what intrinsics are available in the language.**
- **The private copies of the list variables will be initialized with a value that depends on the operation. (Initial values are shown in two slides.)**

Reduction Example

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

- Local copy of *sum* for each thread
- All local copies of *sum* added together and stored in "global" variable

C/C++ Reduction Operations

- A range of associative and commutative operators can be used with reduction
- Initial values are the ones that make sense

Operator	Initial Value
+	0
*	1
-	0
^	0

Operator	Initial Value
&	~0
|	0
&&	1
||	0

Assigning Iterations - three main scheduling clauses for work-sharing loop construct

The schedule clause affects how loop iterations are mapped onto threads

schedule(static [,chunk])

- Blocks of iterations of size "chunk" to threads
- Round robin distribution

schedule(dynamic[,chunk])

- Threads grab "chunk" iterations
- When done with iterations, thread requests next set

schedule(guided[,chunk])

- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than "chunk"

- **Default chunk size for static is to divide the set of iterations into one chunk per thread.**
- **Default chunk size for dynamic is 1.**
- **Default chunk size for guided is 1.**

Which Schedule to Use

Schedule Clause	When To Use
STATIC	Predictable and similar work per iteration
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

Schedule Clause Example

```
#pragma omp parallel for schedule (static, 8)
    for( int i = start; i <= end; i += 2 )
    {
      if ( TestForPrime(i) )  gPrimesFound++;
    }
```

Iterations are divided into chunks of 8

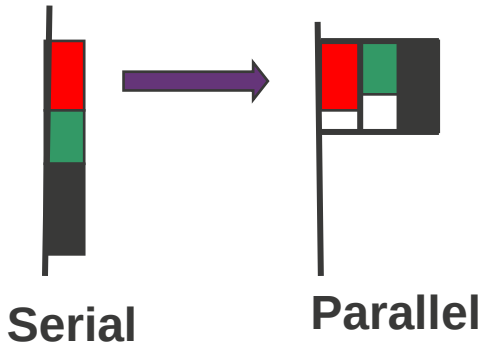
- If start = 3, then first chunk is **i**={3,5,7,9,11,13,15,17}

C example uses STATIC scheduling. Set of iterations is divided up into chunks of size 8 and distributed to threads in round robin fashion.

Parallel Sections

Independent sections of code can execute concurrently

```
#pragma omp parallel sections
{
   #pragma omp section
   phase1();
   #pragma omp section
   phase2();
   #pragma omp section
   phase3();
}
```



- **OpenMP sections for task parallelism.**
- **Note that the outer pragma is plural.**
- **No need to block code within curly braces. The task is composed of all lines of code between the "section" pragmas. Thus, if the third "section" pragma were erased, there would be two tasks defined: A. phase1, and B) phase2 followed by phase3.**
- **There is an implicit barrier at the end of the "sections" pragma.**

Q: What if there are more/less threads than tasks? How are tasks assigned to threads?

A: Scheduling of tasks to threads is implementation dependent. Sections are distributed among the threads in the parallel team. Each section is executed only once and each thread may execute zero or more sections. It's not possible to determine whether or not a section will be executed before another. Therefore, the output of one section should not serve as the input to another. Instead, the section that generates output should be moved before the sections construct.

Single Construct

Denotes block of code to be executed by only one thread

- Thread chosen is implementation dependent

Implicit barrier at end

```
#pragma omp parallel
{
  DoManyThings();
#pragma omp single
  {
    ExchangeBoundaries();
  } // threads wait here for single
  DoManyMoreThings();
}
```

Used when only one thread should execute a portion of code. This could be used when two parallel regions have very little serial code in between. Combine the two regions into a single region (less overhead for fork-join) and add the "single" construct for the serial portion.

Master Construct

Denotes block of code to be executed only by the master thread. No implicit barrier at end

```
#pragma omp parallel
{
  DoManyThings();
#pragma omp master
  {            // if not master skip to next statement
    ExchangeBoundaries();
  }
  DoManyMoreThings();
}
```

Implicit Barriers

Several OpenMP* constructs have implicit barriers

- **parallel**
- **for**
- **single**

Unnecessary barriers hurt performance

- Waiting threads accomplish no work!

Suppress implicit barriers, when safe, with the **nowait** clause

Nowait Clause

```
#pragma omp for nowait
  for(...)
    {...};
```

```
#pragma single nowait
{ [...] }
```

Use when threads would wait between independent computations

```
#pragma omp for schedule(dynamic,1) nowait
 for(int i = 0; i < n; i++)
   a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
 for(int j = 0; j < m; j++)
   b[j] = bigFunc2(j);
```

Schedule in each loop is (dynamic, 1) and computations in each loop are independent of each other (one loop updates a[], other loop updates b[]). Without nowait clause, threads would pause until all work is done in first loop; with nowait clause, when work is exhausted from first loop, threads can begin executing work in second loop. (It is possible that the second loop can complete all work before the work in the first loop is done.)

Barrier Construct

Explicit barrier synchronization

Each thread waits until all threads arrive

```
#pragma omp parallel shared (A, B, C)
{
        DoSomeWork(A,B);
        printf("Processed A into B\n");
#pragma omp barrier
        DoSomeWork(B,C);
        printf("Processed B into C\n");
}
```

Example code likely uses A to update B in first call, and B to update C in second call. Thus, to ensure correct execution, all processing from first call must be completed before starting second call.

Atomic Construct

Special case of a critical section

Applies only to simple update of memory location

```
#pragma omp parallel for shared(x, y, index, n)
  for (i = 0; i < n; i++) {
    #pragma omp atomic
      x[index[i]] += work1(i);
    y[i] += work2(i);
  }
```

- **Only a small set of instruction can be used within the atomic pragma.**
- **Since index[i] can be the same for different i values, the update to x must be protected. In this case, the update to an element of x[] is atomic. The other computations (call to work1() and the computation of the value in index[i]) will not be done atomically.**
- **Use of a critical section would serialize updates to x. Atomic protects individual elements of x array, so that if multiple, concurrent instances of index[i] are different, updates can still be done in parallel.**
- **The operations allowed within atomic are: x <binop>= <expr>; x++; ++x; x—; --x**
  where x is "an lvalue expression of scalar type."

OpenMP* API

Get the thread number within a team

```
int omp_get_thread_num(void);
```

Get the number of threads in a team

```
int omp_get_num_threads(void);
```

Usually not needed for OpenMP codes

- Can lead to code not being serially consistent
- Does have specific uses (debugging)
- Must include a header file

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

Emphasize that API calls are usually not needed. Assignment of loop iterations and other computations to threads is already built into OpenMP.

Summary

- OpenMP
  - Basic architecture
  - Parallelism with #pragma construct
  - Critical section and atomicity
  - Additional constructs (reduction, sections, etc.)