



Session 4: Parallel Programming with OpenMP

Xavier Martorell

Barcelona Supercomputing Center

Agenda

- 10:00 - 11:00 OpenMP fundamentals, parallel regions
- 11:00 - 11:30 Worksharing constructs
- 11:30 - 12:00 Break
- 12:00 - 12:15 Synchronization mechanisms in OpenMP
- 12:15 - 13:00 Practical: heat diffusion
- 13:00 - 14:00 Lunch
- 14:00 - 14:20 Programming using a hybrid MPI/OpenMP approach
- 14:20 - 17:00 Practical: heat diffusion



Part I

OpenMP fundamentals, parallel regions



Outline

- OpenMP Overview
- The OpenMP model
- Writing OpenMP programs
- Creating Threads
- Data-sharing attributes



Outline

- OpenMP Overview
- The OpenMP model
- Writing OpenMP programs
- Creating Threads
- Data-sharing attributes



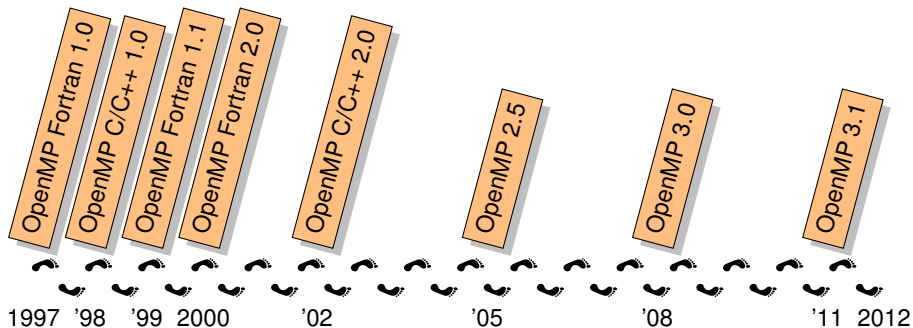
What is OpenMP?

- It's an API extension to the C, C++ and Fortran languages to write parallel programs for shared memory machines
 - Current version is 3.1 (July 2011)
 - ... 4.0 is open for public comments
 - Supported by most compiler vendors
 - Intel, IBM, PGI, Sun, Cray, Fujitsu, HP, GCC...
- Maintained by the Architecture Review Board (ARB), a consortium of industry and academia

<http://www.openmp.org>



A bit of history



Advantages of OpenMP

- Mature standard and implementations
 - Standardizes practice of the last 20 years
- Good performance and scalability
- Portable across architectures
- Incremental parallelization
- Maintains sequential version
- (mostly) High level language
 - Some people may say a medium level language :-)
- Supports both task and data parallelism
- Communication is implicit



Disadvantages of OpenMP

- **Communication is implicit**
- Flat memory model
- Incremental parallelization creates false sense of glory/failure
- No support for accelerators (...yet, maybe in 4.0)
- No error recovery capabilities (...yet, 4.0)
- Difficult to compose
- Lacks high-level algorithms and structures
- Does not run on clusters



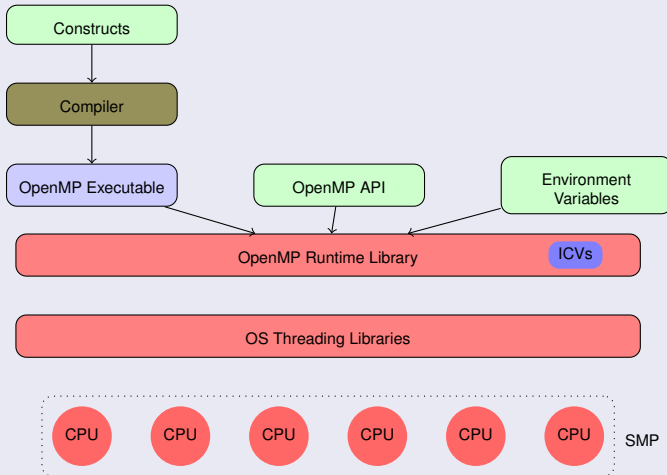
Outline

- OpenMP Overview
- The OpenMP model
- Writing OpenMP programs
- Creating Threads
- Data-sharing attributes



OpenMP at a glance

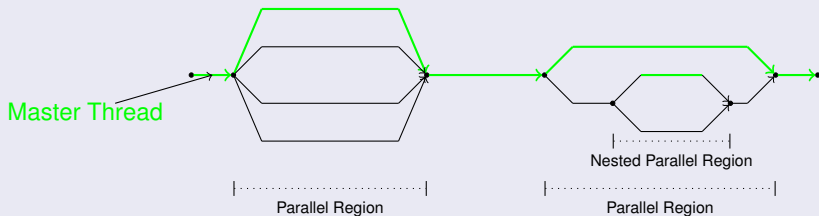
OpenMP components



Execution model

Fork-join model

- OpenMP uses a **fork-join** model
 - The **master** thread spawns a **team** of threads that joins at the end of the parallel region
 - Threads in the same team can **collaborate** to do work



Memory model

- OpenMP defines a relaxed memory model
 - Threads can see different values for the same variable
 - Memory consistency is only guaranteed at specific points
 - Luckily, the default points are usually enough
- Variables can be shared or private to each thread



Outline

- OpenMP Overview
- The OpenMP model
- **Writing OpenMP programs**
- Creating Threads
- Data-sharing attributes



OpenMP directives syntax

In Fortran

Through a specially formatted comment:

```
sentinel construct [clauses]
```

where sentinel is one of:

- !\$OMP or C\$OMP or *\$OMP in fixed format
- !\$OMP in free format

In C/C++

Through a compiler directive:

```
#pragma omp construct [clauses]
```

- OpenMP syntax is ignored if the compiler does not recognize OpenMP



OpenMP directives syntax

In Fortran

Through a specially formatted comment:

```
sentinel construct [clauses]
```

where sentinel is one of:

- !\$OMP or C\$OMP or *\$OMP in fixed format
- !\$OMP in free format

In C/C++

Through a compiler directive:

```
#pragma omp construct [clauses]
```

- OpenMP svntax is ignored if the compiler does not recognize

We'll be using C/C++ syntax through this tutorial



Headers/Macros

C/C++ only

- `omp.h` contains the API prototypes and data types definitions
- The `_OPENMP` is defined by the OpenMP enabled compilers
 - Allows conditional compilation of OpenMP

Fortran only

- The `omp_lib` module contains the subroutine and function definitions



Structured Block

Definition

Most directives apply to a **structured block**:

- Block of one or more statements
- One entry point, one exit point
 - No branching in or out allowed
- Terminating the program is allowed



Outline

- OpenMP Overview
- The OpenMP model
- Writing OpenMP programs
- **Creating Threads**
- Data-sharing attributes



The parallel construct

Directive

```
#pragma omp parallel [clauses]
  structured block
```

where clauses can be:

- **num_threads** (*expression*)
- **if** (*expression*)
- **shared** (*var-list*)
- **private** (*var-list*)
- **firstprivate** (*var-list*)
- **default**(none|shared| **private** | **firstprivate**)
- **reduction** (*var-list*)

Coming shortly!

We'll see it later

Only in Fortran

The parallel construct

Specifying the number of threads

- The number of threads is controlled by an internal control variable (**ICV**) called **nthreads-var**
- When a parallel construct is found a parallel region with a **maximum** of **nthreads-var** is created
 - Parallel constructs can be nested creating **nested parallelism**
- The **nthreads-var** can be modified through
 - the **omp_set_num_threads** API called
 - the **OMP_NUM_THREADS** environment variable
- Additionally, the **num_threads** clause causes the implementation to ignore the ICV and use the value of the clause for that region



The parallel construct

Avoiding parallel regions

- Sometimes we only want to run in parallel under certain conditions
 - E.g., enough input data, not running already in parallel, ...
- The **if** clause allows to specify an *expression*. When evaluates to false the **parallel** construct will only use 1 thread
 - Note that still creates a new team and data environment



Putting it together

Example

```
void main () {  
    #pragma omp parallel  
    ...  
    omp_set_num_threads(2);  
    #pragma omp parallel  
    ...  
    #pragma omp parallel num_threads(random()%4+1) if(N>=128)  
    ...  
}
```



Putting it together

Example

```
void main () {  
    #pragma omp parallel  
    ... ← An unknown number of threads here. Use OMP_NUM_THREADS  
    omp_set_num_threads(2);  
    #pragma omp parallel  
    ...  
    #pragma omp parallel num_threads(random()%4+1) if(N>=128)  
    ...  
}
```



Putting it together

Example

```
void main () {  
    #pragma omp parallel  
    ...  
    omp_set_num_threads(2);  
    #pragma omp parallel  
    ... ← A team of two threads here  
    #pragma omp parallel num_threads(random()%4+1) if(N>=128)  
    ...  
}
```



Putting it together

Example

```
void main () {  
    #pragma omp parallel  
    ...  
    omp_set_num_threads(2);  
    #pragma omp parallel  
    ...  
    #pragma omp parallel num_threads(random()%4+1) if(N>=128)  
    ... ← A team of [1..4] threads here  
}
```



API calls

Other useful routines

<code>int omp_get_num_threads()</code>	Returns the number of threads in the current team
<code>int omp_get_thread_num()</code>	Returns the id of the thread in the current team
<code>int omp_get_num_procs()</code>	Returns the number of processors in the machine
<code>int omp_get_max_threads()</code>	Returns the maximum number of threads that will be used in the next parallel region
<code>double omp_get_wtime()</code>	Returns the number of seconds since an arbitrary point in the past



Outline

- OpenMP Overview
- The OpenMP model
- Writing OpenMP programs
- Creating Threads
- Data-sharing attributes



Data environment

A number of clauses are related to building the data environment that the construct will use when executing

- `shared`
- `private`
- `firstprivate`
- `default`
- `threadprivate`
- `lastprivate`
- `reduction`



Data-sharing attributes

Shared

When a variable is marked as **shared**, the variable inside the construct is the same as the one outside the construct

- In a parallel construct this means all threads see the same variable
 - but not necessarily the same value
- Usually need some kind of synchronization to update them correctly
 - OpenMP has consistency points at synchronizations



Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel shared(x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```



Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel shared(x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

Prints 2 or 3 (three printf's in total)



Data-sharing attributes

Private

When a variable is marked as **private**, the variable inside the construct is a **new** variable of the same type with an **undefined** value

- In a parallel construct this means all threads have a different variable
- Can be accessed without any kind of synchronization



Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel private(x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```



Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel private(x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

Can print anything (twice, same or different)



Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel private(x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x); ← Prints 1
```



Data-sharing attributes

Firstprivate

When a variable is marked as **firstprivate**, the variable inside the construct is a **new** variable of the same type but it is initialized to the original value of the variable

- In a parallel construct this means all threads have a different variable with the same initial value
- Can be accessed without any kind of synchronization



Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel firstprivate(x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```



Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel firstprivate(x) num_threads(2)
{
    x++;
    printf("%d\n",x); ← Prints 2 (twice)
}
printf("%d\n",x);
```



Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel firstprivate(x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x); ← Prints 1
```



Data-sharing attributes

What is the default?

- Static/global storage is **shared**
- Heap-allocated storage is **shared**
- Stack-allocated storage inside the construct is **private**
- Others
 - If there is a **default** clause, what the clause says
 - **none** means that the compiler will issue an error if the attribute is not explicitly set by the programmer
 - Otherwise, depends on the construct
 - For the **parallel** region the default is **shared**



Data-sharing attributes

Example

```
int x,y;
#pragma omp parallel private(y)
{
    x =
    y =
    #pragma omp parallel private(x)
    {
        x =
        y =
    }
}
```



Data-sharing attributes

Example

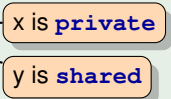
```
int x,y;
#pragma omp parallel private(y)
{
    x = ← x is shared
    y = ← y is private
    #pragma omp private(x)
    {
        x =
        y =
    }
}
```



Data-sharing attributes

Example

```
int x,y;
#pragma omp parallel private(y)
{
    x =
    y =
    #pragma omp parallel private(x)
    {
        x = ← x is private
        y = ← y is shared
    }
}
```



Threadprivate storage

The threadprivate construct

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

- Can be applied to:
 - Global variables
 - Static variables
 - Class-static members
- Allows to create a per-thread copy of “global” variables
- **threadprivate** storage persist across **parallel** regions if the number of threads is the same

Threadprivate persistence across nested regions is complex



Threadprivate storage

Example

```
char* foo ()
{
    static char buffer[BUF_SIZE];
    #pragma omp threadprivate(buffer)

    ...

    return buffer;
}
```



Threadprivate storage

Example

```
char* foo ()  
{  
    static char buffer[BUF_SIZE];  
    #pragma omp threadprivate(buffer) ←  
    ...  
    return buffer;  
}
```

Creates one *static* copy of *buffer* per thread



Threadprivate storage

Example

```
char* foo ()  
{  
    static char buffer[BUF_SIZE];  
    #pragma omp threadprivate(buffer) ←  
    ...  
    return buffer;  
}
```

Now `foo` can be called by multiple threads at the same time



Threadprivate storage

Example

```
char* foo ()  
{  
    static char buffer[BUF_SIZE];  
    #pragma omp threadprivate(buffer)  
  
    ...  
  
    return buffer;←  
}
```

foo returns correct
address to caller



Part II

Worksharing constructs



Outline

- The worksharing concept

- Loop worksharing



Outline

- The worksharing concept
- Loop worksharing



Worksharings

Worksharing constructs divide the execution of a code region among the threads of a team

- Threads **cooperate** to do some work
- Better way to split work than using thread-ids
- Lower overhead than using **tasks**
 - But, less flexible

In OpenMP, there are four worksharing constructs:

- **single**
- loop worksharing
- **section**
- **workshare**

Restriction: worksharings cannot be nested

Outline

- The worksharing concept
- Loop worksharing



Loop parallelism

The for construct

```
#pragma omp for [clauses]  
  for( init-expr ; test-expr ; inc-expr )
```

where clauses can be:

- **private**
- **firstprivate**
- **lastprivate** (*variable-list*)
- **reduction** (*operator:variable-list*)
- **schedule** (*schedule-kind*)
- **nowait**
- **collapse** (*n*)
- **ordered**



The for construct

How it works?

The iterations of the loop(s) associated to the construct are divided among the threads of the team

- Loop iterations must be independent
- Loops must follow a form that allows to compute the number of iterations
- Valid data types for induction variables are: integer types, pointers and random access iterators (in C++)
 - The induction variable(s) are automatically privatized
- The default data-sharing attribute is **shared**

It can be merged with the **parallel** construct:

```
#pragma omp parallel for
```


The for construct

Example

```
void foo (int *m, int N, int M)
{
    int i;
    #pragma omp parallel for private(j)
    for ( i = 0; i < N; i++ )
        for ( j = 0; j < M; j++ )
            m[i][j] = 0;
}
```



The for construct

Example

```
void foo (int *m, int N, int M)
{
    int i;
    #pragma omp parallel for<private>
    for ( i = 0; i < N; i++ )
        for ( j = 0; j < M; j++ )
            m[i][j] = 0;
}
```

New created threads cooperate to execute all the iterations of the loop



The for construct

Example

```
void foo (int *m, int N, int M)
{
  int i;
  #pragma omp parallel for private(i)
  for ( i ← 0;   )
    for ( j = 0, j < M, j++ )
      m[i][j] = 0;
}
```

The *i* variable is automatically privatized



The for construct

Example

```
void foo (int *m, int N, int M)
{
  int i;
  #pragma omp parallel for private(j)
  for ( i = 0; i < N; i++ )
    for ( j ← 0; m[i][j] = 0,

```

Must be explicitly privatized



The for construct

Example

```
void foo ( std::vector<int> &v )
{
    #pragma omp parallel for
    for ( std::vector<int>::iterator it = v.begin() ;
          it < v.end() ;
          it ++ )
        *it = 0;
}
```



The for construct

Example

```
void foo ( std::vector<int> &v )  
{  
    #pragma omp parallel for  
    for ( std::vector<int>::iterator<it  
        it < v.end() ;  
        it ++ )  
        *it = 0;  
}
```

random access iterators
(and pointers) are valid
types



The for construct

Example

```
void foo ( std::vector<int> &v )  
{  
    #pragma omp parallel for  
    for ( std::vector<int>::iterator it = v.begin() :  
          it < v.end()<←; != cannot be used in the test expression  
          it ++ )  
        *it = 0;  
}
```



Removing dependences

Example

```
x = 0;
for ( i = 0; i < n; i++ )
{
    v[ i ] = x;
    x += dx;←
}
```

Each iteration x depends on the previous one. **Can't be parallelized**



Removing dependences

Example

```
x = 0;
for ( i = 0; i < n; i++ )
{
  x = i * dx;
  v[i] = x;
}
```

But x can be rewritten in terms of i .
Now it can be parallelized



The lastprivate clause

When a variable is declared **lastprivate**, a private copy is generated for each thread. Then the value of the variable in the last iteration of the loop is copied back to the original variable

- A variable can be both **firstprivate** and **lastprivate**



The reduction clause

A very common pattern is where all threads accumulate some values into a single variable

- E.g., $n += v[i]$, our `heat` program, ...
- Using `critical` or `atomic` is not good enough
 - Besides being error prone and cumbersome

Instead we can use the `reduction` clause for basic types

- Valid operators are: `+`, `-`, `*`, `|`, `||`, `&`, `&&`, `^`, `min`, `max`
 - User-defined reductions coming soon...
- The compiler creates a `private` copy that is properly initialized
- At the end of the region, the compiler ensures that the `shared` variable is properly (and safely) updated

We can also specify `reduction` variables in the `parallel` construct

The reduction clause

Example

```
int vector_sum (int n, int v[n])
{
    int i, sum = 0;
    #pragma omp parallel for reduction(+:sum)
    {
        for ( i = 0; i < n; i++ )
            sum += v[i];
    }
    return sum;
}
```



The reduction clause

Example

```
int vector_sum (int n, int v[n])
{
    int i, sum = 0;
    #pragma omp parallel for reduction(+:sum)
    {
        for (i = 0; i < n; i++)
            sum += v[i];
    }
    return sum;
}
```

← Private copy initialized here to the identity value

← Shared variable updated here with the partial values of each thread



The schedule clause

The **schedule** clause determines which iterations are executed by each thread

- If no **schedule** clause is present then its implementation is defined

There are several possible options as schedule:

- **STATIC**
- **STATIC, chunk**
- **DYNAMIC [, chunk]**
- **GUIDED [, chunk]**
- **AUTO**
- **RUNTIME**



The schedule clause

Static schedule

The iteration space is broken in chunks of approximately size $N/\text{num} - \text{threads}$. Then these chunks are assigned to the threads in a Round-Robin fashion

Static, N schedule (Interleaved)

The iteration space is broken in chunks of size N . Then these chunks are assigned to the threads in a Round-Robin fashion

Characteristics of static schedules

- Low overhead
- Good locality (usually)
- Can have load imbalance problems

The schedule clause

Dynamic, N schedule

Threads dynamically grab chunks of N iterations until all iterations have been executed. If no chunk is specified, $N = 1$.

Guided, N schedule

Variant of **dynamic**. The size of the chunks decreases as the threads grab iterations, but it is at least of size N . If no chunk is specified, $N = 1$.

Characteristics of dynamic schedules

- Higher overhead
- Not very good locality (usually)
- Can solve imbalance problems

The schedule clause

Auto schedule

In this case, the implementation is allowed to do whatever it wishes

- Do not expect much of it as of now

Runtime schedule

The decision is delayed until the program is run through the **sched-nvar** ICV. It can be set with:

- The **OMP_SCHEDULE** environment variable
- The **omp_set_schedule()** API call



The nowait clause

When a worksharing has a **nowait** clause then the implicit **barrier** at the end of the loop is removed

- This allows to overlap the execution of **non-dependent** loops/tasks/worksharings



The nowait clause

Example

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

First and second loop are independent,
so we can overlap them



The nowait clause

Example

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

Side note: you would better fuse the loops in this case



The nowait clause

Example

```
#pragma omp for nowait←  
for ( i = 0; i < n ; i++ )  
    v[i] = 0;  
#pragma omp for←  
for ( i = 0; i < n ; i++ )  
    a[i] = v[i]*v[i];
```

First and second loops are dependent!
No guarantees that the previous iteration
is finished



The nowait clause

Exception: static schedules

If the two (or more) loops have the same **static** schedule **and** all have the same number of iterations

Example

```
#pragma omp for schedule(static, M) nowait
for ( i = 0; i < n ; i++ )
    v[i] = 0;
#pragma omp for schedule(static, M)
for ( i = 0; i < n ; i++ )
    a[i] = v[i]*v[i];
```



The collapse clause

Allows to distribute work from a set of n nested loops

- Loops must be perfectly nested
- The nest must traverse a rectangular iteration space



The collapse clause

Allows to distribute work from a set of n nested loops

- Loops must be perfectly nested
- The nest must traverse a rectangular iteration space

Example

```
#pragma omp for collapse(2)
for ( i = 0; i < N; i++ )
  for ( j = 0; j < M; j++ )
    foo (i, j);
```

i and *j* loops are folded and iterations distributed among all threads. Both *i* and *j* are privatized





Coffee time! :-)



Part III

Basic Synchronizations



Outline

- Thread barriers
- Exclusive access



Why synchronization?

Mechanisms

Threads need to synchronize to impose some ordering in the sequence of actions of the threads. OpenMP provides different synchronization mechanisms:

- **barrier**
- **critical**
- **atomic**
- **taskwait**
- **ordered**
- **locks**



Outline

- Thread barriers
- Exclusive access



Thread Barrier

The barrier construct

`#pragma omp barrier`

- Threads cannot proceed past a barrier point until all threads reach the barrier **AND** all previously generated work is completed
- Some constructs have an implicit **barrier** at the end
 - E.g., the **parallel** construct



Barrier

Example

```
#pragma omp parallel
{
    foo ();
#pragma omp barrier
    bar ();
}
```



Barrier

Example

```
#pragma omp parallel
{
    foo ();
#pragma omp barrier
    bar ();
}
```

Forces all `foo` occurrences to happen before all `bar` occurrences



Barrier

Example

```
#pragma omp parallel
{
    foo ();
#pragma omp barrier
    bar ();
}←
```

Implicit barrier at the end of the `parallel` region



Outline

- Thread barriers
- Exclusive access



Exclusive access

The critical construct

```
#pragma omp critical [(name)]  
    structured block
```

- Provides a region of mutual exclusion where only one thread can be working at any given time.
- By default all critical regions are the same, but you can provide them with names
 - Only those with the same name synchronize



Critical construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp critical
    x++;
}
printf("%d\n",x);
```



Critical construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp critical
    x++;← Only one thread at a time here
}
printf ("%d\n",x);
```



Critical construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
  #pragma omp critical
  x++; ← Only one thread at a time here
}
printf("%d\n",x); ← Prints 3!
```



Critical construct

Example

```
int x=1,y=0;
#pragma omp parallel num_threads(4)
{
#pragma omp critical (x)
    x++;
#pragma omp critical (y)
    y++;
}
```



Critical construct

Example

```
int x=1,y=0;
#pragma omp parallel num_threads(4)
{
#pragma omp critical (x)
    x++;←
#pragma omp critical (y)
    y++;←
}
```

Different names: One thread can update x while another updates y



Exclusive access

The atomic construct

```
#pragma omp atomic  
    expression
```

- Provides an special mechanism of mutual exclusion to do **read & update** operations
- Only supports simple read & update expressions
 - E.g., `x += 1`, `x = x - foo()`
- Only protects the read & update part
 - `foo()` not protected
- Usually much more efficient than a **critical** construct
- **Not compatible with critical**



Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp atomic
    x++;
}
printf("%d\n",x);
```



Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp atomic
    x++;
}
printf ("%d\n",x);
```

Only one thread at a time updates x here



Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp atomic
    x++;
}
printf("%d\n",x);
```

← Prints 3!



Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp critical
    x++;
#pragma omp atomic
    x++;
}
printf("%d\n",x);
```



Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp critical
    x++;←
#pragma omp atomic
    x++;←
}
printf("%d\n",x);
```

Different threads can update x at the same time!



Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp critical
    x++;
#pragma omp atomic
    x++;
}
printf("%d\n",x);
```

Prints 3,4 or 5 :(



Part IV

Practical: OpenMP heat diffusion



Outline

- Heat diffusion



Outline

- Heat diffusion



Before you start

Enter the [OpenMP](#) directory to do the following exercises

- [Session4.1-exercise](#) contains the serial version of the Heat application
- you can use [SsGrind](#) on heat-tareador to determine parallelism, and observe differences among the three algorithms
- and then annotate the application with OpenMP



Description of the Heat Diffusion app Hands-on

Parallel loops

The file [solver.c](#) implements the computation of the Heat diffusion

- 1 Annotate the [jacobi](#), [redblack](#), and [gauss](#) functions with OpenMP
- 2 Execute the application with different numbers of processors
 - compare the results
 - evaluate the performance





Bon appétit!*

*Disclaimer: actual food may differ from the image! :-)



Part V

Programming using a hybrid MPI/OpenMP approach



Outline

- MPI+OpenMP programming



Outline

- MPI+OpenMP programming



Distributed- vs Shared- Memory Programming

Distributed-memory programming

- Separate processes
 - Private variables are inaccessible from others
- Point-to-point and collective communication
 - Implicit synchronization

Shared-memory programming

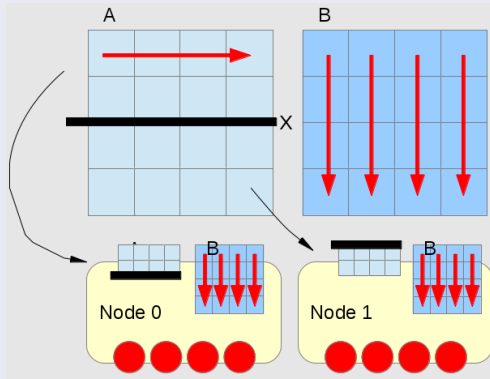
- Multiple threads share same address space
- Explicit synchronization



Hybrid programming

Combining MPI+OpenMP

- Distributed algorithms spread over nodes
- Shared memory for computation within each node



Opportunities

When to use MPI+OpenMP

- Starting from OpenMP and moving to clusters with MPI
- Starting from MPI and exploiting further parallelism inside each node

Improvements

- OpenMP can solve MPI imbalance



Alternatives

MPI + computational kernels in OpenMP

Use OpenMP directives to exploit parallelism between communication phases

- OpenMP parallel will end before new communication calls

MPI inside OpenMP constructs

Call MPI from within for-loops, or tasks

- MPI needs to support multi-threaded mode



Compiling MPI+OpenMP

MPI compiler driver needs the proper OpenMP option

- `mpicc -openmp`
- `mpicc -fopenmp`

Also useful

- `mpicc -show <your command line options and files>`
 - It displays the full command line executed by `mpicc` to compile your program



Part VI

Practical: MPI+OpenMP heat diffusion



- MPI+OpenMP Heat diffusion



Outline

- MPI+OpenMP Heat diffusion



Before you start

Enter the [Session4.2-exercise](#) directory to do the following exercises



Description of the Heat Diffusion app Hands-on

Parallel loops

The file `solver.c` implements the computation of the Heat diffusion

- 1 Use MPI to distribute the work across nodes
- 2 Annotate the `jacobi`, `redblack`, and `gauss` functions with OpenMP tasks
- 3 Execute the application with different numbers of nodes/processors, and compare the results

