INTELLECTUAL PROPERTY RIGHTS NOTICE:

• The User may only download, make and retain a copy of the materials for his/her use for non-commercial and research purposes.
• The User may not commercially use the material, unless has been granted prior written consent by the Licensor to do so; and cannot remove, obscure or modify copyright notices, text acknowledging or other means of identification or disclaimers as they appear.
• For further details, please contact BSC-CNS patc@bsc.es

PRACE TRAINING COURSE
under
PRACE Advance Training Centre at BSC

BSC-CNS http://www.bsc.es/
PRACE project http://www.prace-ri.eu/
PRACE Training Portal http://www.training.prace-ri.eu/
PATC @ BSC Training Program
http://www.bsc.es/marenostrum-support-services/hpc-trainings/prace-trainings
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 - 14:30  Tasking in OpenMP
14:30 - 15:30  Programming using a hybrid MPI/OpenMP approach
15:30 - 16:00  Break
16:00 - 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 3.0 (May 2008)
  - 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

- OpenMP Fortran 1.0: 1997
- OpenMP C/C++ 1.0: 1998
- OpenMP Fortran 1.1: 1999
- OpenMP Fortran 2.0: 2000
- OpenMP C/C++ 2.0: 2002
- OpenMP 2.5: 2005
- OpenMP 3.0: 2008
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
- No error recovery capabilities
- 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

- Constructs
- Compiler
- OpenMP Exec
- OpenMP API
- Environment Variables
- OpenMP Runtime Library
- ICVs
- OS Threading Libraries

CPU

SMP
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

Master Thread

- Parallel Region
- Nested Parallel Region
- Parallel Region
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 syntax 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 OpenMP enabled compiler
  - 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
Creating Threads

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)`
- `reduction(var-list)`
- `copyin(var-list)`

Coming shortly!

Only in Fortran

We’ll see it later

Not today

Xavier Martorell (BSC)
PATC Parallel Programming Workshop
November 26-30, 2012
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
Creating Threads

Putting it together

Example

```c
void main () {
    #pragma omp parallel
    ...
    omp_set_num_threads(2);
    #pragma omp parallel
    ...
    #pragma omp parallel num_threads(random()%4+1) if(0)
    ...
}
```
Putting it together

Example

```c
void main () {
    #pragma omp parallel
    ...
    omp_set_num_threads(2);
    #pragma omp parallel
    ...
    #pragma omp parallel num_threads(random()%4+1) if(0)
    ...
}
```

An unknown number of threads here. Use `OMP_NUM_THREADS`
Putting it together

Example

```c
void main () {
    #pragma omp parallel
    ...
    omp_set_num_threads(2);
    #pragma omp parallel
    ...
    #pragma omp parallel num_threads(random()%4+1) if (0)
    ...
}
```

A team of two threads here.
Putting it together

Example

```c
void main () {
    #pragma omp parallel
    ...
    omp_set_num_threads(2);
    #pragma omp parallel
    ...
    #pragma omp parallel num_threads(random()%4+1) if(0)
    ...
}  // A team of 1 thread here.
```
### Other useful routines

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

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

We’ll see them later

Out of our scope today
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

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

Example

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

Prints 2 or 3
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

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

Example

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

Can print anything
Data-sharing attributes

Example

```c
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 variable value.

- 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

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

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

Prints 2 (twice)
Data-sharing attributes

Example

```c
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

```c
int x, y;
#pragma omp parallel private(y)
{
    x = y =
    #pragma omp parallel private(x)
    {
        x = y =
    }
}
```
**Data-sharing attributes**

**Example**

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

- `x` is **shared**
- `y` is **private**
Data-sharing attributes

Example

```c
int x, y;
#pragma omp parallel private(y)
{
    x =
    y =
    #pragma omp parallel private(x)
    {
        x = \textcolor{red}{\textbf{x is private}}
        y = \textcolor{blue}{\textbf{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
Data-sharing attributes

Threaprivate storage

Example

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

    ...

    return buffer;
}
```

Creates one static copy of buffer per thread
Now foo can be called by multiple threads at the same time
Data-sharing attributes

Threaprivate storage

**Example**

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

    ...

    return buffer;
}
```

*Creates one *static* copy of* `buffer` *per thread*
Data-sharing attributes

Threaprivate storage

Example

```c
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.
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 worksharing

Loop parallelism

The for construct

```c
#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`  
  We’ll see it later

Xavier Martorell (BSC)

PATC Parallel Programming Workshop

November 26-30, 2012
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 inductions 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

```c
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;
}
```
Loop worksharing

The for construct

```c
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;
}
```

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

Example
Loop worksharing

The for construct

Example

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

The `i` variable is automatically privatized

Must be explicitly privatized
The for construct

Example

```c
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 `i` variable is automatically privatized

Must be explicitly privatized
The for construct

Example

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

The for construct

Example

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

random access iterators (and pointers) are valid types
The for construct

Example

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

Random access iterators (and pointers) are valid types. `!=` cannot be used in the test expression.
Removing dependences

Example

```c
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

```c
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 \textit{pi} program, ...
- Using \textit{critical} or \textit{atomic} is not good enough
  - Besides being error prone and cumbersome

Instead we can use the \textit{reduction} clause for basic types.

- Valid operators are: \(+, -, *, |, ||, &\), \&\&, ^
- The compiler creates a \textit{private} copy that is properly initialized
- At the end of the region, the compiler ensures that the \textit{shared} variable is properly (and safely) updated.

We can also specify \textit{reduction} variables in the \textit{parallel} construct.
Loop worksharing

The reduction clause

Example

```c
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

```c
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
Loop worksharing

The schedule clause

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

- If no schedule clause is present then is implementation 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 $\frac{N}{\text{num} - \text{threads}}$. 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
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

```c
#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

```c
#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;
```

On a side note, you would be better by fusing the loops in this case
The nowait clause

Example

```c
#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 loop 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

```c
#pragma omp for schedule(static,2) nowait
for ( i = 0; i < n ; i++ )
    v[i] = 0;
#pragma omp for schedule(static,2)
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
Loop worksharing

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

```c
#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

We’ll see them later
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

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

Forces all foo occurrences to happen before all bar occurrences. Implicit barrier at the end of the parallel region.
Thread barriers

**Barrier**

Example

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

Forces all `foo` occurrences too happen before all `bar` occurrences
Thread barriers

Barrier

Example

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

Implicit barrier at the end of the `parallel` region
Outline

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

Example
Example

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

Only one thread at a time here
**Critical construct**

**Example**

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

*Only one thread at a time here*

*Prints 3!*
Critical construct

Example

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

```c
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
```
The atomic construct

```c
#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

```c
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

Prints 3!
Atomic construct

Example

```c
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

```c
int x = 1;
#pragma omp parallel num_threads(2)
{
#pragma omp atomic
    x ++;
}
printf("%d\n", x);  \rightarrow  Prints 3!
```

Only one thread at a time updates `x` here

Prints 3!
Atomic construct

Example

```c
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 :(
Atomic construct

Example

```c
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

```c
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: heat diffusion
Outline

- Heat diffusion
Outline

- Heat diffusion
Before you start

Enter the OpenMP 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. Annotate the `jacobi`, `redblack`, and `gauss` functions with OpenMP
2. Execute the application with different numbers of processors, and compare the results
Bon appétit!*

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

Task Parallelism in OpenMP
Outline

- OpenMP tasks
- Task synchronization
- The single construct
- Task clauses
- Common tasking problems
Outline

- OpenMP tasks
- Task synchronization
- The single construct
- Task clauses
- Common tasking problems
Task parallelism in OpenMP

Task parallelism model

- Parallelism is extracted from “several” pieces of code
- Allows to parallelize very unstructured parallelism
  - Unbounded loops, recursive functions, ...

Team

Task pool
What is a task in OpenMP?

- Tasks are work units whose execution **may** be deferred
  - they can also be executed immediately

- Tasks are composed of:
  - **code** to execute
  - a **data** environment
    - Initialized at creation time
  - internal control variables (**ICVs**)

- Threads of the team **cooperate** to execute them
Creating tasks

The task construct

```c
#pragma omp task [clauses]
structed block
```

Where clauses can be:

- shared
- private
- firstprivate
  - Values are captured at creation time
- default
- `if(expression)`
- untied
OpenMP tasks

When are tasks created?

- **Parallel** regions create tasks
  - One **implicit** task is created and assigned to each thread
    - So all task-concepts have sense inside the parallel region
  - Each thread that encounters a **task** construct
    - Packages the code and data
    - Creates a new **explicit** task
Default task data-sharing attributes

When there are no clauses ...

If no default clause

- **Implicit rules** apply
  - e.g., global variables are shared

Otherwise...

- **firstprivate**
- **shared** attribute is lexically inherited
Task default data-sharing attributes

In practice...

Example

```c
int a;
void foo() {
    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;

            a = b = c = d = e =
        }
    }
```
Task default data-sharing attributes

In practice...

Example

```c
int a;
void foo() {
    int b, c;
    #pragma omp parallel shared (b)
    #pragma omp parallel private (b)
    {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b =
            c =
            d =
            e =
        }
    }
```

Xavier Martorell (BSC)
Task default data-sharing attributes

In practice...

Example

```c
int a;
void foo() {
    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b = firstprivate
            c =
            d =
            e =
        }
    }
}
```
Task default data-sharing attributes

In practice...

Example

```c
int a;
void foo() {
    int b, c;
#pragma omp parallel shared(b)
#pragma omp parallel private(b)
{
    int d;
#pragma omp task
{
    int e;

    a = shared
    b = firstprivate
    c = shared
    d =
    e =
}
}
```
Task default data-sharing attributes

In practice...

Example

```c
int a;
void foo() {
    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b = firstprivate
            c = shared
            d = firstprivate
            e =
        }
    }
}}
```
Task default data-sharing attributes

In practice...

Example

```c
int a;
void foo() {
    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;
            a = shared
            b = firstprivate
            c = shared
            d = firstprivate
            e = private
        }
    }
}
```
Task default data-sharing attributes

In practice...

Example

```c
int a;
void foo() {
    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;
            a = shared
            b = firstprivate
            c = shared
            d = firstprivate
            e = private
        }
    }
```

Tip: `default (none)` is your friend if you do not see it clearly
List traversal

Example

```c
void traverse_list ( List l )
{
    Element e;
    for ( e = l->first ; e ; e = e->next )
        #pragma omp task
        process(e);
}
```

e is firstprivate
Outline

- OpenMP tasks
- Task synchronization
- The single construct
- Task clauses
- Common tasking problems
Task synchronization

There are two main constructs to synchronize tasks:

- **barrier**
  - Remember: all previous work (including tasks) must be completed
- **taskwait**
Task synchronization

Waiting for children

The taskwait construct

```
#pragma omp taskwait
```

Suspends the current task until all *children* tasks are completed

- Just direct children, not descendants
Task synchronization

**Taskwait**

---

**Example**

```c
void traverse_list ( List l )
{
    Element e;
    for ( e = l->first ; e ; e = e->next )
        #pragma omp task
        process(e);

    #pragma omp taskwait

    All tasks guaranteed to be completed here
}
```
Task synchronization

Taskwait

Example

```c
void traverse_list ( List l )
{
    Element e;
    for ( e = l->first ; e ; e = e->next )
        #pragma omp task
        process(e);
    #pragma omp taskwait
}
```

Now we need some threads to execute the tasks
List traversal
Completing the picture

Example

```
List l

#pragma omp parallel
traverse_list(l);
```
List traversal
Completing the picture

Example

```c
List l
#pragma omp parallel
traverse_list(l);
```

This will generate multiple traversals
List traversal
Completing the picture

Example

```
List l

#pragma omp parallel
traverse_list(l);
```

We need a way to have a single thread execute `traverse_list`
Outline

- OpenMP tasks
- Task synchronization
- The single construct
- Task clauses
- Common tasking problems
The single construct

The single construct

#pragma omp single [clauses]
structured block

where clauses can be:

- private
- firstprivate
- nowait
- copyprivate

Only one thread of the team executes the structured block

There is an implicit barrier at the end
The single construct

Example

```c
int main (int argc, char **argv )
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("Hello world!\n");
        }
    }
}
```
The single construct

Example

```c
int main ( int argc, char **argv )
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("Hello world!\n");
        }
    }
}
```

This program outputs just one "Hello world"
List traversal
Completing the picture

Example

List l

#pragma omp parallel
#pragma single
traverse_list(l);
List traversal
Completing the picture

Example

```
List l
#pragma omp parallel
#pragma single
traverse_list(l);
```

One thread creates the tasks of the traversal
List traversal
Completing the picture

Example

List l

#pragma omp parallel
#pragma single
traverse_list(l);

All threads cooperate to execute them
Outline

- OpenMP tasks
- Task synchronization
- The single construct
- Task clauses
- Common tasking problems
Task scheduling

How it works?

Tasks are **tied** by default

- **Tied** tasks are executed always by the **same thread**
  - Not necessarily the creator
- **Tied** tasks have **scheduling restrictions**
  - Deterministic scheduling points (creation, synchronization, ... )
    - Tasks can be suspended/resumed at these points
  - Another constraint to avoid deadlock problems
- **Tied** tasks may run into performance problems
The untied clause

A task that has been marked as **untied** has none of the previous scheduling restrictions:

- Can *potentially* switch to any thread
- Can *potentially* switch at any moment
- Bad mix with thread based features
  - thread-id, critical regions, threadprivate
- Gives the runtime more flexibility to schedule tasks
The if clause

- If the expression of an `if` clause evaluates to `false`:
  - The encountering task is suspended.
  - The new task is executed immediately:
    - with its own data environment.
    - different task with respect to synchronization.
  - The parent task resumes when the task finishes.
  - Allows implementations to optimize task creation:
    - For very fine grain task you may need to do your own if
Outline

- OpenMP tasks
- Task synchronization
- The single construct
- Task clauses

Common tasking problems
Common tasking problems

Search problem

Example

```c
void search (int n, int j, bool *state) 
{
    int i, res;
    
    if (n == j) {
        /* good solution, count it */
        solutions++;
        return;
    } 
    
    /* try each possible solution */
    for (i = 0; i < n; i++)
    {
        state[j] = i;
        if (ok(j+1, state)) {
            search(n, j+1, state);
        }
    }
}
```
Common tasking problems

Search problem

Example

```c
void search (int n, int j, bool *state) {
    int i, res;

    if (n == j) {
        /* good solution, count it */
        solutions++;
        return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
        #pragma omp task
        {
            state[j] = i;
            if (ok(j+1,state)) {
                search(n,j+1,state);
            }
        }
}
```
Common tasking problems

Search problem

Example

```c
void search (int n, int j, bool *state)
{
    int i, res;

    if (n == j) {
        /* good solution, count it */
        solutions++;
        return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
        #pragma omp task
        {
            state[j] = i;
            if (ok(j+1, state)) {
                search(n, j+1, state);
            }
        }
}
```

Data scoping

Because it’s an orphaned task all variables are firstprivate
Common tasking problems

Search problem

Example

```c
void search (int n, int j, bool *state)
{
    int i, res;

    if (n == j) {
        /* good solution, count it */
        solutions++;
        return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
        #pragma omp task
        {
            state[j] = i;
            if (ok(j+1, state)) {
                search(n, j+1, state);
            }
        }
}
```

Data scoping

Because it’s an orphaned task all variables are firstprivate

State is not captured

Just the pointer is captured not the pointed data
Common tasking problems

Search problem

Example

```c
void search (int n, int j, bool *state)
{
    int i, res;

    if (n == j) {
        /* good solution, count it */
        solutions++;
        return;
    } /* try each possible solution */

    for (i = 0; i < n; i++)
        #pragma omp task
        {
            state[j] = i;
            if (ok(j+1, state)) {
                search(n, j+1, state);
            }
        }
}
```

Problem #1

Incorrectly capturing pointed data
Problem #1
Incorrectly capturing pointed data

**Problem**

`firstprivate` does not allow to capture data through pointers

**Solutions**

1. Capture it manually
2. Copy it to an array and capture the array with `firstprivate`
Common tasking problems

Search problem

Example

```c
void search (int n, int j, bool *state)
{
    int i, res;
    if (n == j) {
        /* good solution, count it */
        solutions++;
        return;
    }
    /* try each possible solution */
    for (i = 0; i < n; i++)
        #pragma omp task
        {
            bool *new_state = alloca(sizeof(bool)*n);
            memcpy(new_state, state, sizeof(bool)*n);
            new_state[j] = i;
            if (ok(j+1,new_state)) {
                search(n,j+1,new_state);
            }
        }
}
```

Common tasking problems

Search problem

Example

```c
void search (int n, int j, bool *state)
{
    int i, res;

    if (n == j) {
        /* good solution, count it */
        solutions++;
        return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
        #pragma omp task
        {
            bool *new_state = (bool *)malloc(sizeof(bool) * n);
            memcpy(new_state, state, sizeof(bool) * n);
            new_state[j] = i;
            if (ok(j + 1, new_state)) {
                search(n, j + 1, new_state);
            }
        }
}
```

Caution!
Will new_state still be valid by the time memcpy is executed?
Common tasking problems

Search problem

Example

```c
void search (int n, int j, bool *state)
{
    int i, res;

    if (n == j) {
        /* good solution, count it */
        solutions++;
        return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
        #pragma omp task
        {
            bool *new_state = alloca(sizeof(bool)*n);
            memcpy(new_state, state, sizeof(bool)*n);
            new_state[j] = i;
            if (ok(j+1, new_state)) {
                search(n, j+1, new_state);
            }
        }
}
```

Problem #2

Data can go out of scope!
Problem #2
Out-of-scope data

Problem
Stack-allocated parent data can become invalid before being used by child tasks
- Only if not captured with firstprivate

Solutions
1. Use firstprivate when possible
2. Allocate it in the heap
   - Not always easy (we also need to free it)
3. Put additional synchronizations
   - May reduce the available parallelism
Common tasking problems

Search problem

Example

```c
void search (int n, int j, bool *state) {
    int i, res;
    if (n == j) {
        /* good solution, count it */
        solutions++;
        return;
    }
    /* try each possible solution */
    for (i = 0; i < n; i++)
        #pragma omp task
        {
            bool *new_state = alloca(sizeof(bool)*n);
            memcpy(new_state, state, sizeof(bool)*n);
            new_state[j] = i;
            if (ok(j+1, new_state)) {
                search(n, j+1, new_state);
            }
        }
        #pragma omp taskwait
}
```
Search problem

Example

```c
void search (int n, int j, bool *state) {
    int i, res;

    if (n == j) {
        /* good solution, count it: */
        solutions++;
        return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
        #pragma omp task
        {
            bool *new_state = alloca(sizeof(bool)*n);
            memcpy(new_state, state, sizeof(bool)*n);
            new_state[j] = i;
            if (ok(j+1, new_state)) {
                search(n, j+1, new_state);
            }
        }

    #pragma omp taskwait
}
```

*Shared variable needs protected access*
Search problem

Example

```c
void search (int n, int j, bool *state) {
    int i, res;

    if (n == j) {
        /* good solution, count it */
        solutions++;
        return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
        #pragma omp task
        {
            bool *new_state = alloca(sizeof(bool)*n);
            memcpy(new_state, state, sizeof(bool)*n);
            new_state[j] = i;
            if (ok(j+1, new_state)) {
                search(n, j+1, new_state);
            }
        }

    #pragma omp taskwait
}
```

Solutions
- Use `critical`
- Use `atomic`
- Use `threadprivate`
Common tasking problems

Reductions for tasks

Example

```c
int solutions = 0;
int mysolutions = 0;
#pragma omp threadprivate(mysolutions)

void start_search ()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            bool initial_state[n];
            search(n, 0, initial_state);
        }
        #pragma omp atomic
        solutions += mysolutions;
    }
}
```

- Use a separate counter for each thread
- Accumulate them at the end
Common tasking problems

Search problem

Example

```c
void search (int n, int j, bool *state)
{
    int i, res;

    if (n == j) {
        /* good solution, count it */
        mysolutions++;
        return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
        #pragma omp task
        {
            bool *new_state = alloc(sizeof(bool)*n);
            memcpy(new_state, state, sizeof(bool)*n);
            new_state[j] = i;
            if (ok(j+1, new_state)){
                search(n, j+1, new_state);
            }
        }
    
    #pragma omp taskwait
}
```
Part VI

Programming using a hybrid MPI/OpenMP approach
Outline

- MPI+OpenMP programming
Outline

- MPI+OpenMP programming
MPI + OpenMP programming

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 gets the proper OpenMP option

- mpicc -openmp
- mpicc -fopenmp
Coffee time! :-)

![Coffee Image]
Part VII

Practical: heat diffusion
Outline

- MPI+OpenMP Heat diffusion
Outline

- MPI+OpenMP Heat diffusion
Before you start

Enter the MPI+OpenMP 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