OpenMP loops
Outline

- Expressing parallelism
  - Understanding parallel threads

- Memory Data management
  - Data clauses

- Synchronization
  - Barriers, locks, critical sections

- Work partitioning
  - Loops, sections, single work, tasks…

- Execution devices
  - Target
What we saw so far..

✓ Threads
  – How to create and properly manage a team of threads
  – How to join them with barriers

✓ Memory
  – How to create private and shared variables storages
  – How to properly ensure memory consistency among parallel threads

✓ Data synchronization
  – How to create locks to implement, e.g., mutual exclusion
  – How to identify Critical Sections
  – How to ensure atomicity on single statements
Work sharing between threads

✓ But..how can we split an existing workload among parallel threads?
  – Say, a loop

✓ Typical scenario
  1. Analyze sequential code from customer/boss
  2. Parallelize it with OpenMP (for a "generic" parallel machine)
  3. Tune num_threads for specific machine
  4. Get money/congratulations from customer/boss

✓ Might not be as easy as with PI Montecarlo!

How to do 2. without rewriting/re-engineering the code?
Exercise

✓ Create an array of $N$ elements
  – Put inside each array element its index, multiplied by '2'
  – $\text{arr}[0] = 0; \text{arr}[1] = 2; \text{arr}[2] = 4; \ldots$ and so on..

✓ Now, do it in parallel with a team of $T$ threads
  – $N = 19$, $T \neq 19$, $N < T$
  – Hint: Act on the boundaries of the loop
  – Hint #2: `omp_get_thread_num()`, `omp_get_num_threads()`

✓ Example:

<table>
<thead>
<tr>
<th>T = 3</th>
<th>T = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>13</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
</tr>
</tbody>
</table>
Loop partitioning among threads

- Case #1: \( N \) multiple of \( T \)
  - Say, \( N = 20, \ T = 4 \)
- \( chunk = \) #iterations for each thread
- Very simple..

\[ chunk = \frac{N}{T}; \]

\[ i_{\text{start}} = \text{thread}_{ID} \times chunk; \quad i_{\text{end}} = i_{\text{start}} + chunk \]
Loop partitioning among threads

✓ Case #2: $N$ not multiple of $T$
  
  - Say, $N = 19$, $T = 4$

✓ $\text{chunk} = \#\text{iterations for each thread (but last)}$
  
  - Last thread has less! ($\text{chunk}_{\text{last}}$)

\[
\text{chunk} = \frac{N}{T} + 1; \quad \text{chunk}_{\text{last}} = N \% \text{chunk}
\]

\[
i_{\text{start}} = \text{thread}_{\text{ID}} \times \text{chunk}; \quad i_{\text{end}} = \begin{cases} 
  i_{\text{start}} + \text{chunk} & \text{if not last thread} \\
  i_{\text{start}} + \text{chunk}_{\text{last}} + 1 & \text{if last thread}
\end{cases}
\]
Unfortunately, we don't know which thread will be "last" in time

But...we don't actually care the order in which iterations are executed
  - If there are not dependencies..
  - And...we do know that

\[0 \leq \text{omp\_get\_thread\_num()} < \text{omp\_get\_num\_threads()}\]

We choose that last thread as highest number
Let's put them together!

✓ Case #1 (N not multiple of T)

\[ \text{chunk} = \frac{N}{T} \]

\[ \text{i}_{\text{start}} = \text{thread}_{ID} \times \text{chunk}; \]

\[ \text{i}_{\text{end}} = \text{i}_{\text{start}} + \text{chunk} \]

✓ Case #2 (N multiple of T)

\[ \text{chunk} = \frac{N}{T} + 1; \quad \text{chunk}_{\text{last}} = N \mod \text{chunk} \]

\[ \text{i}_{\text{start}} = \text{thread}_{ID} \times \text{chunk}; \]

\[ \text{i}_{\text{end}} = \begin{cases} \text{i}_{\text{start}} + \text{chunk} & \text{if not last thread} \\
\text{i}_{\text{start}} + \text{chunk}_{\text{last}} + 1 & \text{if last thread} \end{cases} \]
✓ A way to distribute work among parallel threads
  – In a simple, and "elegant" manner
  – Using pragmas

✓ OpenMP was born for this
  – OpenMP 2.5 targets regular, loop-based parallelism

✓ OpenMP 3.x targets irregular/dynamic parallelism
  – We will see it later
The **for** construct

```c
#pragma omp for [clause [[,] clause]...] new-line
  for-loops
```

Where clauses can be:

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `linear(list[ : linear-step])`
- `reduction(reduction-identifier : list)`
- `schedule([modifier [, modifier]:]kind[, chunk_size])`
- `collapse(n)`
- `ordered[(n)]`
- `nowait`

- The iterations will be executed in parallel by threads in the team
- The iterations are distributed across threads executing the parallel region to which the loop region binds
- **for-loops** must have **Canonical loop form**
Canonical loop form

\[
\text{for (init-expr; test-expr; incr-expr)} \\
\text{structured-block}
\]

✓ init-expr; test-expr; incr-expr not void

✓ Eases programmers' life
  – More structured
  – Recommended also for "sequential programmers"

✓ Preferrable to while and do..while
  – If possible
Create an array of \( N \) elements
- Put inside each array element its index, multiplied by '2'
- \( \text{arr}[0] = 0; \text{arr}[1] = 2; \text{arr}[2] = 4; \) ...and so on..

Now, do it in parallel with a team of \( T \) threads
- Using the \texttt{for} construct
#pragma omp for [clause [,] clause]... new-line
  for-loops

Where clauses can be:

private(list)
firstprivate(list)
lastprivate(list)
linear(list[ : linear-step])
reduction(reduction-identifier : list)
schedule([modifier [, modifier]:]kind[, chunk_size])
collapse(n)
ordered[(n)]
nowait

✓ first/private, reduction we already know...
  – Private storage, w/ or w/o initialization
✓ linear, we won't see
The lastprivate clause

✓ A list item that appears in a lastprivate clause is subject to the private clause semantics

✓ Also, the value is updated with the one from the sequentially last iteration of the associated loops
lastprivate variables and memory

✓ Create a new storage for the variables, local to threads, and initialize

```c
int a = 11;

#pragma omp for lastprivate(a) \ num_threads(4)
{
    a = ...;
}
```

Process memory
Create a new storage for the variables, local to threads, and initialize

```c
int a = 11;

#pragma omp for lastprivate(a) num_threads(4)
{
    a = ...
}
```
Create a new storage for the variables, local to threads, and initialize

```c
int a = 11;

#pragma omp for lastprivate(a) \
num_threads(4)
{
    a = ...  
}
```
Create a new storage for the variables, local to threads, and initialize

```c
int a = 11;

#pragma omp for lastprivate(a) \
num_threads(4)
{
    a = ... 
}
```
Modify the "PI Montecarlo" exercise
  - Use the for construct

Up to now, each threads executes its "own" loop
  - i from 0 to 2499

Using the for construct, they actually share the loop
  - No need to modify the boundary!!!
  - Check it with printf
Create an array of \( N \) elements
- Put inside each array element its index, multiplied by '2'
- \( arr[0] = 0; \ arr[1] = 2; \ arr[2] = 4; \) ...and so on..

Declare the array as lastprivate
- So you can print its value after the parreg, in the sequential zone
- Do this at home
OpenMP provides three work-sharing constructs

- Loops
- Single
- Sections
The single construct

```c
#pragma omp single [clause [,, clause]...] new-line
  structured-block
```

Where clauses can be:

- `private(list)`
- `firstprivate(list)`
- `copyprivate(list)`
- `nowait`

- The enclosed block is executed by only one threads in the team
- ..and what about the other threads?
Each worksharing construct has an implicit barrier at its end
- Example: a loop
- If one thread is delayed, it prevents other threads to do useful work!!

```c
#pragma omp parallel num_threads(4)
{
    #pragma omp for
    for(int i=0; i<N; i++)
    {
        ...
    } // (implicit) barrier

    // USEFUL WORK!!

} // (implicit) barrier
```
Each worksharing construct has an implicit barrier at its end

- Example: a loop
- If one thread is delayed, it prevents other threads to do useful work!!
Nowait clause in the for construct

The **nowait** clause removes the barrier at the end of a worksharing (WS) construct

- Applies to all of WS constructs
- Does not apply to parregs!
Worksharing constructs and barriers

✓ Removed the barrier at the end of WS construct
  – Still, there is a barrier at the end of parreg

```c
#pragma omp parallel num_threads(4) \ nowait
{
  #pragma omp for
  for(int i=0; i<N; i++)
  {
    ...
  }
  // no barrier
  // USEFUL WORK!!
}
  // (implicit) barrier
```
Each section contains code that is executed by a single thread
  – A "switch" for threads
✓ Clauses, we already know..
  – lastprivate items are updated by the section executing last (in time)
Sections vs. loops

✓ Loops implement data-parallel paradigm
  – Same work, on different data
  – Aka: data decomposition, SIMD, SPMD

✓ Sections implement task-based paradigm
  – Different work, on the same or different data
  – Aka: task decomposition, MPSD, MPMD
The master construct

```c
#pragma omp master new-line
    structured-block
```

- The structured block is executed only by master thread
  - "Similar" to the single construct

- It is not a work-sharing construct
  - There is no barrier implied!!
✓ For each WS construct, there is also a compact form
  – In this case, clauses to both constructs apply

```c
#pragma omp parallel
{
  #pragma omp for
  for(int i=0; i<N; i++)
  {
    ...
  }
} // (implicit) barrier
```

```c
#pragma omp parallel
#pragma omp for
for(int i=0; i<N; i++)
{
  ...
} // (implicit) barrier
```

```c
#pragma omp parallel for
for(int i=0; i<N; i++)
{
  ...
} // (implicit) barrier
```
How to run the examples

✓ Download the Code/ folder from the course website

✓ Compile
✓ $ gcc -fopenmp code.c -o code

✓ Run (Unix/Linux)
$ ./code
✓ Run (Win/Cygwin)
$ ./code.exe
References

✓ "Calcolo parallelo" website
  – http://algogroup.unimore.it/people/marko/courses/programmazione_parallela/

✓ My contacts
  – paolo.burgio@unimore.it
  – http://hipert.mat.unimore.it/people/paolob/

✓ OpenMP specifications
  – http://www.openmp.org

✓ A lot of stuff around…
  – https://computing.llnl.gov/tutorials/openMP/