**Pthreads**

- `pthread_create`, `pthread_exit`, `pthread_join`
- **Mutex**: locked/unlocked; used to protect access to shared variables (read/write)
- **Condition variables:**
  - used to allow threads to become idle and wake up when a condition becomes true.
  - used in combination with a mutex to protect access to the condition variable (boolean)
  - `cond_wait`
  - `cond_signal`
OpenMP
**Pthreads/OpenMP**

- Pthreads gives you maximum flexibility.
- It’s a low level API that allows you to implement pretty much any parallel computation exactly the way you want it.
- However, in many cases, the user only wants to parallelize certain common situations:
  - For loop: partition the loop into chunks and have each thread process one chunk.
  - Hand-off a block of code (computation) to a separate thread
- This is where OpenMP is useful. It simplifies the programming significantly.
- In some cases, adding one line in a C code is sufficient to make it run in parallel.
- As a result, OpenMP is the standard approach in scientific computing for multicore processors.
**OpenMP**

What is OpenMP?

- OpenMP is an Application Programming Interface (API), jointly defined by a group of major computer hardware and software vendors.
- OpenMP provides a portable, scalable model for developers of shared memory parallel applications.
- The API supports C/C++ and Fortran on a wide variety of architectures.

Hence, it is more portable and general than Pthreads.

- OpenMP website: openmp.org
- Wikipedia: en.wikipedia.org/wiki/OpenMP
ARB: Architecture Review Board
COMPILING YOUR CODE

First things first
• Header file:
  `#include <omp.h>`

• This is only needed if you explicitly use the OpenMP API.
• Compiler flags:

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>icc</td>
<td>-openmp</td>
</tr>
<tr>
<td>icpc</td>
<td></td>
</tr>
<tr>
<td>ifort</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>-fopenmp</td>
</tr>
<tr>
<td>g++</td>
<td></td>
</tr>
<tr>
<td>g77</td>
<td></td>
</tr>
<tr>
<td>gfortran</td>
<td></td>
</tr>
</tbody>
</table>
Parallel regions
DIRECTIVES

• OpenMP is based on directives.
• Powerful because of simple syntax.
• Dangerous because you may not understand exactly what the compiler is doing.
The most basic directive is

```c
#pragma omp parallel
{ // structured block ... }
```

This starts a new parallel region. OpenMP follows a fork-join model:

Upon entering a region, if there are no further directives, a team of threads is created and all threads execute the code in the parallel region.
**Basic Example**

```plaintext
g++ -o hello_world hello_world_openmp.cc -fopenmp
```
**Pi algorithm**

In our code, Pi is computed using:

\[
\frac{\pi}{2} = 1 + \frac{1}{3} \left( 1 + \frac{2}{5} \left( 1 + \frac{3}{7} \left( 1 + \frac{4}{9} \left( 1 + \cdots \right) \right) \right) \right)
\]

Using this expansion, can you show that the code computes the digits of Pi, *4 at a time*, assuming that:

\[
\text{carry} + \text{sum} / \text{SCALE} < 10,000
\]
• Is the previous algorithm parallel?
• Is this a good multicore implementation?
• How would you improve it?
• Computing pi in parallel is difficult.
• Many algorithms use sequential calculations using high-precisions arithmetic, that is you compute using numbers with a lot of digits.
• This leads to the natural question:

**Is it possible to compute the n-th digit of pi independently from the others?**
\[ \pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left( \frac{4}{8k + 1} - \frac{2}{8k + 4} - \frac{1}{8k + 5} - \frac{1}{8k + 6} \right) \]
COMPUTING THE N-TH BIT

This problem can now be reformulated as:
Can we compute the fractional part of

\[ 16^n \pi \]

Take:

\[
\sum_{k=0}^{\infty} \frac{16^{n-k}}{8k + 1} = \sum_{k=0}^{n} \frac{16^{n-k}}{8k + 1} + \sum_{k=n+1}^{\infty} \frac{16^{n-k}}{8k + 1}
\]

Only a few terms are needed

whole numbers can be removed
COMPUTING THE FIRST SUM

\[ \sum_{k=0}^{n} \frac{16^{n-k}}{8k + 1} \]

This can be easily computed
Clause

• This is one of the tricky points of OpenMP.
• Recall in Pthreads that:
  • Variables passed as argument to a thread are shared (they might be pointers in a struct for example)
  • Variables inside the function that a thread is executing are private to the thread.
• OpenMP needs a similar mechanism: some variables are going to be shared (all threads can read and write), others need to be private.
• There are (complicated) rules to figure out whether a variable is private or shared.
**SHARED/PRIVATE**

- See `shared_private_openmp.cc`

- In a parallel construct, variables defined outside are shared by default.

- You can declare explicitly whether a variable is shared or private using
  ```
  private(variable_name)
  shared(variable_name)
  ```
the value of a private variable like should _be_ after the program exits the parallel region?

The value it had during the last assignment on line 36

**Undefined**

*Start the presentation to activate live content*

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app
What is the problem with read_write?

All threads are trying to write

All threads are trying to read

All threads are trying to modify its value

Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app
**SHARED VARIABLE**

- Those are typically READ ONLY.
- Using a shared variable w/ READ/WRITE is dangerous.

Variable refers to the same memory location for all threads.
PRIVATE VARIABLE

Memory

Variable refers to a different memory location for each thread. Those variables are typically READ/WRITE.
WORKSHARING CONSTRUCTS
**Parallel for Loop**

The most common approach to parallelize a computation on a multicore processor is to parallelize a `for` loop.

OpenMP has some special constructs to do that.

```c
#pragma omp for [clause [clause] ... ]
for (i = lower bound; i op upper bound; incr expr) {
  ...
}
```

![Diagram showing parallel execution of a loop across threads](image.png)
Example

• Let’s consider again the matrix-matrix example we used for Pthreads.

• See matrix_prod_openmp.cc

• One line of code is sufficient to parallelize the calculation! This is the power of OpenMP.

$ ./matrix_prod_openmp -n 4000 -p 32
$ top
Is it possible to move `#pragma omp parallel for` to line 122?

Yes, it makes no difference.

Yes but this is not recommended.
SCHEDULING FOR LOOPS

• How are the iterates in a for loop split among threads? This is important to fine-tune the optimization of your code.
• This is a problem of load-balancing: how should we distribute the work so that we minimize the total execution time?

1. schedule(\texttt{static}, \texttt{block\_size}): iterations are divided into pieces of size \texttt{block\_size} and then statically assigned to threads. This is the best \texttt{default} option.

2. schedule(\texttt{dynamic}, \texttt{block\_size}): iterations are divided into pieces of size \texttt{block\_size}, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. This is useful when the work per iteration is irregular.

3. schedule(\texttt{guided}, \texttt{block\_size}): specifies a dynamic scheduling of blocks but with decreasing size. It is appropriate for the case in which the threads arrive at varying times at a \texttt{for} construct (with each iteration requiring about the same amount of work).
Other worksharing constructs: sections

• There are situations where two independent pieces of work can be executed concurrently. For example, we may need to update two vectors independently.
• In that case, we would like to assign one thread to do each operation in parallel.
• This can be done using sections.
• The compiler is allowed to schedule the execution of the code inside each section concurrently.

• See section.cc
Default number of threads (24)

Two threads

Start the presentation to activate live content
If you see this message in presentation mode, install the add-in or get help at PollEv.com/app
**SUMMARY**

Large number of iterates. Parallel for loop.

Small and fixed number of independent tasks. Parallel sections.