CME 213

Spring 2017

Eric Darve

Stanford University

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

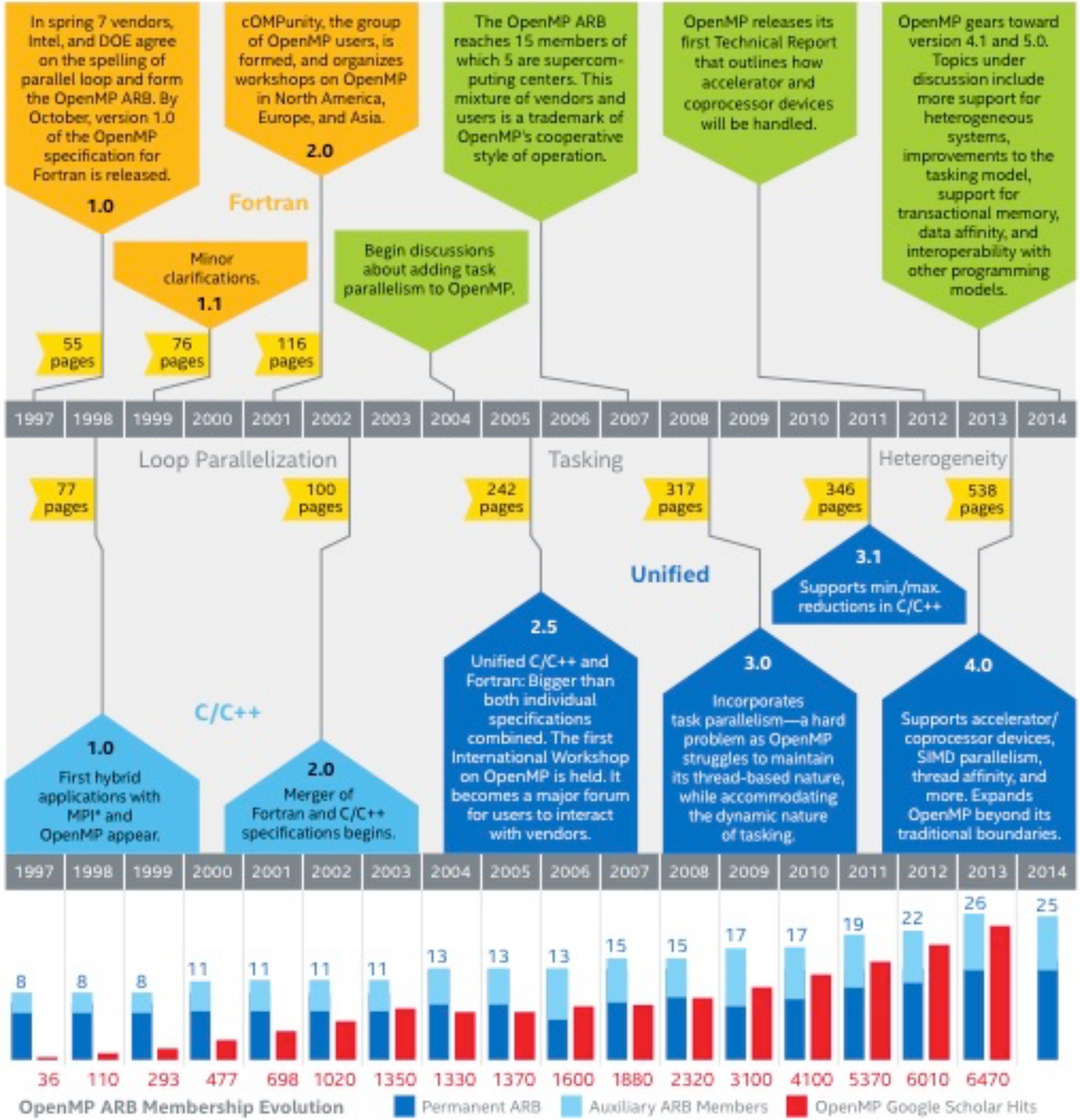
What is OpenMP?

- 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>
```

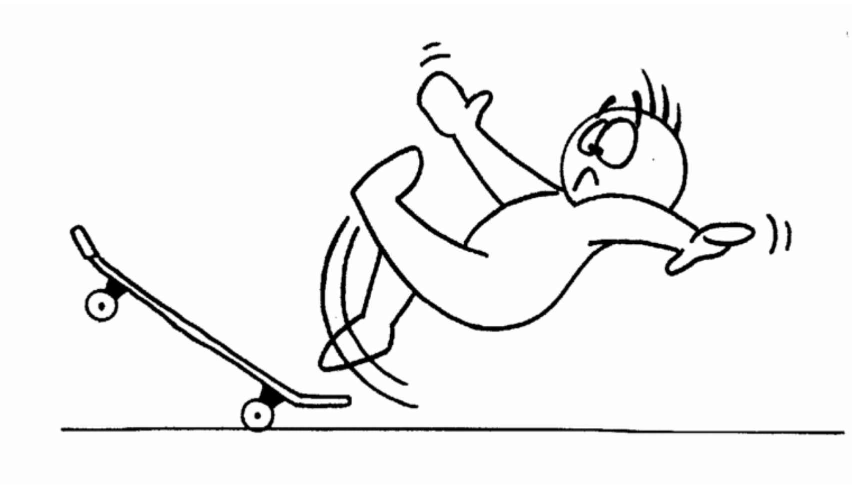
- **This is only needed if you explicitly use the OpenMP API.**
- **Compiler flags:**

Compiler	Flag
icc icpc ifort	-openmp
gcc g++ g77 gfortran	-fopenmp

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.

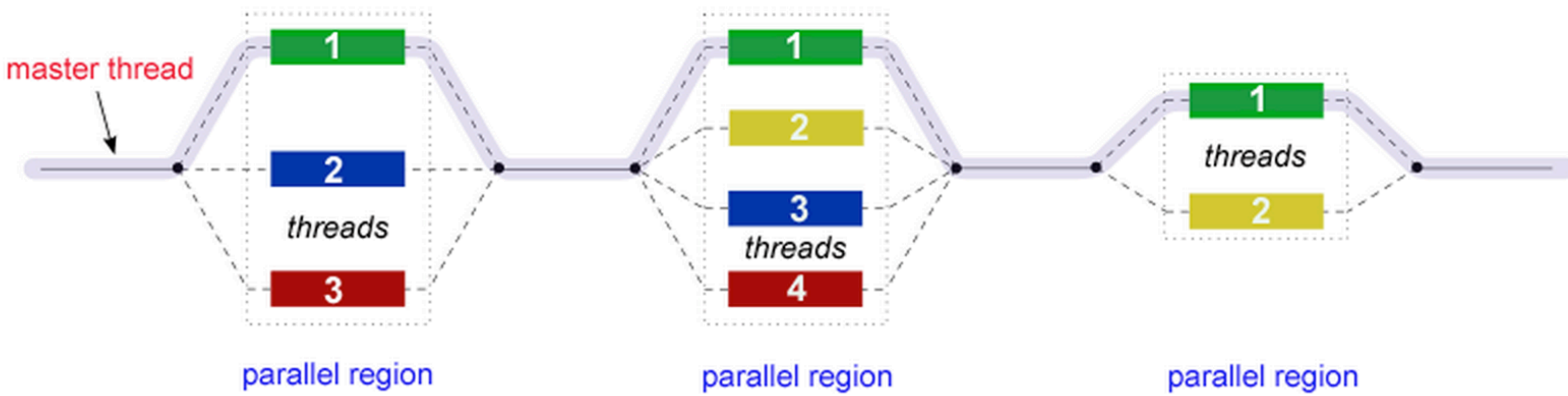


PARALLEL REGION

The most basic directive is

```
#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

`hello_world_openmp.cc`

Compilation:

```
g++ -o hello_world hello_world_openmp.cc -fopenmp
```

3. 1415926535897932384626433832795028841971693993751058209749445
9230781640628620899862803482534211706798214808651328230664709384
446095505822317**2535940812848111745028410270193852110555**9644
62294895493**038196442881097566593344612847564823378678**31652
712019091**45648566923460348610454326648213393607260249**14127
37245870**0660631558817488152092096282925409171536436789**2590
3600113**3053054882046652138414695194151160943305727036**57595
919530**9218**611738193**261179**31051185480**7446237**996274956735188575
27248**9122**7938183011**949129**833673362440**656643**086021394946395224
73719**07**021798609437**027705**3921717629317**67523**8467481846766940513
2000**56**812714526356082**7785**7713427577896**09173**637178721468440901 2
24953430146549585371**05079**22796892589**2354201**9956112129021960864
0344181598136297747**713099**605187072113**499999**9837297804995105973
17328160963185950244**59455**34690830264**2522308**2533446850352619311
881710100031378387528**865**875332083814**2061717**7669147303598253490
4287554687311595628**63882**353787593751**957781**857780532171226806 61
3001927876611195909**21642**01989380952**5720106**548586327886593615 33
818279682303019520**353018**529689957736**225994**1389124972177528347 9
1315155748572424541**50695**950829533116**861727**8558890750983817546 3
7464939319255060**4009277**0167113900984**882401**285836160356370766 01
04710181942955596**198946**767837449448**2553797**747268471040475346 46
208046684259069**4912933**136770289891**5210475**216205696602405803 81
5019351125338**243003558**764024749647**32639141**9927260426992**2**79678
23547816360**0934172164**121992458631**5030286182**974555706749**8**38505
4945885869**2699569092**7210797509302**9553211653**4498720275**59**602364
806654991**1988183479**775356636980742**6542527862551818417**5746728
9097777**2793800081647**0600161452491921**73217214772350141**4419735
68548161**36115735255**21334757418494684**385233239073941**433345477
62416862**5189835694**85562099219222184272**550254256887**67179049460
16534668049**88627**23279178608578438382796797**668145**41009538837863
6095068006422512520511739298489608412848862694560424196528502222
1066118630674427862203919494504712371378696095636437191728746776

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:

```
carry + sum / 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?

Bailey–Borwein–Plouffe formula

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right)$$

Computing the n-th bit

This problem can now be reformulated as:

Can we compute the fractional part of

$$16^n \pi$$

Only a few terms are needed

Take:

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

whole numbers can be removed

Computing the first sum

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

$$\sum_{k=0}^{n} \frac{16^{n-k} \bmod (8k+1)}{8k+1}$$

This can be easily computed

CLAUSE

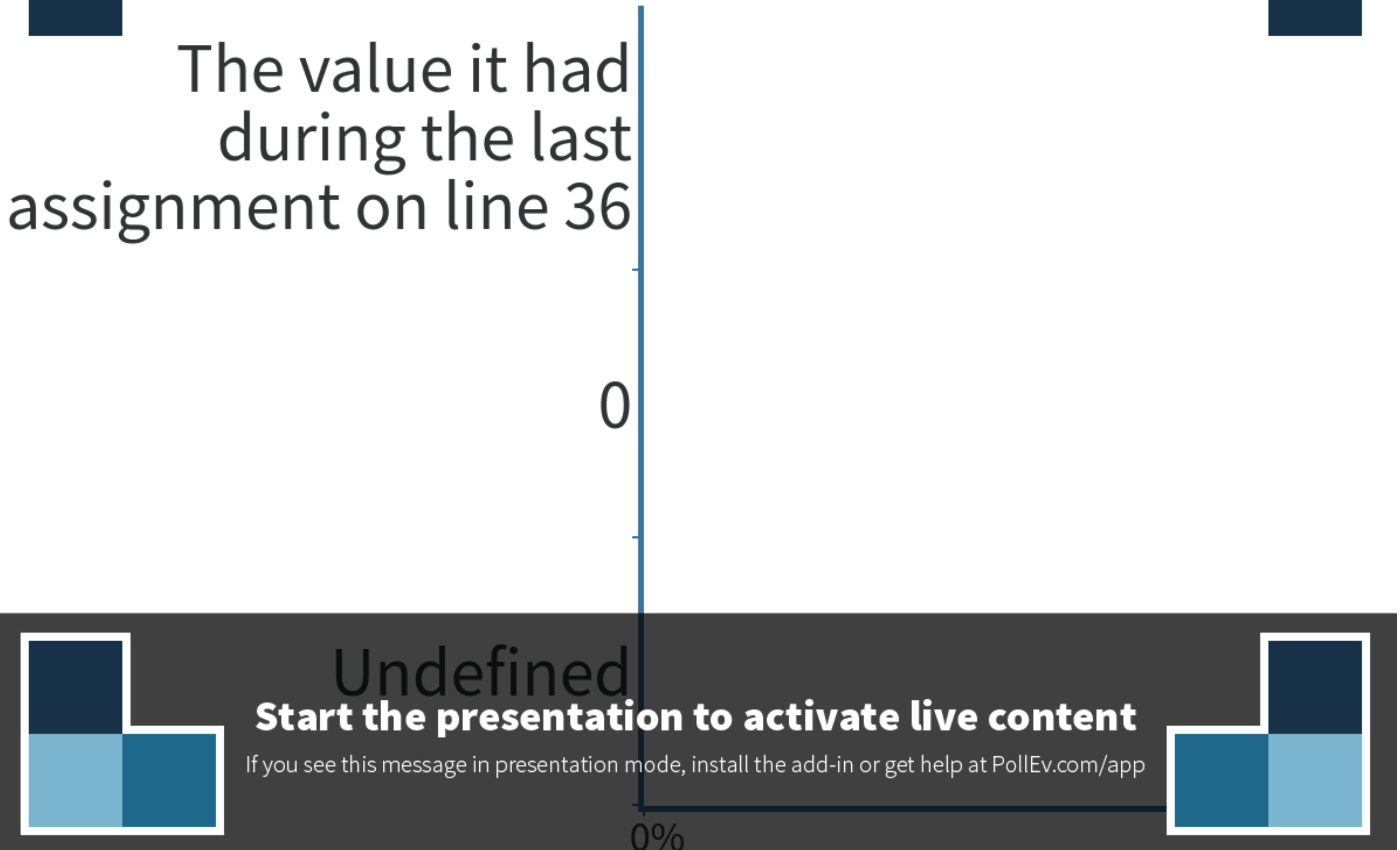
- This is one of the tricky points of OpenMP.
- Recall in Pthreads that:
  - Variables passed as argument to a thread are **shared** (they might be pointers in a `struct` for example)
  - **Variables** inside the function that a thread is executing are **private** to the thread.
- OpenMP needs a similar 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?



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

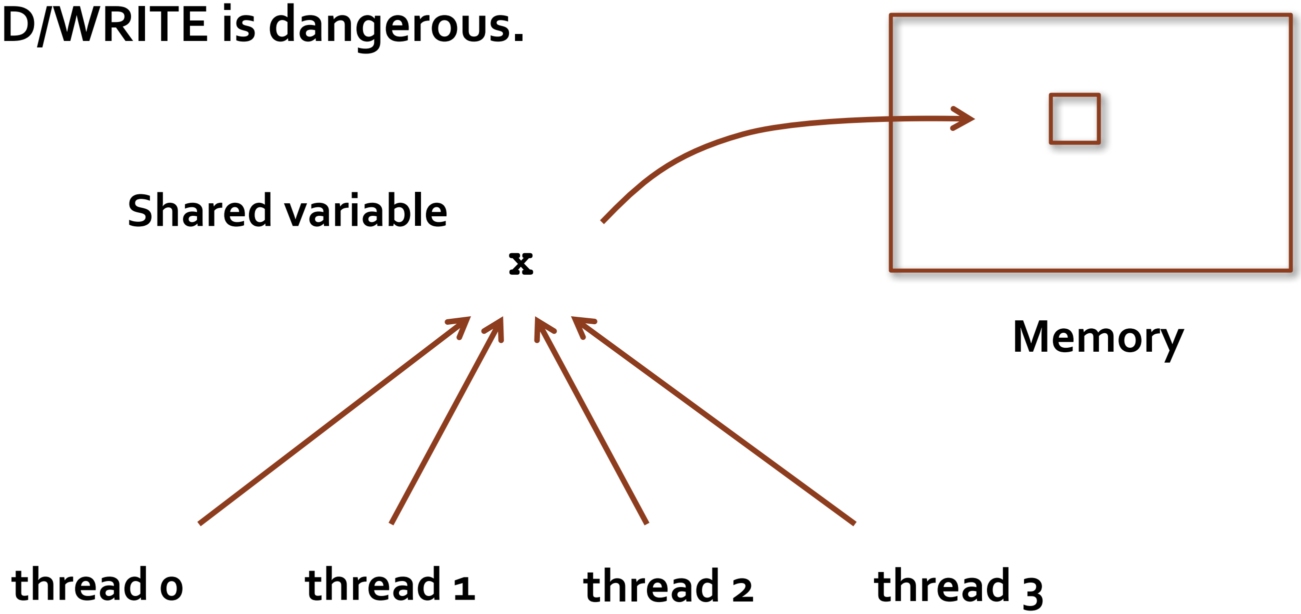
What is the problem with read_write?

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

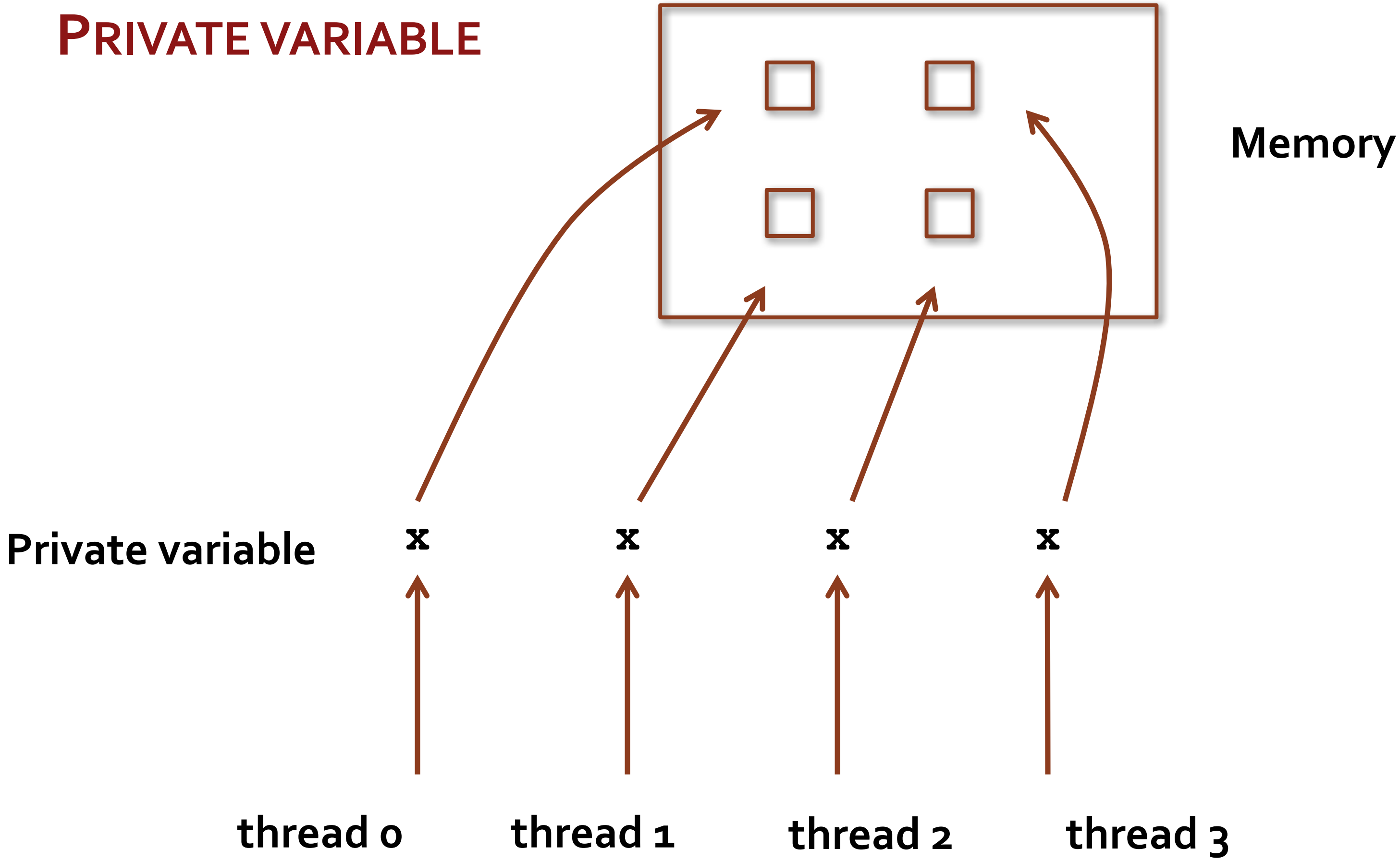
SHARED VARIABLE

- 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



Variable refers to a different memory location for each thread. Those variables are typically READ/WRITE.

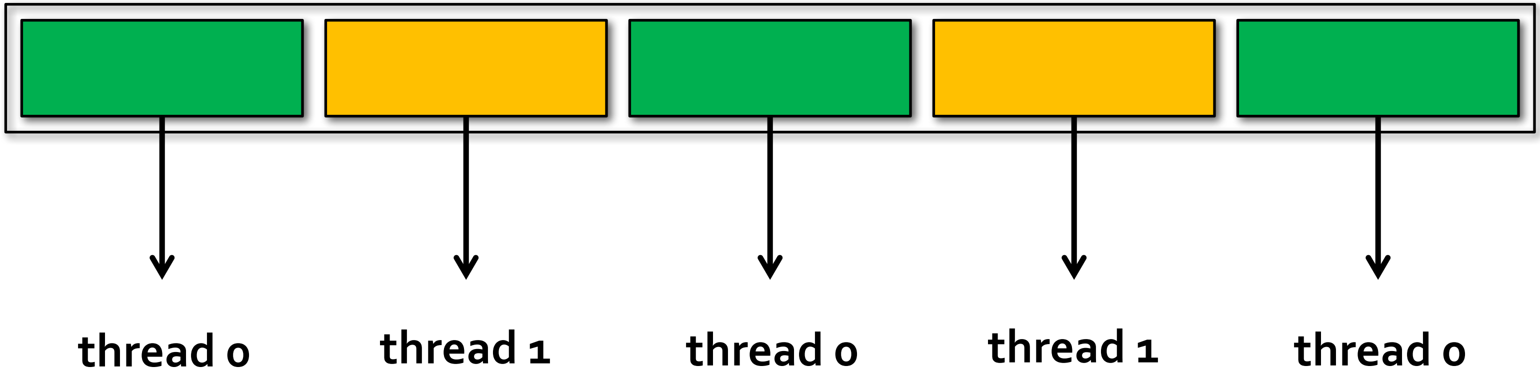
WORKSHARING CONSTRUCTS

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.

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



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
```

sible to move #pragma omp parallel for to lin line 122?

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

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(static, block_size)`: iterations are divided into pieces of size `block_size` and then statically assigned to threads. This is the best default option.
2. `schedule(dynamic, block_size)`: iterations are divided into pieces of size `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(guided, 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 `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.

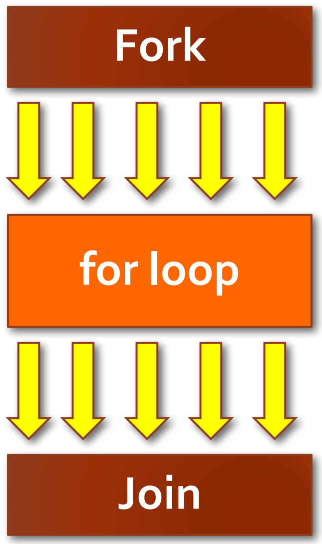
- See `section.cc`

w many threads have work to do in section.c

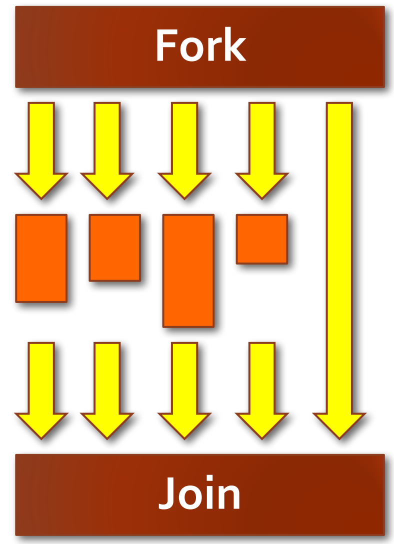
Start the presentation to activate live content

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.