CSC 4005
Tutorial

Fall 2019

Yingruli@link.cuhk.edu.cn

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

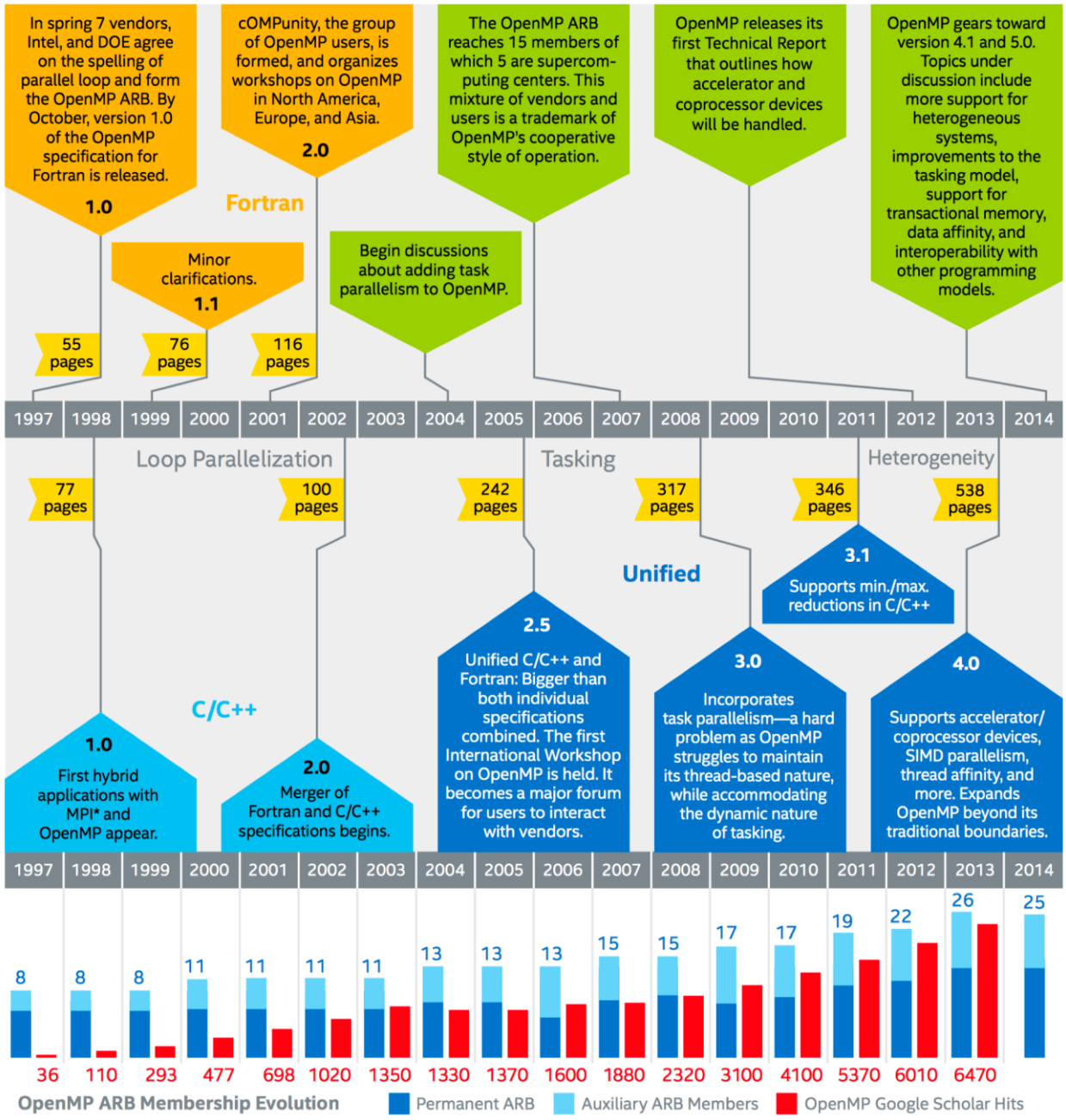
What is OpenMP?

- 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 threads.

- OpenMP website: `openmp.org`
- Wikipedia: `en.wikipedia.org/wiki/OpenMP`
- LLNL tutorial `https://computing.llnl.gov/tutorials/openMP/`

- The Parallel Universe
- 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

For loop

For loops

- This is the most basic construct in OpenMP.
- You will need to use it for future Homework Assignment.

- Let's go through an example together.

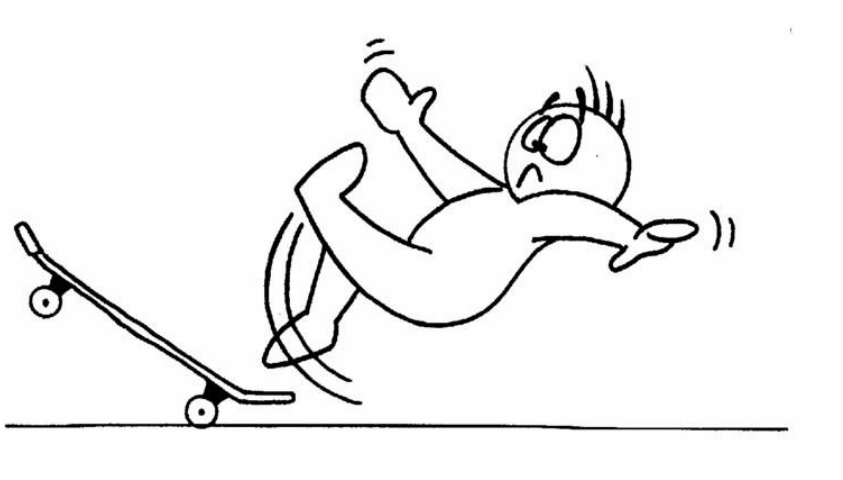
- Download the files from the BlackBoard and open

`for_loop.cpp`

```
#pragma omp parallel for
    for (int i = 0; i < n; ++i)
    {
        x[i] = i;
        y[i] = 2 * i;
    }

    print_vector(x);
    print_vector(y);

    /* todo: write your own omp parallel for loop here */

    print_vector(z);

    for (int i = 0; i < n; ++i)
        assert(z[i] == x[i] + y[i]);
    /* you must pass the test above if your for loop is correct. */
```

Directives

- OpenMP is based on directives.
- Powerful becauseof simple syntax.
- Dangerous because you may not understand exactly what the compiler is doing.

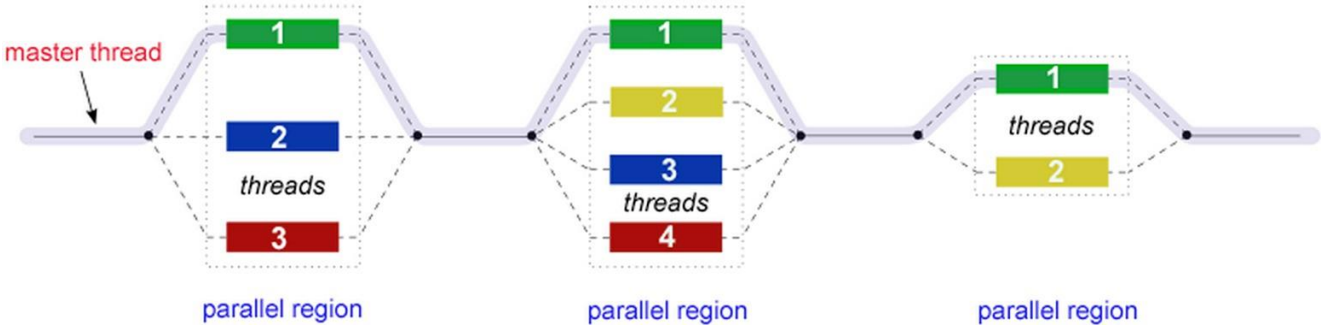


The most general concept: omp parallel region

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.

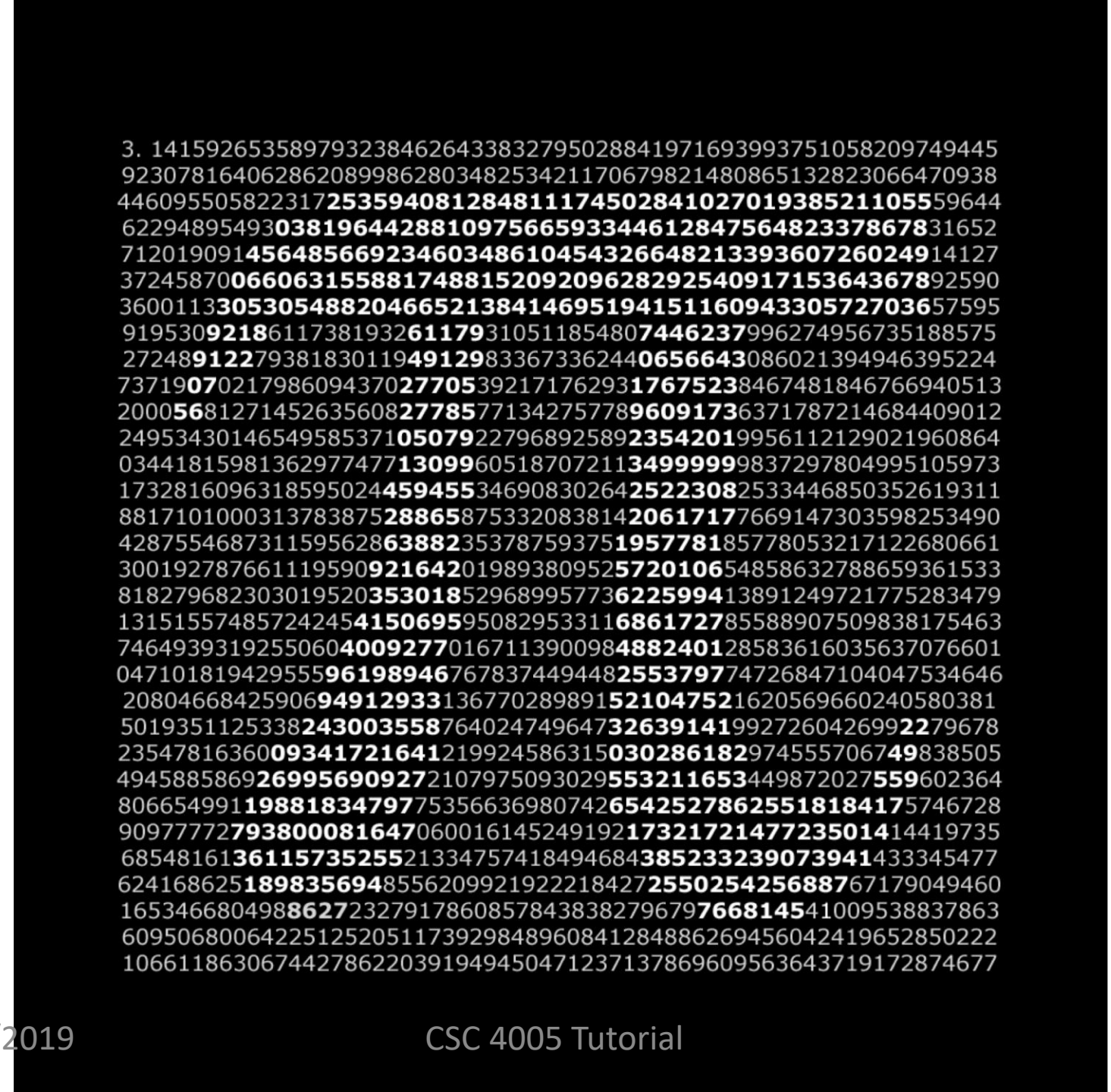
Basic example

`hello_world_openmp.cpp`

Compilation:

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

```
#pragma omp parallel
    {
        long tid = omp_get_thread_num();
        // Only thread 0 does this
        if (tid == 0)
        {
            int n_threads = omp_get_num_threads();
            printf("Number of threads = %d\n", n_threads);
        }
        // Print the thread ID
        printf("Hello World from thread = %ld\n", tid);
        // Compute some of the digits of pi
        DoWork(tid);
    } // All threads join master thread and terminate
```


3. 14159265358979323846264338327950288419716939937510582097494459
230781640628620899862803482534211706798214808651328230664709384
460955058223172535940812848111745028410270193852110555964
4
622948954930381964428810975665933446128475648233786783165
2
712019091456485669234603486104543266482133936072602491412
7
37245870660631558817488152092096282925409171536436789259
0
360011330530548820466521384146951941511609433057270365759
5
91953092186117381932611793105118548074462379962749567351885755
27248912279381830119491298336733624406566430860213949463952247
3719070217986094370277053921717629317675238467481846766940513
2000568127145263560827785771342757789609173637178721468440901
2
24953430146549585371050792279689258923542019956112129021960864
0344181598136297747713099605187072113499999837297804995105973
1732816096318595024459455346908302642522308253344685035261931
1
88171010003137838752886587533208381420617177669147303598253490
428755468731159562863882353787593751957781857780532171226806613
0019278766111959092164201989380952572010654858632788659361533
8182796823030195203530185296899577362259941389124972177528347913151557485724245415069595082953311686172785588907509838175463
7464939319255060400927701671139009848824012858361603563707660104710181942955596198946767837449448255379774726847104047534646
208046684259064941293313677028989152104752162056966024058038150193511253382430035587640247496473263914199272604269922796782354781636093417216412199245863150302861829745557067498385054945885869269956909272107975093029553211653449872027559602364
8066549911988183479775356636980742654252786255181841757467289097777279380008164706001614524919217321721477235014144197356854816136115735255213347574184946843852332390739414333454776241686251898356948556209921922218427255025425688767179049460165346680498862723279178608578438382796797668145410095388378636095068006422512520511739298489608412848862694560424196528502221066118630674427862203919494504712371378696095636437191728746774

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 π 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

$$16^n \pi$$

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

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

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

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.

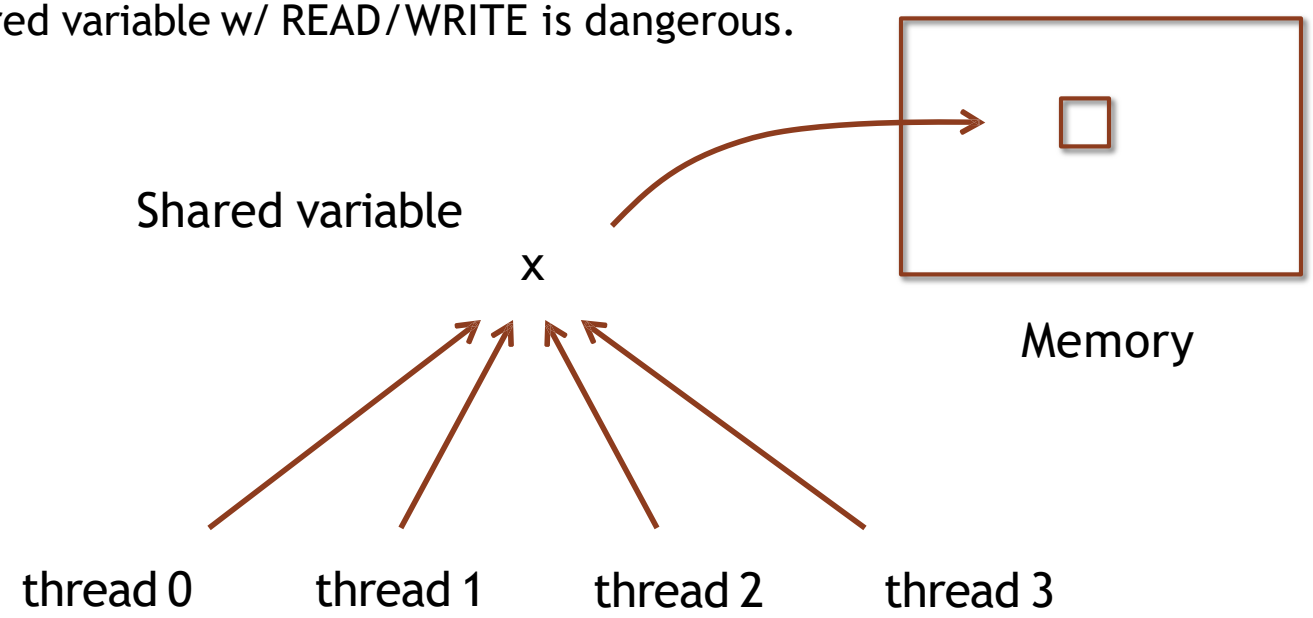
Shared/private

- See `shared_private_openmp.cpp`
- 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)
```

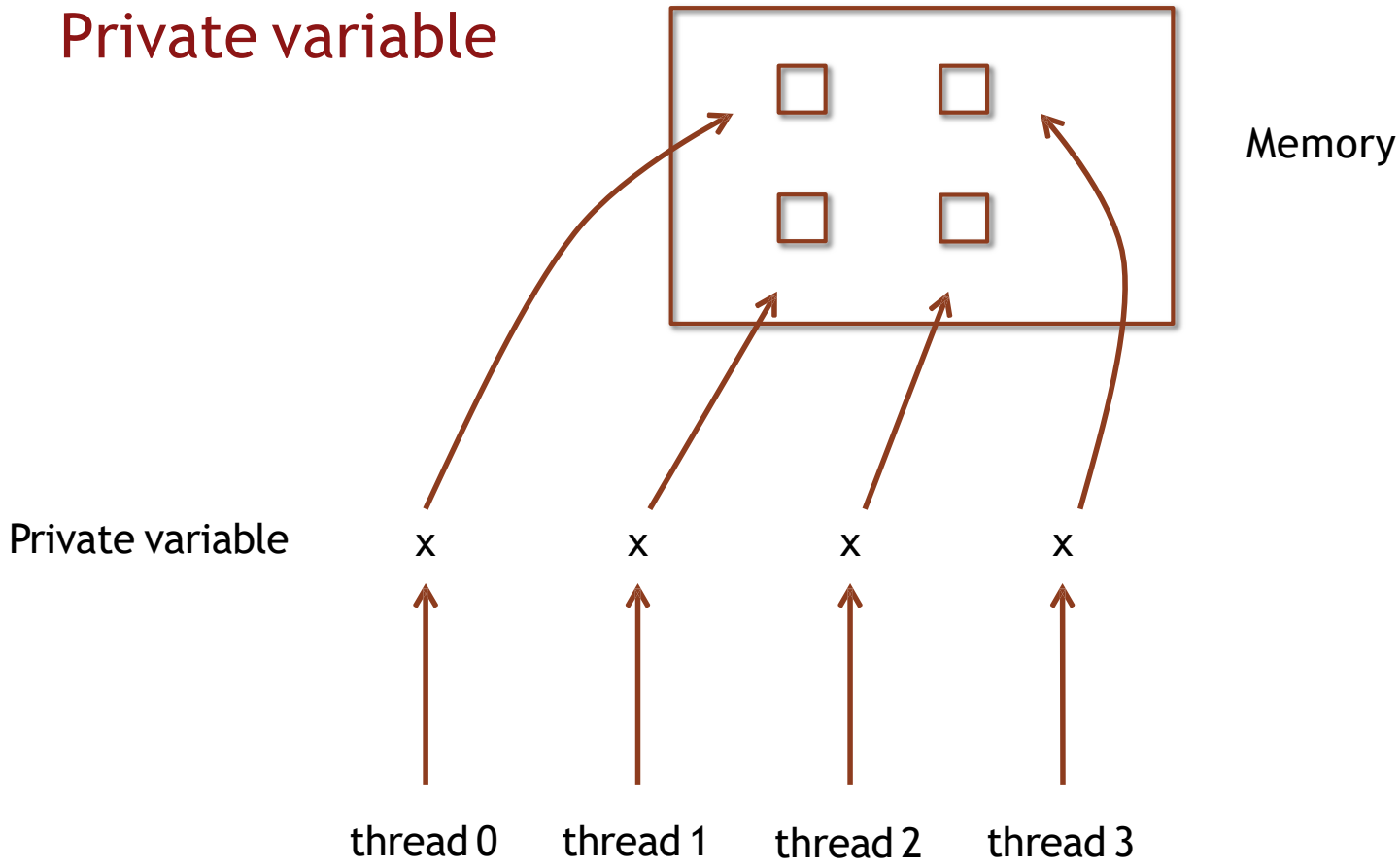
Shared variable

- 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.

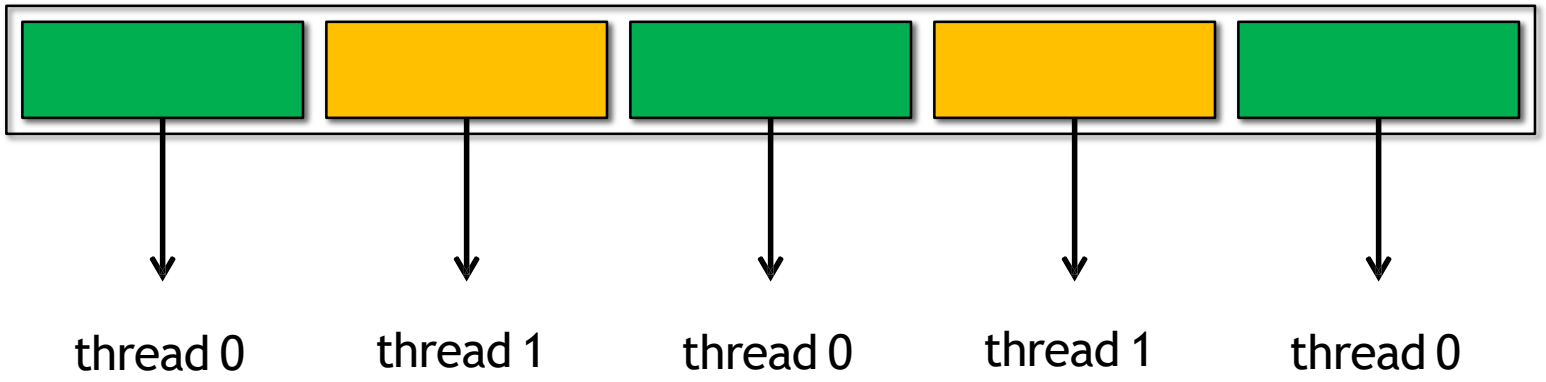
Worksharing constructs

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.cpp`
- One line of code is sufficient to parallelize the calculation! This is the power of OpenMP.

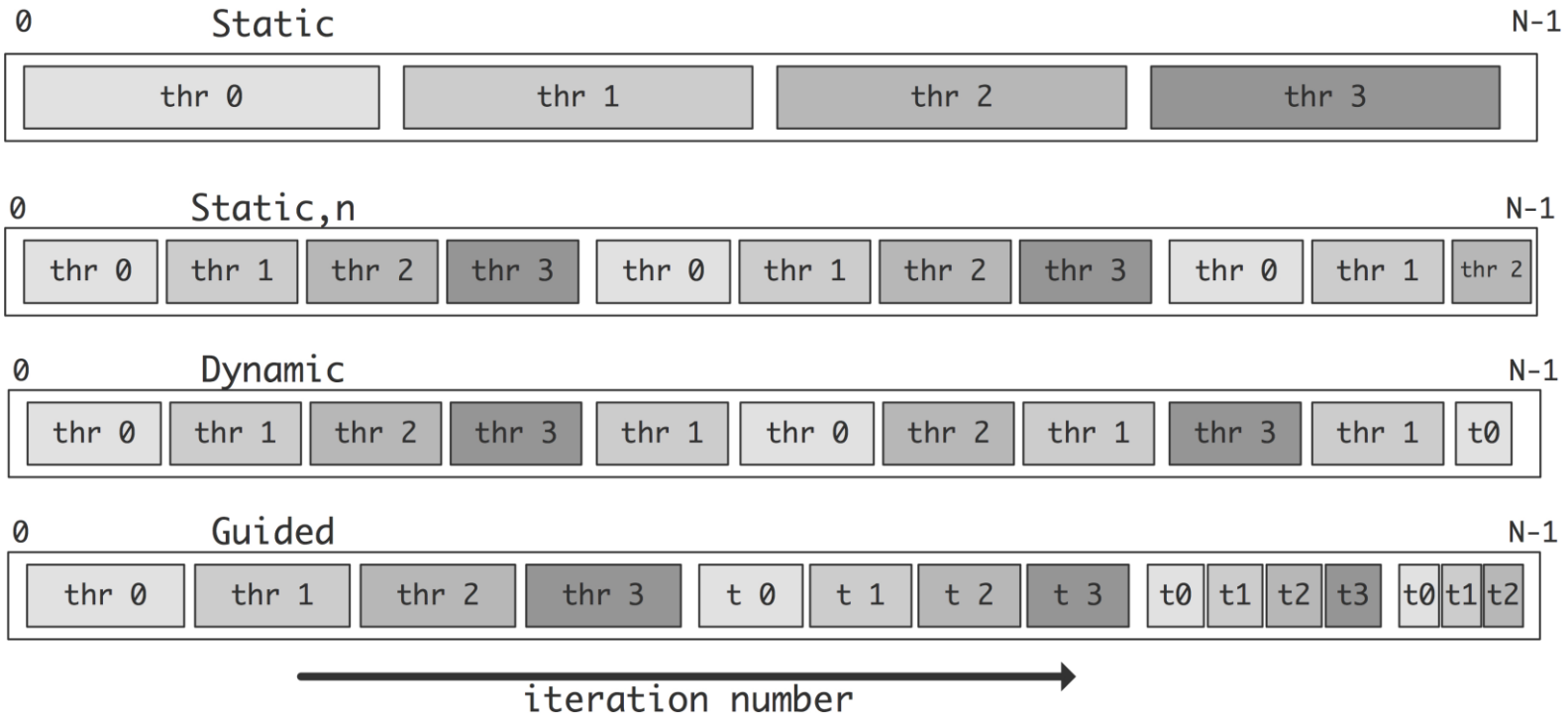
```
$ ./matrix_prod_openmp -n 4000 -p 16
$ top
```

```
#pragma omp parallel for
    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)
        {
            float c_ij = 0;
            for (int k = 0; k < size; ++k)
            {
                c_ij += MatA(i, k) * MatB(k, j);
            }
            mat_c[i * size + j] = c_ij;
        }
```

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).


0
Static
N-1
thr 0
thr 1
thr 2
thr 3
0
Static,n
N-1
thr 0
thr 1
thr 2
thr 3
thr 0
thr 1
thr 2
thr 3
thr 0
thr 1
thr 2
0
Dynamic
N-1
thr 0
thr 1
thr 2
thr 3
thr 1
thr 0
thr 2
thr 1
thr 3
thr 1
t0
0
Guided
N-1
thr 0
thr 1
thr 2
thr 3
t 0
t 1
t 2
t 3
t0
t1
t2
t3
t0
t1
t2
iteration number

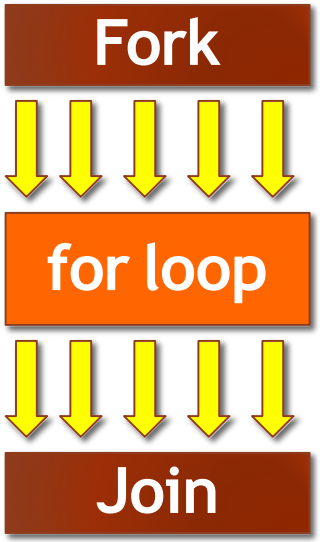
Other worksharing constructs: sections

- 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.

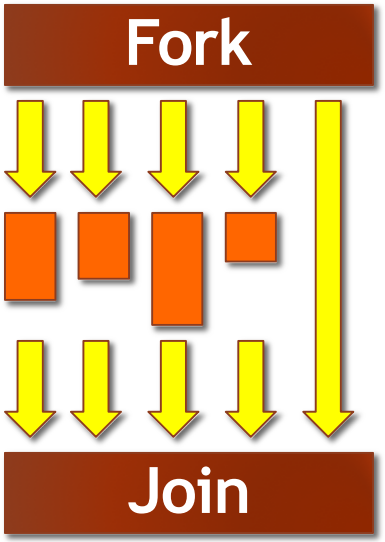
See `section.cpp`

```
#pragma omp parallel
    {
#pragma omp sections
        {
#pragma omp section
            for (unsigned i(0); i < size; i++)
                for (unsigned k(0); k < inner; k++)
                    c[i] += a[(i * k) % size];
#pragma omp section
            for (unsigned i(0); i < size; i++)
                for (unsigned k(0); k < inner; k++)
                    d[i] += a[(b[i] + i * k) % size];
        } // end of sections
    }     // end of parallel block
```

Summary



Large number of iterates.
Parallel for loop



Small and fixed number of independent tasks.
Parallel sections

Questions?

Acknowledgement:
Stanford CME 213