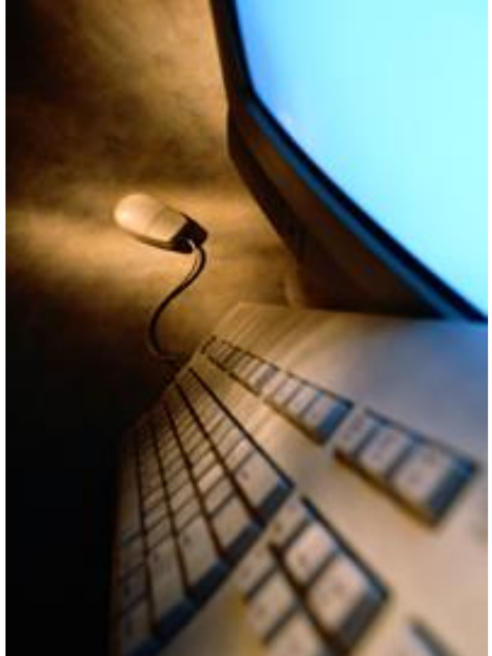
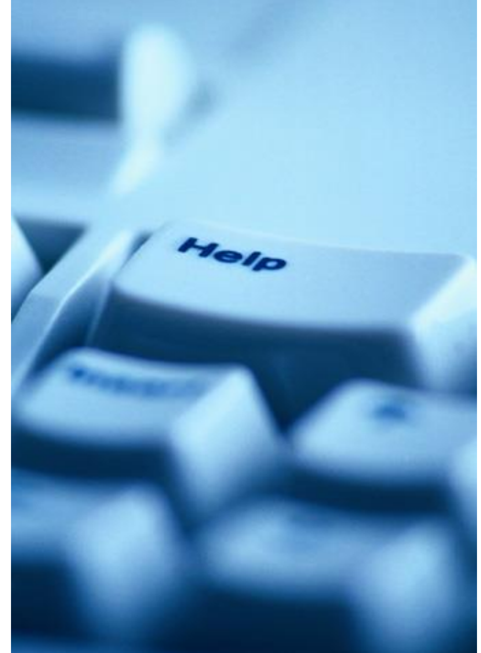
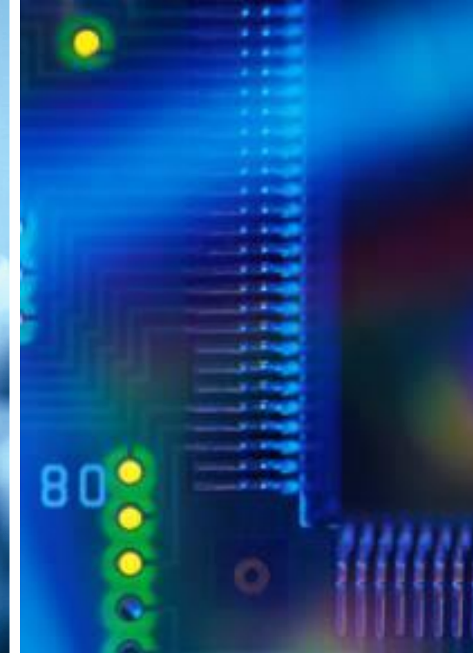






并行与分布式计算

Parallel & Distributed Computing

陈鹏飞
数据科学与计算机学院
2018-04-13

Lecture 5 — OpenMP Programming

Pengfei Chen

School of Data and Computer Science

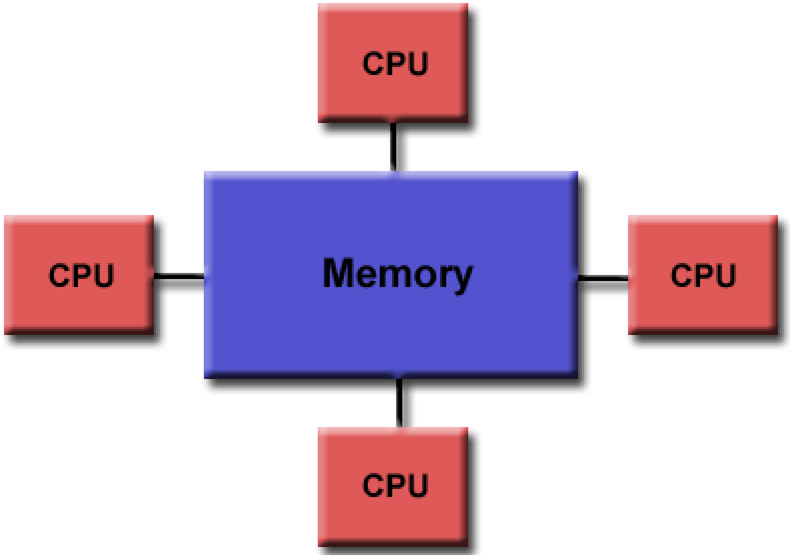
April 13, 2018

Outline:

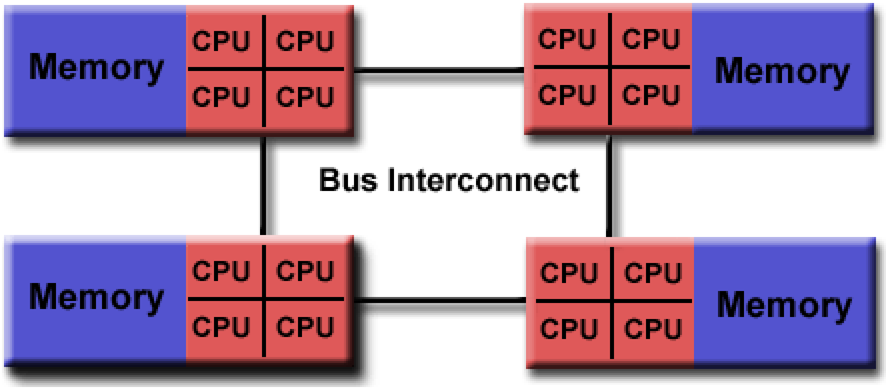
OpenMP programming

Overview

Architecture for Shared Memory Model



Uniform Memory Access



Non-uniform Memory Access

Thread Base Parallelism

- OpenMP programs accomplish parallelism （显式）exclusively （仅仅）through the use of threads
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system
  - The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is
- Threads exist within the resources of a single process
  - Without the process, they cease （停止） to exist
- Typically, the number of threads match the number of machine processors/cores
  - However, the actual use of threads is up to the application

Explicit Parallelism

- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization
- Parallelization can be as simple as taking a serial program and inserting compiler directives....
- Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks

What is OpenMP?

- **An abbreviation for**
  - **Short version**
    - **Open Multi-Processing （开放多处理过程）**
  - **Long version**
    - **Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia**

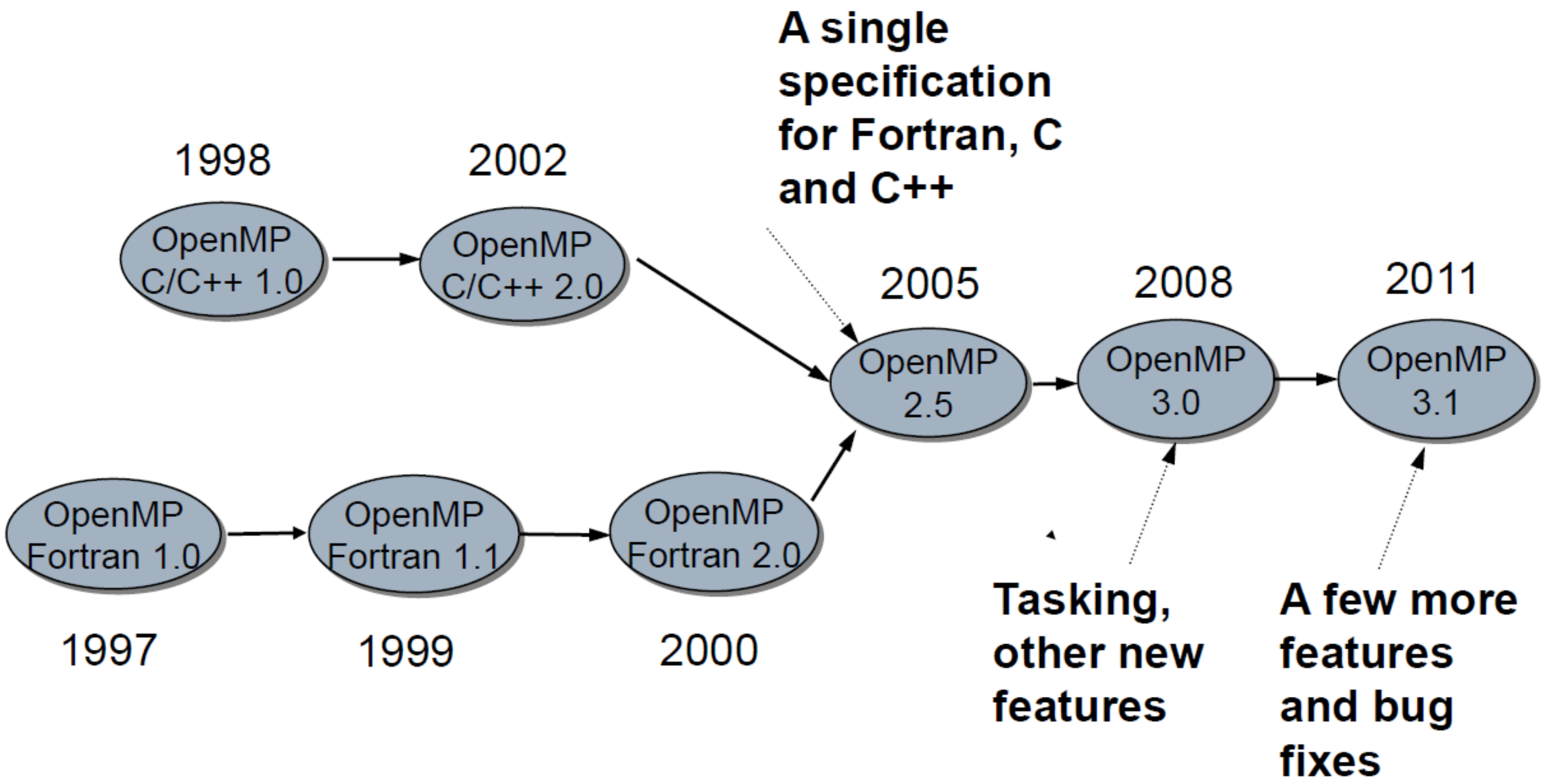
OpenMP Overview

```
C$OMP FLUSH
#pragma omp critical
C$OMP THREADPRIVATE(/ABC/)
CALL OMP_SET_NUM_THREADS(10)
C$OM
C$OM
C$C
ED
C
#p
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice

OpenMP Release History

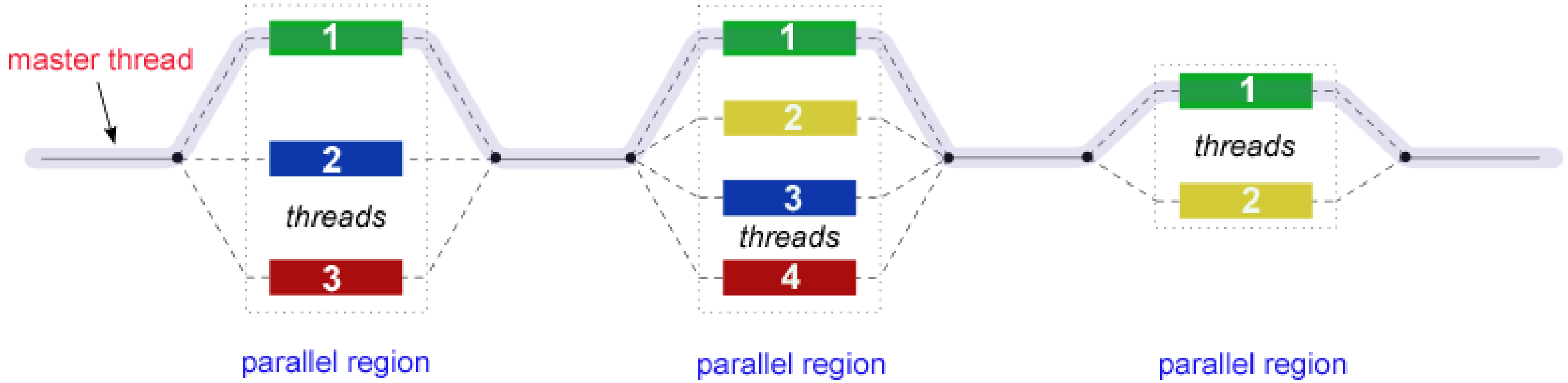


OpenMP Overview: How do Threads Interact?

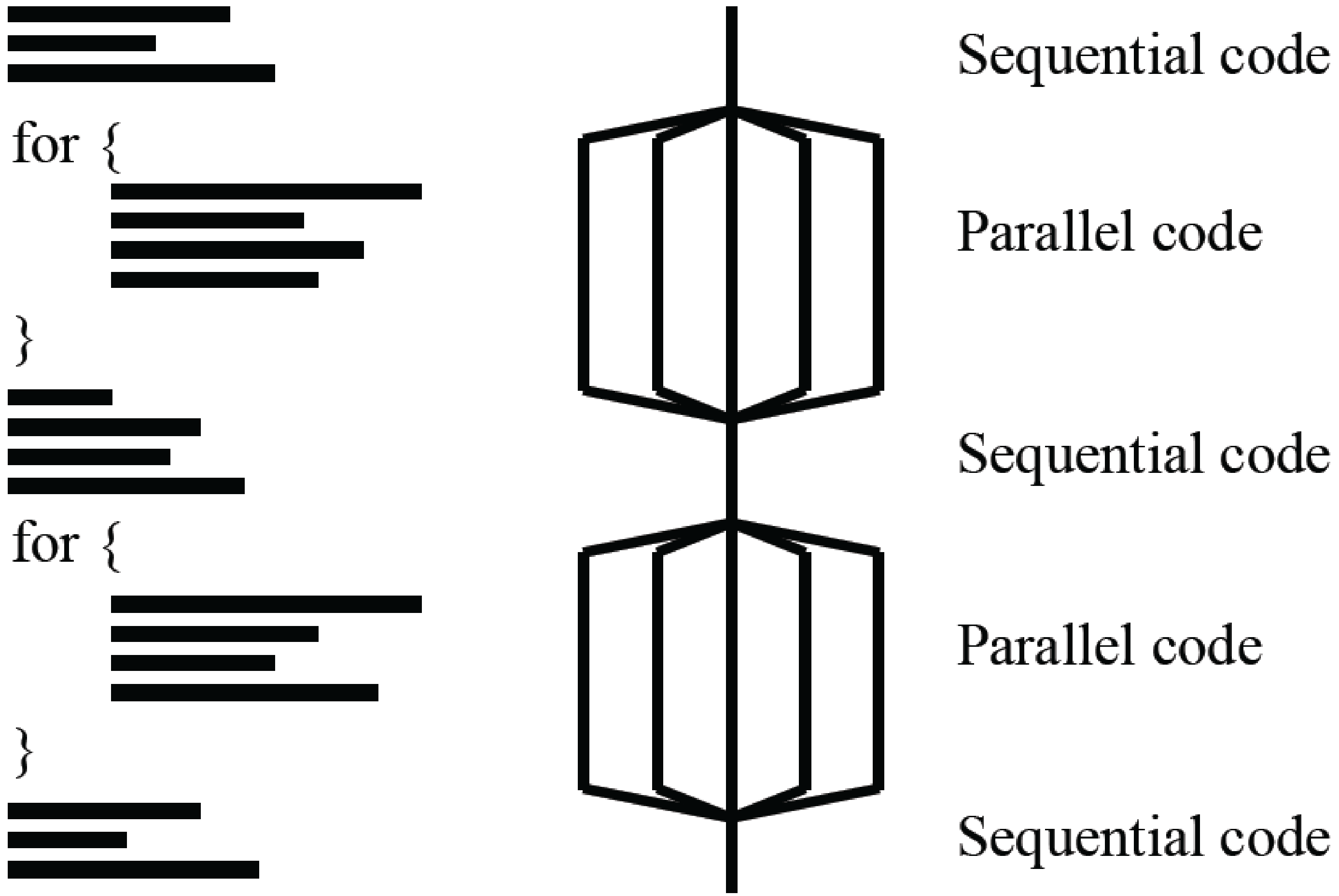
- OpenMP is a multi-threading, shared address model
  - Threads communicate by sharing variables
- Unintended sharing of data causes race conditions
  - Race condition: when the program's outcome changes as the threads are scheduled differently
- To control race conditions
  - Use synchronization to protect data conflicts
- Synchronization is expensive
  - Change how data is accessed to minimize the need for synchronization

Fork-Join Model

- **FORK**
  - The master thread then creates (or awakens) a team of parallel threads.
- **JOIN**
  - When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.



Relating Fork/Join to Code



OpenMP Components

- **Three API components**
  - **Compiler directives**
  - **Runtime library routines**
  - **Environment variables**

Syntax of Compiler Directives （指令）

- *pragma*: **a C/C++ compiler directive （编译开关）**
  - **(other compiler directives:** #include**,** #define**, …)**
  - **Stands for "pragmatic information （附注信息）"**
  - **A way for the programmer to communicate with the compiler**
  - **Pragmas are handled by the preprocessor**
  - **Compilers are free to ignore pragmas**
- **All OpenMP pragmas have the syntax:**
  - #pragma omp <directive-name> [clause, ...]
- **Pragmas appear immediately *before* relevant construct**

Hello World

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
  int nthreads, tid;
  /* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(nthreads, tid)
  {
    /* Obtain thread number */
    tid = omp_get_thread_num();
    /* Only master thread does this */
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
    printf("Hello World from thread = %d\n", tid);
  }  /* All threads join master thread and disband */
}
```

Output – Non-deterministic!

```
[cong@lnxsrvg1 ~/cs133_examples]$ ./a.out
Hello World from thread = 5
Hello World from thread = 7
Number of threads = 16
Hello World from thread = 0
Hello World from thread = 8
Hello World from thread = 3
Hello World from thread = 1
Hello World from thread = 6
Hello World from thread = 15
Hello World from thread = 14
Hello World from thread = 10
Hello World from thread = 2
Hello World from thread = 9
Hello World from thread = 11
Hello World from thread = 13
Hello World from thread = 12
Hello World from thread = 4
[cong@lnxsrvg1 ~/cs133_examples]$ ./a.out
Hello World from thread = 6
Hello World from thread = 10
Hello World from thread = 8
Number of threads = 16
Hello World from thread = 0
Hello World from thread = 5
Hello World from thread = 14
Hello World from thread = 15
Hello World from thread = 13
Hello World from thread = 12
Hello World from thread = 4
Hello World from thread = 3
Hello World from thread = 9
Hello World from thread = 2
Hello World from thread = 7
Hello World from thread = 1
Hello World from thread = 11
```

Supplemental: printf & cout

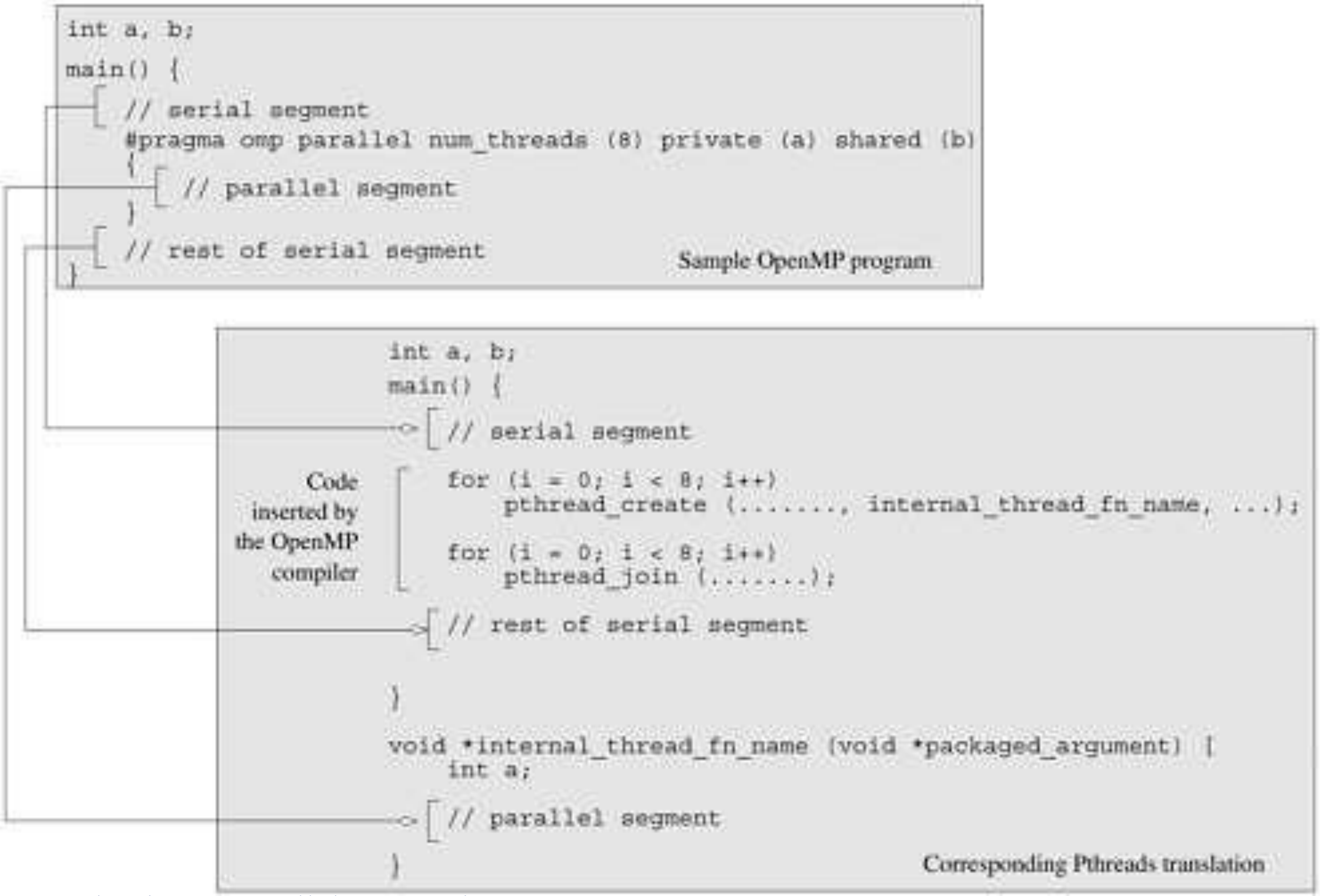
- C/C++ is not aware of “threads,” but POSIX is.
    - http://pubs.opengroup.org/onlinepubs/9699919799/functions/flockfile.html
    - All functions that reference (FILE *) objects shall behave as if they use flockfile() and funlockfile() internally to obtain ownership of these (FILE *) objects.
- What about replacing the printf statement … ?

```
printf("Hello World from thread = %d\n", tid);
```

by

```
cout << "Hello World from thread = " << tid << endl;
```

OpenMP Compilers Perform the Translations to Threads (e.g. pthreads)



Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Matching Threads with CPUs

- **Function** *omp_get_num_procs* **returens the number of physical processors available to the parallel program**

  int omp_get_num_procs(void);

- **Function** *omp_set_num_threads* **allow you to set the number of threads that should be active in parallel sections of code**

  void omp_set_num_threads(int t);

  - **The function can be called with different arguments at different points in the program**

Pragma: *parallel for*

- **The compiler directive**

  #pragma omp parallel for

  tells the compiler that the *for* loop which immediately follows can be executed in parallel

  - **The number of loop iterations must be computable at run time before loop executes**
  - **Loop must not contain a *break*, *return*, or *exit***
  - **Loop must not contain a *goto* to a label outside loop**

Example: *parallel for*

```
int a[1000], b[1000], s[1000];
...
#pragma omp parallel for
for (i = 0; i < 1000; i ++)
    s[i] = a[i] + b[i];
```

- **Threads are assigned an independent set of iterations**
- **Threads must wait at the end of construct**

Which Loop to Make Parallel?

```
int main() {
    int i, j, k;
    float **a, **b;
    … // initialize a[][], b[][] as the 1-hop distance matrix
    for (k = 0; k < N; k++) {
        for (i = 0; i < N; i++)
            for (j = 0; j < N; j++)
                a[i][j] = min(a[i][j], b[i][k] + b[k][j]);
        … // copy a[][] to b[][]
    }
```

Which Loop to Make Parallel?

```
int main() {
    int i, j, k;
    float **a, **b;
    … // initialize b[][] as the 1-hop distance matrix
    for (k = 0; k < N; k++)            // Loop-carried dependences
        for (i = 0; i < N; i++)        // Can execute in parallel
            for (j = 0; j < N; j++)    // Can execute in parallel
                a[i][j] = min(a[i][j], b[i][k] + b[k][j]);
    … // copy a[][] to b[][]
```

Minimizing Threading Overhead

- **There is a fork/join for every instance of**

```
#pragma omp parallel for
for (…) {
…
}
```

- **Since fork/join is a source of overhead, we want to maximize the amount of work done for each fork/join; i.e., the *grain size***
- **Hence we choose to make the middle loop parallel**

Almost Right, but Not Quite

Problem: j is a shared variable

```
int main() {
    int i, j, k;
    float **a , **b;
    … // initialize b[][] as the 1-hop distance matrix
    for (k = 0; k < N; k++) {
        #pragma omp parallel for
        for (i = 0; i < N; i++)
            for (j = 0; j < N; j++)
                a[i][j] = min(a[i][j], b[i][k] + b[k][j]);
        … // copy a[][] to b[][]
    }
```

Clause: *private*

- **Clause**
  - **An optional, additional component to a pragma**
- **Private clause: directs compiler to make one or more variables private**

#pragma omp … private (<variable list>)

Problem Solved with *private* **Clause**

Tell compiler to make listed variables private

```
int main() {
    int i, j, k;
    float **a , **b;
    … // initialize b[][] as the 1-hop distance matrix
    for (k = 0; k < N; k++) {
        #pragma omp parallel for private (j)
        for (i = 0; i < N; i++)
            for (j = 0; j < N; j++)
                a[i][j] = min(a[i][j], b[i][k] + b[k][j]);
        … // copy a[][] to b[][]
    }
```

Another Example

```
int i;
float *a, *b, *c, tmp;
…
for (i = 0; i < N; i++) {
tmp = a[i] / b[i];
c[i] = tmp * tmp;
}
```

Loop is perfectly parallelizable except for shared variable ***tmp***

Solution

```
int i;
float *a, *b, *c, tmp;
…
#pragma omp parallel for private (tmp)
for (i = 0; i < N; i++) {
tmp = a[i] / b[i];
c[i] = tmp * tmp;
}
```

More About Private Variables (私有变量)

- **Each thread has its own copy of the private variables**
- **If *j* is declared private, then inside the *for* loop no thread can access the "other" *j* (the *j* in shared memory)**
- **No thread can use a previously defined value of *j***
- **No thread can assign a new value to the shared *j***
- **Private variables are undefined at loop entry and loop exit, reducing execution time**

Clause: *firstprivate*

- **The *firstprivate* clause tells the compiler that the private variable should inherit the value of the shared variable upon loop entry**
- **The value is assigned once per thread, not once per loop iteration**

Example: *firstprivate*

```
a[0] = 0.0;
for (i = 1; i < N; i++)
    a[i] = alpha(i, a[i-1]);

#pragma omp parallel for firstprivate (a)
for (i = 0; i < N; i++)
    b[i] = beta(i, a[i]);
    a[i] = gamma(i);
    c[i] = delta(a[i], b[i]);
}
```

Clause: *firstprivate*

- pragma omp … firstprivate(x)
  - *x* **is a fundamental data type**
    - **Private** *x* **is directly copied from the shared** *x*
  - *x* **is an array**
    - **Copy the data with** *sizeof(x)* **to the private memory**
  - *x* **is a pointer**
    - **Private** *x* **points to the same location as the shared** *x*
  - *x* **is a class instance**
    - **Copy constructor is called to create the private** *x*

Clause: *firstprivate*

- pragma omp … firstprivate(x)
  - ***x*** **is a fundamental data type**
    - **Private *x* is directly copied from the shared *x***
  - ***x*** **is an array**
    - **Copy the data with *sizeof(x)* to the private memory**
  - ***x*** **is a pointer**
    - **Private *x* points to the same location as the shared *x***
  - ***x*** **is a class instance**
    - **Copy constructor is called to create the private *x***

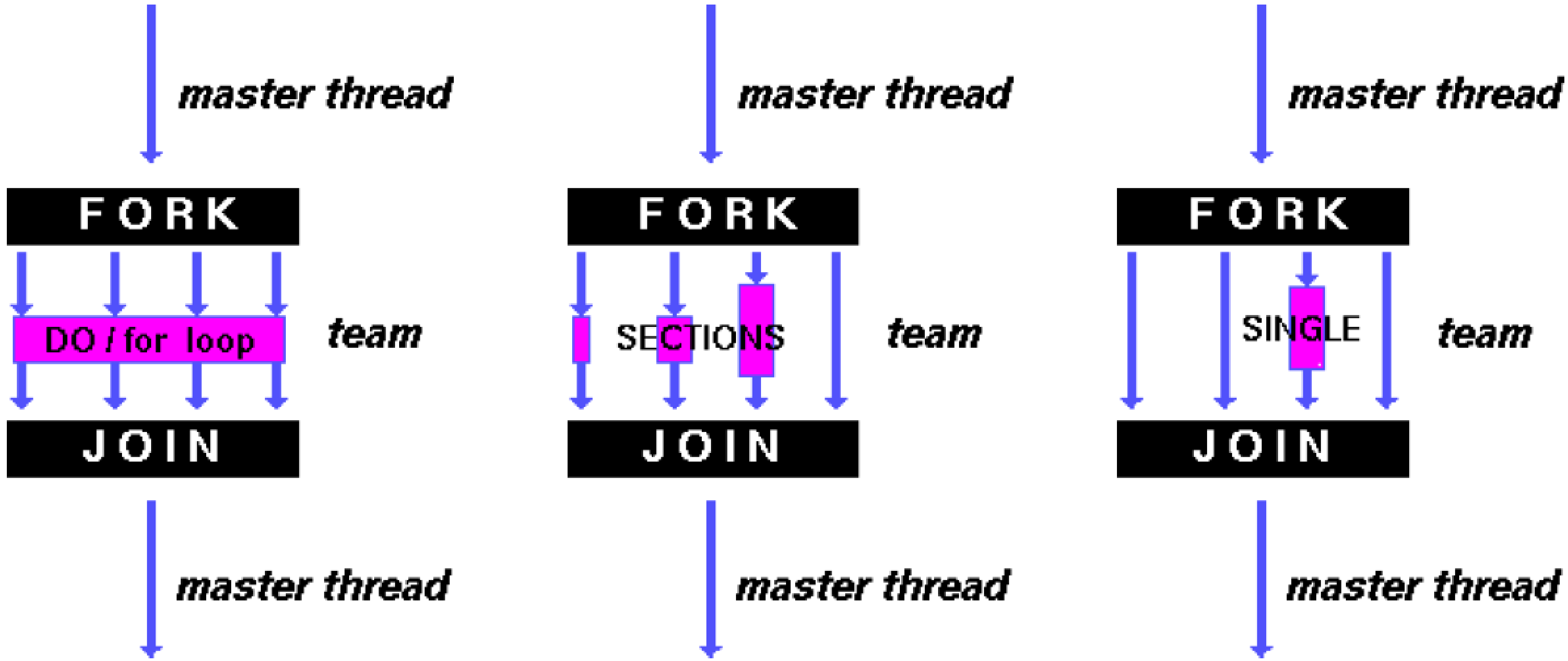
Clause: *lastprivate*

- **The *lastprivate* clause tells the compiler that the value of the private variable after the *sequentially last* loop iteration should be assigned to the shared variable upon loop exit**
  - **In other words, when the thread responsible for the sequentially last loop iteration exits the loop, its copy of the private variable is copied back to the shared variable**

Example: *lastprivate*

```
#pragma omp parallel for lastprivate (x)
for (i = 0; i < N; i++)
x = foo(i);
y[i] = bar(i, x);
}
last_x = x; // == foo(N-1)
```

Work-Sharing Constructs



a type of "data parallelism".

a type of "functional parallelism".

serializes a section of code

Pragma: *parallel*

- **In the effort to increase grain size, sometimes the code that should be executed in parallel goes beyond a single *for* loop**
  - **The *parallel* pragma is used when a block of code should be executed in parallel**
  - **SPMD-style programming**
  - **Single program, multiple data**

Pragma: *for*

- **The *for* pragma is used inside a block of code already marked with the *parallel* pragma**
  - **It indicates a *for* loop whose iterations should be divided among the active threads**
  - **There is a *barrier synchronization* of the threads at the end of the *for* loop**

Pragma parallel ***and Pragma*** for

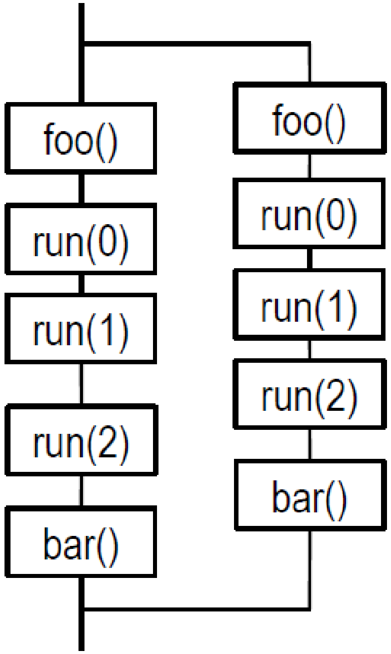
- #pragma omp for
  - **Used inside a block of code already marked by *parallel***
  - **Distribute the iterations to the active threads**

```
#pragma omp parallel \
 num_threads(2)
{
     foo();
     #pragma omp for
     for (int i = 0; i < 3; i++)
          run(i);
     bar();
}
```

```
#pragma omp parallel \
 num_threads(2)
{
     foo();
     for (int i = 0; i < 3; i++)
          run(i);
     bar();
}
```

Pragma parallel *and Pragma* for

- #pragma omp for
  - **Used inside a block of code already marked by *parallel***
  - **Distribute the iterations to the active threads**



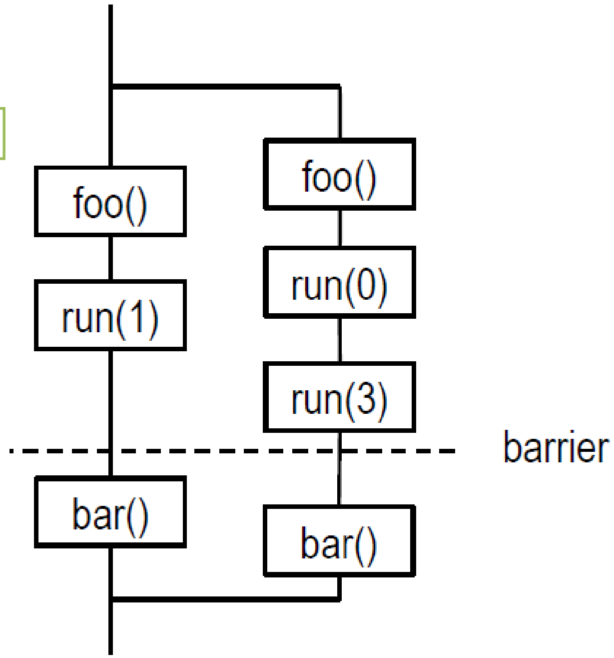

```
#pragma omp parallel \
 num_threads(2)
{
      foo();
      for (int i = 0; i < 3; i++)
            run(i);
      bar();
}
```

Pragma parallel *and Pragma* for

- #pragma omp for
  - **Used inside a block of code already marked by** *parallel*
  - **Distribute the iterations to the active threads**

```
#pragma omp parallel \
 num_threads(2)
{
    foo();
    #pragma omp for
    for (int i = 0; i < 3; i++)
        run(i);
    bar();
}
```

只能在并行结构中使用



Pragma: single

- **The *single* pragma is used inside a parallel block of code**
  - **It tells the compiler that only a single thread should execute the statement or block of code immediately following**
  - **May be useful when dealing with sections of code that are not thread safe (such as I/O)**
  - **Threads in the team that do not execute the** single **directive, wait at the end of the enclosed code block, unless a** nowait **clause is specified.**

Example: *master* **and** *single nowait*

```
tid = omp_get_thread_num();
if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
}
```

=

```
#pragma omp master
{
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
}
```

≈

```
#pragma omp single nowait
{
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
}
```

Pragma: *sections* ***and*** *section*

- **Directive** sections **specifies that the enclosed section(s) of code are to be divided among the threads in the team.**
- **Independent section directives are nested within a** sections **directive.**
  - **Each** section **is executed once by a thread in the team.**
  - **Different sections may be executed by different threads.**
  - **It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such.**

Example: *sections* ***and*** *section*

```
#pragma omp parallel shared(a,b,c,d) private(i)
{
#pragma omp sections
   {
#pragma omp section
      {
        for (i=0; i<N; i++)
          c[i] = a[i] + b[i];
      }
#pragma omp section
      {
        for (i=0; i<N; i++)
          d[i] = a[i] * b[i];
      }
   }  /* end of sections */
 }  /* end of parallel section */
```

Clause: *reduction （归并）*

- **Reductions are so common that OpenMP provides a reduction clause for the *parallel*, *for*, and *sections***

  #pragma omp … reduction (op : list)

  - **A private copy of each list variable is created and initialized depending on the *op***
    - **The identity value *op* (e.g., 0 for addition)**
  - **These copies are updated locally by threads**
  - **At end of construct, local copies are combined through *op* into a single value and combine the value in the original shared variable**

C/C++ Reduction Operation

结合率

- Reduction with an associative binary operator $\oplus$

$$a_1 \oplus a_2 \oplus a_3 \oplus \ldots \oplus a_n$$

- A range of associative and commutative operators can be used with reduction
- Initial values are the ones that make sense

<table>
<tr><th>Operato r</th><th>Initial Value</th></tr>
<tr><td>+</td><td>0</td></tr>
<tr><td>*</td><td>1</td></tr>
<tr><td>-</td><td>0</td></tr>
<tr><td>^</td><td>0</td></tr>
</table>

<table>
<tr><th>Operato r</th><th>Initial Value</th></tr>
<tr><td>&</td><td>~0</td></tr>
<tr><td>|</td><td>0</td></tr>
<tr><td>&&</td><td>1</td></tr>
<tr><td>||</td><td>0</td></tr>
</table>

Reduction: an Artificial Example

```
int sum = 3;
int prod = 5;
#pragma omp parallel for \
  reduction(+:sum)  \
  reduction(*:prod) \
  num_threads(2)
for (int i=0; i < 3; ++i) {
  int tid =
    omp_get_thread_num();
  sum |= i;
  prod += i;
  printf("thread(%d) "
         "sum=%d prod=%d\n",
         tid, sum, prod);
}
printf("results: "
       "sum=%d prod=%d\n",
       tid, sum, prod);
```

- **Assume**
  - thread 0 executes the 1st and 2nd iterations, and
  - thread 1 executes the 3rd iteration
- **Possible outputs**

```
thread(0) sum=0 prod=1
thread(1) sum=2 prod=3
thread(0) sum=1 prod=2
results: sum=6 prod=30
```

Reduction: an Artificial Example

```
int sum = 3;
int prod = 5;
#pragma omp parallel for \
  reduction(+:sum)  \
  reduction(*:prod) \
  num_threads(2)
for (int i=0; i < 3; ++i) {
  int tid =
    omp_get_thread_num();
  sum |= i;
  prod += i;
  printf("thread(%d) "
         "sum=%d prod=%d\n",
         tid, sum, prod);
}
printf("results: "
       "sum=%d prod=%d\n",
       tid, sum, prod);
```

- initial $sum_{private} = 0$
  - for the reduction of +
- initial $prod_{private} = 1$
  - for the reduction of *
- At the end of "parallel for" with reduction
  - $sum_{shared} \mathrel{+}= \Sigma_{thread} sum_{private}$
  - $prod_{shared} \mathrel{*}= \Pi_{thread} prod_{private}$

Strengths and Weaknesses of OpenMP

- **Strengths**
  - **Incremental parallelization & sequential equivalence**
  - **Well-suited for domain decompositions**
  - **Available on *nix and Windows**
- **Weaknesses**
  - **Not well-tailored for functional decompositions**
  - **Compilers do not have to check for such errors as deadlocks and race conditions**

Thank You！