Programming Models Case Study: OpenMP

Ruini Xue

School of Computer Science and Engineering

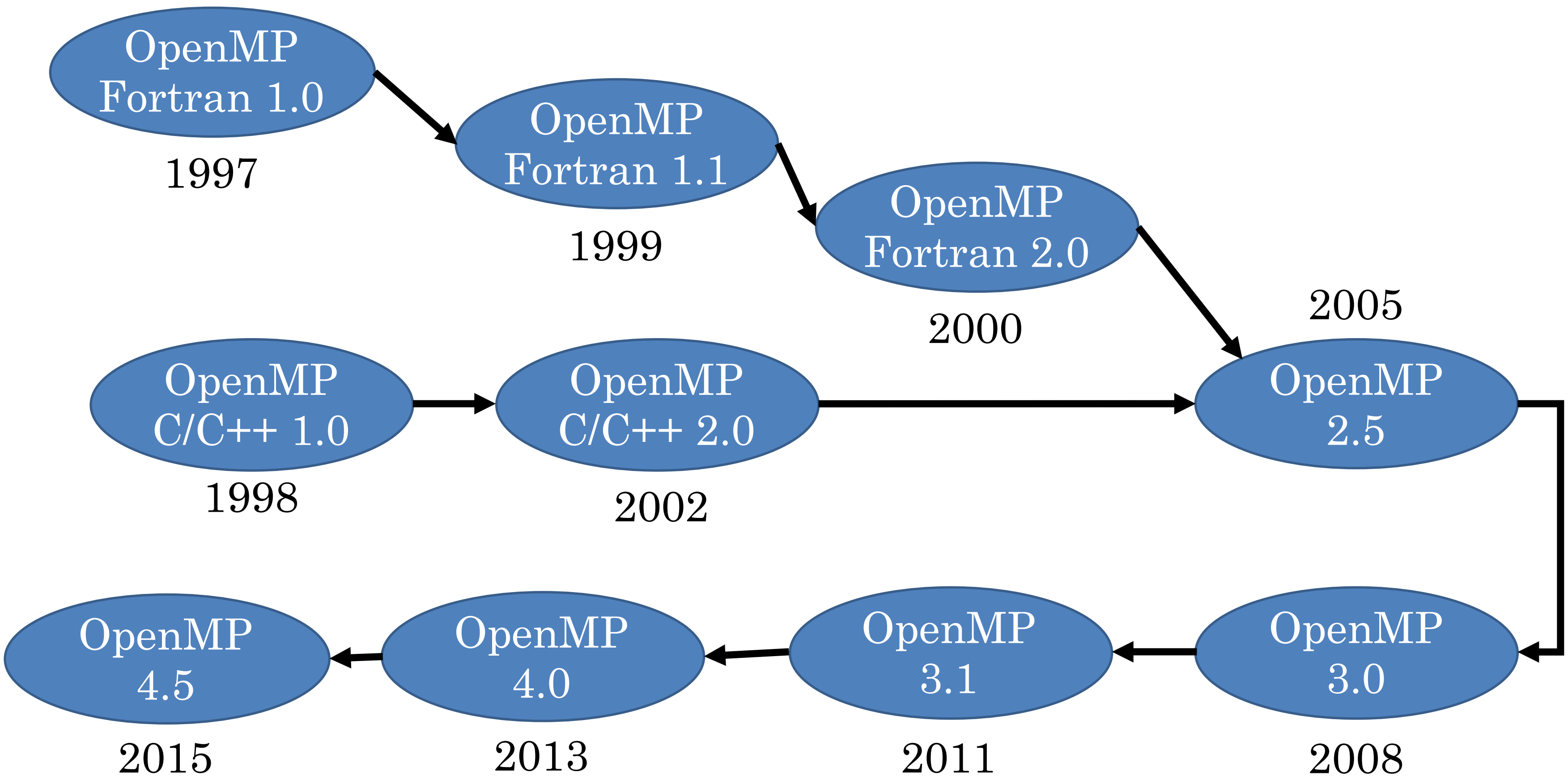
University of Electronic Science and Technology of China

2016

What is OpenMP?

- OpenMP is an API for parallel programming
- Designed for shared-memory multiprocessors
- Set of compiler directives, library functions, and environment variables, but not a language
- Can be used with C, C++, or Fortran
- Based on fork/join model of threads

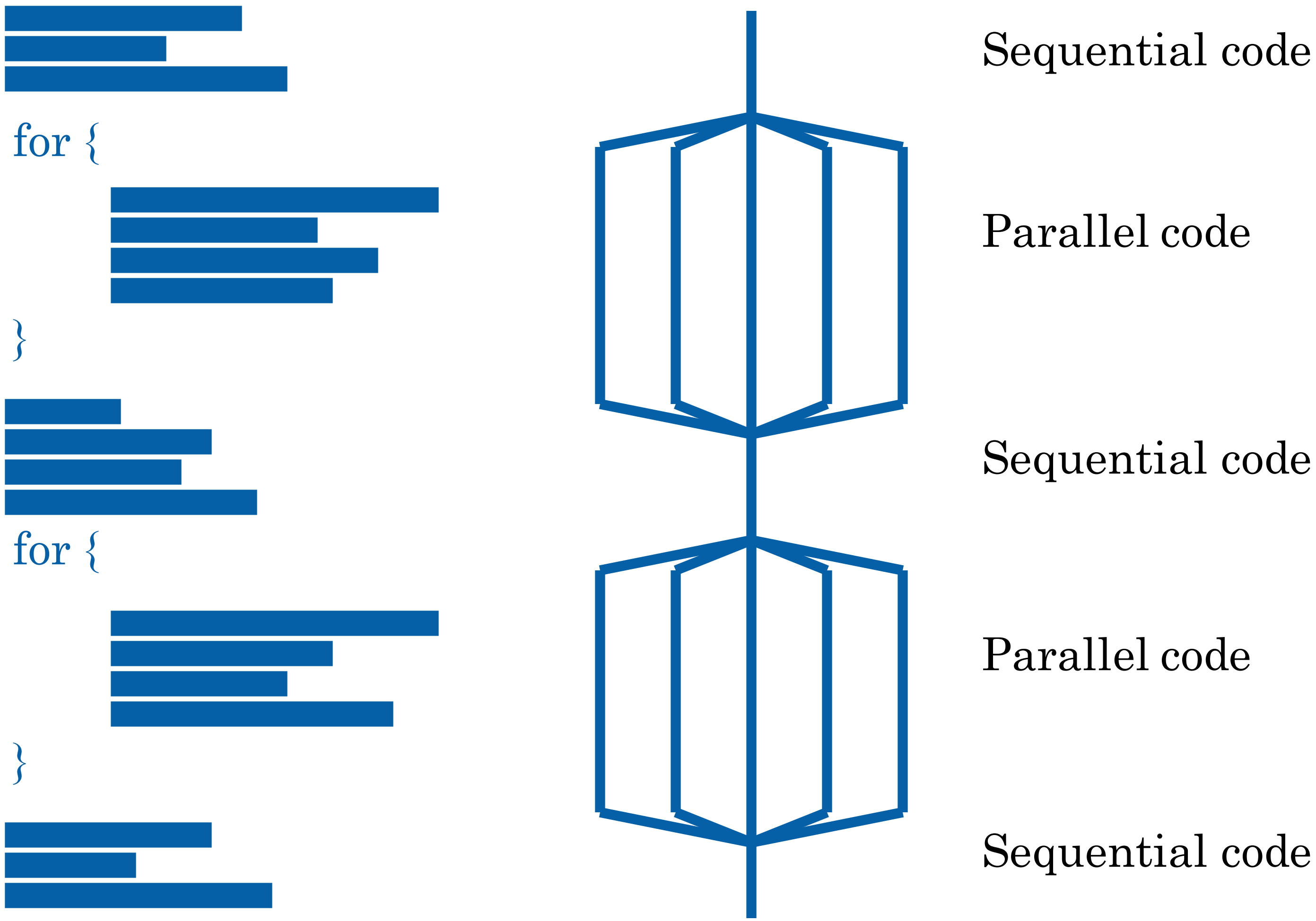
Release History



Fork/Join Programming Model

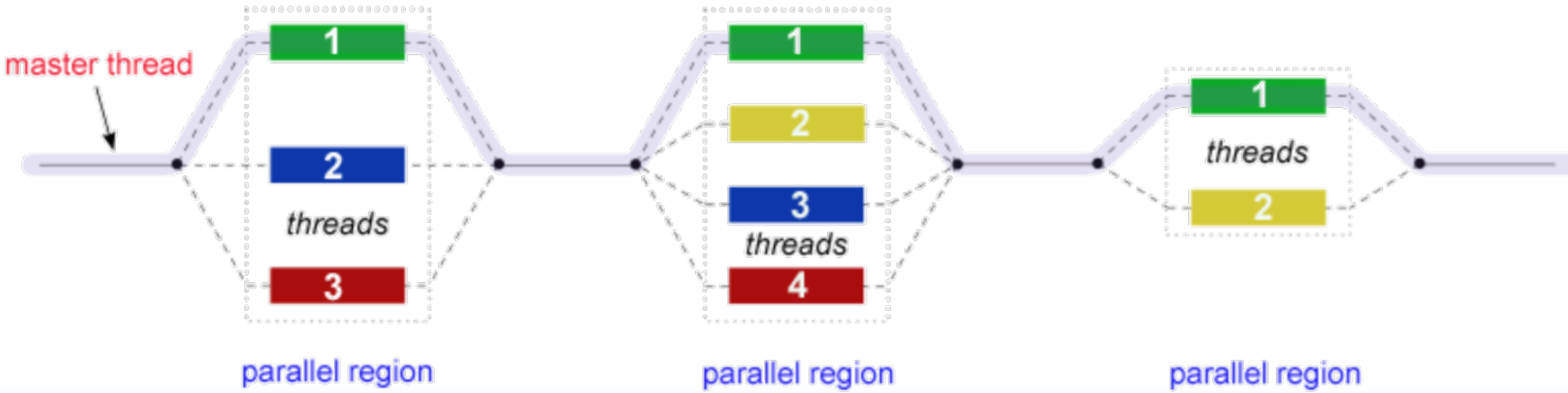
- When program begins execution, only master thread is active
- Master thread executes sequential portions of the program
- For parallel portions of program, master thread forks (creates or awakens) additional threads
- At join (end of parallel section of code), extra threads are suspended or die

Relating Fork/Join to Code



Another view

- All OpenMP programs begin as a single process: the master thread.
    - The master thread executes sequentially until the first parallel region construct is encountered
- Master thread spawns a team of threads as needed
- Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program



Incremental Parallelization

- Sequential program is a special case of threaded program
- Programmers can add parallelism incrementally
- Profile program execution
- Repeat
  - Choose best opportunity for parallelization
  - Transform sequential code into parallel code
- Until further improvements not worth the effort
- Difficult for distributed memory programming

#pragma

- Directive is the method specified by the C standard for providing additional information to the compiler
  - it tells the compiler to do something, set some option, take some action, override some default, etc.
- E.g.

```
#pragma GCC error "message"
```

```
#pragma once
// header file code
```

```
#pragma warning (disable : 4018 )
```

Syntax of Compiler Directives

- A C/C++ compiler directive is called a *pragma*
- Pragmas are handled by the preprocessor
- All OpenMP pragmas have the syntax:
- `#pragma omp` <*rest of pragma*>
- *structured block*
- Pragmas appear immediately before relevant construct

Pragma: 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

parallel for **[2.11.1] [2.10.1]**

Shortcut for specifying a **parallel** construct containing one or more associated loops and no other statements.

#pragma omp parallel for *[clause[[,]clause] ...]*
    for-loop

Example

- **`int first, *marked, prime, size;`**
- **`...`**
- **`#pragma omp parallel for`**
- **`for (i = first; i < size; i += prime)`**
- **`    marked[i] = 1;`**

1. Threads are assigned an independent set of iterations
2. Barrier/join: threads must wait at the end of construct

Pragma: parallel

- 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

parallel **[2.5] [2.5]**

Forms a team of threads and starts parallel execution.

#pragma omp parallel *[clause[[,]clause] ...]*
    structured-block

```
#pragma omp parallel
{
  DoSomeWork(res, M);
  DoSomeOtherWork(res, M);
}
```

Pragma: for

for [2.7.1] [2.7.1]

Specifies that the iterations of associated loops will be executed in parallel by threads in the team in the context of their implicit tasks.

Pragma: for

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

```
#pragma omp parallel
{
  DoSomeWork(res, M);
 #pragma omp for
  for (i = 0; i < M; i++){
     res[i] = huge();
   }
  DoSomeMoreWork(res, M);
}
```

Which loop to make parallel?

- Loop-carried dependence: dependence exists across iterations; i.e., if the loop is removed, the dependence no longer exists.
- Loop-independent dependence: dependence exists within an iteration; i.e., if the loop is removed, the dependence still exists.

Loop dependencies

```
for (i=1; i<n; i++) {
  S1: a[i] = a[i-1] + 1;
  S2: b[i] = a[i];
}

for (i=1; i<n; i++)
  for (j=1; j< n; j++)
    S3: a[i][j] = a[i][j-1] + 1;

for (i=1; i<n; i++)
  for (j=1; j< n; j++)
    S4: a[i][j] = a[i-1][j] + 1;
```

`S1[i]` →T **`S1[i+1]`**: loop-carried

`S1[i]` →T **`S2[i]`**: loop-independent

`S3[i,j]` →T **`S3[i,j+1]`**:

- loop-carried on **`for`** `j` loop
- no loop-carried dependence in **`for`** `i` loop

`S4[i,j]` →T **`S4[i+1,j]`**:

- no loop-carried dependence in **`for`** `j` loop
- loop-carried on **`for`** `i` loop

Which Loop to Make Parallel?

```
main () {
int i, j, k;
float **a, **b;
...
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], a[i][k] + a[k][j]);
```

Loop-carried dependences (for k loop)

Can execute in parallel (for i loop)

Can execute in parallel (for j loop)

Floyd's algorithm

Minimizing Threading Overhead

- There is a fork/join for every instance of
- Since fork/join is a source of overhead, we want to maximize the amount of work done for each fork/join
- Hence we choose to make the middle loop parallel
  - n fork/joins
  - For inner loop parallel, n^2 fork/joins

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

Almost Right, but Not Quite

```
• main () {
• int i, j, k;
• float **a, **b;
• ...
• 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], a[i][k] + a[k][j]);
```

Problem: j is a shared variable

Problem Solved with private Clause

- `main () {`
- `int i, j, k;`
- `float **a, **b;`
- `...`
- `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], a[i][k] + a[k][j]);`

private(*list*)
Declares one or more list items to be private to a task or a SIMD lane. Each task that references a list item that appears in a **private** clause in any statement in the construct receives a new list item.

Tells compiler to make listed variables private

The Private Clause

- Reproduces the variable for each thread
  - Variables are un-initialized; C++ object is default constructed
  - Any value external to the parallel region is undefined

```
void work(float* c, int N)
{
  float x, y; int i;
  #pragma omp parallel for private(x,y)
  for(i = 0; i < N; i++) {
    x = a[i]; y = b[i];
    c[i] = x + y;
  }
}
```

Example: Dot Product

- Why won't the use of the **private** clause work in this example?

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
  #pragma omp parallel for private(sum)
  for(int i = 0; i < N; i++) {
    sum += a[i] * b[i];
  }
  return sum;
}
```

Reductions

- Given associative binary operator ⊕ the expression
  - $a_1 \oplus a_2 \oplus a_3 \oplus \ldots \oplus a_n$
- is called a *reduction*

OpenMP `reduction` Clause

- Reductions are so common that OpenMP provides a `reduction` clause for the `parallel for` pragma
- **`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 combined with the value in the original SHARED variable

reduction(*reduction-identifier*:*list*)

Specifies a *reduction-identifier* and one or more list items. The *reduction-identifier* must match a previously declared *reduction-identifier* of the same name and type for each of the list items.

Operators for reduction (initialization values)			
+	(0)	**|**	(0)
*****	(1)	**^**	(0)
-	(0)	**&&**	(1)
&	(~0)	**||**	(0)
max (Least representable number in **reduction** list item type)			
min (Largest representable number in **reduction** list item type)			

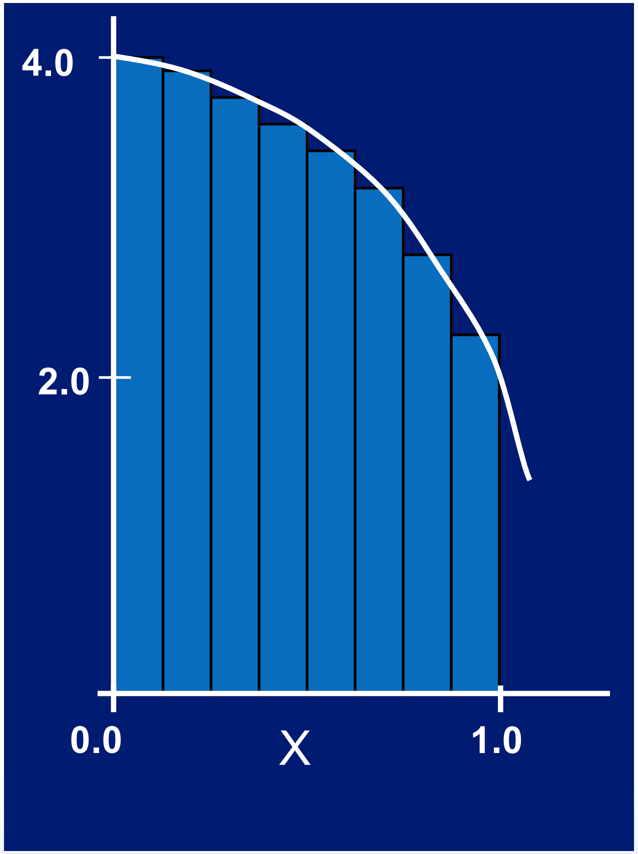
Reduction Example

- Local copy of **sum** for each thread
- All local copies of sum added together and stored in shared copy

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
  #pragma omp parallel for reduction(+:sum)
  for(int i = 0; i < N; i++) {
    sum += a[i] * b[i];
  }
  return sum;
}
```

Numerical Integration Example

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$




```
static long num_rects=100000;
double width, pi;

void main()
{   int i;
    double x, sum = 0.0;

    width = 1.0/(double) num_rects;
    for (i = 0; i < num_rects; i++){
        x = (i+0.5)*width;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = width * sum;
    printf("Pi = %f\n",pi);
}
```

Numerical Integration: What's Shared?

```
static long num_rects=100000;
double width, pi;

void main()
{   int i;
    double x, sum = 0.0;

    width = 1.0/(double) num_rects;
    for (i = 0; i < num_rects; i++){
        x = (i+0.5)*width;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

What variables can be shared?

`width, num_rects`

Numerical Integration: What's Private?

```
static long num_rects=100000;
double width, pi;

void main()
{   int i;
    double x, sum = 0.0;

    width = 1.0/(double) num_rects;
    for (i = 0; i < num_rects; i++){
        x = (i+0.5)*rects;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

What variables need to be private?

`x, i`

Numerical Integration: Any Reductions?

```
static long num_rects=100000;
double width, pi;

void main()
{   int i;
    double x, sum = 0.0;

    width = 1.0/(double) num_rects;
    for (i = 0; i < num_rects; i++){
        x = (i+0.5)*width;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

What variables should be set up for reduction?

`sum`

Solution to Computing Pi

```
static long num_rects=100000;
double width, pi;

void main()
{   int i;
    double x, sum = 0.0;
#pragma omp parallel for private(x) reduction(+:sum)
    width = 1.0/(double) num_rects;
    for (i = 0; i < num_rects; i++){
        x = (i+0.5)*width;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```