CME 213

SPRING 2012-2013

Eric Darve

Stanford University


LELAND STANFORD JUNIOR UNIVERSITY
DIE LUFT DER FREIHEIT WEHT
1891

OpenMP

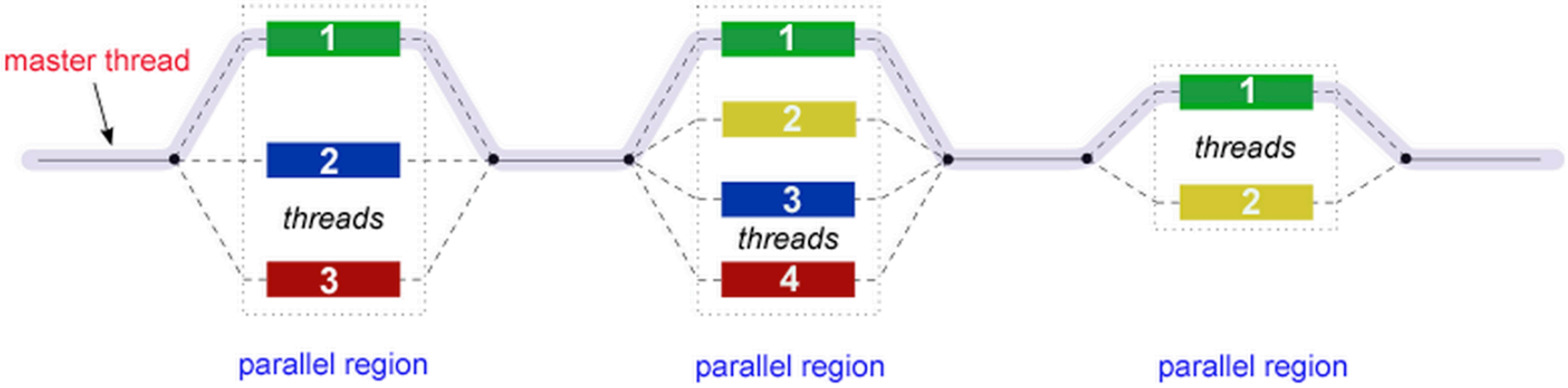
PARALLEL REGIONS

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.

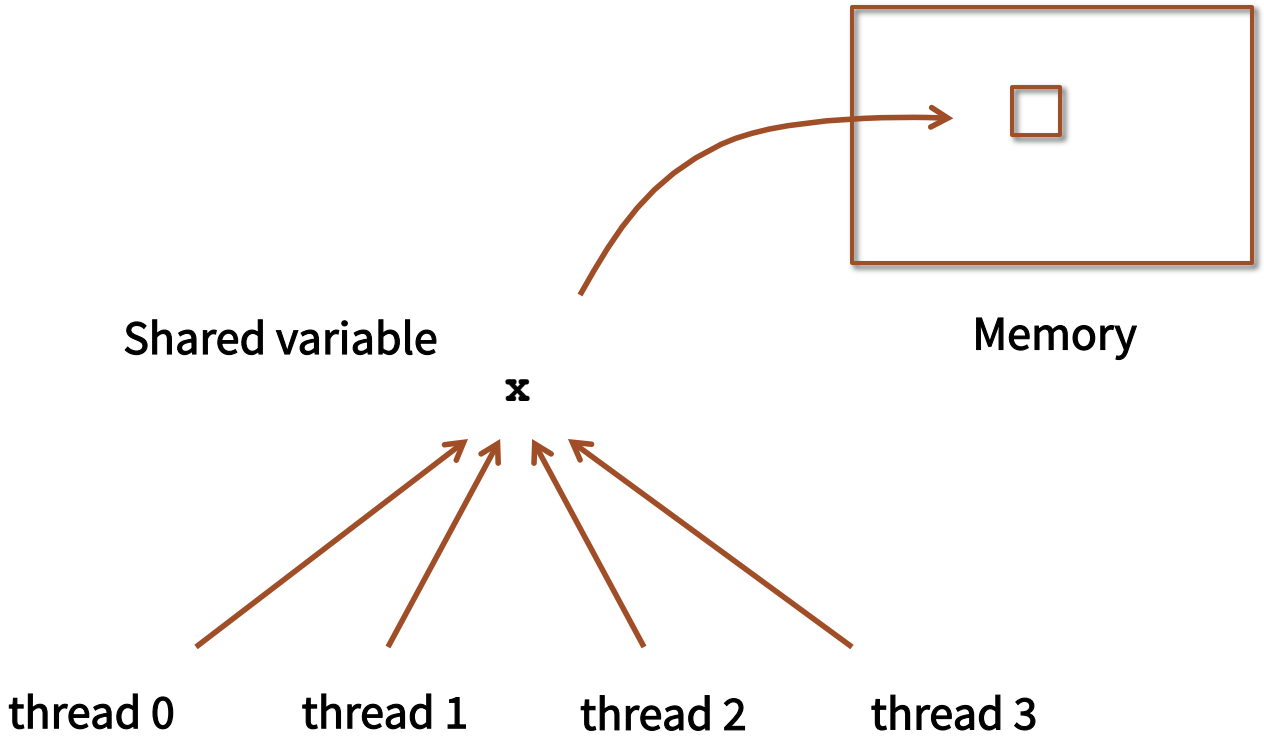
CLAUSE

- 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()

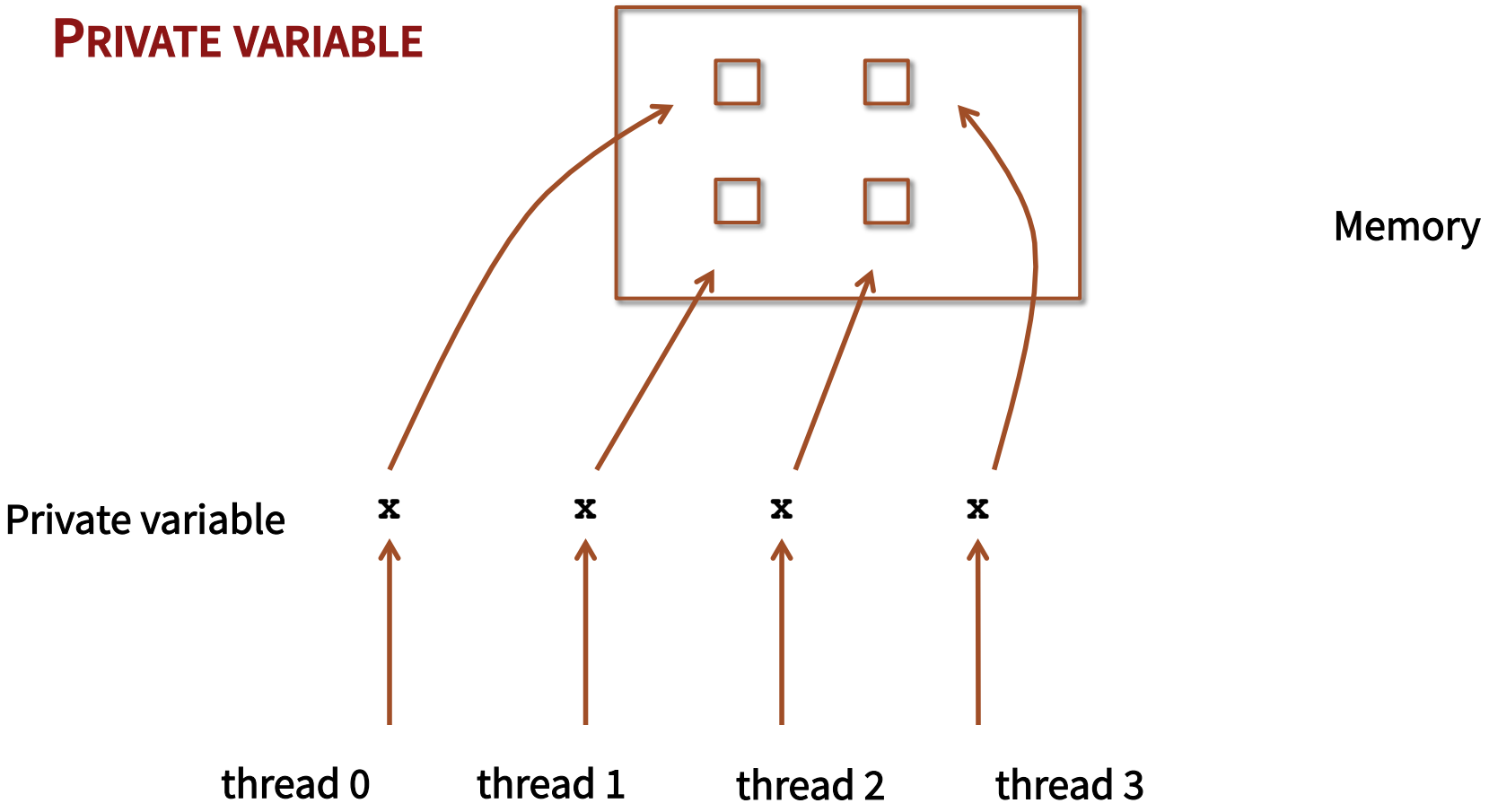
```
#include <omp.h>

void subdomain(float *x, int istart, int ipoints) {
     int i;

     for (i = 0; i < ipoints; i++)
          x[istart+i] = 123.456;
}

void sub(float *x, int npoints) {
     int iam, nt, ipoints, istart;

#pragma omp parallel shared(x,npoints) private(iam,nt,ipoints,istart)
     {
          iam = omp_get_thread_num();
          nt =  omp_get_num_threads();
          ipoints = npoints / nt;  /* size of partition */
          istart = iam * ipoints;  /* starting array index */
          if (iam == nt-1)         /* last thread may do more */
               ipoints = npoints - istart;
          subdomain(x, istart, ipoints);
     }
}

int main() {
     float array[10000];
     sub(array, 10000);
     return 0;
}
```

SHARED VARIABLE



Variable refers to the same memory location for all threads

PRIVATE VARIABLE



Variable refers to a different memory location for each thread

We will see in more details towards the end how the data-sharing attribute of a variable is determined.

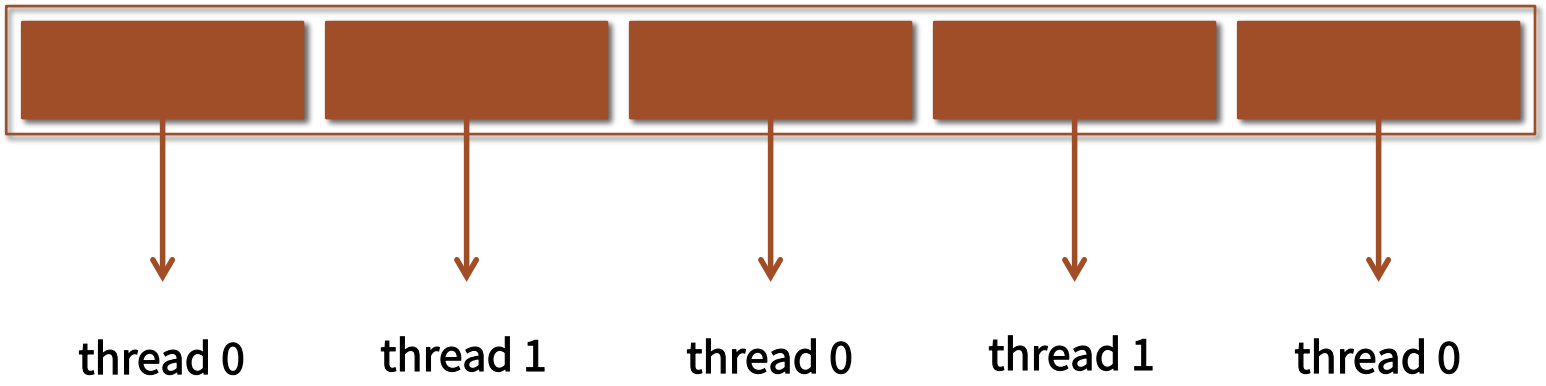
In the meantime…

WORKSHARING CONSTRUCTS

PARALLEL FOR LOOP

This is probably the most important construct in OpenMP: how to parallelize a for loop.

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



SCHEDULING FOR LOOPS (SIMPLIFIED)

How is the iteration in a for loop split among threads?

1. **`schedule(static, block_size)`**: iterations are divided into pieces of size **`block_size`** and then statically assigned to threads. If **`block_size`** is not specified, the iterations are evenly divided contiguously among the threads. Usually this is the best 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. This is similar to dynamic but the number of blocks to schedule may be smaller, which reduces the overhead.
4. If not specified, a default scheduling is chosen.

Next slide: example of a matrix multiplication: MC = MA*MB

```
#include <omp.h>

double MA[100][100], MB[100][100], MC[100][100];
int i, row, col, size = 100;

int main() {
  read_input(MA, MB);
  #pragma omp parallel shared(MA,MB,MC,size) private(row,col,i)
  {
    #pragma omp for schedule(static)
    for (row = 0; row < size; row++) {
      for (col = 0; col < size; col++)
        MC[row][col] = 0.0;
    }
    #pragma omp for schedule(static)
    for (row = 0; row < size; row++) {
      for (col = 0; col < size; col++)
        for (i = 0; i < size; i++)
          MC[row][col] += MA[row][i] * MB[i][col];
    }
  }
  write_output(MC);
}
```

OTHER WORKSHARING CONSTRUCTS: SECTIONS

- This allows specifying chunks of code that can be executed concurrently by different threads.
- A single thread executes each section.

```
#include <omp.h>
#define N 1000

int main () {

    float a[N], b[N], c[N], d[N];

    /* Some initializations */
    for (int i=0; i < N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
    }

    #pragma omp parallel shared(a,b,c,d)
    {
        #pragma omp sections
        {
            #pragma omp section
            for (int i=0; i < N; i++)
                c[i] = a[i] + b[i];

            #pragma omp section
            for (int i=0; i < N; i++)
                d[i] = a[i] * b[i];
        }  /* end of sections */

    }  /* end of parallel section */
    return 0;
}
```

SINGLE

The portion of code cannot be executed by more than one thread.

```
#include <stdio.h>
void work1() {}
void work2() {}

void single_example() {
  #pragma omp parallel
  {
    #pragma omp single
        printf("Beginning work1().\n");

    work1();

    #pragma omp single
        printf("Finishing work1().\n");

    #pragma omp single
        printf("Finished with work1(); beginning work2().\n");

    work2();
  }
}
```

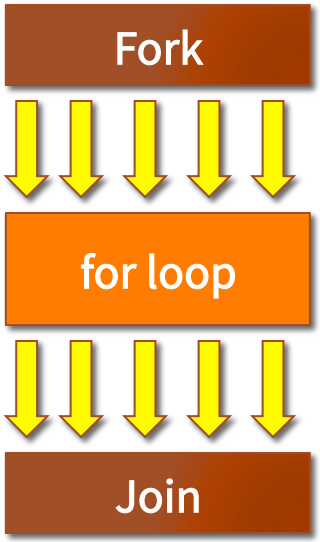
COMBINED CONSTRUCTS

Simply a shortcut:

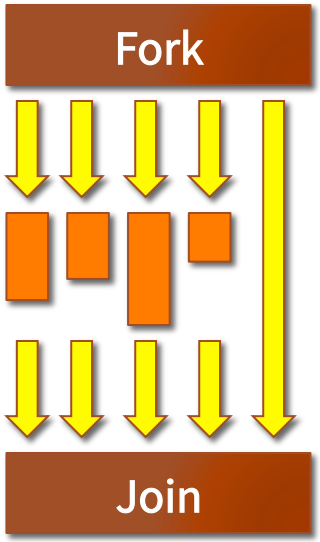
```
#pragma omp parallel for [clause [clause] ...]
for (i = lower bound; i op upper bound; incr expr) {
  { // loop body ... }
}

#pragma omp parallel sections [clause [clause] ...]
{
  [#pragma omp section]
  { // structured block ... }

  [#pragma omp section
  { // structured block ... }
  ]
}
```

These constructs combine omp parallel and omp for/sections in one line.

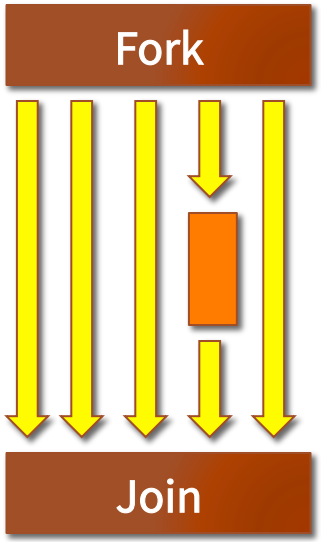
SUMMARY



Parallel for loop



Parallel sections



Single

TASKING CONSTRUCTS

TASKING CONSTRUCTS

- In the most recent version of OpenMP, the programmer can create tasks.
- This is somewhat related to sections but it is more flexible.
- Tasks are quite useful for example to process a tree using recursive functions.

```
struct node {
  struct node *left, *right;
};

void traverse( struct node *p ) {
  if (p->left)
    #pragma omp task
    traverse(p->left);
  if (p->right)
    #pragma omp task
    traverse(p->right);
  process(p);
}

int main() {
    node *root = ...;
#pragma omp parallel
#pragma omp single
    traverse(root);
}
```

TASK

- When a thread encounters a task construct, a task is generated for the associated structured block.
- The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task.
- task should be called from within a parallel region for the different specified tasks to be executed in parallel.
- The tasks will be executed in no specified order because there are no synchronization directives.

TASKWAIT

```
void postorder_traverse( struct node *p ) {
  if (p->left)
    #pragma omp task
    postorder_traverse(p->left);

  if (p->right)
    #pragma omp task
    postorder_traverse(p->right);

  #pragma omp taskwait
  process(p);
}
```

What do you think this code does?
How does the execution differ from the previous case?

In the previous code, the thread encountering **`taskwait`** needs to wait for completion of the two tasks it generated:

```
postorder_traverse(p->left);
```

and

```
postorder_traverse(p->right);
```

TECHNICAL EXPLANATION ON TASKWAIT

- OpenMP defines the concept of child task. A child task of a piece of code (region) is a task generated by a directive
  `#pragma omp task`
  found in that piece of code.
- For example, in the previous code `postorder_traverse(p->left)` and `postorder_traverse(p->right)` are child tasks of the enclosing region.
- `taskwait` specifies a wait on the completion of the child tasks of the current task (precisely the region the current task is executing).
- Note that `taskwait` requires to wait for completion of the child tasks, but not completion of all descendant tasks (e.g., child tasks of child tasks).

TREE TRAVERSAL USING SECTIONS

If we wanted to use sections we would write:

```
void traverse( struct node *p ) {
  #pragma omp parallel sections
  {
    #pragma omp section
    if (p->left)
      traverse(p->left);
    #pragma omp section
    if (p->right)
      traverse(p->right);
  }
  process(p);
}
```

- The problem with the previous code is that each thread entering one of the sections will call **`traverse`**, which leads to the creation of a new parallel region because of

```
#pragma omp parallel sections
```

- The result is that this makes it more difficult in general to control the number of threads being generated by this implementation.