UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Shared memory: OpenMP

Implicit threads: motivations

- Implicit threading frameworks and libraries take care of much of the minutiae needed to create, manage, and (to some extent) synchronize threads.

- They greatly **simplify concurrent programming** hiding details, **at the expense of expressiveness and flexibility**.

- So far we have seen explicit threading libraries for Java and C/C++ …

- … in this lecture we are going to see an example of implicit threading framework for C/C++ and Fortran: OpenMP

Implicit threads: motivations

- OpenMP **does not require restructuring** the serial program. The use only needs to add compiler directives to reconstruct the serial program into a parallel one.

- If no OpenMP library function is used, i.e. only compiler directives are used, then the structure of the program remains the same !

OpenMP

- OpenMP is an API for shared-memory parallel programming.

- The “MP” in OpenMP stands for “multiprocessing,” a term that is synonymous with shared-memory parallel computing.

- OpenMP is designed for systems in which each thread or process can potentially have access to all available memory. When using OpenMP, we view our system as a collection of cores or CPUs, all of which have access to main memory,

- It’s not just a library: it’s a set of compiler directives (must be supported by the compiler), library routines, and environment variables.

OpenMP

- Compiler directives: instruct the compiler on how to parallelize the code.
  - This is the core of OpenMP.
- Runtime library functions: modify and check the number of threads and check how may processors there are in the multiprocessor system.
- Environment variables to alter the execution of OpenMP applications (e.g. the number of max. threads to use).

OpenMP and compilers

- OpenMP has to be supported by the compiler.
- Modern C/C++ compilers support OpenMP:
  - Microsoft Visual Studio
  - Intel C/C++ for Windows and Linux
  - GCC
  - Clang
    - notable exception: Apple Clang - when programming under OS X install clang or gcc, e.g. using Macports or Homebrew
  - Use the latest version of compilers to use the latest standard. Typically Clang was lagging behind gcc but now it’s almost on parity.

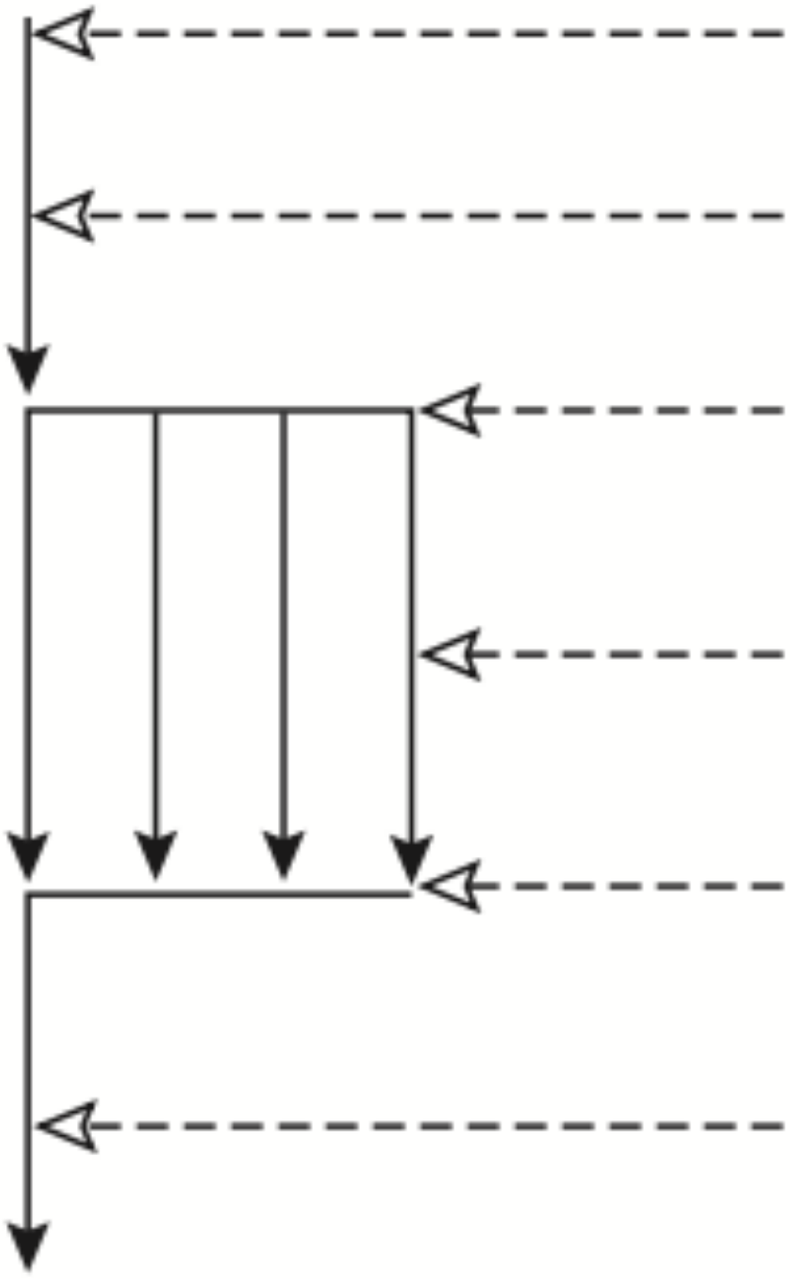
OpenMP and compilers

- A compiler that support OpenMP defines the symbol `_OPENMP`
- OpenMP is an active standard: check what version is supported by the compiler.
  - E.g. using `task` directive require support for OpenMP 3.0
- Current version is 4.5

Thread model

- The OpenMP implicit threading follows the fork-join model:
  - the programmer indicates the regions of code that can be executed in parallel
  - when the compiled program founds those regions the main program continues as “main thread” and…
    - … multiple threads are forked, executing the parallel region.
    - Once a thread of this pool finishes it waits the end of the other threads (join)
    - When all threads have reached the “barrier” of joins the program continues executing the first instruction after the parallel region
- The parallel execution mode is SPMD, but it is possible to assign different tasks to different threads.

Thread model



- Master thread executes serial portion of the code.
- Master thread enters a routine with parallelized section.
- Master thread encounters parallel directive.
  Creates slave threads.
- Master and slave threads divide iterations of parallel code and execute them concurrently.
- Implicit barrier: Wait for all threads to finish their iterations.
- Master thread resumes execution after the parallelized code. Slave threads disappear.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Sequential consistency

Sequential Consistency

- It has been defined as the property that requires that:

"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

Sequential Consistency

- The result of a parallel program is always the same as the sequential program, irrespective of the statement interleaving that is a result of parallel execution

Sequential program	Parallel program 1	Parallel program n
`a = 5;`	`a = 5;`	`x = 1;`
`x = 1;`	`x = 1;`	`y = x+3;`
…	…	…
`y = x+3;`	`y = x+3;`	`a = 5;`
…	…	`z = x+y;`
`z = x+y;`	`z = x+y;`	…

Sequential Consistency

- The result of a parallel program is always the same as the sequential program, irrespective of the statement interleaving that is a result of parallel execution

Parallel programs have any order of permitted statement interleaving

Sequential program

```
a = 5;
x = 1;
…
y = x+3;
…
z = x+y;
```

Parallel program 1

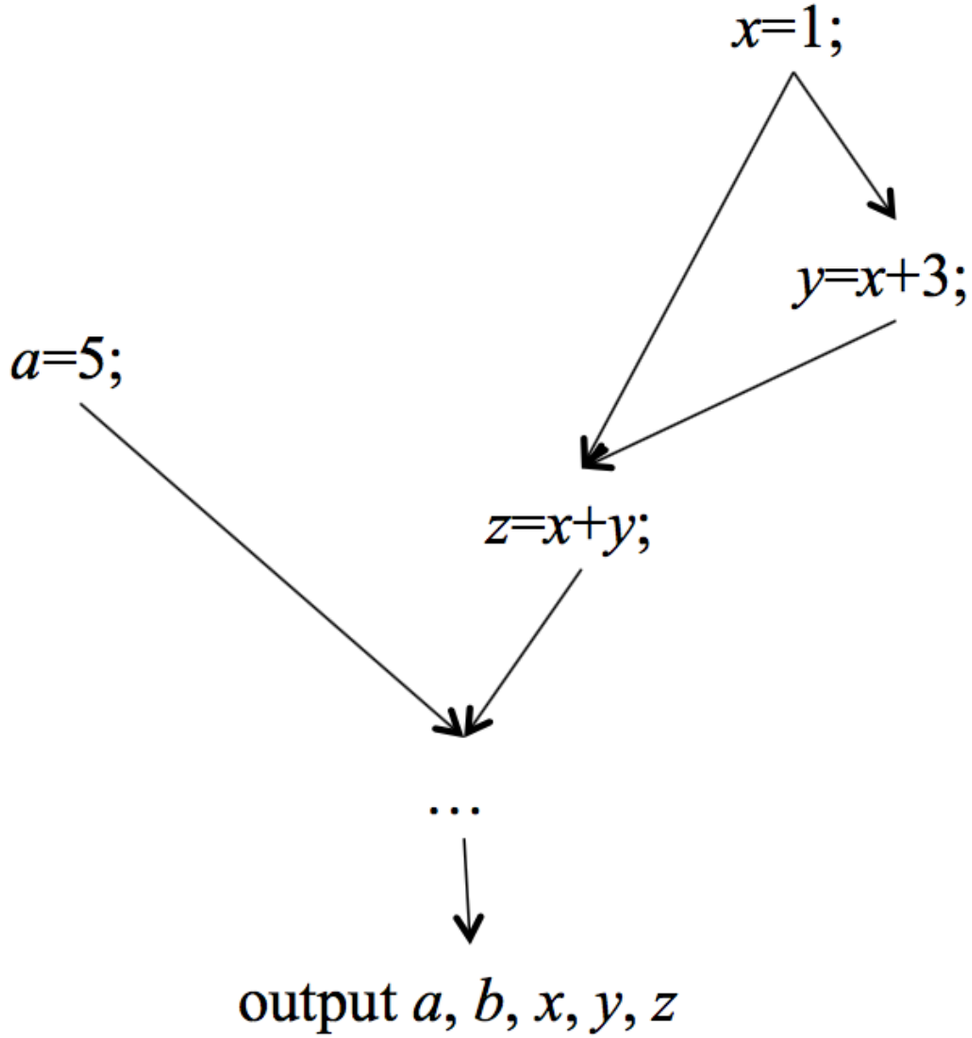
```
a = 5;
x = 1;
…
y = x+3;
…
z = x+y;
```

Parallel program n

```
x = 1;
y = x+3;
…
a = 5;
z = x+y;
…
```

Flow dependances

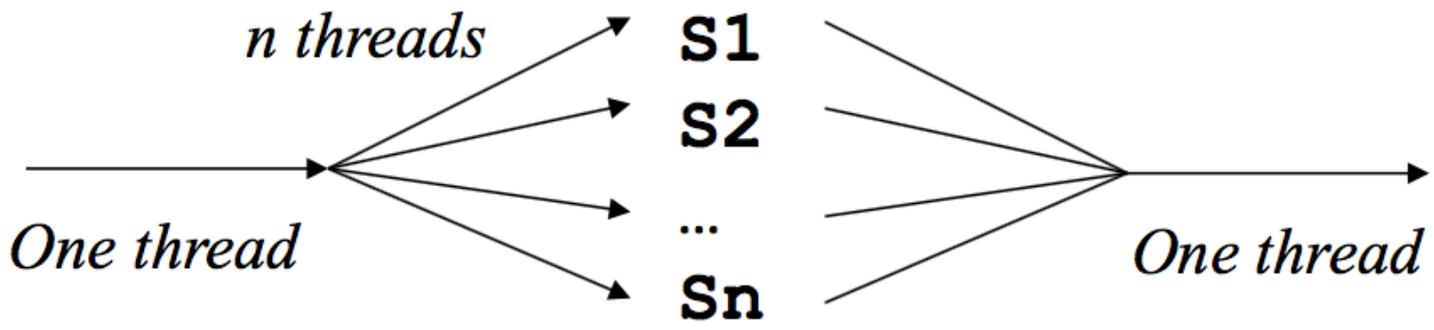
- Flow dependences determine the parallel execution schedule: each operation waits until operands are produced



Parallel programming constructs

- With the par construct the statements in the body are executed concurrently:

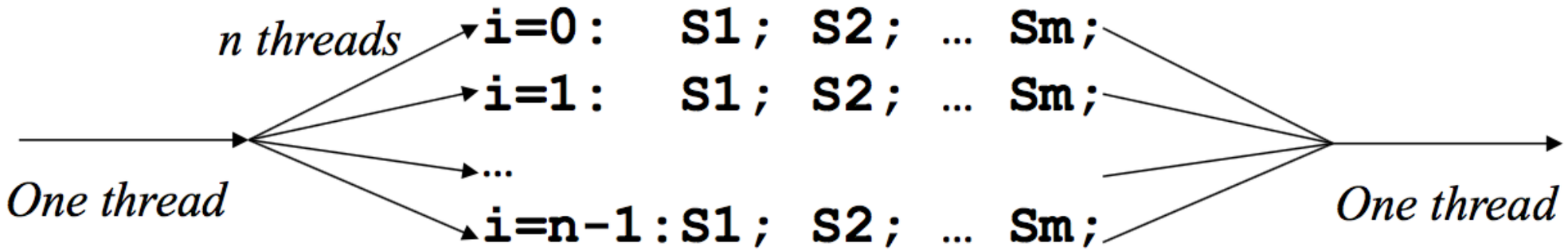
```
par {
   S1;
   S2;
   . . .
   Sn;
}
```



Parallel programming constructs

- With the `parfor` construct (also known as `forall`) the statements in the body are executed in serial order by n threads `i=0..n-1` in parallel

```
parfor (i=0; i<n; i++) {
   S1;
   S2;
   . . .
   Sn;
}
```



Data flow and dependance

- Considering data flow, we notice that each operation requires completion of operands first.
  If data dependancies are preserved the sequential consistency is guaranteed.

```
par {          x = 1;          x = 1;
  a = 5;       . . .           . . .
  x = 1;       par {           par {
}                a = 5;          a = 5;
. . .            y = x+3;        y = x+3;
y = x+3;     }                   z = x+y;
. . .        . . .           }
z = x+y;     z = x+y;        . . .
```

Data flow and dependance

- Considering data flow, we notice that each operation requires completion of operands first.
  If data dependancies are preserved the sequential consistency is guaranteed.

```
par {
   a = 5;
   x = 1;
}
. . .
y = x+3;
. . .
z = x+y;
```

```
x = 1;
. . .
par {
   a = 5;
   y = x+3;
}
. . .
z = x+y;
```

```
x = 1;
. . .
par {
   a = 5;
   y = x+3;
   z = x+y;
}
. . .
```

Bernstein's Conditions

- Processes cannot execute in parallel if one affects values used by the other.
  Nor can they execute in parallel if any subsequent process uses data affected by both, i.e. whose value might depend on the order of execution.

- I_i is the set of memory locations read by process P_i
- O_j is the set of memory locations altered by process P_j
- Processes P_1 and P_2 can be executed concurrently if all of the following conditions are met
- $I_1 \cap O_2 = \varnothing$
- $I_2 \cap O_1 = \varnothing$
- $O_1 \cap O_2 = \varnothing$

These three constraints are too rigid if we want to share memory; however, if we provide some means to enforce a precedence among process sharing memory then we can relax them.

Dependence analysis

- Dependence analysis performed by a compiler determines that Bernstein's conditions are not violated when optimizing and/or parallelizing a program

independent	RAW	WAR	WAW
P_1: A=x+y; P_2: B=x+z;	P_1: A=x+y; P_2: B=x+A;	P_1: A=x+B; P_2: B=x+z;	P_1: A=x+y; P_2: A=x+z;
$I_1 \cap O_2 = \varnothing$ $I_2 \cap O_1 = \varnothing$ $O_1 \cap O_2 = \varnothing$	$I_1 \cap O_2 = \varnothing$ $I_2 \cap O_1 = \{A\}$ $O_1 \cap O_2 = \varnothing$	$I_1 \cap O_2 = \{B\}$ $I_2 \cap O_1 = \varnothing$ $O_1 \cap O_2 = \varnothing$	$I_1 \cap O_2 = \varnothing$ $I_2 \cap O_1 = \varnothing$ $O_1 \cap O_2 = \{A\}$

Dependence analysis

- Dependence analysis performed by a compiler determines that Bernstein's conditions are not violated when optimizing and/or parallelizing a program

independent	RAW	WAR	WAW

par {
A=x+y;
B=x+z;
}
	P_1: A=x+y;
	P_2: B=x+A;
		P_1: A=x+B;
		P_2: B=x+z;
			P_1: A=x+y;
			P_2: A=x+z;

$I_1 \cap O_2 = \varnothing$	$I_1 \cap O_2 = \varnothing$	$I_1 \cap O_2 = \{B\}$	$I_1 \cap O_2 = \varnothing$
$I_2 \cap O_1 = \varnothing$	$I_2 \cap O_1 = \{A\}$	$I_2 \cap O_1 = \varnothing$	$I_2 \cap O_1 = \varnothing$
$O_1 \cap O_2 = \varnothing$	$O_1 \cap O_2 = \varnothing$	$O_1 \cap O_2 = \varnothing$	$O_1 \cap O_2 = \{A\}$

Bernstein's Conditions in Loops

- A parallel loop is valid when any ordering of its parallel body yields the same result

- ```
  parfor (I=4;I<7;I++)
      S1: A[I] = A[I-3]+B[I];
  ```

S1(4): A[4] = A[1]+B[4];
S1(5): A[5] = A[2]+B[5];
S1(6): A[6] = A[3]+B[6];

S1(6): A[6] = A[3]+B[6];
S1(5): A[5] = A[2]+B[5];
S1(4): A[4] = A[1]+B[4];

S1(5): A[5] = A[2]+B[5];
S1(4): A[4] = A[1]+B[4];
S1(6): A[6] = A[3]+B[6];

S1(6): A[6] = A[3]+B[6];
S1(4): A[4] = A[1]+B[4];
S1(5): A[5] = A[2]+B[5];

S1(4): A[4] = A[1]+B[4];
S1(6): A[6] = A[3]+B[6];
S1(5): A[5] = A[2]+B[5];

S1(5): A[5] = A[2]+B[5];
S1(6): A[6] = A[3]+B[6];
S1(4): A[4] = A[1]+B[4];
  ```

Loops and parallelization

```
for(i=1; i<10; i++)
  A[i] = A[i-1];// S1
```

The instances of S_1 can be executed only in sequential order because of flow dependance

```
for(i=1; i<10; i++)
  A[i] = A[i+1];// S1
```

can be parallelized with:

```
parfor(i=1;i<10;i++)
  B[i] = A[i+1]

parfor(i=1;i<10;i++)
  A[i] = B[i]
```

```
for(i=1; i<10; i++)
  A[i] = A[i-k];// S1
```

allows a degree k of parallelization

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Overview of OpenMP in C/C++

#pragma

- The #pragma directives offer a way for each compiler to offer machine- and operating system-specific features.

- If the compiler finds a pragma it does not recognize, it issues a warning, but compilation continues.

OpenMP directive

- The general form of an OpenMP directive is

```
#pragma omp directive [clauses [ ] ...]
```

- written in a single line. Use \ to split the directive on more lines, as usual for preprocessor directives, e.g.:

```
#pragma omp directive \
        [clauses [ ] ...]
```

- The clauses are optional and are different for different directives. Clauses are used to influence the behavior of a directive. Directives are case sensitive.

OpenMP directive

- The general form of an OpenMP directive is

  #p .]

- wi on
  m .g.:

```
#pragma omp directive \
        [clauses [ ] ...]
```

The clause list contains information about:

- conditional parallelization
- degree of concurrency
- data handling between serial/parallel code

- The clauses are optional and are different for different directives. Clauses are used to influence the behavior of a directive. Directives are case sensitive.

Parallel region

- Parallel regions are indicated using a `#pragma` pre-processor directive (`#pragma omp`)

- Regions are single instructions or block of code (indicated with { and })

- Example on how to parallelize a region of code:

```
#pragma omg parallel
{
    // code
}
```

Compiling with OpenMP

- Tell the compiler to use OpenMP with an appropriate switch
  - `-fopenmp`
    - Both clang and gcc require this switch
- You may need to tell the compiler where are the headers and libraries
  - e.g. on OSX you have to manually install them with Macports or Homebrew (libomp)

Compiling with OpenMP

- Tell the compiler to use OpenMP with an appropriate switch
  - `-fopenmp`

```
set(CMAKE_CXX_COMPILER "/opt/local/bin/clang++-mp-3.9")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -fopenmp")
```

- You may need to tell the compiler where are the headers and libraries
  - e.g. on OSX you have to manually install them with Macports or Homebrew (libomp)

Hello World with OpenMP

CMakeLists.txt × | main.cpp ×

```cpp
#include <omp.h> // for OpenMP library functions
#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();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

    }  /* All threads join master thread and disband */

}
```

Hello World with OpenMP

CMakeLists.txt × | main.cpp ×

```
#include <omp.h> // for OpenMP library functions
#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();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

    }  /* All threads join master thread and disband */

}
```

pragma directive

Hello World with OpenMP

include directive to use OpenMP library functions

pragma directive

CMakeLists.txt × main.cpp ×

```cpp
#include <omp.h> // for OpenMP library functions
#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();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

    }  /* All threads join master thread and disband */

}
```

Hello World with OpenMP

CMakeLists.txt × main.cpp ×

```
#include <omp.h> // for OpenMP library functions
#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();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

    }  /* All threads join master thread and disband */

}
```

include directive to use OpenMP library functions

pragma directive

OpenMP library functions

OpenMP library functions

Hello World with OpenMP

CMakeLists.txt × | main.cpp ×

```
#include <omp.h> // for OpenMP library functions
#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();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

    }  /* All threads join master thread and disband */

}
```

pragma directive

OpenMP library functions

OpenMP library functions

Hello World with OpenMP

CMakeLists.txt × | main.cpp ×

```cpp
#include <omp.h> // for OpenMP library functions
#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();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

    }  /* All threads join master thread and disband */

}
```

OpenMP library functions

OpenMP library functions

Hello World with OpenMP

CMakeLists.txt × | main.cpp ×

```cpp
#include <omp.h> // for OpenMP library functions
#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();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

    }  /* All threads join master thread and disband */

}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Hello World with OpenMP

CMakeLists.txt × | main.cpp ×

```cpp
#include <omp.h> // for OpenMP library functions
#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();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

    }  /* All threads join master thread and disband */

}
```

Parallel code

Parallel loops

- Loops are good candidates for parallelization.
- OpenMP has a special construct: `parallel for` to divide loop blocks to the threads of the team
- It is possible to use **dynamic** scheduling (blocks are assigned to threads once they finish their previous block) or **static** scheduling (predefined # of threads each receiving only one block)
- Use a **chunk** argument to set the size of each block

Variables and threads

- By default variables in an OpenMP threaded program are shared between threads, **except**:
  - the loop index variable associated with a loop construct (each thread must have its own copy in order to correctly iterate through the assigned set of iterations);
  - variables declared within a parallel region or declared within a function that is called from within a parallel region;
  - any other variable that is placed on the thread's stack (e.g., function parameters).
- If you need a local copy use a `private` clause (as in the previous example). A local copy of the variables in the list will be allocated for each thread. The initial value of variables that are listed within the private clause will be undefined, and you must assign value to them before they are read within the region of use.

Synchronization

- A `critical` construct acts like a lock around a critical region. Only one thread may execute within a protected critical region at a time.

- An `atomic` construct ensures that statements will be executed in an atomic, uninterruptible manner.
  There is a restriction on which types of statements you can use with the atomic construct, and you can only protect a single statement.

- The `single` and `master` constructs will control execution of statements within a parallel region so that only one thread will execute those statements (as opposed to allowing only one thread at a time).

  - `single` will use the first thread that encounters the construct,
  - `master` will allow only the master thread (the thread that executes outside of the parallel regions) to execute the protected code.

Synchronization

- A `barrier` directive defines a point where each thread waits for all other threads to arrive. Once all the threads arrive at that point, they can all continue execution past the barrier.

- Each thread is therefore guaranteed that all the code before the barrier has been completed across all other threads.

Reduction

- A reduction clause may help in reducing synchronization issues due to critical sections: often there's need to apply the same operation on a shared variable… but we do not want to pay sync cost.

- A **reduction operator** is a **binary operation** (such as addition or multiplication) and a reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result. Furthermore, all of the intermediate results of the operation should be stored in the same variable: the reduction variable.

- OpenMP creates a private variable for each thread, and the run-time system stores each thread's result in this private variable. OpenMP also creates a critical section and the values stored in the private variables are added in this critical section.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

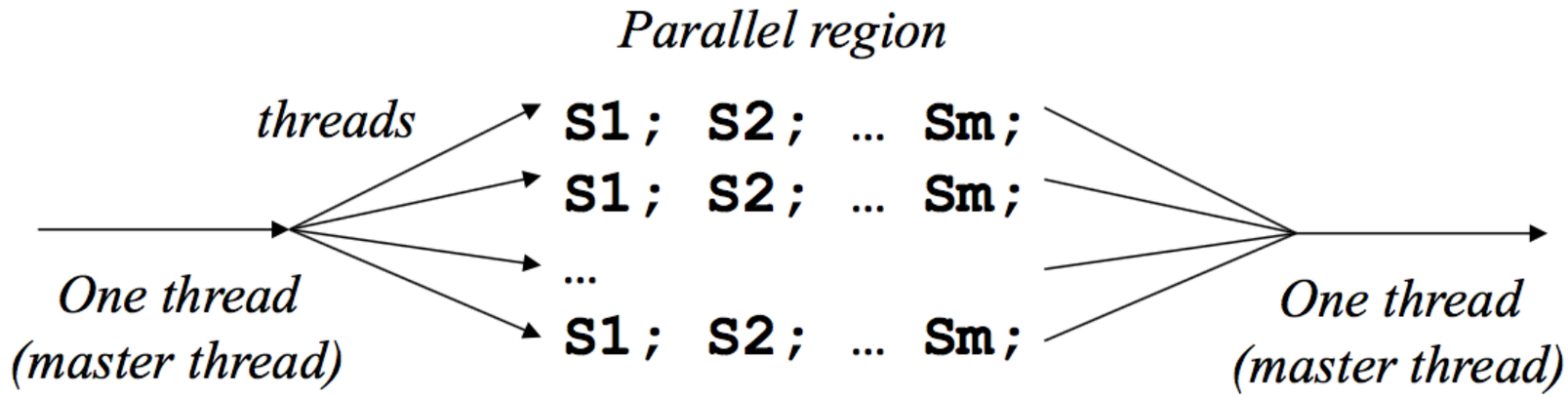
Overview of OpenMP directives

Parallel

- A team of threads all execute the body statements and joins when done

  - Note: is not the same as **par**

- ```
  #pragma omp parallel
  {
     S1;
     S2;
     ...
     Sm;
  }
  ```


  ```

`shared` / `private` clauses

- Specific clauses instruct wether variables are shared by thread or private, or set the default visibility of variables, e.g.:

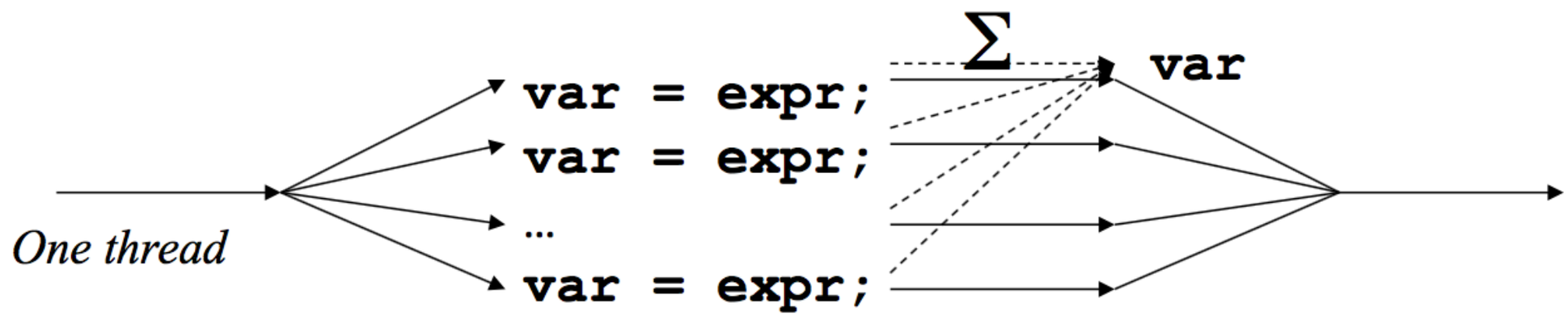
```
#pragma omp parallel default(none) \
    shared(var1, var2) private(var3)
```

- means that variables should not be shared by default, that `var1` and `var2` are shared by threads while `var3` is private in each thread

Parallel with reduction clause

- ```
  #pragma omp parallel reduction(+:var)
  {
     var = expr;
     ...
  }
  ... = var;
  ```

- Performs a global reduction operation over privatized variable(s) and assigns final value to master's private variable(s) or to the shared variable(s) when shared

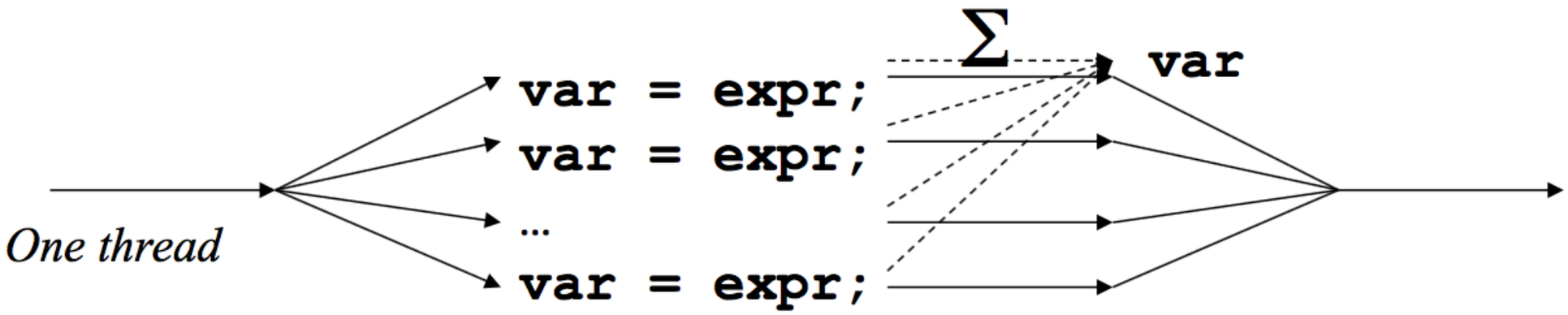

  ```

Parallel with reduction clause

- ```
  #pragma omp parallel reduction(+:var)
  {
     var = expr;
     ...
  }
  ... = var;
  ```

+, *, -, &, ^, |, &&, ||

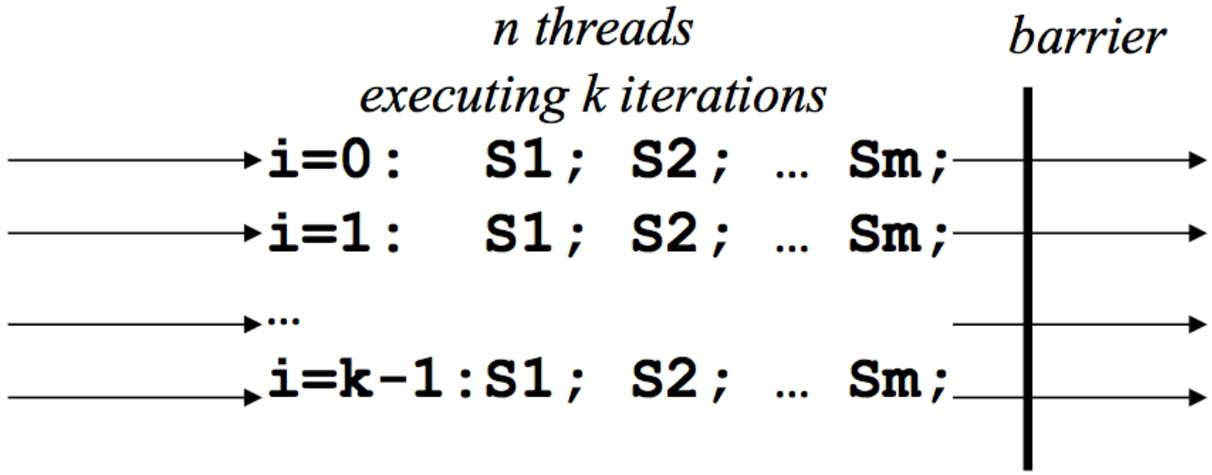
- Performs a global reduction operation over privatized variable(s) and assigns final value to master's private variable(s) or to the shared variable(s) when shared


  ```

for loop

- An existing team of threads is used to execute a loop concurrently
- Loop iterations are executed concurrently by *n* threads
  Use nowait clause to remove the implicit barrier.

```
#pragma omp parallel
. . .
#pragma omp for
for (i=0; i<k; i++) {
   S1;
   S2;
   . . .
   Sm;
}
```



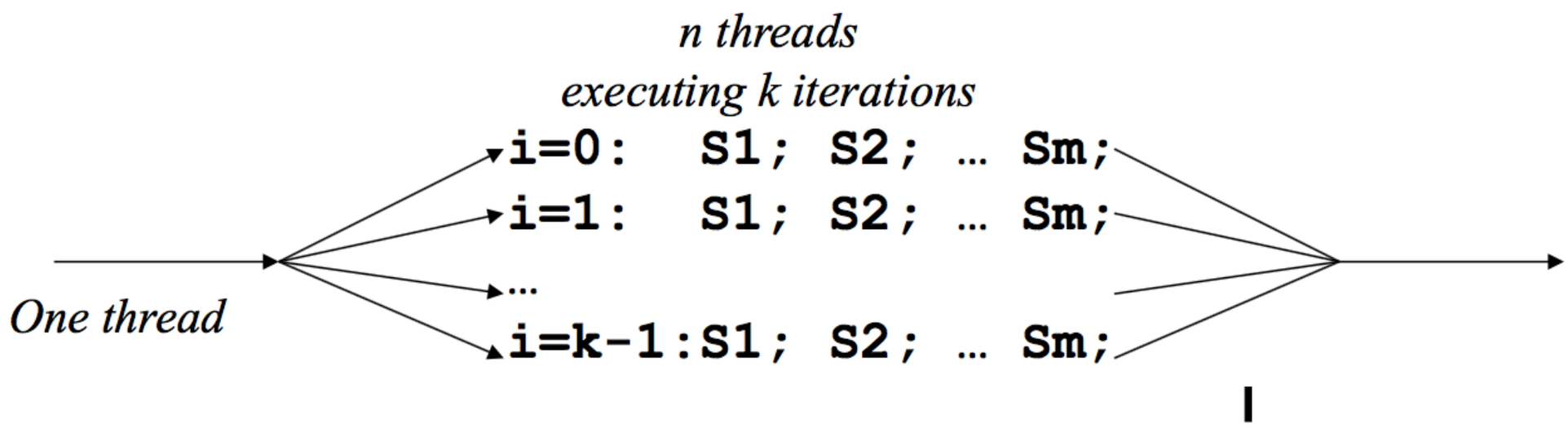
for loop

- An existing team of threads is used to execute a loop concurrently
- Loop iterations are executed concurrently by *n* threads
  Use nowait clause to remove the implicit barrier.

Simplify with:

```
#pragma omp parallel for
```

```
for (i=0; i<k; i++) {
   S1;
   S2;
   . . .
   Sm;
}
```

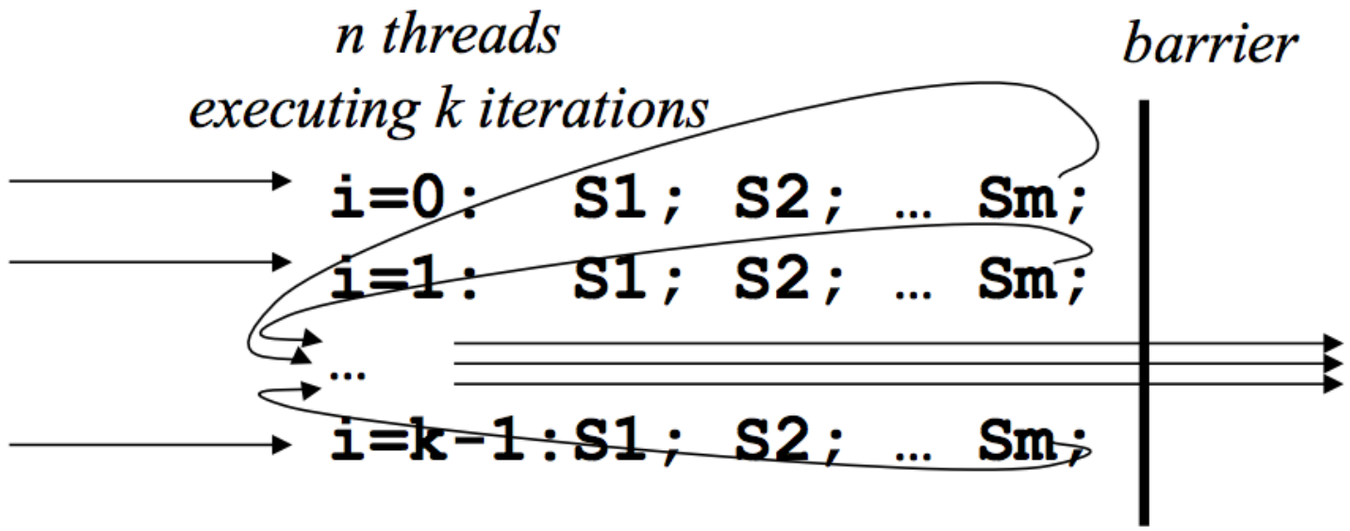


`for` loop scheduling

- Assignment of work loads to threads is controlled with a schedule clause, e.g.:

- `#pragma omp for schedule(dynamic)`

- When k>n, threads execute randomly chosen loop iterations until all iterations are completed



for loop scheduling

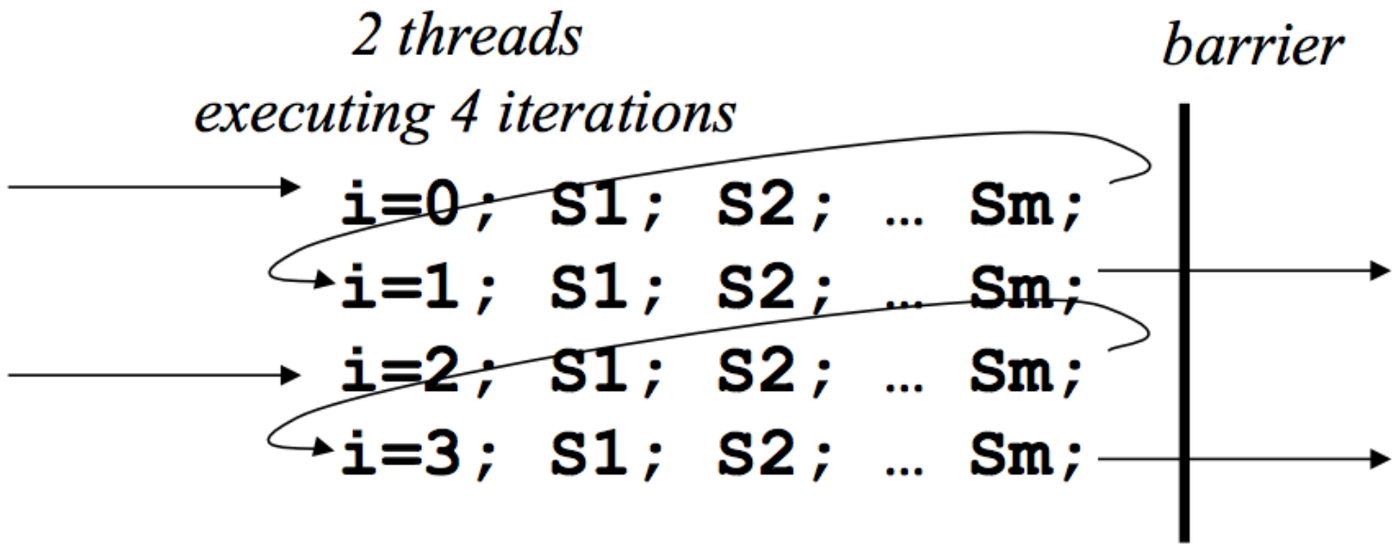
- Assignment of work loads to threads is controlled with a schedule clause, e.g.:

`for` loop scheduling

- Assignment of work loads to threads is controlled with a schedule clause, e.g.:

- `#pragma omp for schedule(static)`

- When k>n, threads are assigned to k/n chunks of the iteration space

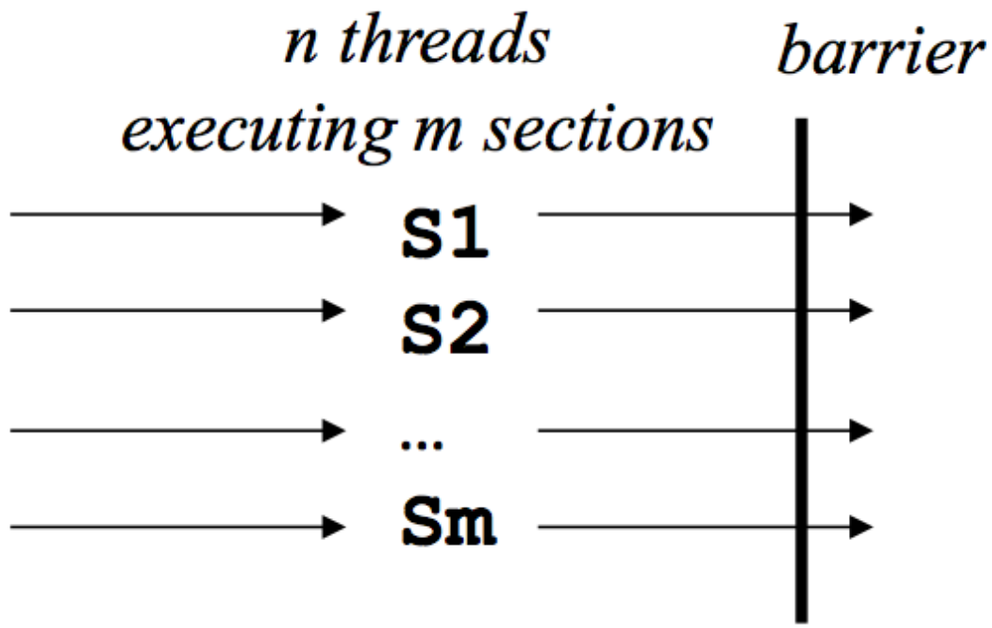


sections

- The sections construct is for work-sharing, where a current team of threads is used to execute statements of each section concurrently

```
#pragma omp parallel
. . .
#pragma omp sections
{
  #pragma omp section
  S1;

  #pragma omp section
  S2;
  . . .
  #pragma omp section
  Sm;
}
```



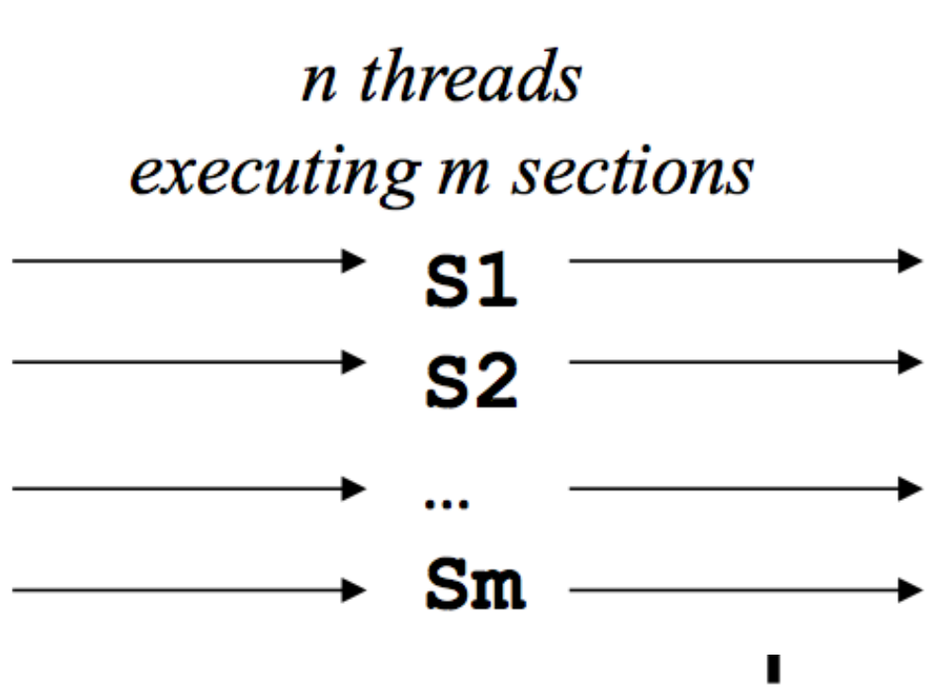
sections

- The sections construct is for work-sharing, where a current team of threads is used to execute statements of each section concurrently

```
#pragma omp parallel
. . .
#pragma omp sections
{
  #pragma omp section
  S1;

  #pragma omp section
  S2;
  . . .
  #pragma omp section
  Sm;
}
```

Use a `nowait` clause to remove implicit barrier

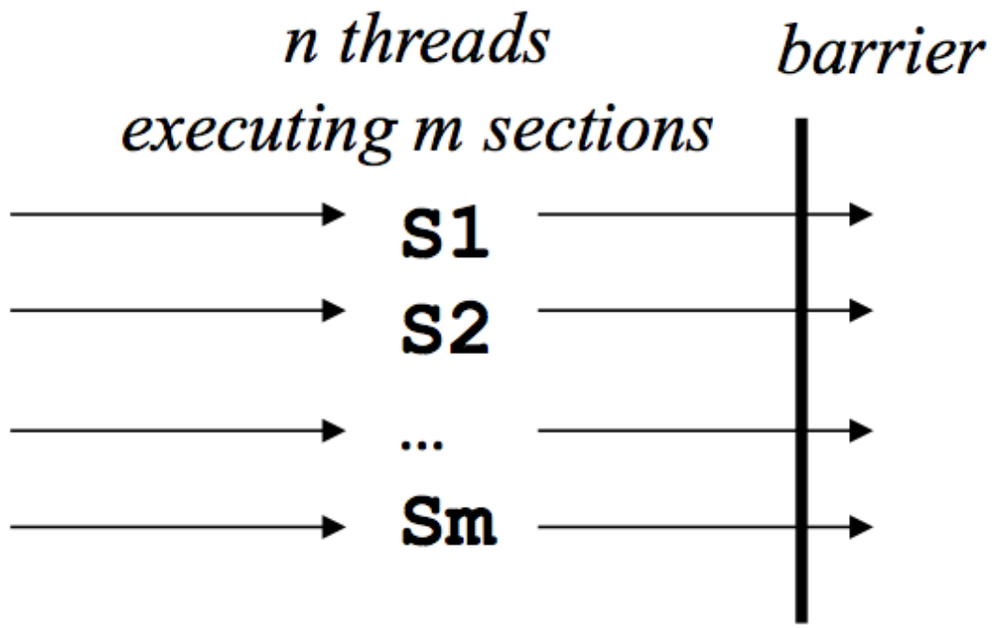


sections

- The sections construct is for work-sharing, where a current team of threads is used to execute statements of each section concurrently

```
#pragma omp parallel
. . .
#pragma omp sections
{
  #pragma omp section
  S1;

  #pragma omp section
  S2;
  . . .
  #pragma omp section
  Sm;
}
```



sections

- The sections construct is for work-sharing, where a current team of threads is used to execute statements of each section concurrently

```
#pragma omp parallel
. . .
#pragma omp sections
{
  #pragma omp section
  S1;

  #pragma omp section
  S2;
  . . .
  #pragma omp section
  Sm;
}
```

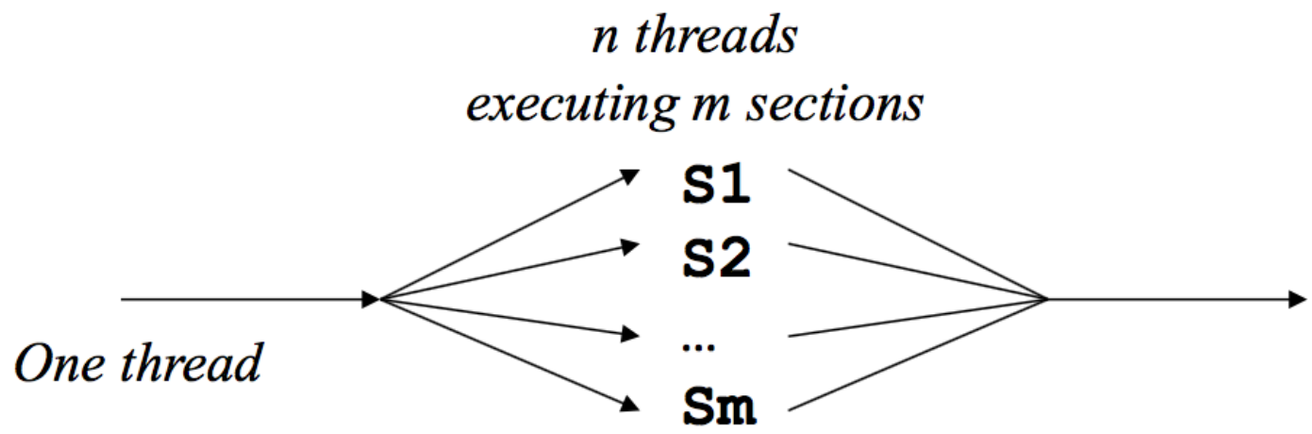
sections

- The sections construct is for work-sharing, where a current team of threads is used to execute statements of each section concurrently

Simplify with:
`#pragma omp parallel sections`

```
{
  #pragma omp section
  S1;

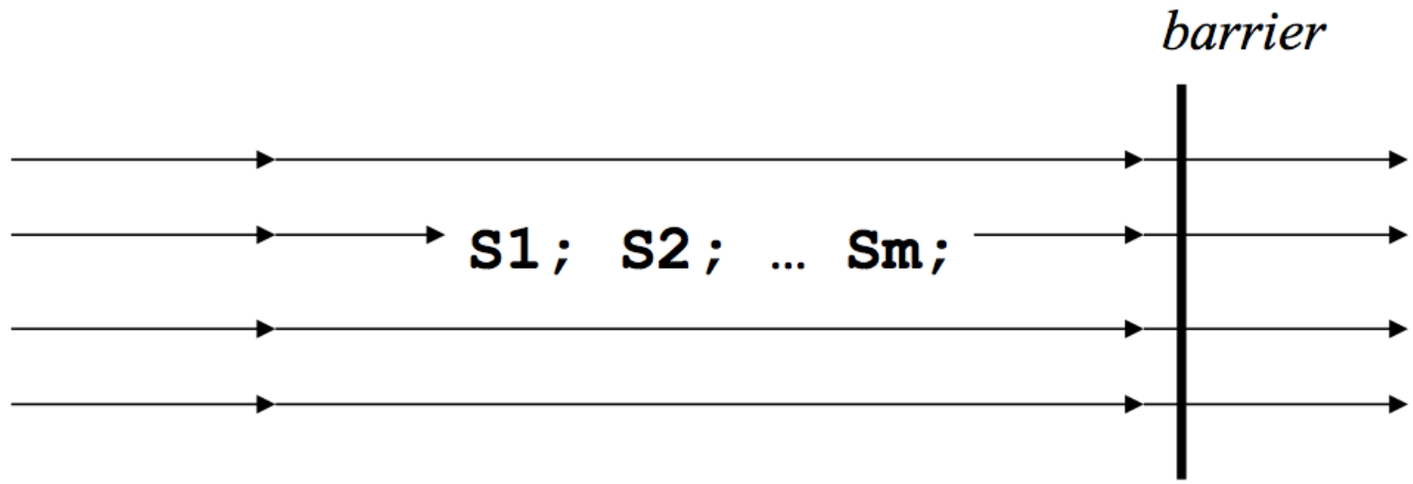
  #pragma omp section
  S2;
  . . .
  #pragma omp section
  Sm;
}
```



`single` execution

- The single construct selects one thread of the current team of threads to execute the body

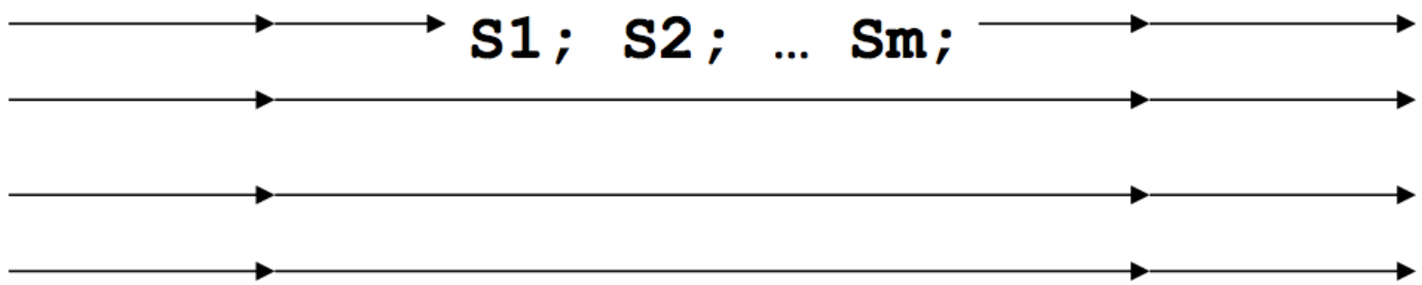
```
#pragma omp parallel
. . .
#pragma omp single
{
   S1;
   S2;
   ...
   Sm;
}
```



master execution

- The master construct selects the master thread of the current team of threads to execute the body, no barrier is inserted

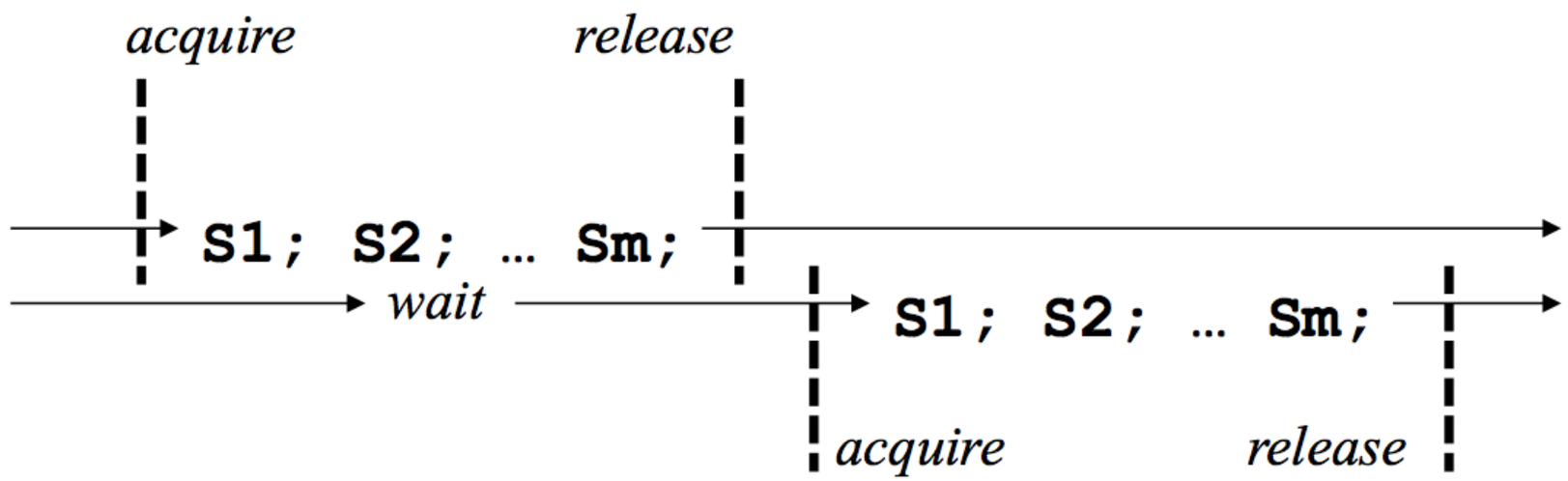
```
#pragma omp parallel
...
#pragma omp master
{
   S1;
   S2;
   ...
   Sm;
}
```



critical section

- The critical construct defines a critical section. Mutual exclusion is enforced on the body using a (named) lock

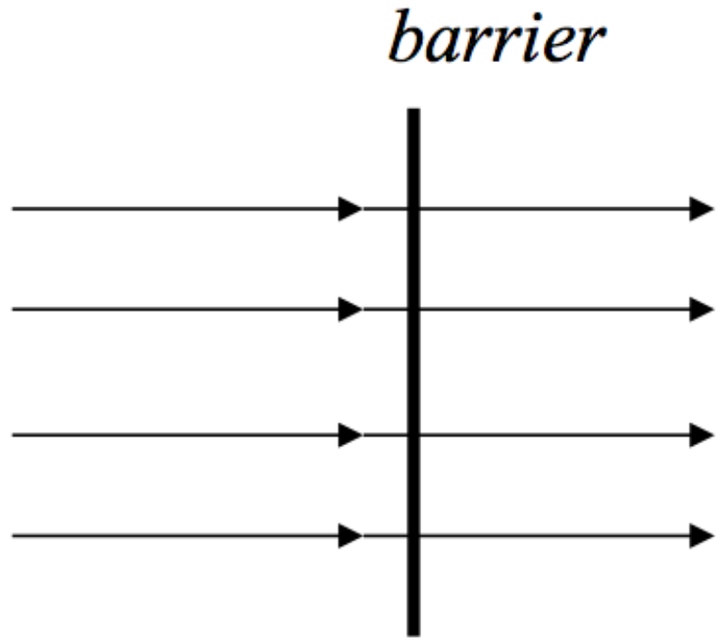
```
#pragma omp parallel
. . .
#pragma omp critical [name]
{
   S1;
   S2;
   . . .
   Sm;
}
```



`barrier` synchronization

- The barrier construct synchronizes the current team of threads

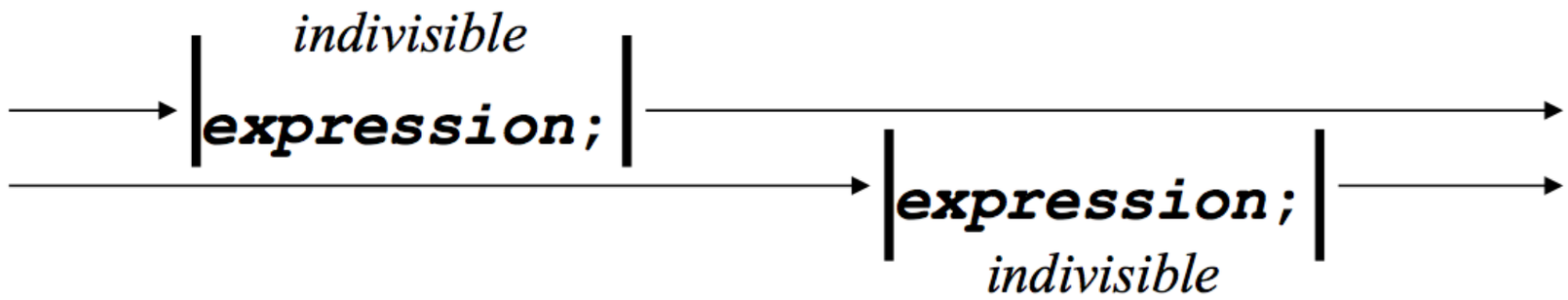
```
#pragma omp parallel
{
...
#pragma omp barrier
}
```



atomic execution

- The atomic construct executes an expression atomically (expressions are restricted to simple updates)

```
#pragma omp parallel
...
#pragma omp atomic
expression;
```



Locking

- Mutex locks, with additional “nestable” versions of locks that can be locked multiple times by the same thread

```
omp_lock_t lck;

omp_init_lock(&lck);
omp_set_lock(&lck);
...
... critical section ...
...
omp_unset_lock(&lck);
omp_destroy_lock(&lck);
```

Credits

- These slides report material from:
  - Prof. Robert van Engelen (Florida State University)
  - Prof. Jan Lemeire (Vrjie Universiteit Brussel)
  - Prof. Robert M. Keller (Harvey Mudd College)

Books

- The Art of Concurrency, Clay Breshears, O'Reilly - Chapt. 5

- Principles of Parallel Programming, Calvin Lyn and Lawrence Snyder, Pearson - Chapt. 6

- Parallel Programming for Multicore and Cluster Systems, Thomas Dauber and Gudula Rünger, Springer - Chapt. 6

- An introduction to parallel programming, Peter S. Pacheco, Morgan Kaufman - Chapt. 5