Programming with Shared Memory PART II

HPC Fall 2012

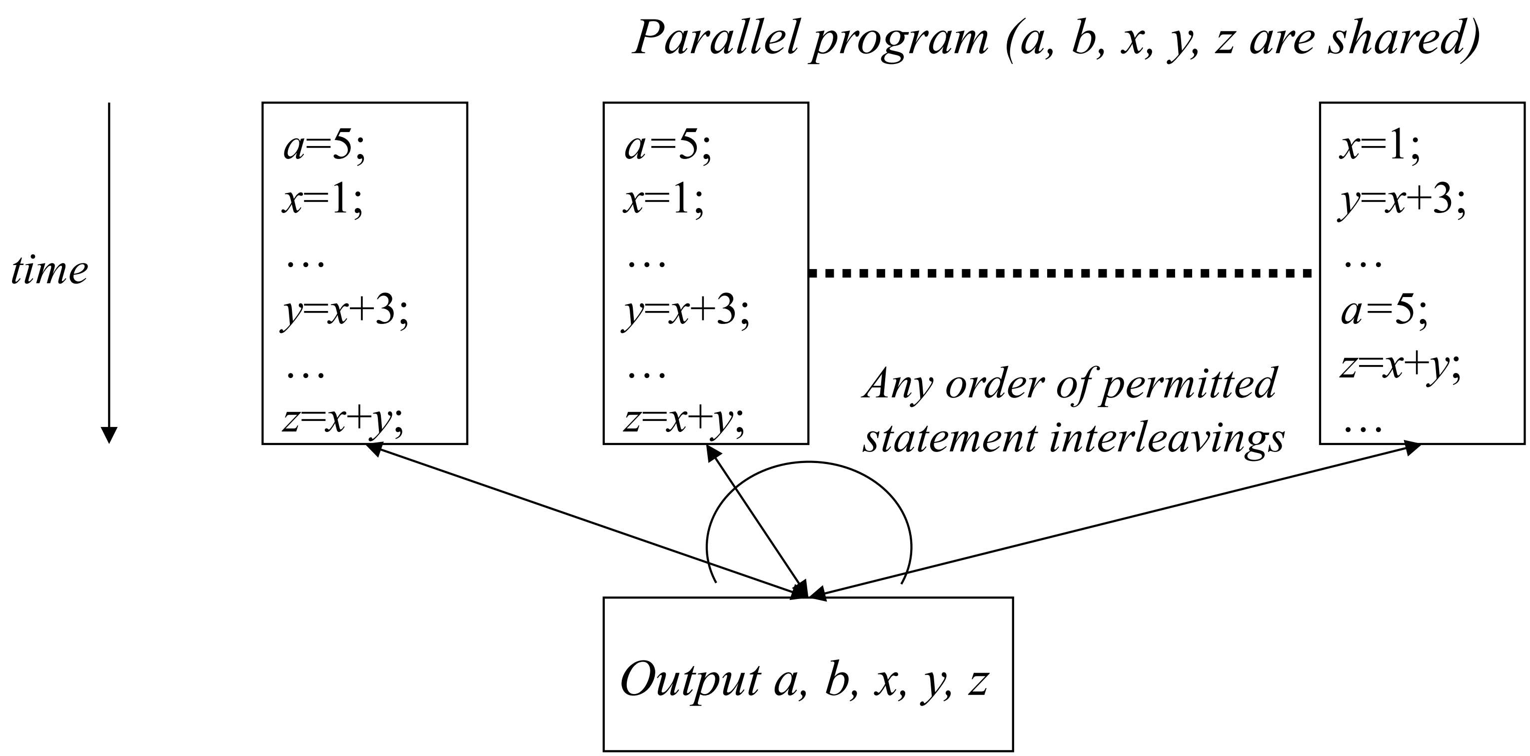
Prof. Robert van Engelen

Overview

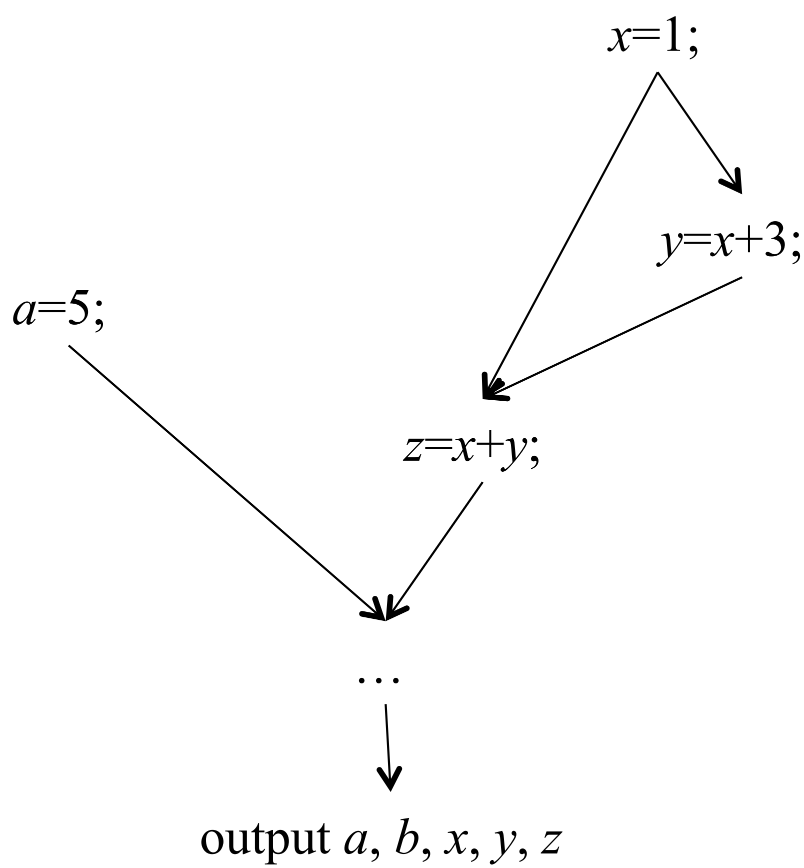
- Sequential consistency
- Parallel programming constructs
- Dependence analysis
- OpenMP
- Autoparallelization
- Further reading

Sequential Consistency

- *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



Data Flow: Implicitly Parallel



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

Explicit Parallel Programming Constructs

- Declaring shared data, when private is implicit

```
shared int x;
```
A shared variable

```
shared int *p;
```
Private pointer to a shared value

- Declaring private data, when private is explicit

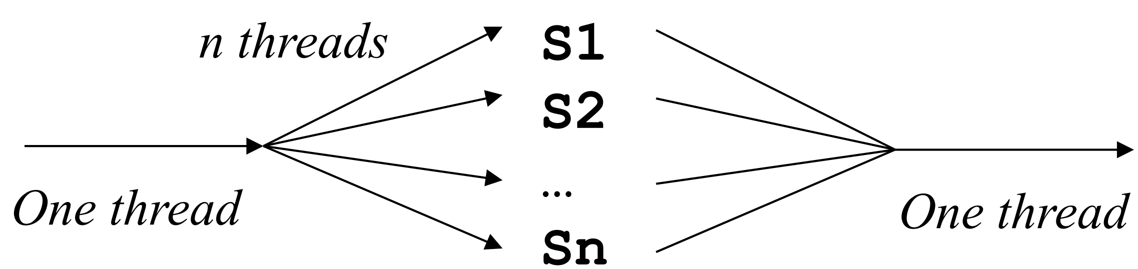
```
private int x;

private int *p;
```
Would this make any sense?

Explicit Parallel Programming Constructs

- The `par` construct

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

Statements in the body are executed concurrently

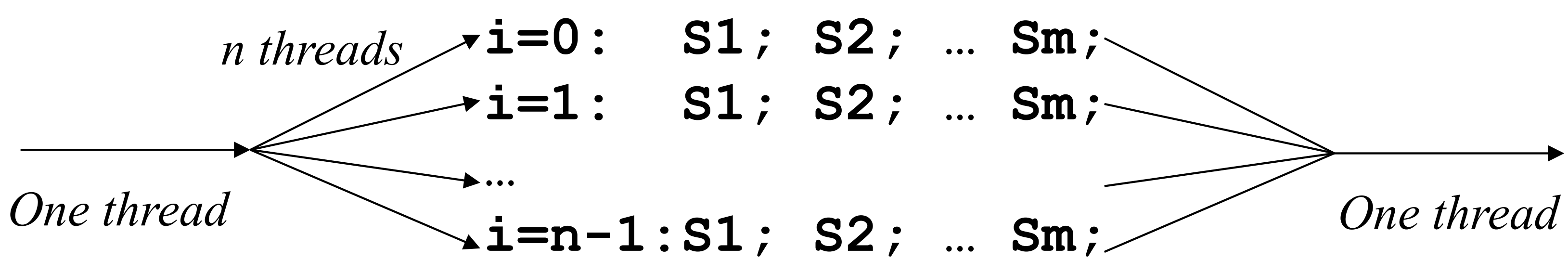


Explicit Parallel Programming Constructs

- The **`forall`** construct (also called **`parfor`**)

```
forall (i=0; i<n; i++) {
  S1;
  S2;
  …
  Sm;
}
```

Statements in the body are executed in serial order by n threads i=0..n-1 in parallel



Explicit Parallel: Many Choices, Which is Safe?

```
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;
}
…
```

Think about data flow: each operation requires completion of operands first

Data dependences preserved ➡ *sequential consistency guaranteed*

Bernstein's Conditions

- 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

$$
\begin{aligned}
I_1 &\cap O_2 = \varnothing \\
I_2 &\cap O_1 = \varnothing \\
O_1 &\cap O_2 = \varnothing
\end{aligned}
$$

Bernstein's Conditions Verified by Dependence Analysis

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

independent

P_1: `A = x + y;`
P_2: `B = x + z;`

$$I_1 \cap O_2 = \varnothing$$
$$I_2 \cap O_1 = \varnothing$$
$$O_1 \cap O_2 = \varnothing$$

RAW

P_1: `A = x + y;`
P_2: `B = x + A;`

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

WAR

P_1: `A = x + B;`
P_2: `B = x + z;`

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

WAW

P_1: `A = x + y;`
P_2: `A = x + z;`

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

Bernstein's Conditions Verified by Dependence Analysis

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

independent

P_1: `A = x + y;`
P_2: `B = x + z;`

$$I_1 \cap O_2 = \varnothing$$
$$I_2 \cap O_1 = \varnothing$$
$$O_1 \cap O_2 = \varnothing$$

```
   par {
P1: A = x + y;
P2: B = x + z;
   }
```

Recall:
instruction scheduling for instruction-level parallelism (ILP)

Bernstein's Conditions in Loops

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

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

OpenMP

- OpenMP is a portable implementation of common parallel constructs for shared memory machines

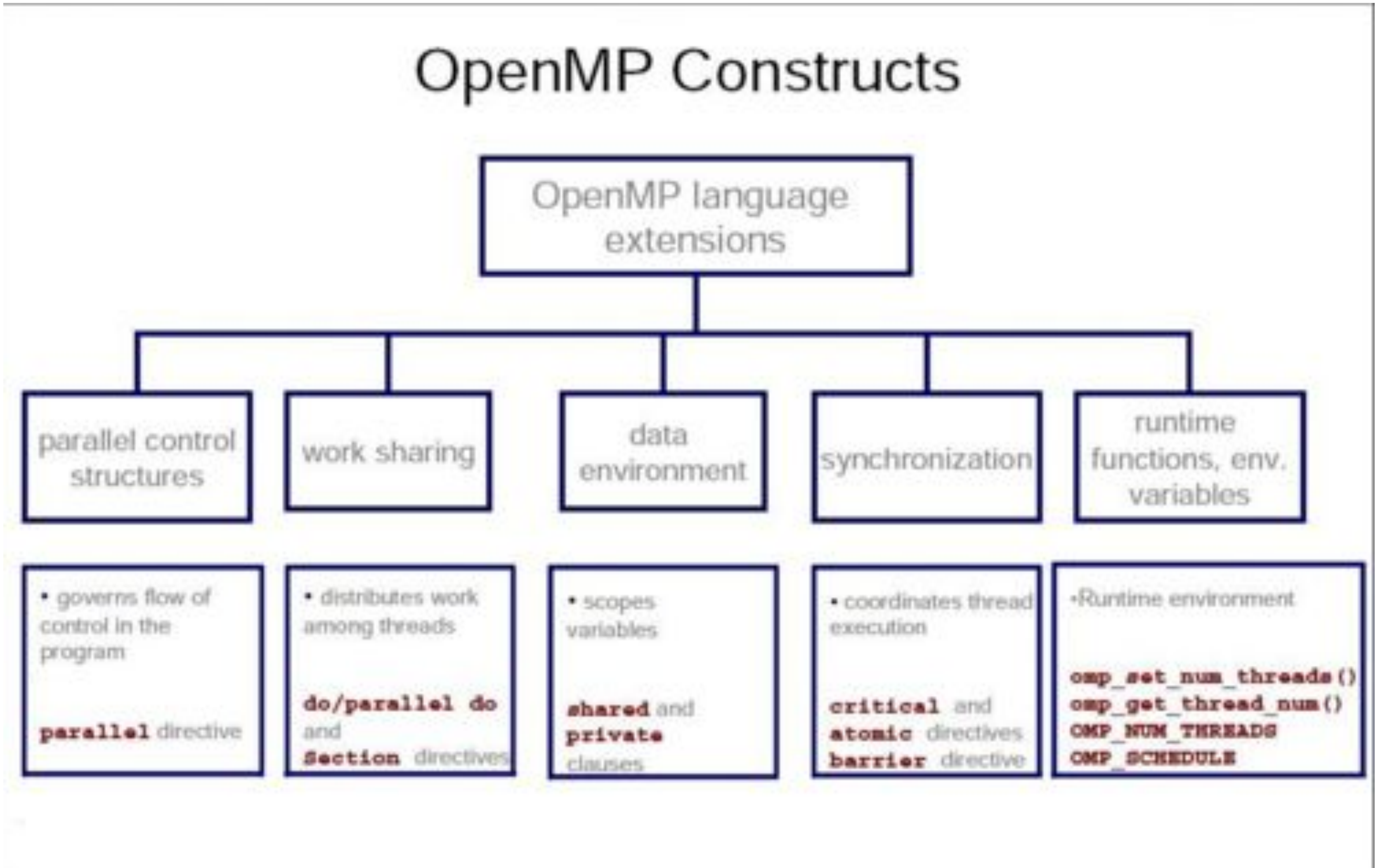
- OpenMP in C

```
#pragma omp directive_name
  statement_block
```

- OpenMP in Fortran

```
!$OMP directive_name
      statement_block
!$OMP end directive name
```

OpenMP

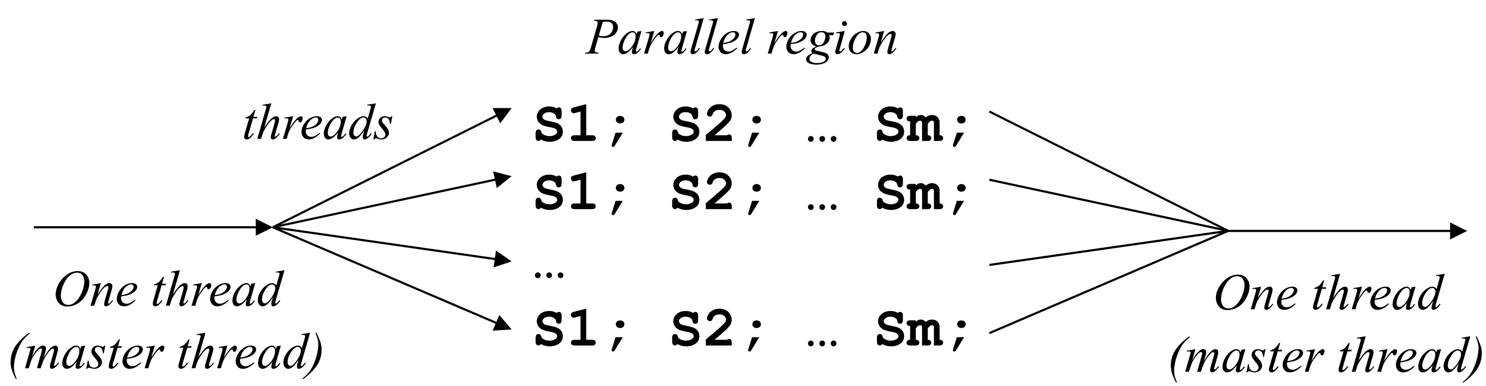

OpenMP Constructs
OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
• governs flow of control in the program
parallel directive
• distributes work among threads
do/parallel do and Section directives
• scopes variables
shared and private clauses
• coordinates thread execution
critical and atomic directives
barrier directive
•Runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

OpenMP Parallel

- The **`parallel`** construct in OpenMP is not the same as **`par`**

```
#pragma omp parallel
{
  S1;
  S2;
  …
  Sm;
}
```

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

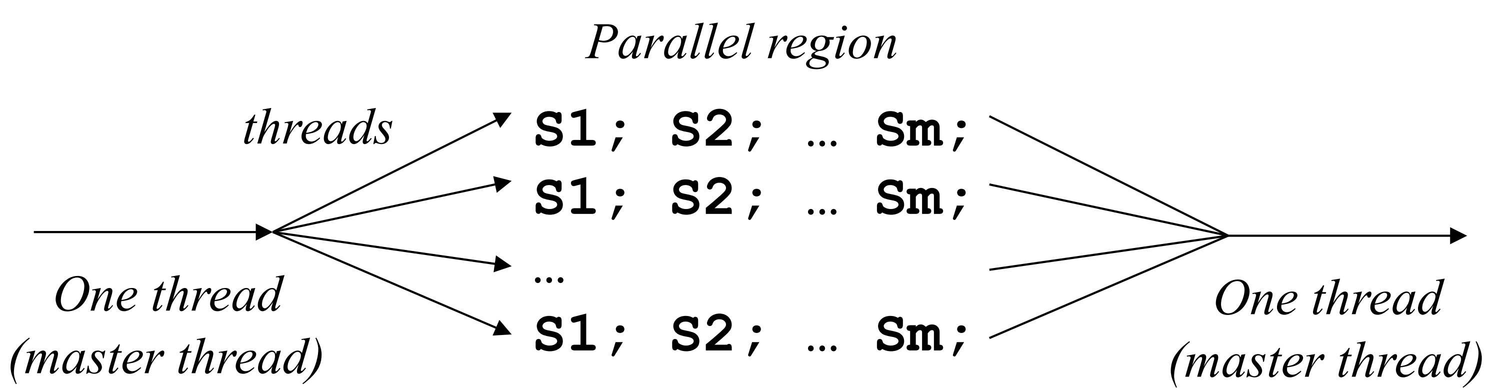


OpenMP Parallel

- The **`parallel`** construct

```
#pragma omp parallel default(none) shared(vars)
{
  S1;
  S2;
  …
  Sm;
}
```

This specifies that variables should not be assumed to be shared by default



OpenMP Parallel

- The **`parallel`** construct

```
#pragma omp parallel private(n, i)
{
  n = omp_get_num_threads();
  i = omp_get_thread_num();
  …
}
```

Use **`private`** *to declare private data*

`omp_get_num_threads()`
returns the number of threads that are currently being used

`omp_get_thread_num()`
returns the thread id (0 to n-1)

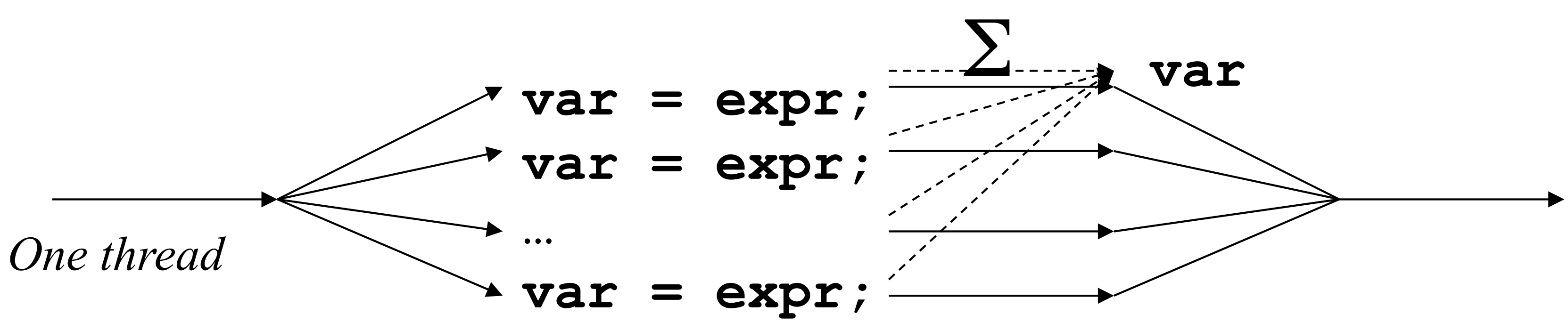
OpenMP Parallel with Reduction

- The **`parallel`** construct with reduction clause

operation: +, *, -, &, ^, |, &&, ||

```
#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



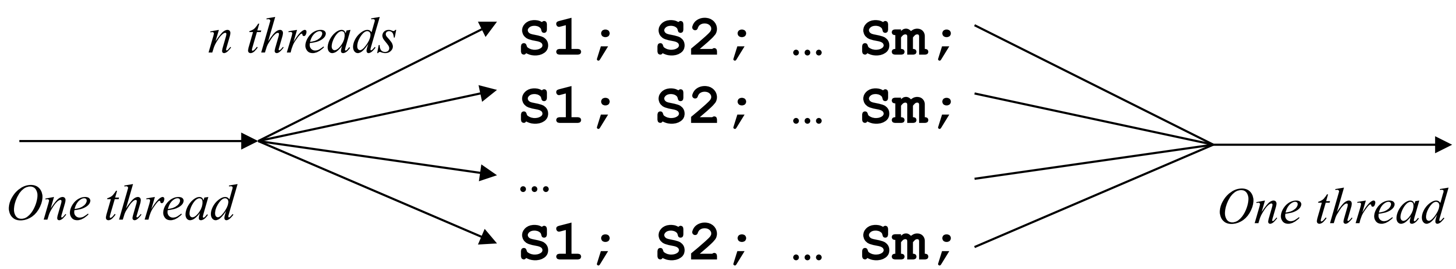
OpenMP Parallel

- The **`parallel`** construct

```
#pragma omp parallel num_threads(n)
{
  S1;
  S2;
  …
  Sm;
}
```

Number of threads we want

Alternatively, use `omp_set_num_threads()`
or set environment variable `OMP_NUM_THREADS`

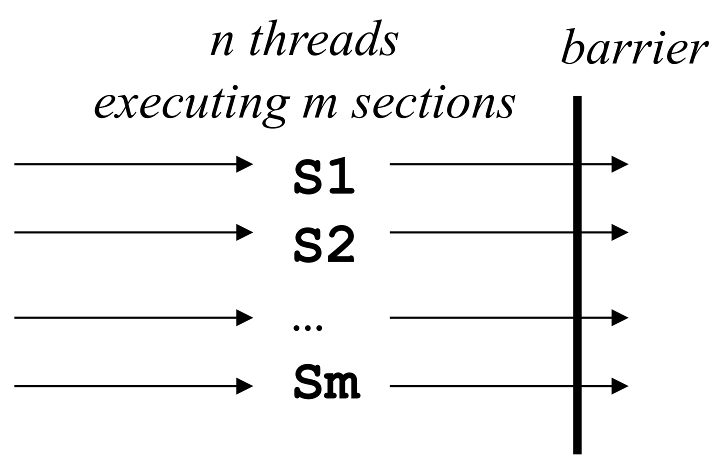


OpenMP Parallel Sections

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

```
#pragma omp parallel
…
#pragma omp sections
{
  #pragma omp section
    S1;
  #pragma omp section
    S2;
  …
  #pragma omp section
    Sm;
}
```

Statements in the sections are executed concurrently

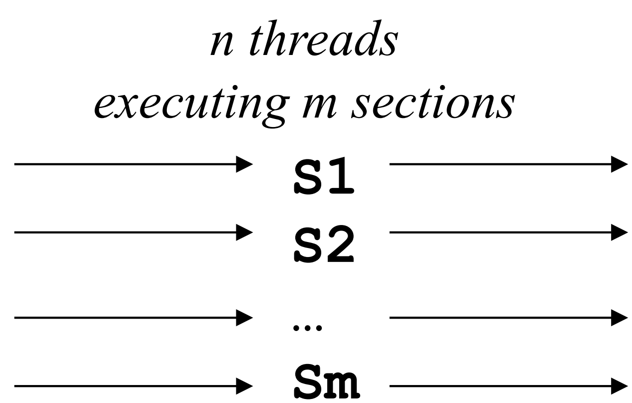


OpenMP Parallel Sections

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

```
#pragma omp parallel
…
#pragma omp sections nowait
{
  #pragma omp section
    S1;
  #pragma omp section
    S2;
  …
  #pragma omp section
    Sm;
}
```

Use **`nowait`** *to remove the implicit barrier*

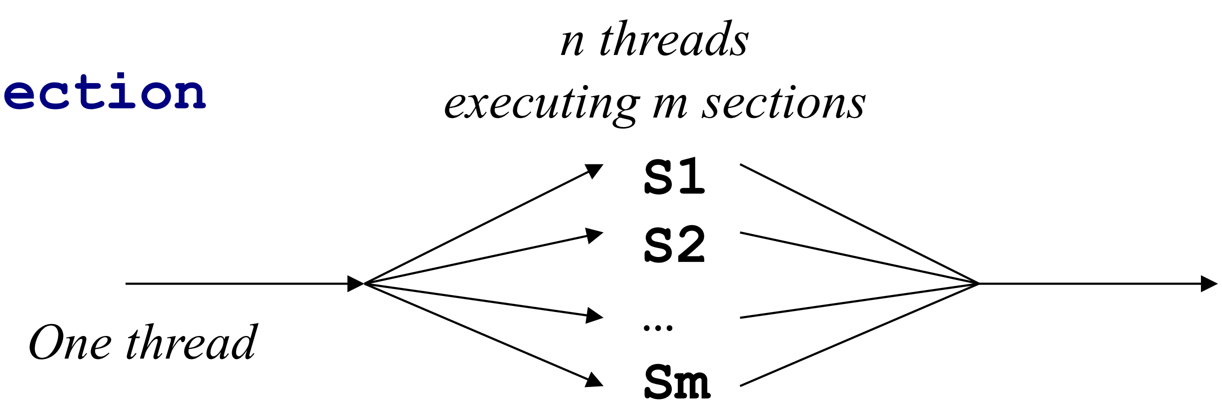


OpenMP Parallel Sections

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

```
#pragma omp parallel sections
{
  #pragma omp section
    S1;
  #pragma omp section
    S2;
  …
  #pragma omp section
    Sm;
}
```

Use **`parallel sections`** *to combine* **`parallel`** *with* **`sections`**



OpenMP For/Do

- The **`for`** construct (**`do`** in Fortran) is for *work-sharing*, where a current team of threads is used to execute a loop concurrently

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

Loop iterations are executed concurrently by n threads

Use **`nowait`** *to remove the implicit barrier*

n threads
executing k iterations

barrier

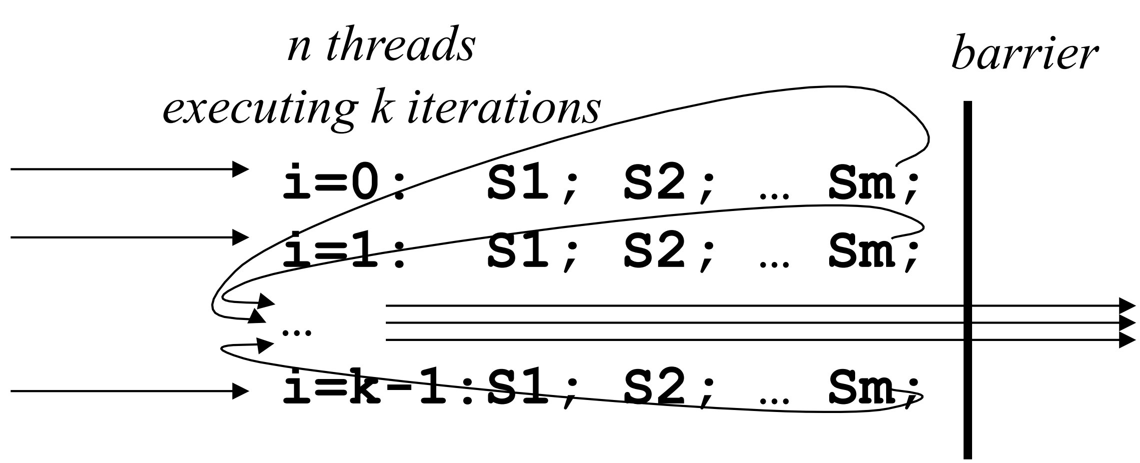
```
i=0:   S1; S2; … Sm;
i=1:   S1; S2; … Sm;
…
i=k-1:S1; S2; … Sm;
```

OpenMP For/Do

- The `for` construct is for *work-sharing*, where a current team of threads is used to execute a loop concurrently

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

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

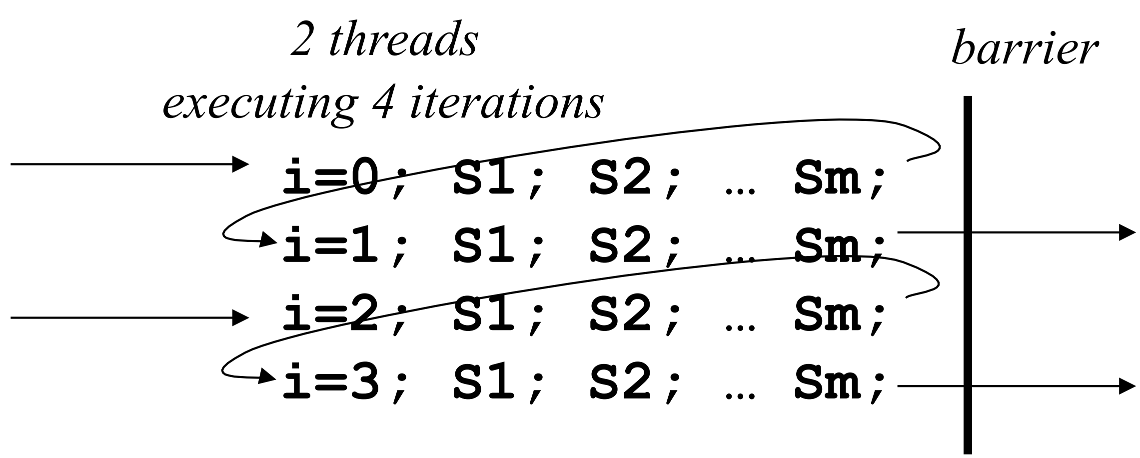


OpenMP For/Do

- The **`for`** construct is for *work-sharing*, where a current team of threads is used to execute a loop concurrently

```
#pragma omp parallel
…
#pragma omp for schedule(static)
for (i=0; i<4; i++)
{
   S1;
   S2;
   …
   Sm;
}
```

When k>n, threads are assigned to $\lceil k/n \rceil$ chunks of the iteration space



OpenMP For/Do

- The **`for`** construct is for *work-sharing*, where a current team of threads is used to execute a loop concurrently

Chunk size

```
#pragma omp parallel
…
#pragma omp for schedule(static, 2)
for (i=0; i<8; i++)
{
   S1;
   S2;
   …
   Sm;
}
```

```
i=0; S1; S2; … Sm;
i=1; S1; S2; … Sm;
i=2; S1; S2; … Sm;
i=3; S1; S2; … Sm;
i=4; S1; S2; … Sm;
i=5; S1; S2; … Sm;
i=6; S1; S2; … Sm;
i=7; S1; S2; … Sm;
```

barrier

2 threads executing 8 iterations using chunk size 2 in a round-robin fashion

OpenMP For/Do

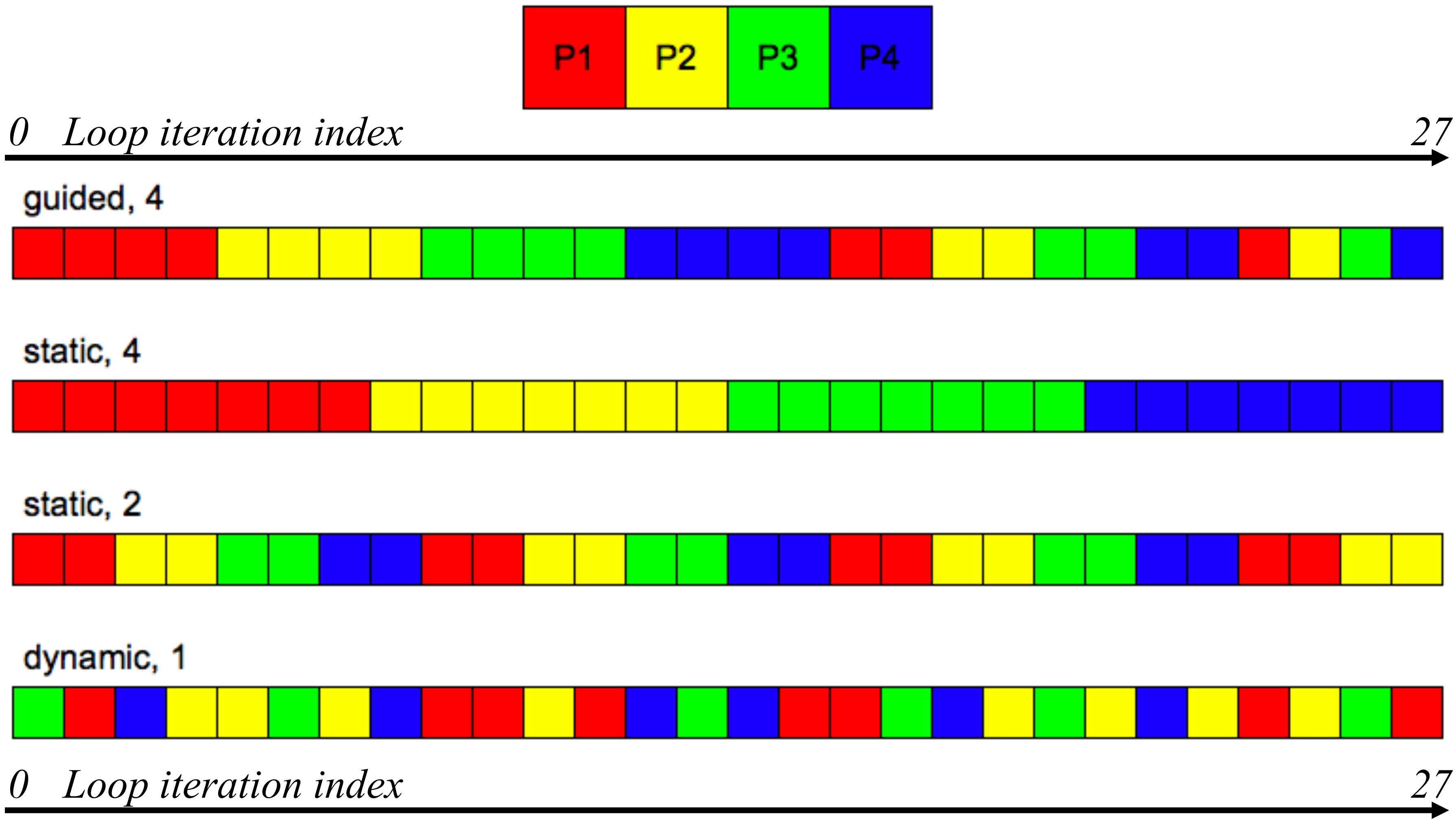
- The **`for`** construct is for *work-sharing*, where a current team of threads is used to execute a loop concurrently

Chunk size

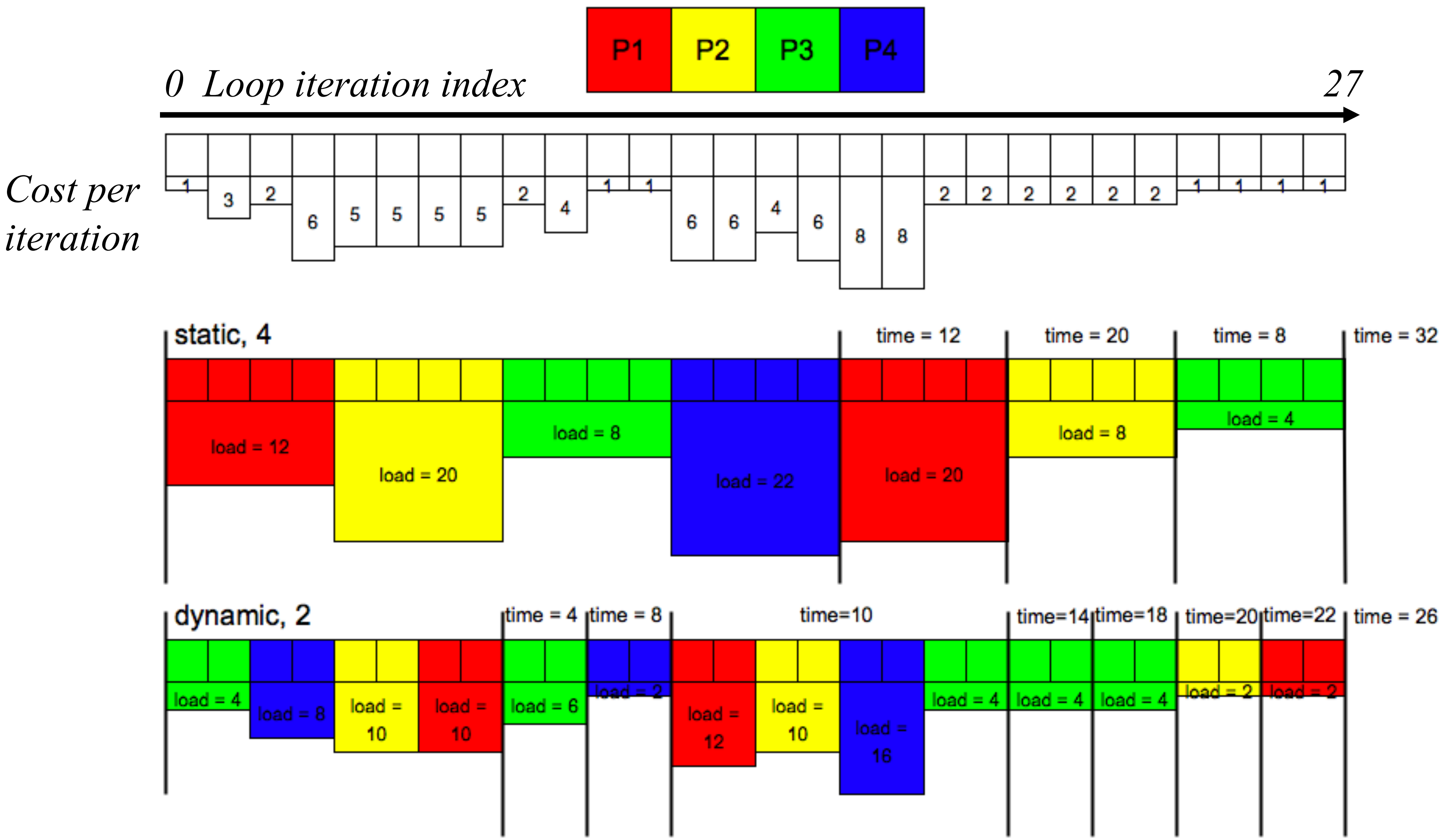
```
#pragma omp parallel
…
#pragma omp for schedule(guided, 4)
for (i=0; i<k; i++)
{
   S1;
   S2;
   …
   Sm;
}
```

Exponentially decreasing chunk size, for example: 4, 2, 1

OpenMP For/Do Scheduling Comparison



OpenMP For/Do Scheduling with Load Imbalances



OpenMP For/Do

- The `for` construct is for *work-sharing*, where a current team of threads is used to execute a loop concurrently

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

Controlled by environment variable **`OMP_SCHEDULE`**:

```
setenv OMP_SCHEDULE "dynamic"
setenv OMP_SCHEDULE "static"
setenv OMP_SCHEDULE "static,2"
```

OpenMP For/Do

- The **`for`** construct is for *work-sharing*, where a current team of threads is used to execute a loop concurrently

operation: +, *, -, &, ^, |, &&, ||

```
#pragma omp parallel
…
#pragma omp for reduction(+:s)
for (i=0; i<k; i++)
{
   s += a[i];
}
```

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

```
i=0:   s += a[0];
i=1:   s += a[1];
…
i=k-1: s += a[k-1];
```

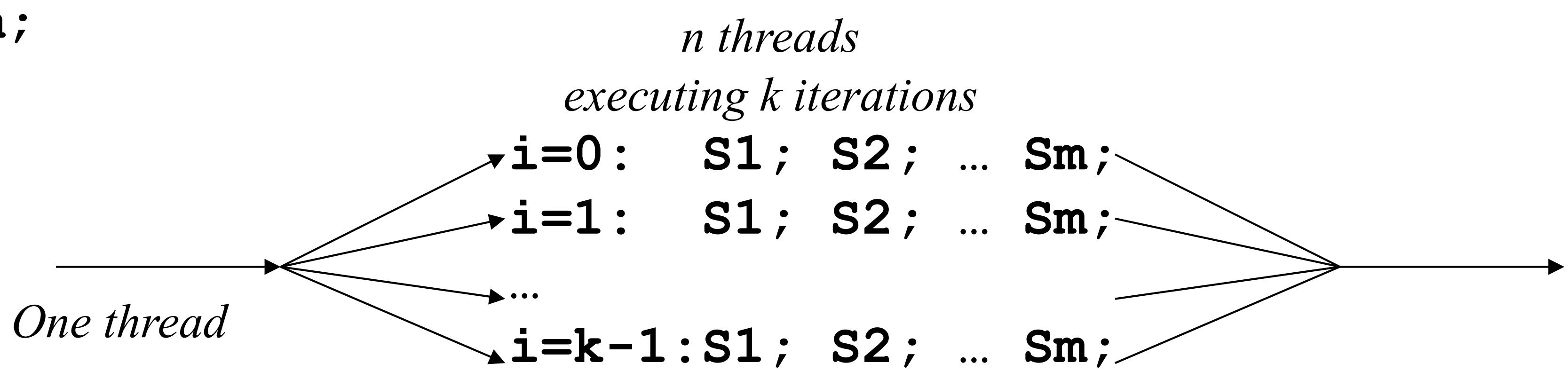
Σ s

OpenMP For/Do

- The **`for`** construct is for *work-sharing*, where a current team of threads is used to execute a loop concurrently

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

Use **`parallel for`** *to combine* **`parallel`** *with* **`for`**



OpenMP Firstprivate and Lastprivate

- The **`parallel`** construct with **`firstprivate`** and/or **`lastprivate`** clause

```
x = …;
#pragma omp parallel firstprivate(x) lastprivate(y)
{ x = x + …;
  #pragma omp for
  for (i=0; i<k; i++)
  { …
    y = i;
  }
}
… = y;
```

Use **`firstprivate`** *to declare private variables that are initialized with the main thread's value of the variables*

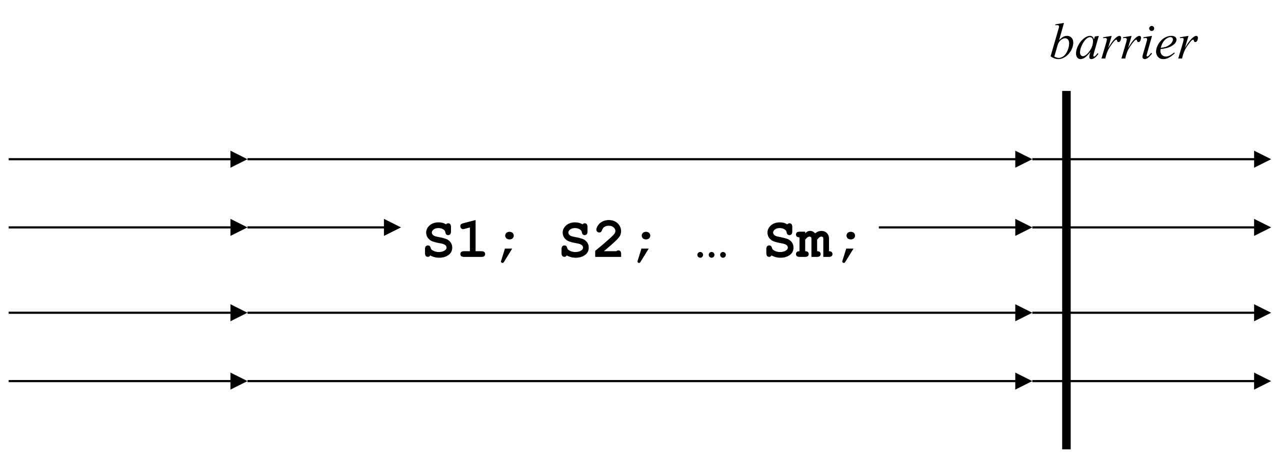
Likewise, use **`lastprivate`** *to declare private variables whose values are copied back out to main thread's variables by the thread that executes the last iteration of a parallel for loop, or the thread that executes the last parallel section*

OpenMP Single

- 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;
}
```

One thread executes the body

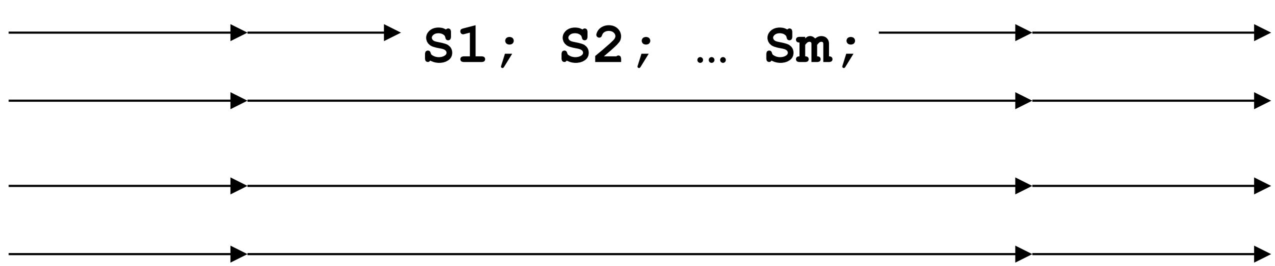


OpenMP Master

- The **`master`** construct selects the master thread of the current team of threads to execute the body

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

The "master" thread executes the body, no barrier is inserted

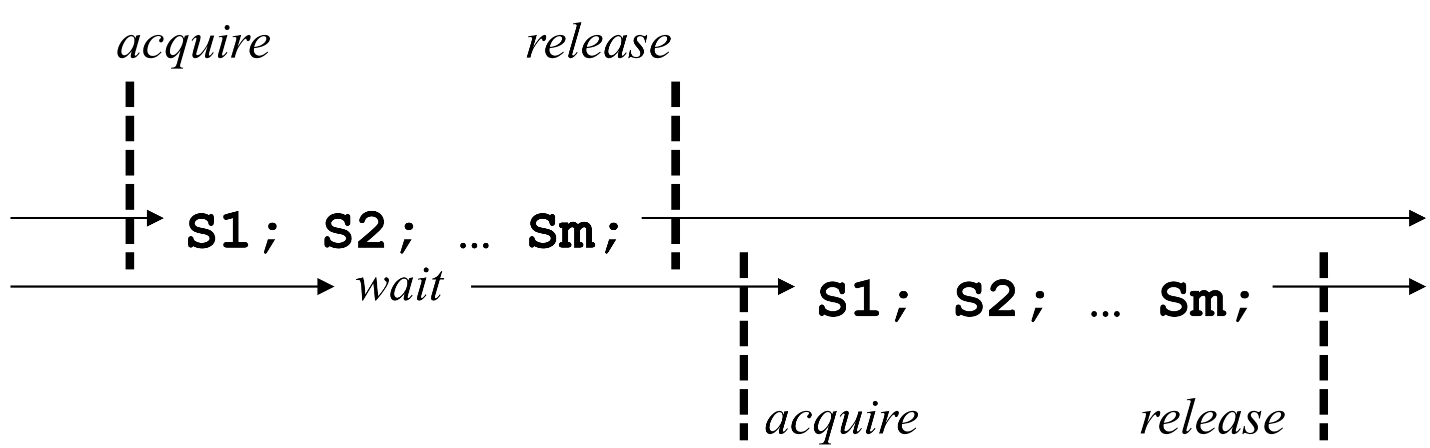


OpenMP Critical

- The **`critical`** construct defines a critical section

```
#pragma omp parallel
…
#pragma omp critical name
{
   S1;
   S2;
   …
   Sm;
}
```

Mutual exclusion is enforced on the body using a named lock



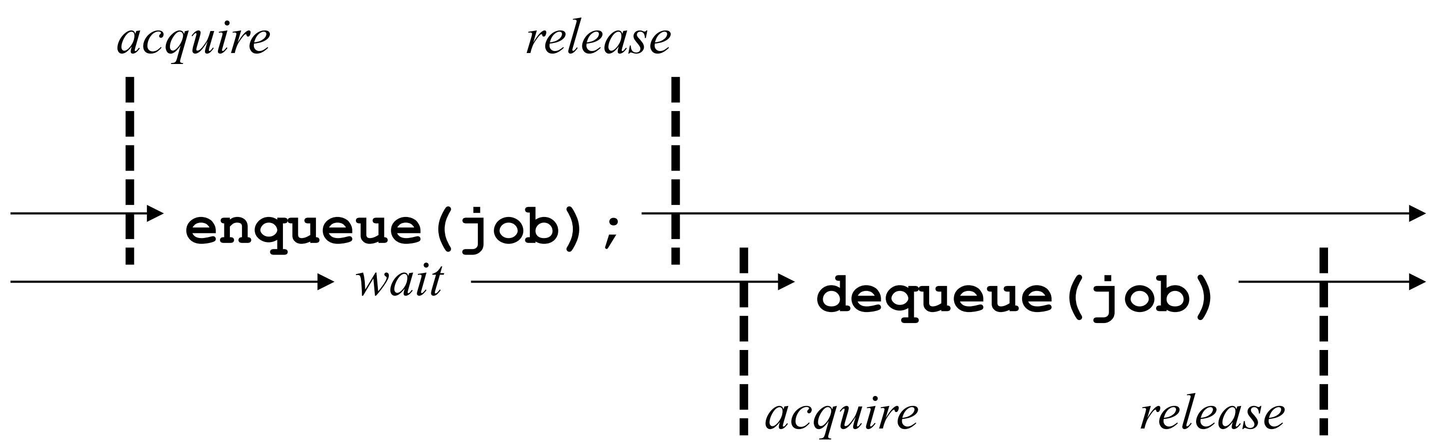
OpenMP Critical

- The **`critical`** construct defines a critical section

```
#pragma omp parallel
…
#pragma omp critical qlock
{ enqueue(job);
}
…
#pragma omp critical qlock
{ dequeue(job);
}
```

One thread is here

Another thread is here

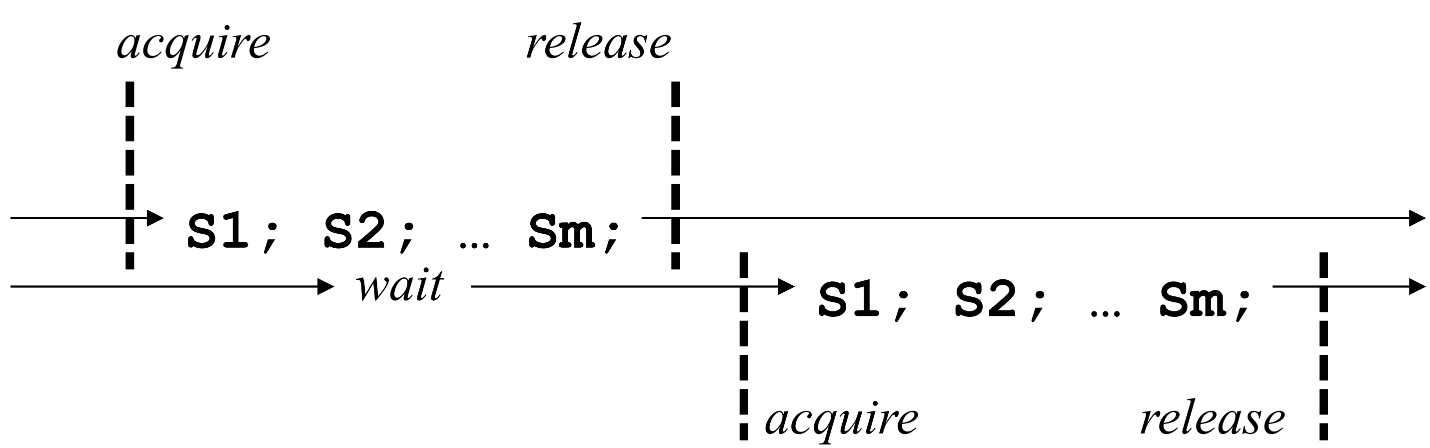


OpenMP Critical

- The **`critical`** construct defines a critical section

```
#pragma omp parallel
…
#pragma omp critical
{
   S1;
   S2;
   …
   Sm;
}
```

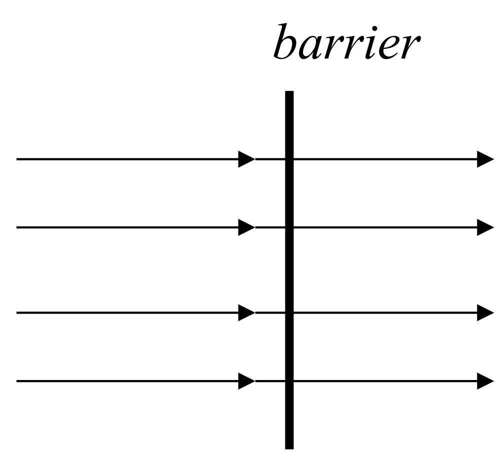
Mutual exclusion is enforced on the body using an anonymous lock



OpenMP Barrier

- The **`barrier`** construct synchronizes the current team of threads

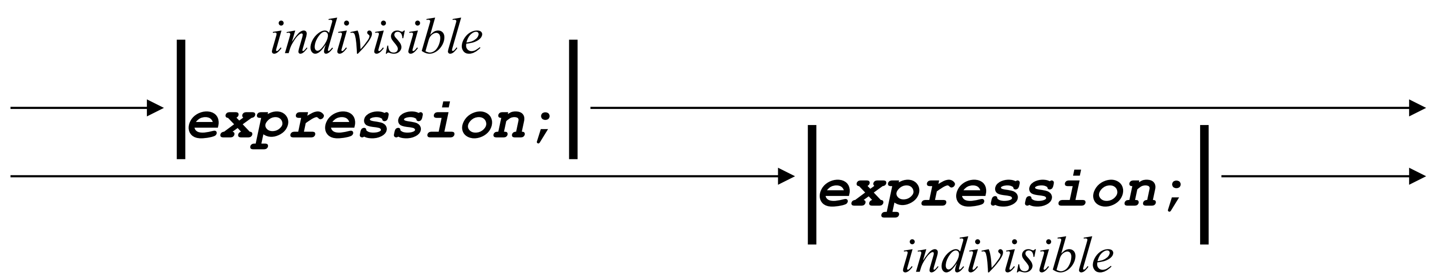
```
#pragma omp parallel
…
#pragma omp barrier
```



OpenMP Atomic

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

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



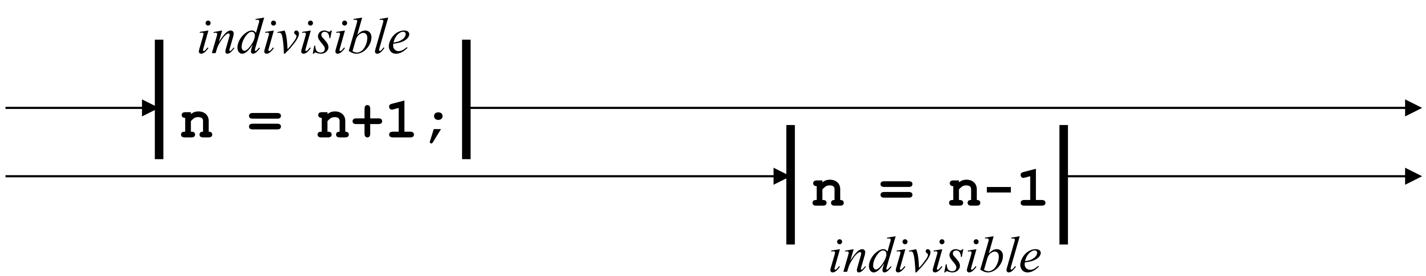
OpenMP Atomic

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

```
#pragma omp parallel
…
#pragma omp atomic
n = n+1;
…
#pragma omp atomic
n = n-1;
```

One thread is here

Another thread is here

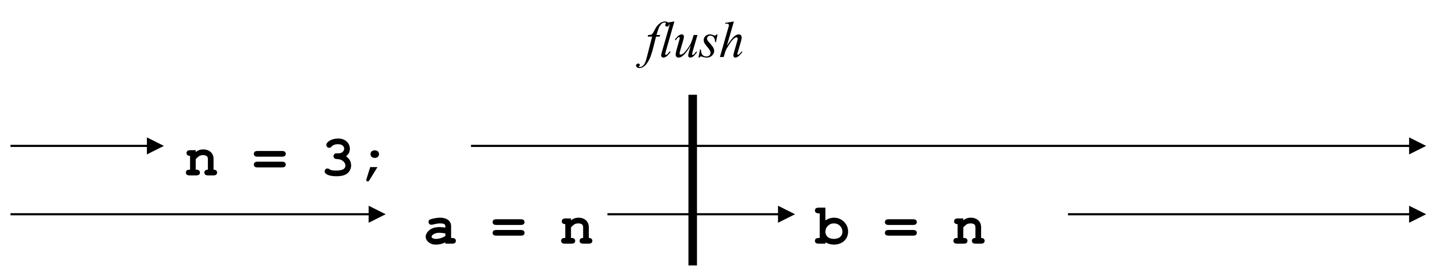


OpenMP Flush

- The **`flush`** construct flushes shared variables from local storage (registers, cache) to shared memory
- OpenMP adopts a *relaxed consistency model* of memory

```
#pragma omp parallel
…
#pragma omp flush(variables)
```

`b = 3`, *but there is no guarantee that* **`a`** *will be* **`3`**

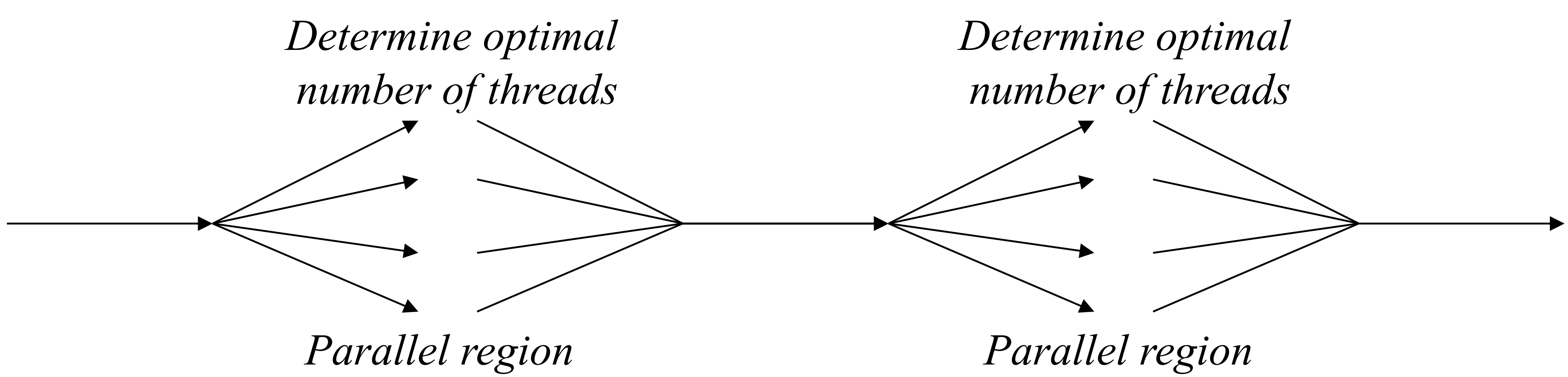


OpenMP Relaxed Consistency Memory Model

- *Relaxed consistency* means that memory updates made by one CPU may not be immediately visible to another CPU
  - Data can be in registers
  - Data can be in cache
    (cache coherence protocol is slow or non-existent)
- Therefore, the updated value of a shared variable that was set by a thread may not be available to another
- An OpenMP *flush* is automatically performed at
  - Entry and exit of `parallel` and `critical`
  - Exit of `for`
  - Exit of `sections`
  - Exit of `single`
  - Barriers

OpenMP Thread Scheduling

- Controlled by environment variable **`OMP_DYNAMIC`**
- When set to **`FALSE`**
  - Same number of threads used for every parallel region
- When set to **`TRUE`**
  - The number of threads is adjusted for each parallel region
  - **`omp_get_num_threads()`** returns actual number of threads
  - **`omp_get_max_threads()`** returns **`OMP_NUM_THREADS`**



OpenMP Threadprivate

- The **`threadprivate`** construct declares variables in a global scope private to a thread across multiple parallel regions
  - Must use when variables should stay private, even outside of the current scope, e.g. across function calls

```
int counter;                        Global counter
#pragma omp threadprivate(counter)

#pragma omp parallel
{ counter = 0;                      Each thread has a local copy of counter
  …
}
…
#pragma omp parallel
{ counter++;                        Each thread has a local copy of counter
  …
}
```

OpenMP Locks

- Mutex locks, with additional “nestable” versions of locks

```
omp_lock_t lck;

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

omp_lock_t
the lock type

omp_init_lock()
initialization

omp_set_lock()
blocks until lock is acquired

omp_unset_lock()
releases the lock

omp_destroy_lock()
deallocates the lock

Compiler Options for OpenMP

- GOMP project for GCC 4.2 (C and Fortran)

- Use `#include <omp.h>`
  - Note: the `_OPENMP` define is set when compiling with OpenMP

- Intel compiler:
  `icc -openmp` …
  `ifort -openmp`

- Sun compiler:
  `suncc -xopenmp` …
  `f95 -xopenmp`

Autoparallelization

- Compiler applies dependence analysis and parallelizes a loop (or entire loop nest) automatically when possible
  - Typically task-parallelizes *outer loops* (more parallel work), possibly after loop interchange, fusion, etc.
  - Similar to adding **`#pragma parallel for`** to loop(s), with appropriate **`private`** and **`shared`** clauses

- Intel compiler:

  `icc -parallel` …

  `ifort -parallel` …

- Sun compiler:

  `suncc -xautopar` …

  `f95 -xautopar` …

Further Reading

- [PP2] pages 248-271
- Optional:
  OpenMP tutorial at Lawrence Livermore
  `http://www.llnl.gov/computing/tutorials/openMP/`