Introduction to OpenMP for CPUs

International HPC Summer School 2024

Ludovic Capelli

EPCC

July 8, 2024

Individuals

The material presented in this lecture is inspired from content developed by:

- Dr Mark Bull
- Prof John Urbanic

Contributors

In this material:

- The slides are created using LaTeX *Beamer* available at `https://ctan.org/pkg/beamer`.
- The sequence diagrams are created using an extended version of the *pgf-umlsd* package available at `https://ctan.org/pkg/pgf-umlsd`.

License - Creative Commons BY-NC-SA-4.0[1]

Non-Commercial you may not use the material for commercial purposes.

Shared-Alike if you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

Attribution you must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

[1] You can find the full documentation about this license at: `https://creativecommons.org/licenses/by-nc-sa/4.0/`

Nota bene

Format

This set of slides is designed to be covered as a **live presentation**, and thus contains little text and explanations. It is **less suitable** to be studied as a stand-alone document.

Timetable

Designed as 30-min blocks

- 20 minutes of teaching
- 10 minutes of practice

Shared memory = ?

Figure: Link to the survey: "Which words come to mind when thinking of shared memory programming?"

Moment of truth

Before we start:

- Connect to the Bridges node

```
ssh your_username@login.cirrus.ad.uk
```

- This repository is on Github.
- Clone the repository used in this session

```
git clone https://github.com/capellil/
    IHPCSS_Introduction_to_OpenMP_CPU_examples
```

Moment of truth

- You will see it contains multiple folders, numbered.
- Each of which will be an illustration to a concept we will see.

Moment of truth - Running on login node

- Go to the repository, inside the first folder.

```
cd <repository>/<language>/1.Preamble
```

- Compile the source code in it

```
make
```

- Run the executable.

```
./bin/preamble
```

Table of content

Table of Contents

Clock frequency

Clock frequency

1998 First 0.1GHz CPU, Pentium II Xeon 400.

Clock frequency

1998 First 0.1GHz CPU, Pentium II Xeon 400.

1999 First 1GHz CPU, AMD Athlon.

Clock frequency

1998 First 0.1GHz CPU, Pentium II Xeon 400.

1999 First 1GHz CPU, AMD Athlon.

2001 First 2GHz CPU, Intel Pentium 4.

Clock frequency

1998 First 0.1GHz CPU, Pentium II Xeon 400.

1999 First 1GHz CPU, AMD Athlon.

2001 First 2GHz CPU, Intel Pentium 4.

2002 First 3GHz CPU, Intel Pentium 4.

Clock frequency

1998 First 0.1GHz CPU, Pentium II Xeon 400.

1999 First 1GHz CPU, AMD Athlon.

2001 First 2GHz CPU, Intel Pentium 4.

2002 First 3GHz CPU, Intel Pentium 4.

2012 First 4GHz CPU, AMD FX-4170.

Clock frequency

1998 First 0.1GHz CPU, Pentium II Xeon 400.

1999 First 1GHz CPU, AMD Athlon.

2001 First 2GHz CPU, Intel Pentium 4.

2002 First 3GHz CPU, Intel Pentium 4.

2012 First 4GHz CPU, AMD FX-4170.

2013 First 5GHz CPU, AMD FX-9590.

Clock frequency

1998 First 0.1GHz CPU, Pentium II Xeon 400.

1999 First 1GHz CPU, AMD Athlon.

2001 First 2GHz CPU, Intel Pentium 4.

2002 First 3GHz CPU, Intel Pentium 4.

2012 First 4GHz CPU, AMD FX-4170.

2013 First 5GHz CPU, AMD FX-9590.

2023 First 6GHz CPU, Intel Core i9-13900KS.

Dawn of multi-threading

- Can only increase clock frequency so much, due to physics.

Dawn of multi-threading

- Can only increase clock frequency so much, due to physics.
- The higher the frequency:
  - the bigger the loss (i.e.: heat generated)

Dawn of multi-threading

- Can only increase clock frequency so much, due to physics.
- The higher the frequency:
  - the bigger the loss (i.e.: heat generated)
  - the bigger the current leak

Dawn of multi-threading

- Can only increase clock frequency so much, due to physics.
- The higher the frequency:
  - the bigger the loss (i.e.: heat generated)
  - the bigger the current leak
- Instead of trying to increase the clock frequency further, what about using multiple cores?

Early days of multi-threading

- In the early days of parallel programming, everybody was developing their own library.
- Challenging situation for everybody:

Early days of multi-threading

- In the early days of parallel programming, everybody was developing their own library.
- Challenging situation for everybody:
  - **As a vendor**: optimising every library independently is unreasonable.

Early days of multi-threading

- In the early days of parallel programming, everybody was developing their own library.
- Challenging situation for everybody:
  - **As a vendor**: optimising every library independently is unreasonable.
  - **As a user**: few to no vendor optimisations available, and no code portability between libraries.
- Need for **standardisation**.

Standardisation multi-threading

- In the 90s, efforts were made towards the development of a standard, named Open Multi-Processing, or OpenMP.

Standardisation multi-threading

- In the 90s, efforts were made towards the development of a standard, named Open Multi-Processing, or OpenMP.
- Members of this team effort became the OpenMP **A**rchitecture **R**eview **B**oard (ARB).

Wide community (as of 2023) - Companies

- AMD
- ARM
- Fujitsu
- HPE
- IBM
- Intel
- Micron
- NVIDIA
- Samsung
- Siemens
- SiFive
- SUSE

Wide community (as of 2023) - Research centres

- Argonne National Laboratory
- ASC / Lawrence Livermore National Laboratory
- Barcelona Supercomputing Center
- Brookhaven National Laboratory
- CSC - IT Center for Science
- EPCC
- Lawrence Berkely National Laboratory
- Leibniz Supercomputing Centre
- Los Alamos National Laboratory
- NEC
- NASA

Wide community (as of 2023) - Research centres

- Oak Ridge National Laboratory
- Pawsey Supercomputing Research Centre
- RWTH Aachen University
- Sandia National Laboratory
- Stony Brook University
- Texas Advanced Computing Center
- University of Basel
- University of Bristol
- University of Delaware
- University of Tennessee

Sustained efforts

- The OpenMP ARB published the version 1.0 of the OpenMP standards in 1997.

Sustained efforts

- The OpenMP ARB published the version 1.0 of the OpenMP standards in 1997.
- 650-page PDF.

Sustained efforts

- The OpenMP ARB published the version 1.0 of the OpenMP standards in 1997.
- 650-page PDF.
- Current version 5.0 as of November 2018.

Sustained efforts

- The OpenMP ARB published the version 1.0 of the OpenMP standards in 1997.
- 650-page PDF.
- Current version 5.0 as of November 2018.
- Minor versions 5.1 and 5.2 published, major version 6.0 in the works.

Sustained efforts

- The OpenMP ARB published the version 1.0 of the OpenMP standards in 1997.
- 650-page PDF.
- Current version 5.0 as of November 2018.
- Minor versions 5.1 and 5.2 published, major version 6.0 in the works.
- It is **directive-based**.

Sustained efforts

- The OpenMP ARB published the version 1.0 of the OpenMP standards in 1997.
- 650-page PDF.
- Current version 5.0 as of November 2018.
- Minor versions 5.1 and 5.2 published, major version 6.0 in the works.
- It is **directive-based**.
- It follows a **fork-join** pattern.

Table of Contents

Structure of a construct

In C:

```
1 #pragma omp <directive-name> [clause(...), ...]
2 {
3   <structured-block>
4 }
```

In FORTRAN:

```
1 !$OMP <directive-name> [clause(...), ...]
2   <structured-block>
3 !$OMP END
```

Structure of a construct

```
1 #pragma omp <directive-name> [clause(...), ...]
2 {
3   <structured-block>
4 }
```

Structure of a construct

```
1 #pragma omp <directive-name> [clause(...), ...]
2 {
3   <structured-block>
4 }
```

Sentinel A bit of code that indicates the presence of an OpenMP directive.

- In C: `#pragma omp`
- In FORTRAN: `!$OMP` / `!$OMP END`

Structure of a construct

```
1 #pragma omp <directive-name> [clause(...), ...]
2 {
3   <structured-block>
4 }
```

Sentinel A bit of code that indicates the presence of an OpenMP directive.

- In C: `#pragma omp`
- In FORTRAN: `!$OMP` / `!$OMP END`

Directive-name The name of the OpenMP directive to use.

Structure of a construct

```
1 #pragma omp <directive-name> [clause(...), ...]
2 {
3   <structured-block>
4 }
```

Clauses A list of additional features / options to enable.

Structure of a construct

```
1 #pragma omp <directive-name> [clause(...), ...]
2 {
3   <structured-block>
4 }
```

Clauses A list of additional features / options to enable.

Directive The entire line, comprising the sentinel, the directive-name and all the clauses.

Structure of a construct

```
1 #pragma omp <directive-name> [clause(...), ...]
2 {
3   <structured-block>
4 }
```

Clauses A list of additional features / options to enable.

Directive The entire line, comprising the sentinel, the directive-name and all the clauses.

Construct The directive and the structured block associated with it.

Table of Contents

Hello world example

In C:

```c
// Main thread alone
printf("Hello world.\n");
```

In FORTRAN:

```fortran
! Main thread alone
WRITE(*, '(A)') 'Hello world'
```

Hello world example

```
1 // Main thread alone
2 #pragma omp parallel
3 {
4   // All threads
5   printf("Hello world.\n");
6 }
7 // Main thread alone
```

- Informed the compiler about OpenMP directives incoming using the **sentinel**.
- Passed the `parallel` construct.

Hello world example

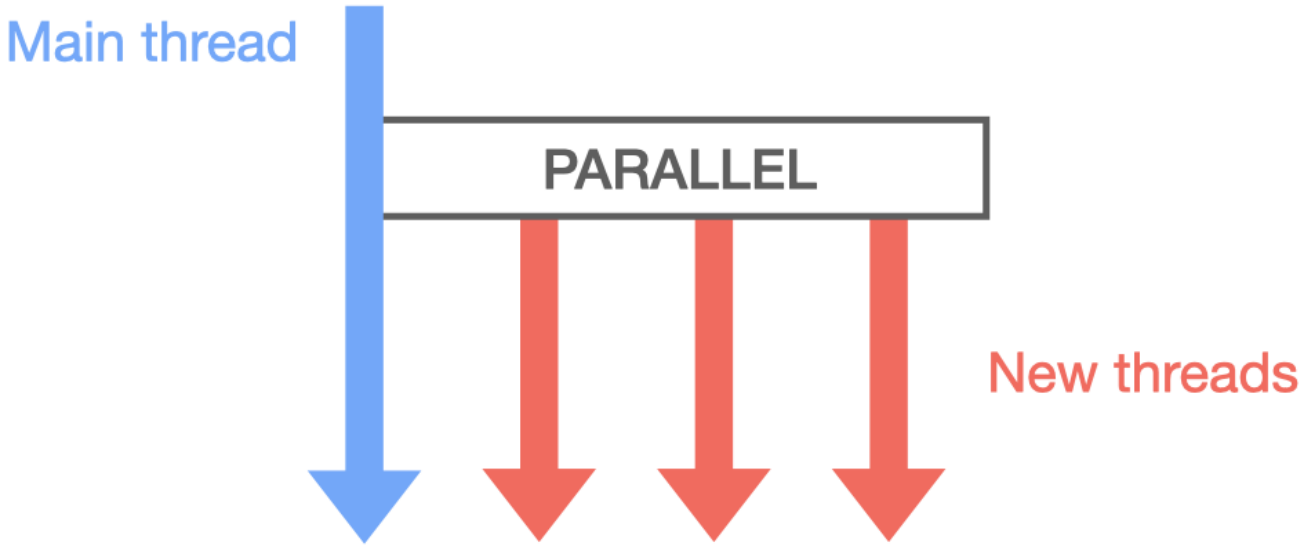
```
1 ! Main thread alone
2 !$OMP PARALLEL
3   ! All threads
4   WRITE(*, '(A)') 'Hello world'
5 !$OMP END PARALLEL
6 ! Main thread alone
```

- Informed the compiler about OpenMP directives incoming using the **sentinel**.
- Passed the `parallel` construct.

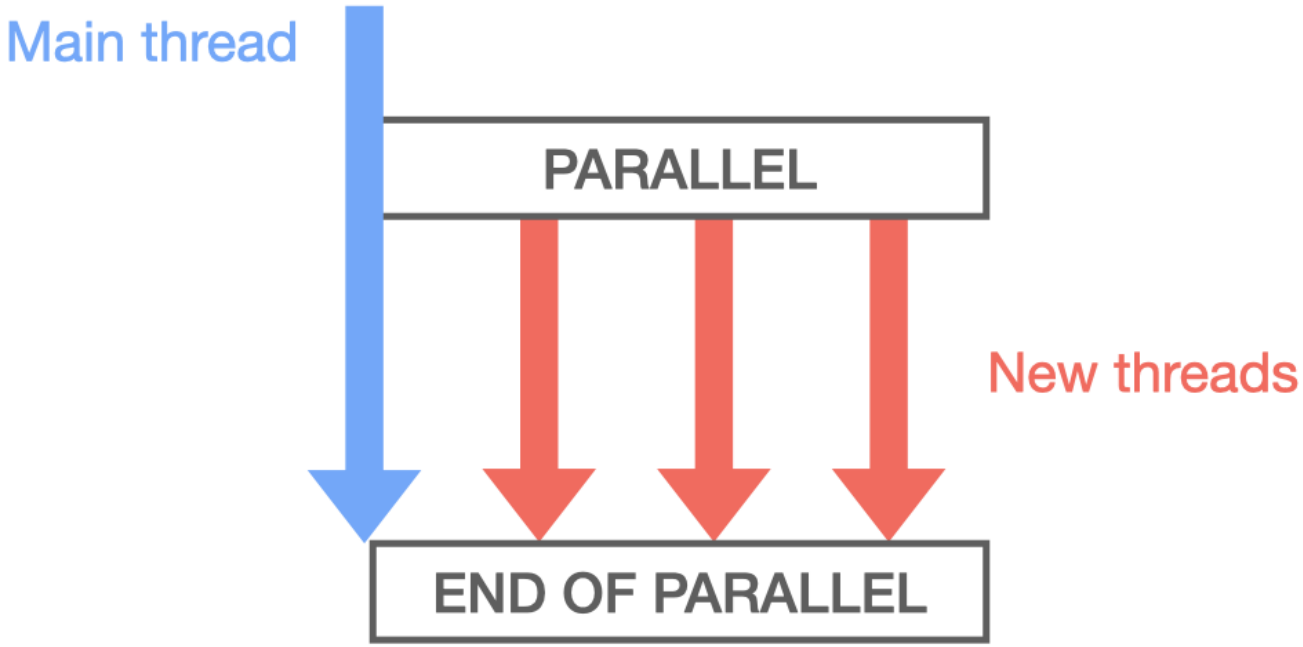
Before the `parallel` construct

Entering `parallel` construct

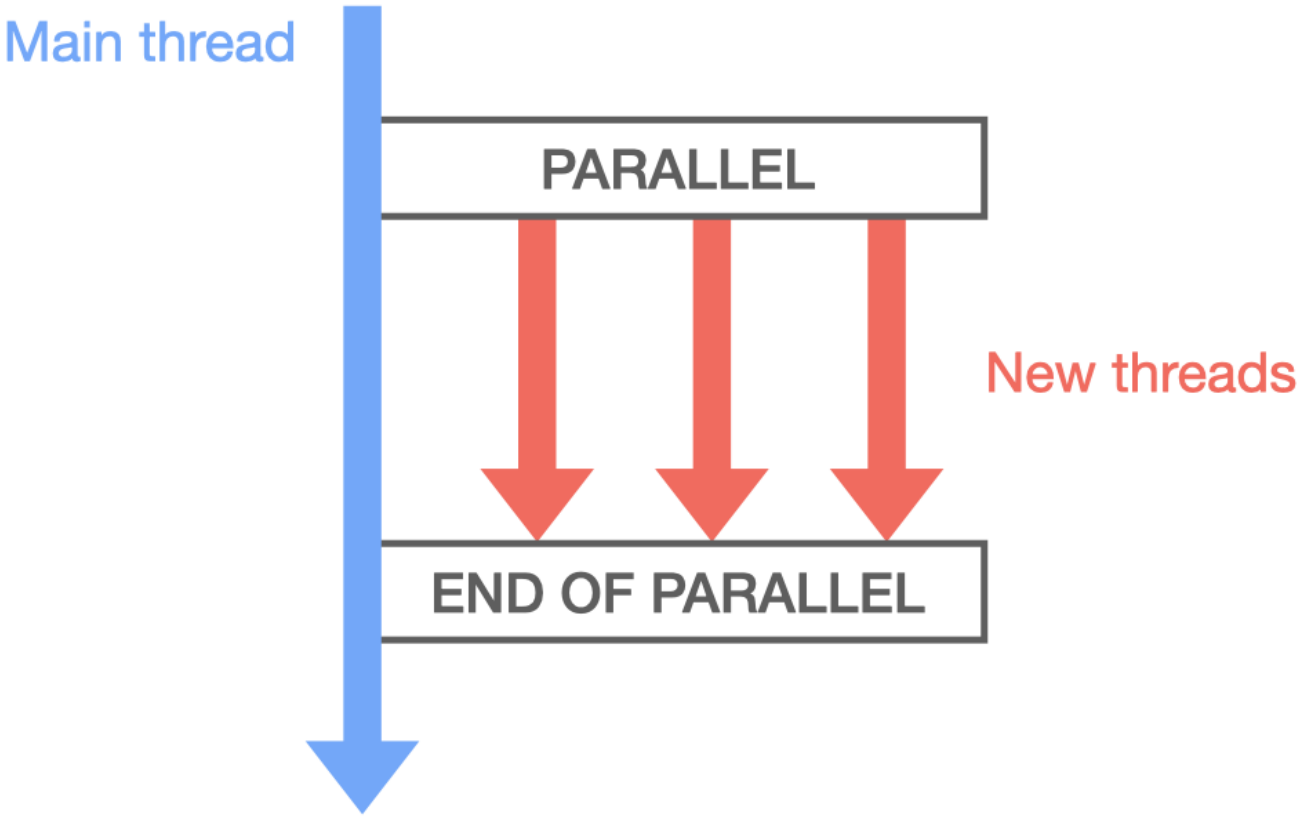
Inside the `parallel` construct



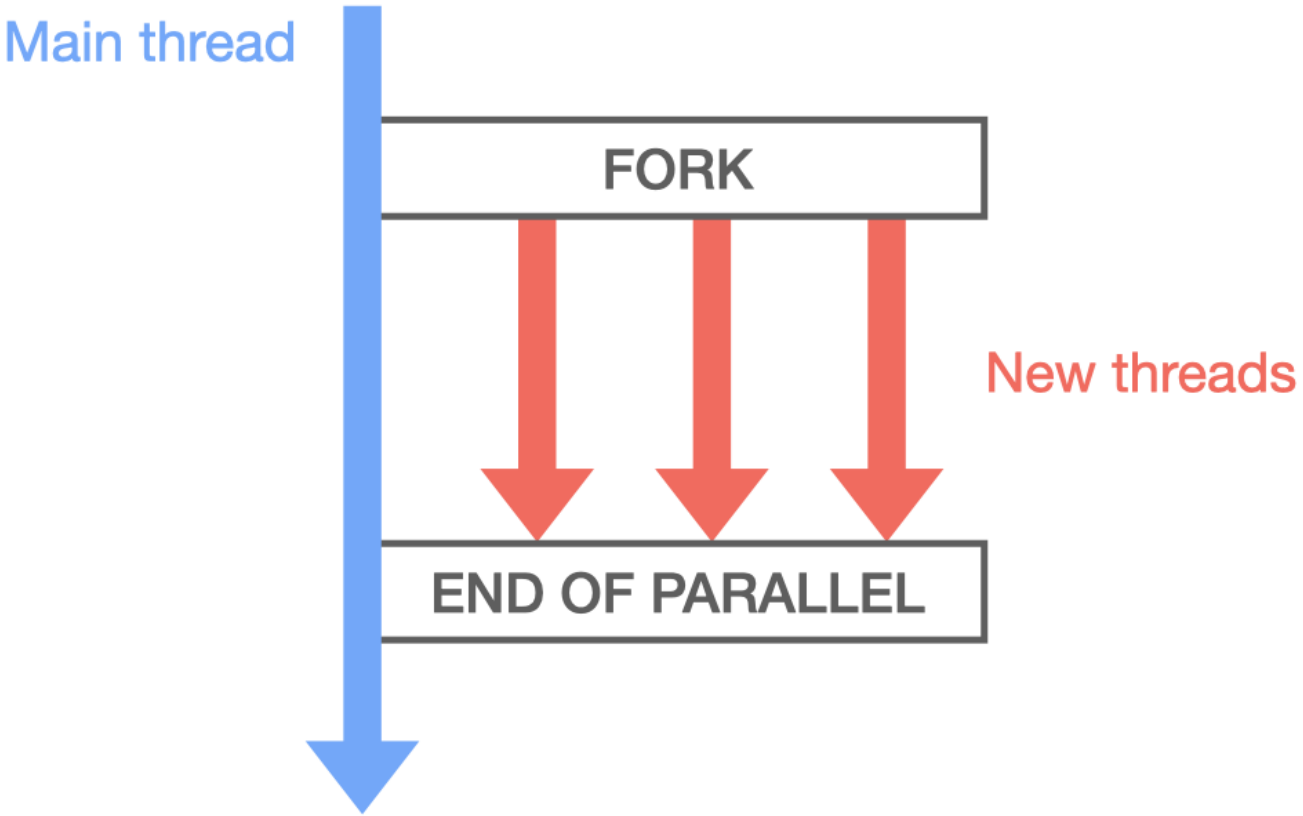
Exiting the `parallel` construct



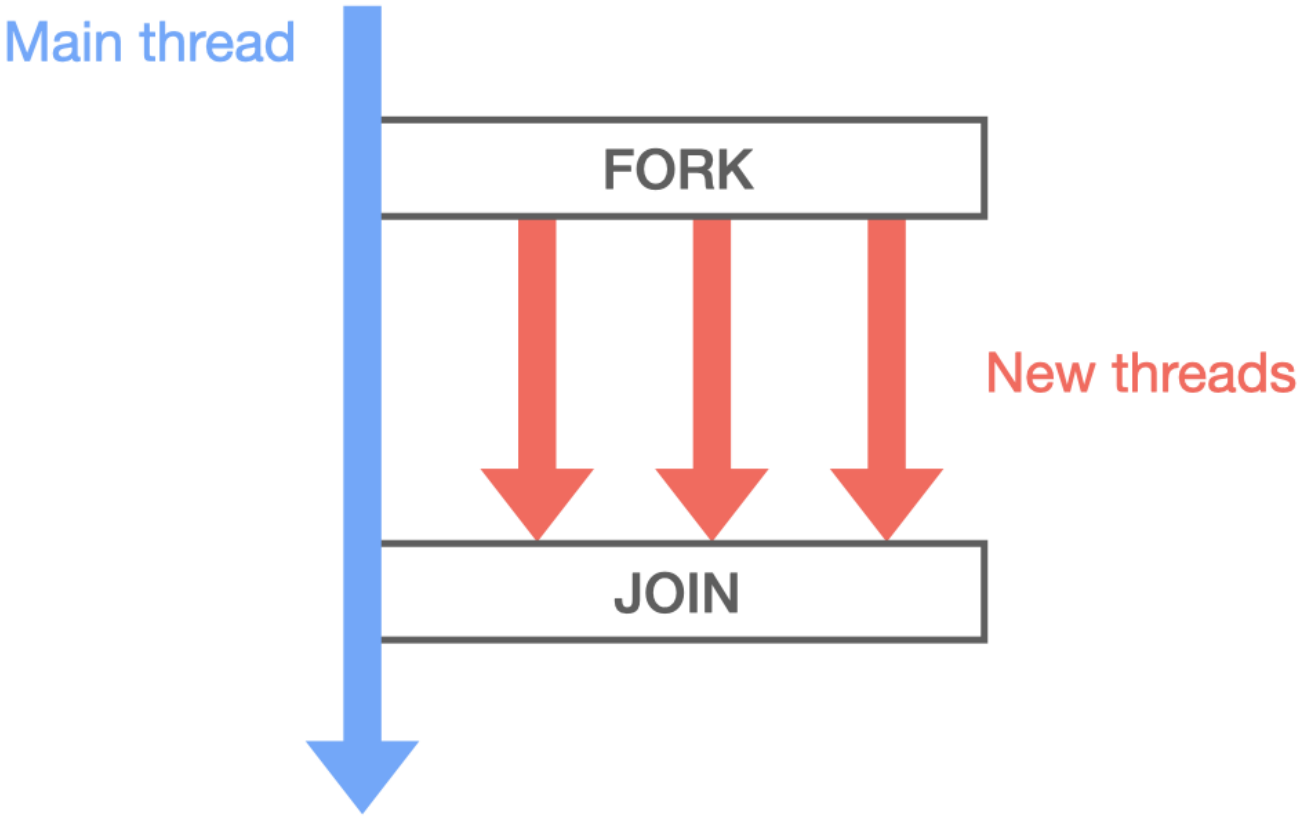
After `parallel` construct



Here comes the fork



Here comes the join



What output? (assuming 4 threads)

```c
// Main thread alone
#pragma omp parallel
{
  // All threads
  printf("Hello world.\n");
}
// Main thread alone
```

```fortran
! Main thread alone
!$OMP PARALLEL
  ! All threads
  WRITE(*, '(A)') 'Hello world'
!$OMP END PARALLEL
! Main thread alone
```

Hello world example output.

Assuming we use 4 threads, the previous source code would produce the following:

```
1 Hello world.
2 Hello world.
3 Hello world.
4 Hello world.
```

Note

Although lines are identical, the order in which they are printed is not guaranteed to be consistent.

Setting the number of threads - Environment level

You can set the OpenMP environment variable

```
export OMP_NUM_THREADS=4;
./MyProgram1 # Uses 4 threads
./MyProgram2 # Uses 4 threads
./MyProgram3 # Uses 4 threads
./MyProgram4 # Uses 4 threads
```

Setting the number of threads - Application level

You can set / overwrite the number of threads to use for a specific execution:

```
1 OMP_NUM_THREADS=4  ./MyProgram1
2 OMP_NUM_THREADS=8  ./MyProgram2
3 OMP_NUM_THREADS=16 ./MyProgram3
4 OMP_NUM_THREADS=32 ./MyProgram4
```

Setting the number of threads - Region level

You can also set / overwrite the number of threads to spawn for a specific `parallel` construct using the `num_threads` clause.

Setting the number of threads

```
1  #pragma omp parallel num_threads(2)
2  {
3      // There are two threads here
4  }
5  #pragma omp parallel num_threads(5)
6  {
7      // There are five threads here
8  }
9  #pragma omp parallel
10 {
11     // There are <OMP_NUM_THREADS> threads here
12 }
```

Setting the number of threads

```
1  !$OMP PARALLEL NUM_THREADS(2)
2      ! There are two threads here
3  !$OMP END PARALLEL
4  !$OMP PARALLEL NUM_THREADS(5)
5      ! There are five threads here
6  !$OMP END PARALLEL
7  !$OMP PARALLEL
8      ! There are <OMP_NUM_THREADS> threads here
9  !$OMP END PARALLEL
```

Setting the number of threads

Careful

The number of thread is set only for this specific `parallel` construct. If the next `parallel` construct does not have a `num_threads` clause, it will rely on the value that was set by `OMP_NUM_THREADS`[2].

[2] and potentially changed with `omp_set_num_threads`

How to enable support for OpenMP directives?

- Your compiler has built-in support for OpenMP.
- To tell your compiler to enable support for OpenMP directives, need to pass a flag
  - `-fopenmp` for GNU compilers
  - `-qopenmp` for Intel compilers
- Examples for C compilers:
  - `gcc -o main main.c -fopenmp`
  - `icc -o main main.c -qopenmp`
- Examples for FORTRAN compilers:
  - `gfortran -o main main.c -fopenmp`
  - `ifort -o main main.c -qopenmp`

Useful functions

Useful functions

`omp_get_thread_num()` Returns the id of the calling thread.

Useful functions

`omp_get_thread_num()`	Returns the id of the calling thread.
`omp_get_num_threads()`	Returns the number of threads at the calling location.

Useful functions

`omp_get_thread_num()`	Returns the id of the calling thread.
`omp_get_num_threads()`	Returns the number of threads at the calling location.

Warning

If you call `omp_get_num_threads()` outside a `parallel` construct, it always returns 1.

How to enable support for OpenMP functions?

- OpenMP functions such as `omp_get_num_threads` are in a classic OpenMP header / library, which you need to include like any other library:
  - In C: `#include <omp.h>`
  - In FORTRAN: `USE OMP_LIB`

Time to practise: `2.HelloWorld`

Update the source code provided such that it prints:

```
1 Hello world, I am thread X. We are Y threads.
```

Tips

You will need:

- the `parallel` construct
- the `omp_get_thread_num` function
- the `omp_get_num_threads` function

Table of Contents

What are we talking about?

Is a variable passed to a `parallel` region meant to be:

- shared among all threads?
- copied on each threads?
  - with initialised value?
- etc...

What happens in this case?

```c
int a = 123;
#pragma omp parallel
{
  printf("%d.\n", a);
  a = 456;
}
printf("%d\n", a);
```

```fortran
INTEGER :: a := 123;
!$OMP PARALLEL
  PRINT *, a
  a = 456;
!$OMP END PARALLEL
PRINT *, a
```

Quite a few choices

- `shared`
- `private`
- `firstprivate`
- `lastprivate`
- `threadprivate`
- `default(none)`

The `shared` clause

- In a `parallel` construct, threads all access the same instance of a `shared` variable.

The `shared` clause

- In a `parallel` construct, threads all access the same instance of a `shared` variable.
- The `shared` variable enters the `parallel` construct with its existing value.

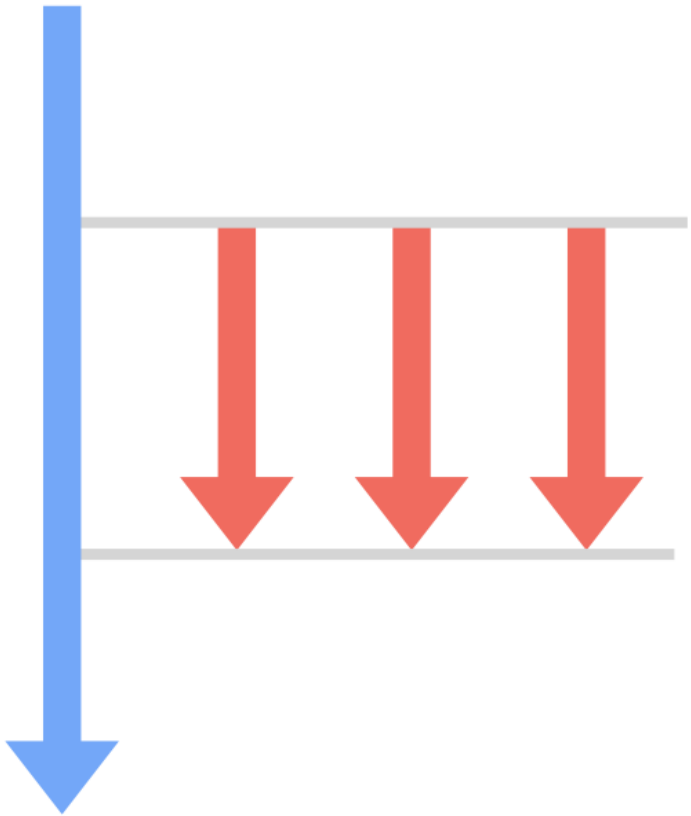
The `shared` clause

- In a `parallel` construct, threads all access the same instance of a `shared` variable.
- The `shared` variable enters the `parallel` construct with its existing value.
- In the `parallel` construct, all threads access the same instance of the original variable.

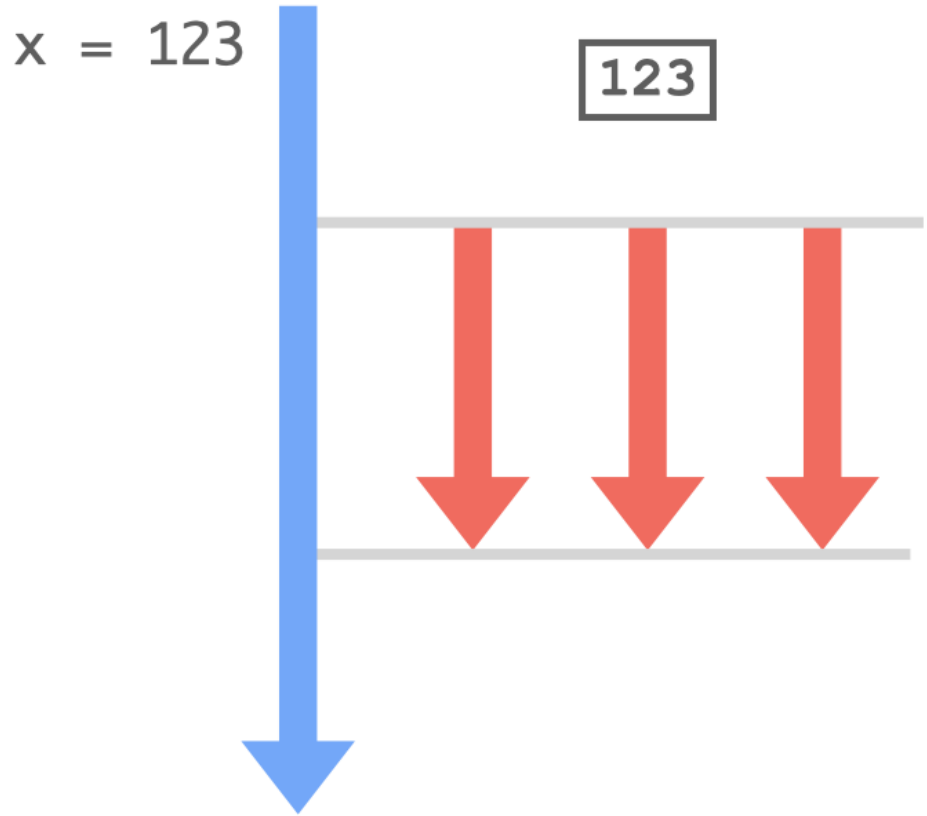
The `shared` clause

- In a `parallel` construct, threads all access the same instance of a `shared` variable.
- The `shared` variable enters the `parallel` construct with its existing value.
- In the `parallel` construct, all threads access the same instance of the original variable.
- When exiting the `parallel` construct, the `shared` variable preserves the value it had in the construct.

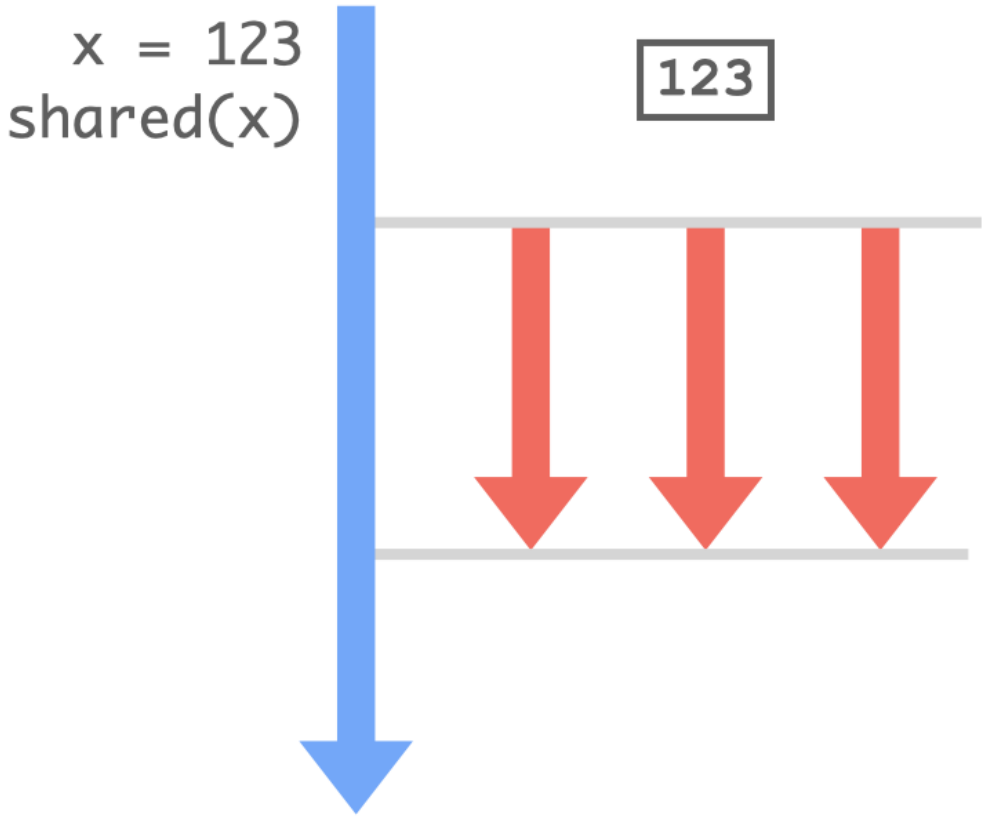
The `shared` clause



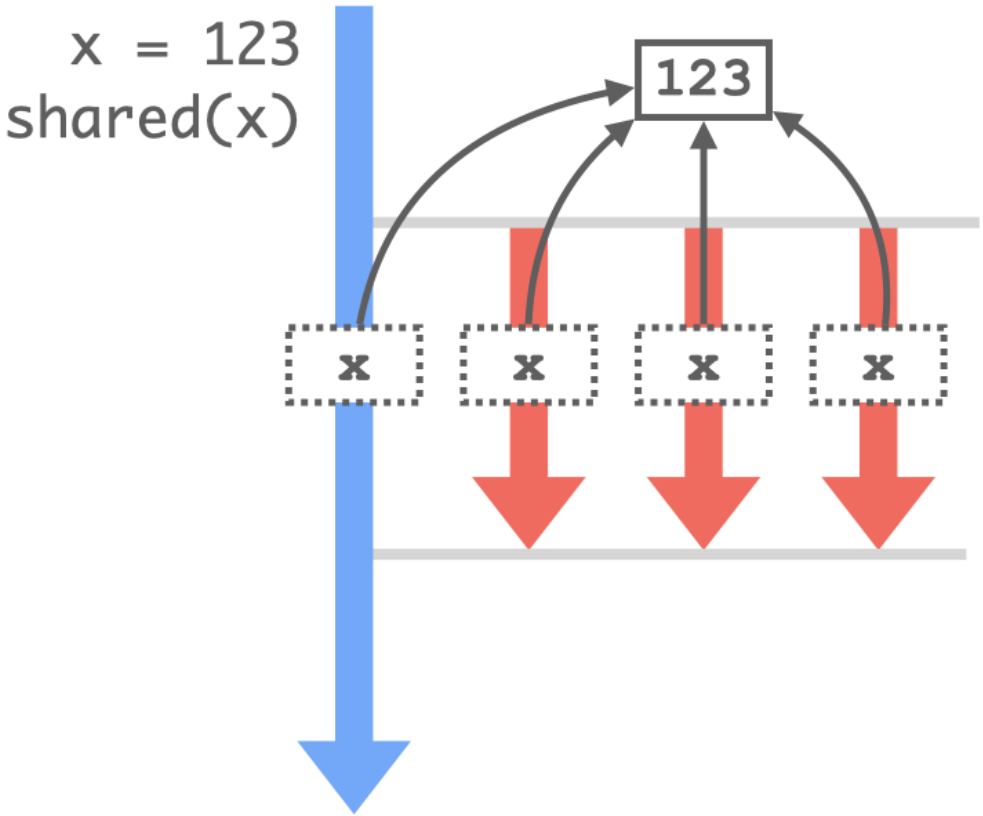
The `shared` clause



The `shared` clause



The `shared` clause



The `shared` clause

```c
int a = 123;
#pragma omp parallel shared(a)
{
  printf("%d.\n", a);
  a = 456;
}
printf("%d\n", a);
```

```fortran
INTEGER :: a := 123
!$OMP PARALLEL SHARED(a)
  PRINT *, a
  a = 456
!$OMP END PARALLEL
PRINT *, a
```

The `private` clause

- In a `parallel` construct, each thread creates its own copy of a `private` variable.

The `private` clause

- In a `parallel` construct, each thread creates its own copy of a `private` variable.
- The `private` variable enters the `parallel` construct with an undefined value.

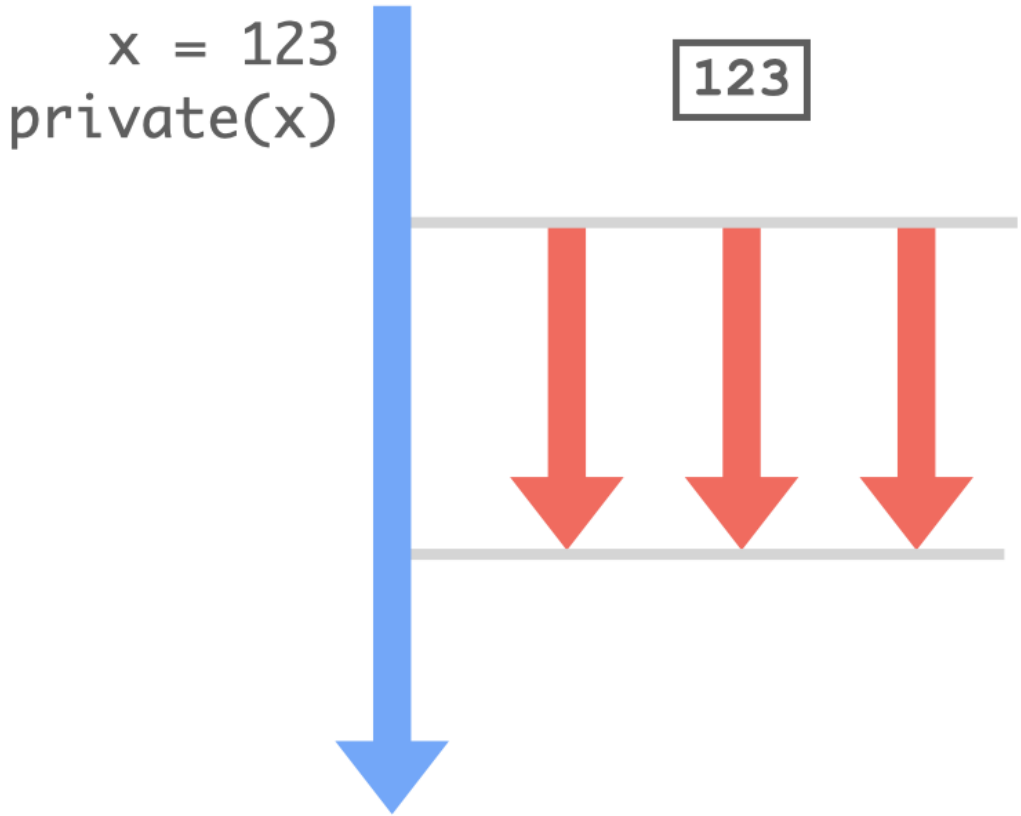
The `private` clause

- In a `parallel` construct, each thread creates its own copy of a `private` variable.
- The `private` variable enters the `parallel` construct with an undefined value.
- In the `parallel` construct, all threads access their own copy of the original variable.

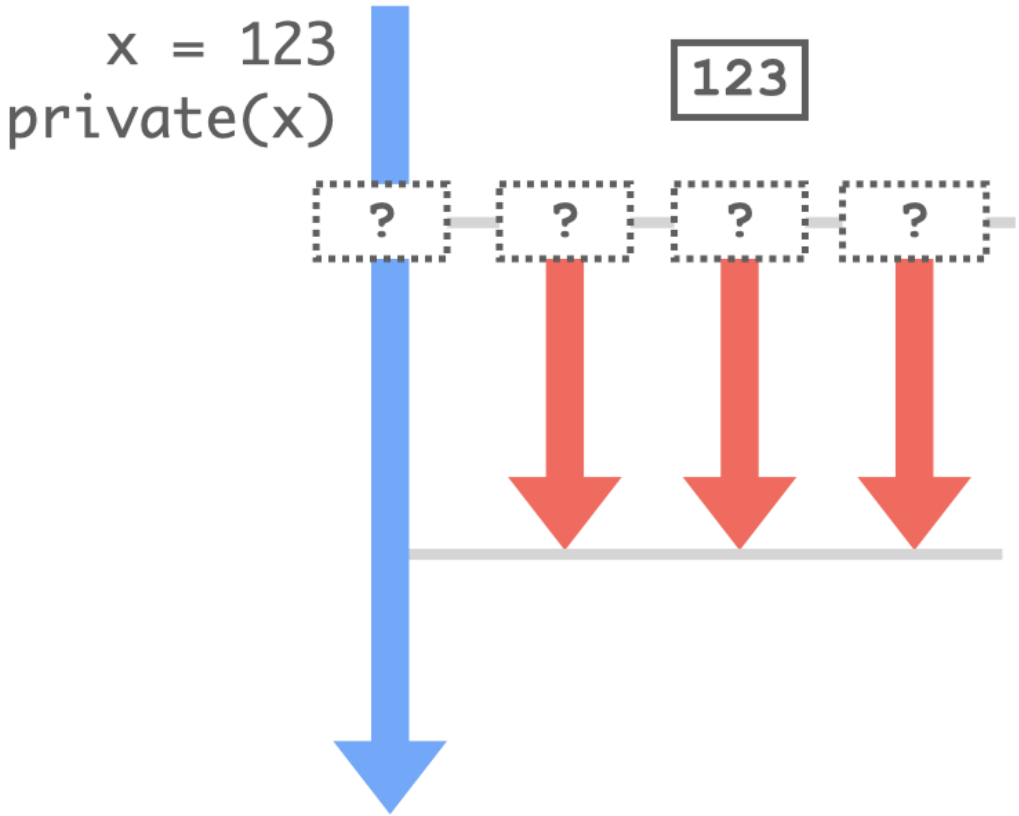
The `private` clause

- In a `parallel` construct, each thread creates its own copy of a `private` variable.
- The `private` variable enters the `parallel` construct with an undefined value.
- In the `parallel` construct, all threads access their own copy of the original variable.
- When exiting the `parallel` construct, the value of the original variable is identical to that before it.

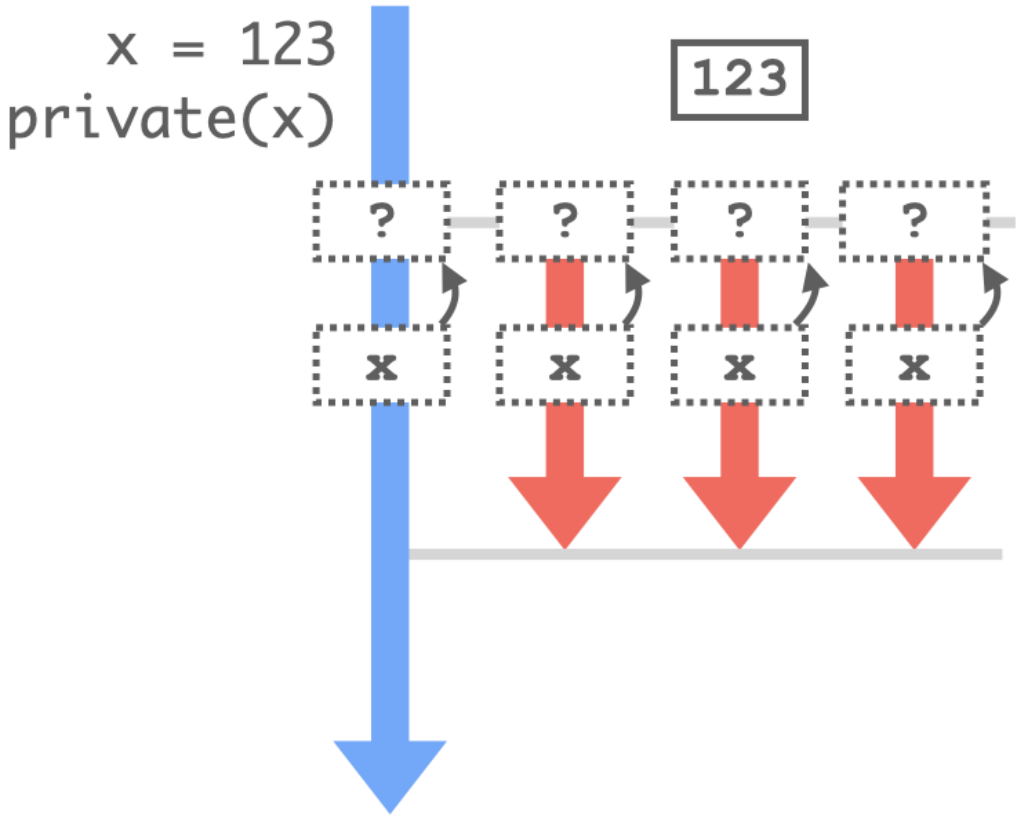
The `private` clause



The `private` clause



The `private` clause



The `private` clause

```c
int a = 123;
#pragma omp parallel private(a)
{
  printf("%d.\n", a);
  a = 456;
}
printf("%d\n", a);
```

```fortran
INTEGER :: a := 123
!$OMP PARALLEL PRIVATE(a)
  PRINT *, a
  a = 456
!$OMP END PARALLEL
PRINT *, a
```

The `firstprivate` clause

- In a `parallel` construct, each thread creates its own copy of a `firstprivate` variable.

The `firstprivate` clause

- In a `parallel` construct, each thread creates its own copy of a `firstprivate` variable.
- The `firstprivate` variable enters the `parallel` construct with the value that the original variable had.

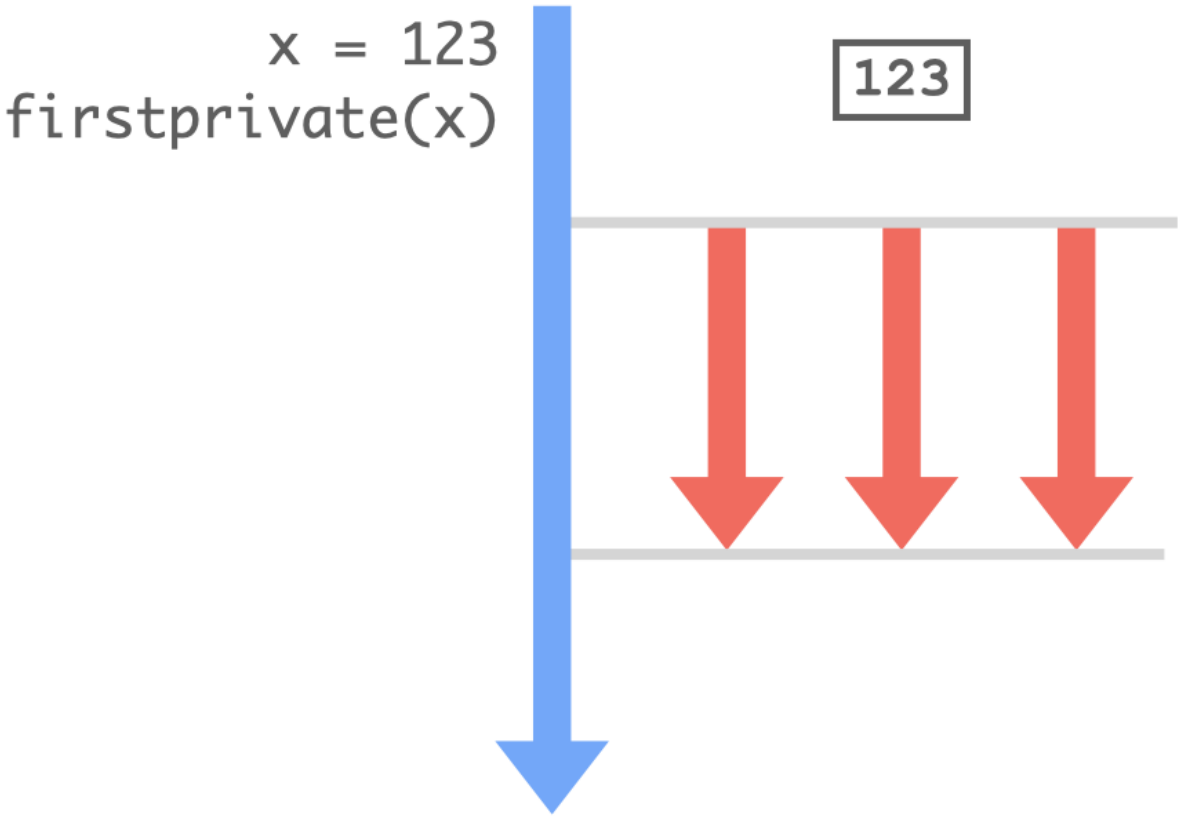
The `firstprivate` clause

- In a `parallel` construct, each thread creates its own copy of a `firstprivate` variable.
- The `firstprivate` variable enters the `parallel` construct with the value that the original variable had.
- In the `parallel` construct, all threads access their own initialised copy of the original variable.

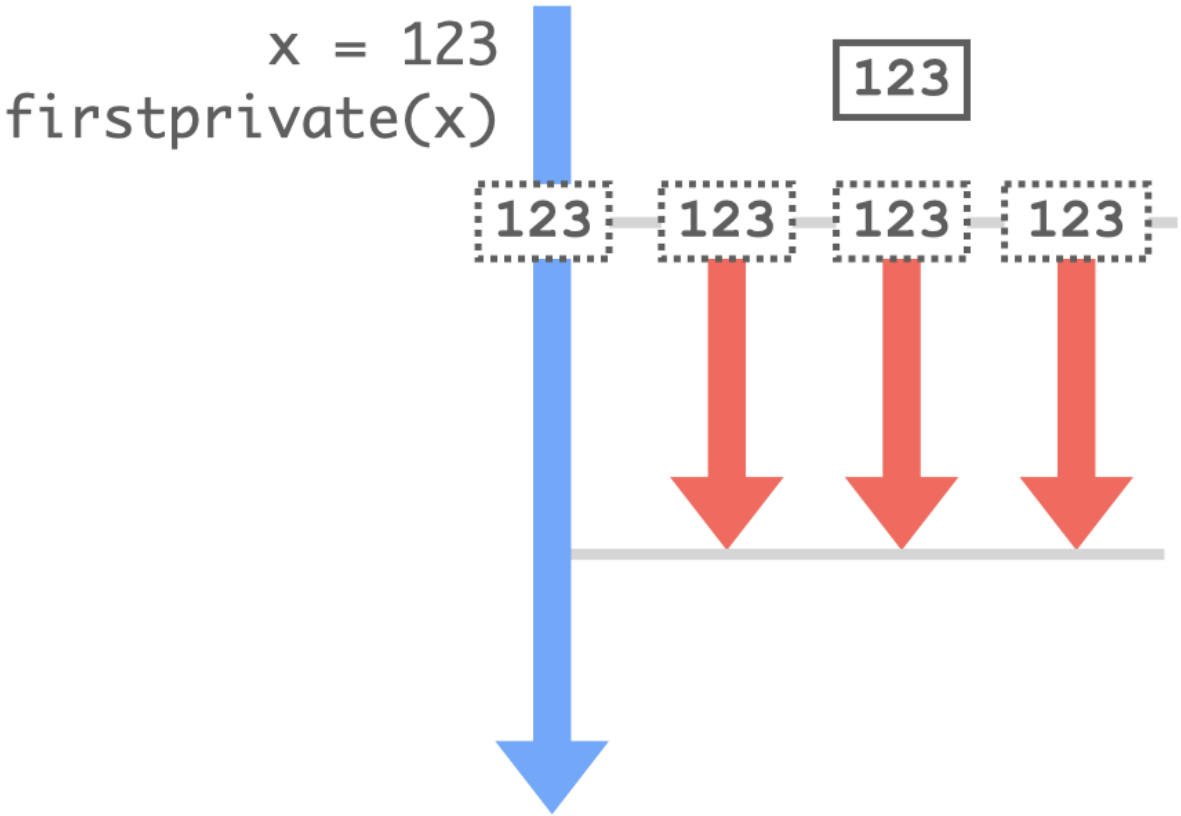
The `firstprivate` clause

- In a `parallel` construct, each thread creates its own copy of a `firstprivate` variable.
- The `firstprivate` variable enters the `parallel` construct with the value that the original variable had.
- In the `parallel` construct, all threads access their own initialised copy of the original variable.
- When exiting the `parallel` construct, the value of the original variable is identical to that before it.

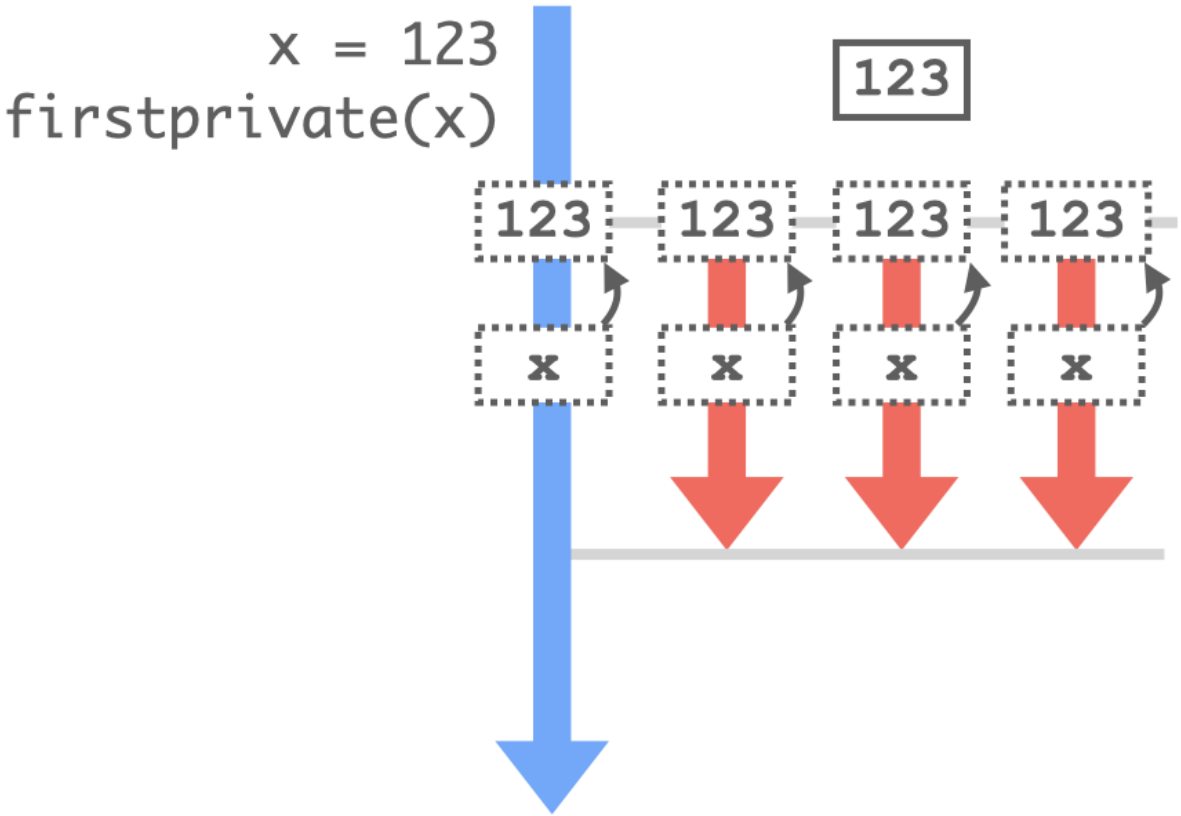
The `firstprivate` clause



The `firstprivate` clause



The `firstprivate` clause



The `firstprivate` clause

```c
int a = 123;
#pragma omp parallel firstprivate(a)
{
  printf("%d.\n", a);
  a = 456;
}
printf("%d\n", a);
```

```fortran
INTEGER :: a := 123
!$OMP PARALLEL FIRSTPRIVATE(a)
  PRINT *, a
  a = 456
!$OMP END PARALLEL
PRINT *, a
```

The `default` clause

- By default, in a `parallel` construct, variables are passed as `shared`.

The `default` clause

- By default, in a `parallel` construct, variables are passed as `shared`.
- However, it is good practice to avoid relying on implicitly declared data-sharing attributes.

The `default` clause

- By default, in a `parallel` construct, variables are passed as `shared`.
- However, it is good practice to avoid relying on implicitly declared data-sharing attributes.
- Specifying the clause `default(none)` requires explicitly passing every variable.

The `default` clause

```
1 int a = 123;
2 #pragma omp parallel default(none)
3 {
4   // Compiler complains because variable "a" is not
      specified.
5   a = 456;
6 }
```

Variables with same data-sharing

When you have multiple variables that are under the same data-sharing attribute, you can chain them using comma ',' as separators:

```
1 #pragma omp parallel default(none) shared(a,b) \
2                                    private(c,d)
3 {
4   // Your code
5 }
```

Variables with same data-sharing

When you have multiple variables that are under the same data-sharing attribute, you can chain them using comma ',' as separators:

```
1 #pragma omp parallel default(none) shared(a,b) \
2                                    private(c,d)
3 {
4   // Your code
5 }
```

What's with '\'?

To write OpenMP directives on multiple line, use the corresponding line breaker, '\' in C and '&' in FORTRAN.

OpenMP directives with line breaks

```
1 #pragma omp parallel default(none) shared(a,b) \
2                                    private(c,d)
3 {
4   // Your code
5 }
```

```
1 !$OMP PARALLEL DEFAULT(NONE) SHARED(a,b) &
2 !$OMP PRIVATE(c,d)
3   ! Your code
4 !$OMP END PARALLEL
```

Time to practise: `3.DataSharing`

Update the source code provided such that each variable gets assigned the correct data-sharing attribute.

Tips

You will need the following clauses:

- `shared`
- `private`
- `firstprivate`
- `default`

Table of Contents

Worksharing constructs

- `single`
- `master`[3]

[3] Deprecated from OpenMP version 5.2

The `single` construct

- The `single` construct indicates that the associated structured block is to be executed only by one thread, not all threads.
- You do not know which thread will execute it.
- Implicit synchronisation at the end.

The `single` construct - Example

```c
#pragma omp parallel
{
  printf("This is executed by all threads.\n");
  #pragma omp single
  {
    printf("This is executed by one thread.\n");
  }
  printf("This is executed by all threads again.\n");
}
```

```fortran
!$OMP PARALLEL
  PRINT *, "This is executed by all threads."
  !$OMP SINGLE
    PRINT *, "This is executed by one thread."
  !$OMP END SINGLE
  PRINT *, "This is executed by all threads again."
!$OMP END PARALLEL
```

The `master` construct

- Only the `master` thread executes the associated structured block.

The `master` construct

- Only the `master` thread executes the associated structured block.
- The `master` thread is the thread that runs the program and is present outside `parallel` constructs.

The `master` construct - Example

```
1 #pragma omp parallel
2 {
3   // All threads
4   #pragma omp master
5   {
6     // Only master thread
7   }
8   // All threads
9 }
```

```
!$OMP PARALLEL                    1
  ! All threads                   2
  !$OMP MASTER                    3
    ! Only master thread          4
  !$OMP END MASTER                5
  ! All threads                   6
!$OMP END PARALLEL                7
```

Difference between `master` and `single` constructs

- The `single` construct indicates that **one** thread, any thread, will execute the associated structured block.
- The `master` construct indicates that **master** thread, and only this thread, will execute the associated structured block.
- The `single` construct has an implicit barrier at the end while the `master` construct does not.

Time to practise: `4.WhoseTurn`

Update the source code provided such that each statement gets printed by the corresponding thread.

Tips

You will need the following directives:

- `single`
- `master`

Table of Contents

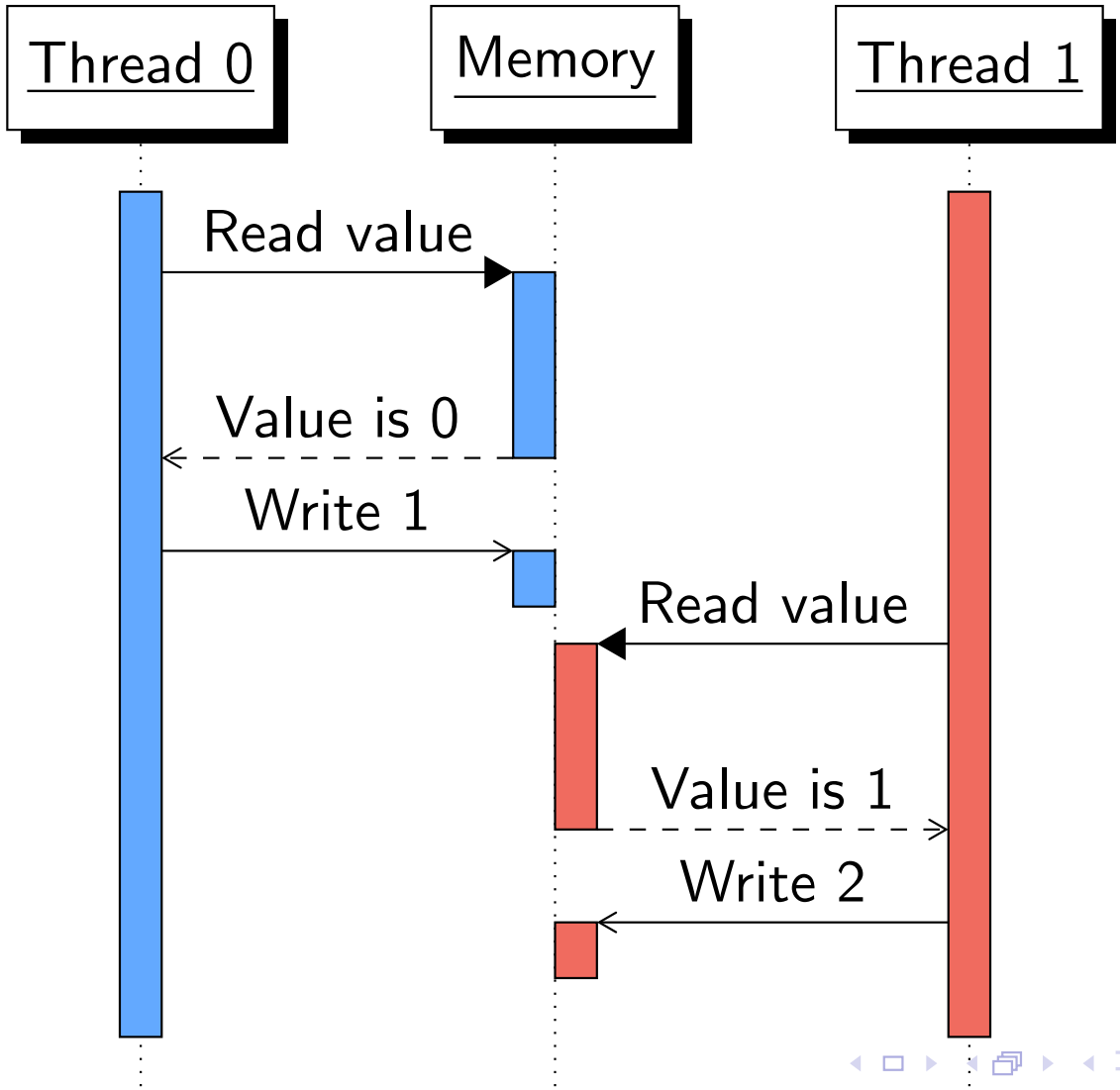
Synchronisation constructs

```
1 int thread_count = 0;
2 #pragma omp parallel default(none) shared(thread_count)
3 {
4     thread_count++;
5     #pragma omp single
6     {
7         printf("There are %d threads.\n", thread_count);
8     }
9 }
```

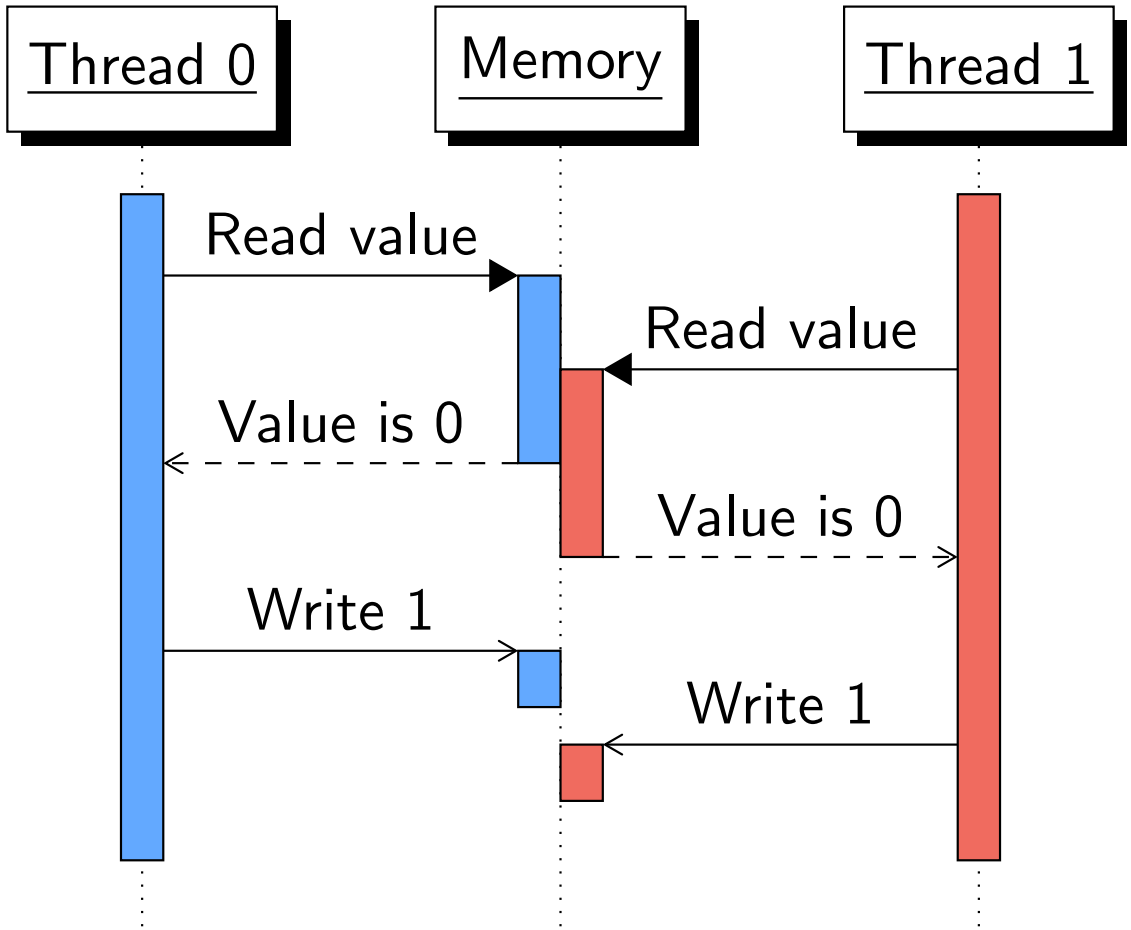
Synchronisation constructs

```
1 INTEGER :: thread_count := 0
2 !$OMP PARALLEL DEFAULT(NONE) SHARED(thread_count)
3     thread_count = thread_count + 1
4     !$OMP SINGLE
5         WRITE(*,'A,I0,A') 'There are ', thread_count, '
      threads.'
6     !$OMP END SINGLE
7 !$OMP END PARALLEL
```

Incrementing a variable - what we expect



Incrementing a variable - what can happen at times



Synchronisation constructs

- `barrier`
- `critical`

The `barrier` construct

- The `barrier` construct is a **stand-alone** directive: it has no associated structured block.

The `barrier` construct

- The `barrier` construct is a **stand-alone** directive: it has no associated structured block.
- When a thread reaches the barrier, it waits until all other threads in the `parallel` construct do the same.

The `barrier` construct

- The `barrier` construct is a **stand-alone** directive: it has no associated structured block.
- When a thread reaches the barrier, it waits until all other threads in the `parallel` construct do the same.
- Once all threads reached the barrier, their execution resumes.

The `barrier` construct - Example

```c
#pragma omp parallel
{
  #pragma omp master
  {
    // Busy for 3 seconds
  }
  #pragma omp barrier
}
```

```fortran
!$OMP PARALLEL
  !$OMP MASTER
    ! Busy for 3 seconds
  !$OMP END MASTER
  !$OMP BARRIER
!$OMP END PARALLEL
```

The `critical` construct

- The structured block associated to a `critical` construct is executed by every thread, but never more than one thread at a time.

The `critical` construct - Example

```c
#pragma omp parallel
{
  // All threads
  #pragma omp critical
  {
    // One thread at a time
  }
  // All threads
}
```

```fortran
!$OMP PARALLEL
  ! All threads
  !$OMP CRITICAL
    ! One thread at a time
  !$OMP END CRITICAL
  ! All threads
!$OMP END PARALLEL
```

Named `critical` construct

The `critical` construct also accepts an optional name clause.

Named `critical` construct

The `critical` construct also accepts an optional name clause.

- `critical` constructs with the same name are mutually exclusive

Named `critical` construct

The `critical` construct also accepts an optional name clause.

- `critical` constructs with the same name are mutually exclusive
- `critical` constructs with no name are given the same default name, so are mutually exclusive

Named `critical` construct - Example

```c
#pragma omp parallel
{
  #pragma omp critical (A)
  {
    // Exclusive with 3rd
  }
  #pragma omp critical (B)
  {
    // No exclusivity
  }
  #pragma omp critical (A)
  {
    // Exclusive with 1st
  }
}
```

```fortran
!$OMP PARALLEL
  !$OMP CRITICAL (A)
    ! Exclusive with 3rd
  !$OMP END CRITICAL
  !$OMP CRITICAL (B)
    ! No exclusivity
  !$OMP END CRITICAL
  !$OMP CRITICAL (A)
    ! Exclusive with 1st
  !$OMP END CRITICAL
!$OMP END PARALLEL
```

Time to practise: `5.Synchronisation`

Update the source code provided, so that the number of threads is incremented correctly. Also, only once the variable has its final value should one thread print it.

Tips

You will need the following directives:

- `critical`
- `barrier`

Table of Contents

Time to parallelise loops!

```
1 for(int i=0; i<8; i++)
2 {
3   a[i] = b[i] + c[i];
4 }
```

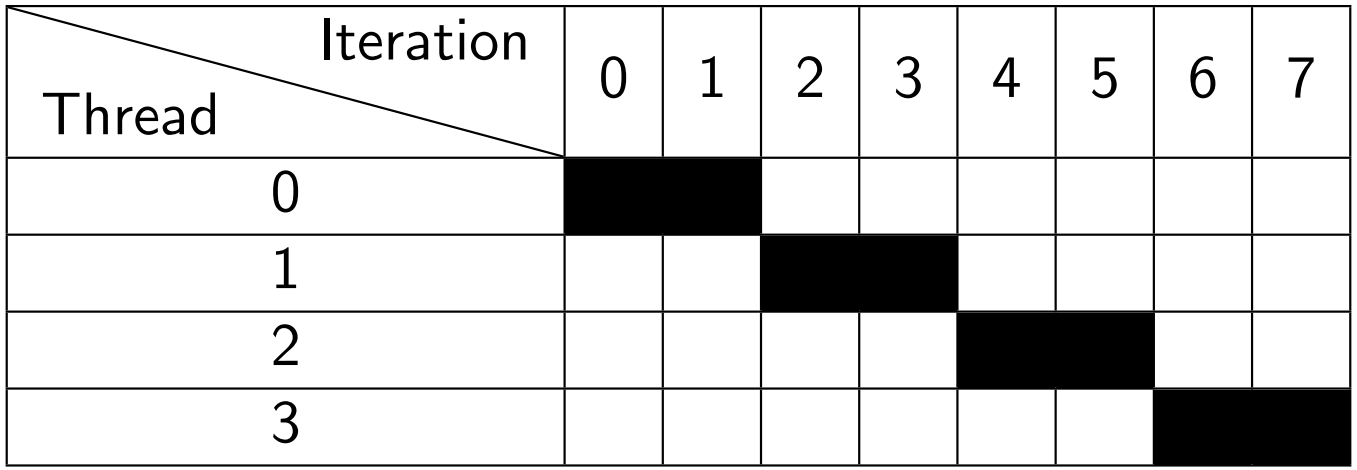
```
INTEGER :: i=0
DO i=1,8
  a[i] = b[i] + c[i];
END DO
```

Time to parallelise loops!

```
1 #pragma omp parallel
2 {
3   for(int i=0; i<8; i++)
4   {
5     a[i] = b[i] + c[i];
6   }
7 }
```

```
INTEGER :: i=0
!$OMP PARALLEL
  DO i=1,8
    a[i] = b[i] + c[i];
  END DO
!$OMP END PARALLEL
```

What we think we asked for



What we actually asked for

Thread \ Iteration	0	1	2	3	4	5	6	7
0								
1								
2								
3								

The `for` / `do` constructs

```
1 #pragma omp parallel
2 {
3   for(int i=0; i<8; i++)
4   {
5     a[i] = b[i] + c[i];
6   }
7 }
```

```
INTEGER :: i=0
!$OMP PARALLEL
  DO i=1,8
    a[i] = b[i] + c[i];
  END DO
!$OMP END PARALLEL
```

The `for` / `do` constructs

```
1  #pragma omp parallel
2  {
3    #pragma omp for
4    for(int i=0; i<8; i++)
5    {
6      a[i] = b[i] + c[i];
7    }
8  }
```

```
1  INTEGER :: i=0
2  !$OMP PARALLEL
3    !$OMP DO
4    DO i=1,8
5      a[i] = b[i] + c[i];
6    END DO
7    !$OMP END DO
8  !$OMP END PARALLEL
```

The `for` / `do` constructs

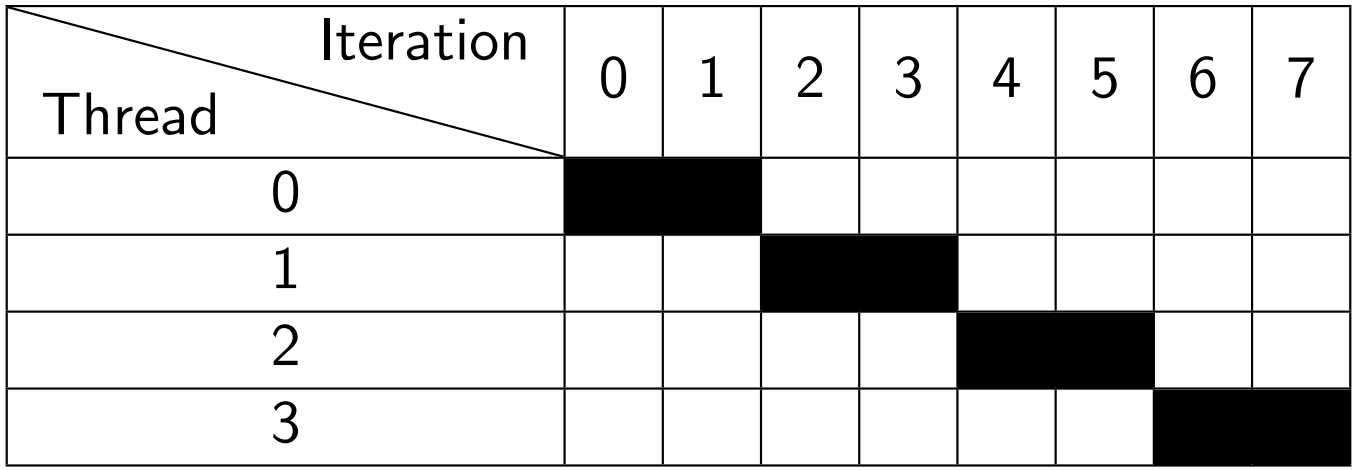
```c
#pragma omp parallel
{
  #pragma omp for
  for(int i=0; i<8; i++)
  {
    a[i] = b[i] + c[i];
  }
}
```

```fortran
INTEGER :: i=0
!$OMP PARALLEL
  !$OMP DO
  DO i=1,8
    a[i] = b[i] + c[i];
  END DO
  !$OMP END DO
!$OMP END PARALLEL
```

Note

In C, there are no brackets following a `for` directive.

Jackpot.



The combined `parallel` and `for` / `do` constructs

```
1 #pragma omp parallel for
2 for(int i=0; i<8; i++)
3 {
4   a[i] = b[i] + c[i];
5 }
```

```
INTEGER :: i=0                          1
!$OMP PARALLEL DO                       2
DO i=1,8                                3
  a[i] = b[i] + c[i];                   4
END DO                                  5
!$OMP END PARALLEL DO                   6
```

Time to conquer the world.

```
1 int total = 0;
2 for(int i=0; i<8; i++)
3 {
4     total++;
5 }
```

```
1 INTEGER :: total=0
2 INTEGER :: i=0
3 DO i=1,8
4     total = total + 1
5 END DO
```

Time to conquer the world.

```
1  int total = 0;
2  #pragma omp parallel for
3  for(int i=0; i<8; i++)
4  {
5      total++;
6  }
```

```
INTEGER :: total=0
INTEGER :: i=0
!$OMP PARALLEL DO
DO i=1,8
    total = total + 1
END DO
!$OMP PARALLEL DO
```

Time to practise: `6.Loops`

Update the source code provided, so that the code relies on a loop to achieve something that is now done in a different way.

Tips

You will need the following directives:

- `for` / `do`

Table of Contents

The `reduction` clause

- The `reduction` clause accepts two arguments:

The `reduction` clause

- The `reduction` clause accepts two arguments:
  - The reduction operation

The `reduction` clause

- The `reduction` clause accepts two arguments:
  - The reduction operation
  - The reduced variable

The `reduction` clause

- The `reduction` clause accepts two arguments:
  - The reduction operation
  - The reduced variable
- When encountering a `reduction` clause, each thread makes its own local copy of the reduced variable, before reducing them back into the original variable at the end of the `reduction` region.

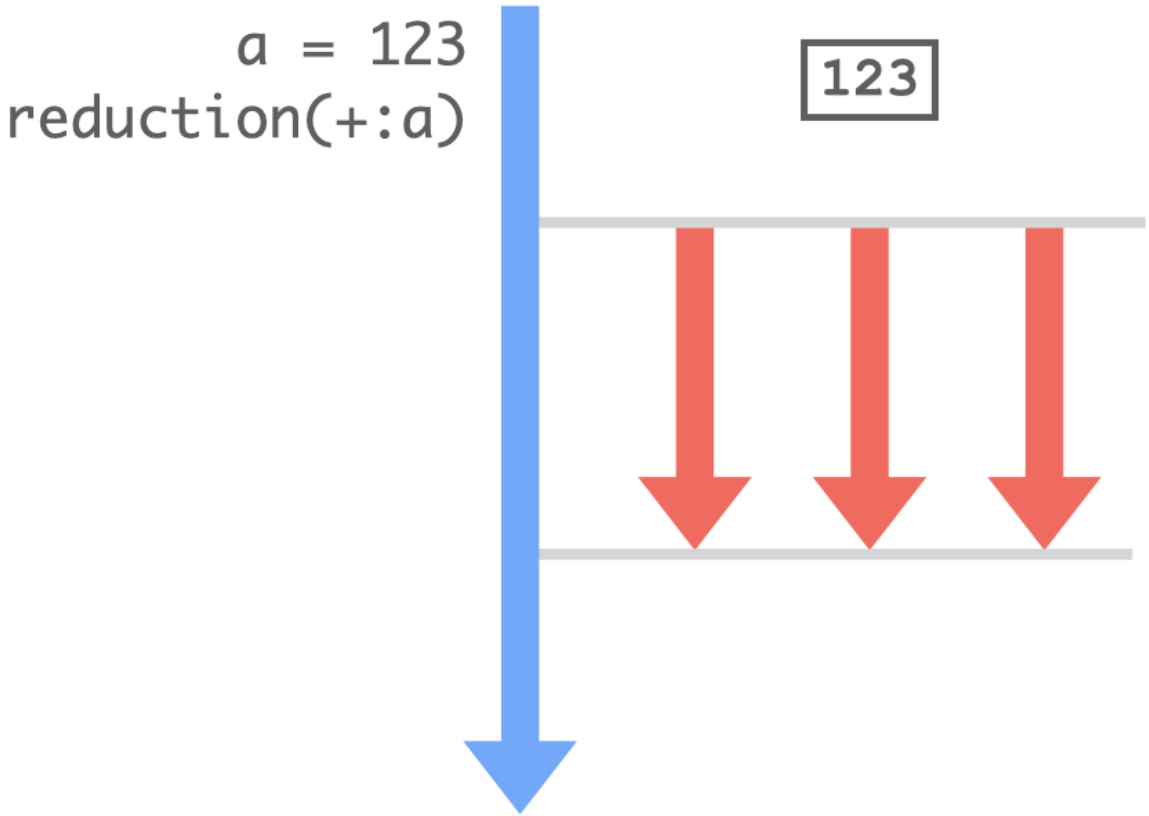
The `reduction` clause

- The `reduction` clause accepts two arguments:
  - The reduction operation
  - The reduced variable
- When encountering a `reduction` clause, each thread makes its own local copy of the reduced variable, before reducing them back into the original variable at the end of the `reduction` region.

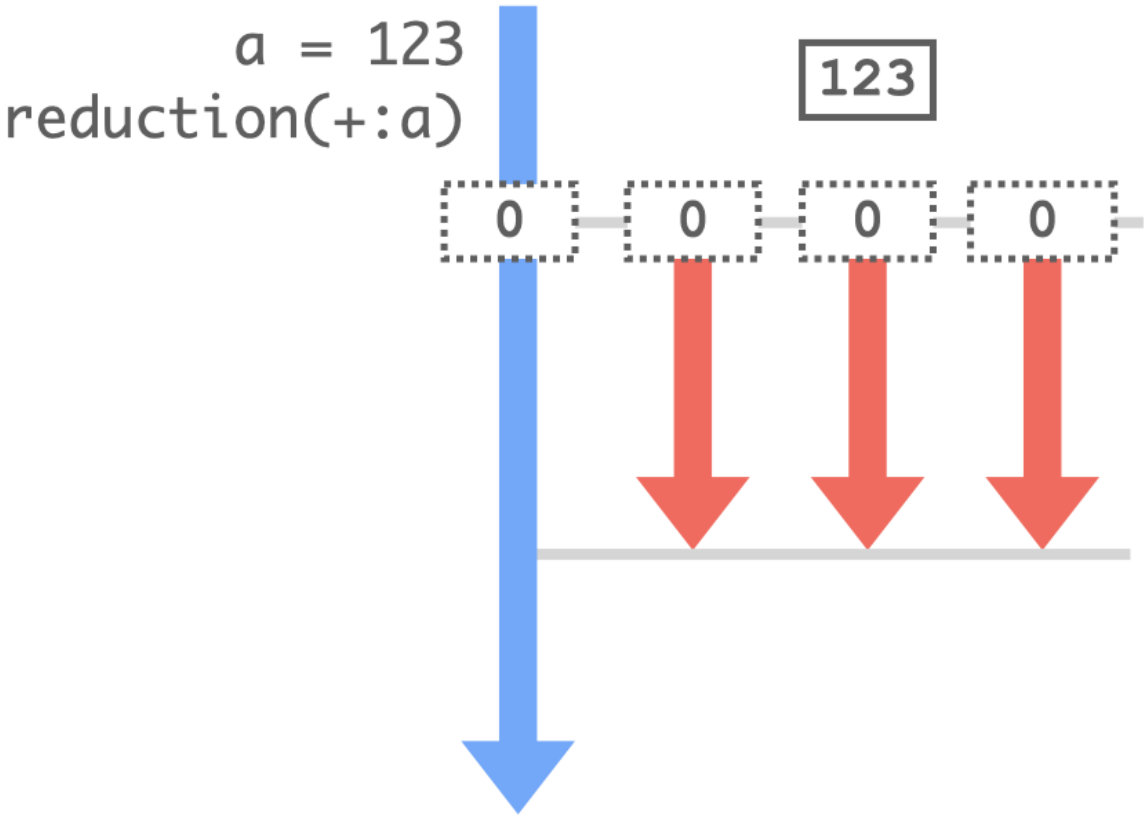
Note

The `reduction` clause is shown here as a clause to a `for` / `for` construct, however, it is a clause that can be used in a `parallel` construct.

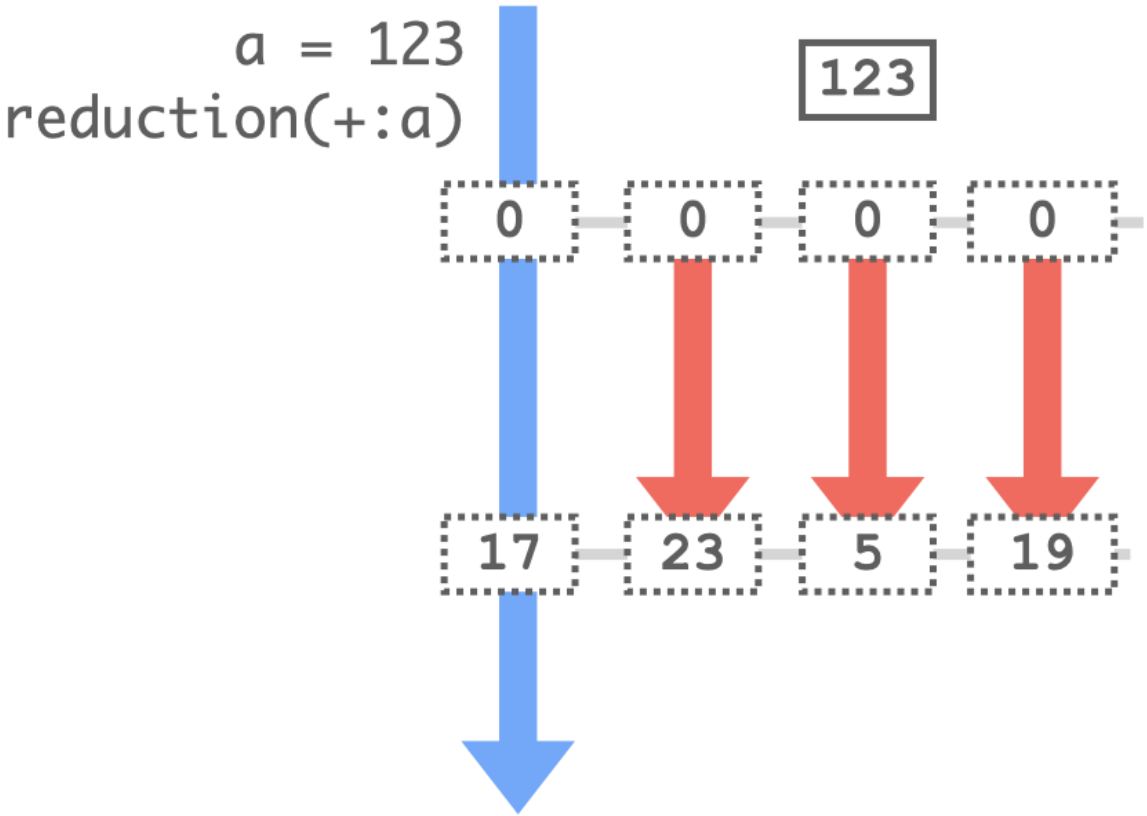
The `reduction` clause



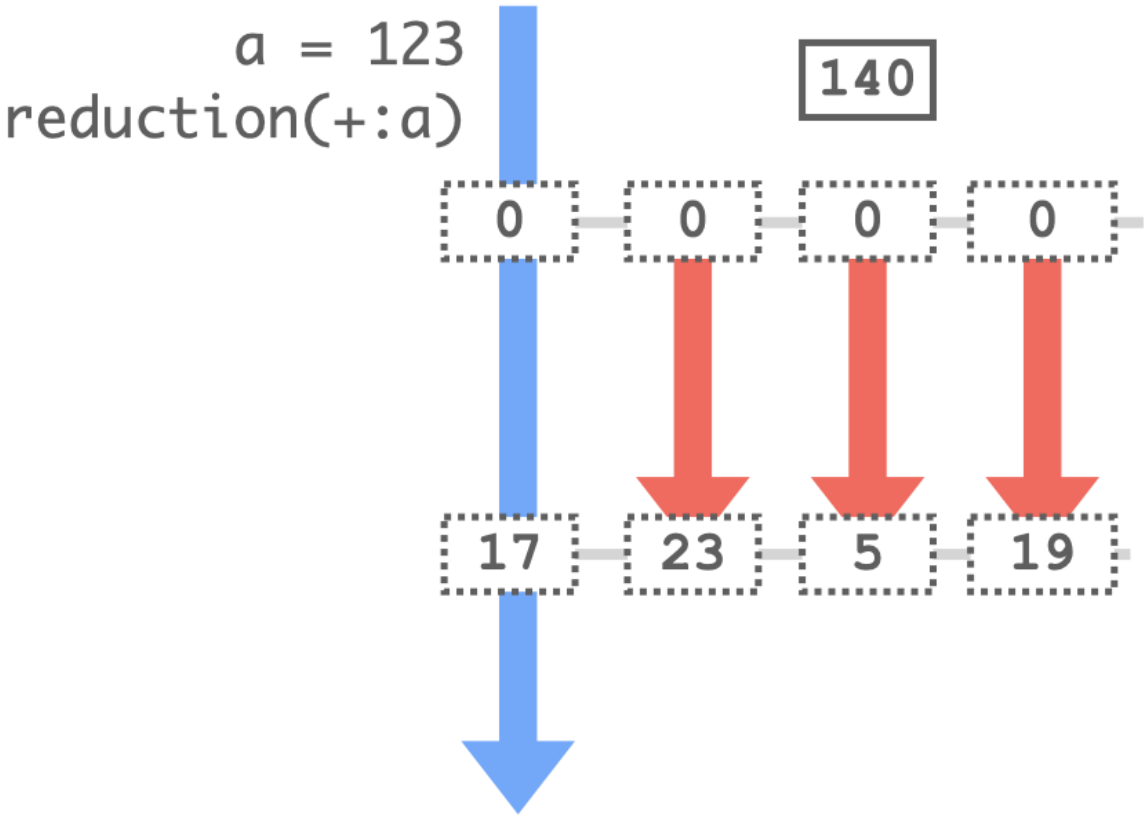
The `reduction` clause



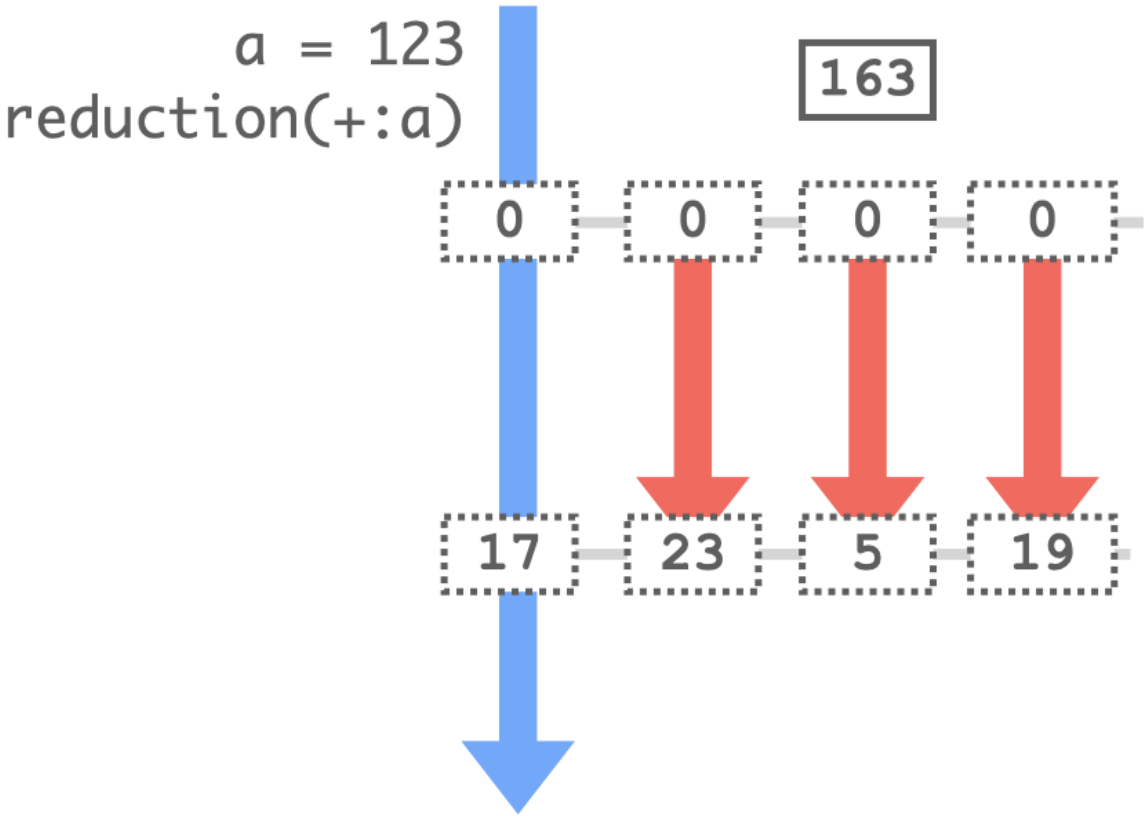
The `reduction` clause



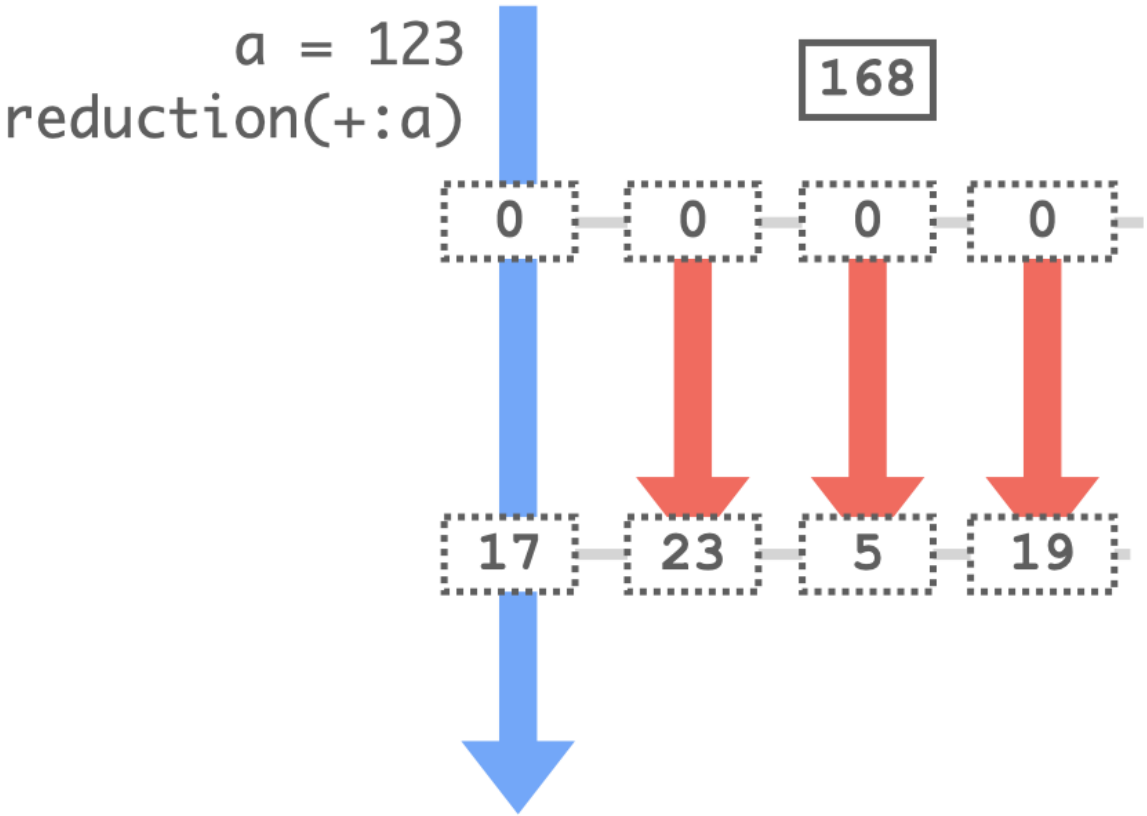
The `reduction` clause



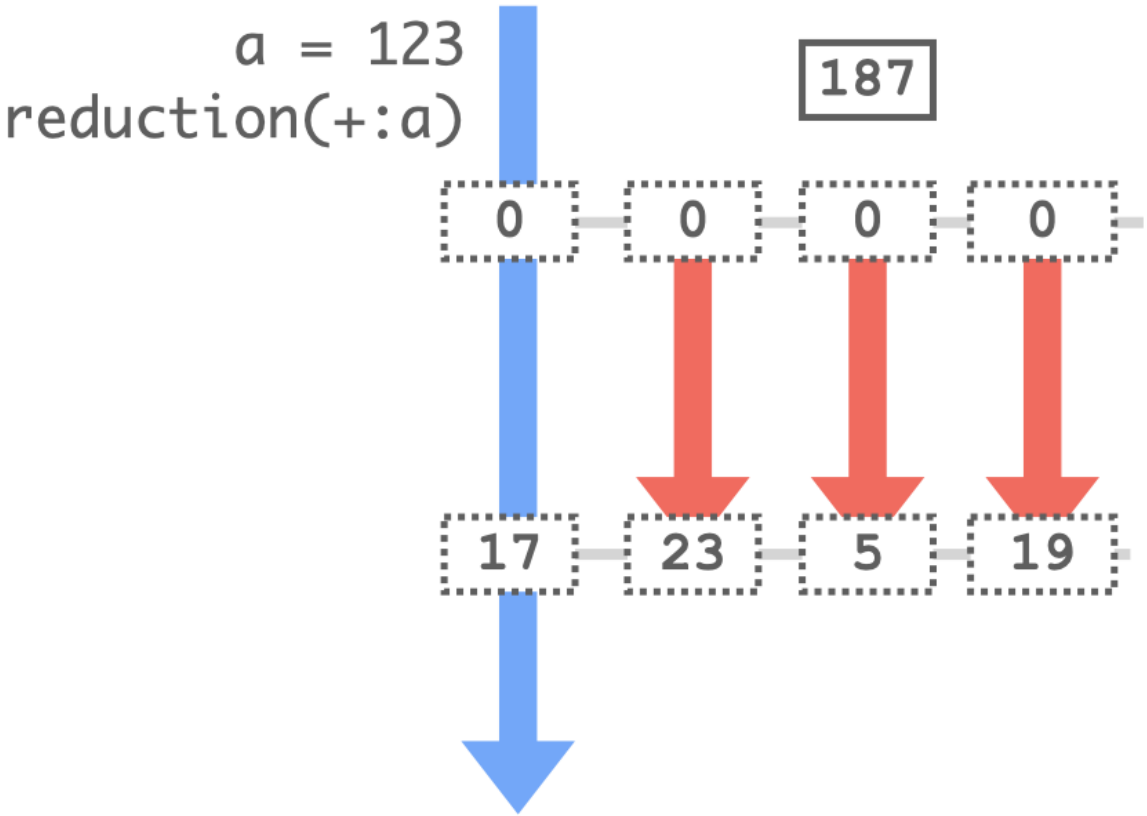
The `reduction` clause



The `reduction` clause



The `reduction` clause



The reduction operators

Operator	C	FORTRAN
Sum	`+`	`+`
Difference	`-`	`-`
Product	`*`	`*`
Minimum	`min`	`min`
Maximum	`max`	`max`
Logical AND	`&&`	`.and.`
Logical OR	`||`	`.or.`
Bit-wise AND	`&`	`iand`
Bit-wise OR	`|`	`ior`
Bit-wise exclusive or	`^`	`ieor`
Logical equivalence	n/a	`.eqv.`
Logical non-equivalence	n/a	`.neqv.`

The `reduction` clause - Example

```c
int total = 0;
#pragma omp parallel for default(none) shared(total)
for(int i=0;i<8;i++)
{
  total++;
}
```

```fortran
INTEGER :: i
!$OMP PARALLEL DO DEFAULT(NONE) SHARED(total)
  DO i=1,8
    total = total + 1
  END DO
!$OMP END PARALLEL DO
```

The `reduction` clause - Example

```
1 #pragma omp parallel for default(none) \
2                          reduction(+:total)
3 for(int i=0;i<8;i++)
4 {
5   total++;
6 }
```

```
1 INTEGER :: i
2 !$OMP PARALLEL DO DEFAULT(NONE) REDUCTION(+:total)
3   DO i=1,8
4     total = total + 1
5   END DO
6 !$OMP END PARALLEL DO
```

The `atomic` construct

- First scenario: we have threads that issue different types of operations on the given variable, so it cannot be encapsulated inside a single reduction (for example, calculating the min of their variable and next line calculating their max of their variable). Example:

```
1 #pragma omp parallel reduction(min/max:a)
2 {
3     a = min(a, some_value);
4     a = max(a, some_other_value);
5 }
```

```
1 !$OMP PARALLEL REDUCTION(MIN/MAX:a)
2     a = MIN(a, some_value);
3     a = MAX(a, some_other_value);
4 !$OMP END PARALLEL
```

The `atomic` construct

- Second scenario: we have threads using a reduction, however they need the result before the end of the parallel region. Example:

```
1 #pragma omp parallel reduction(+:a)
2 {
3     a++;
4     #pragma omp barrier
5 }
```

```
1 !$OMP PARALLEL REDUCTION(+:a)
2     a = a + 1
3     !$OMP BARRIER
4 !$OMP END PARALLEL
```

The atomic construct - Example

```c
int total = 0;
#pragma omp parallel for default(none) shared(total)
for(int i=0;i<8;i++)
{
  total++;
}
```

```fortran
INTEGER :: i
!$OMP PARALLEL DO DEFAULT(NONE) SHARED(+:total)
  DO i=1,8
    total = total + 1
  END DO
!$OMP END PARALLEL DO
```

The atomic construct - Example

```c
#pragma omp parallel for default(none) \
                         shared(total)
for(int i=0;i<8;i++)
{
  #pragma omp atomic
  total++;
}
```

```fortran
INTEGER :: i
!$OMP PARALLEL DO DEFAULT(NONE) SHARED(+:total)
  DO i=1,8
    !$OMP ATOMIC
    total = total + 1
    !$OMP END ATOMIC
  END DO
!$OMP END PARALLEL DO
```

Time to practise: `7.Reduction`

Update the source code provided such that the different calculations performed in the loop are ported to a parallelised loop, without using `critical` or `barrier` constructs.

Tips

You will need:

- the `reduction` clause
- the `atomic` construct

Table of Contents

`schedule` clause

- Use in the `for` / `do` directive.
- Indicates how the iterations are to be distributed across threads.
- Multiple scheduling kinds available.

Scheduling kinds

There are 5 different scheduling kinds available in OpenMP:

- `static`[4]
- `dynamic`
- `guided`
- `auto`
- `runtime`

[4] Although implementation-specific, usually, the `static` scheduling kind is the default.

Scheduling kinds: `static`

```
1 ... schedule(static, chunksize)
```

[5] approximately

Scheduling kinds: `static`

```
1 ... schedule(static, chunksize)
```

- Packs iterations in chunks of `<chunksize>` consecutive iterations

[5] approximately

Scheduling kinds: `static`

```
1 ... schedule(static, chunksize)
```

- Packs iterations in chunks of `<chunksize>` consecutive iterations
- Distributes iterations in a round robin fashion.

[5] approximately

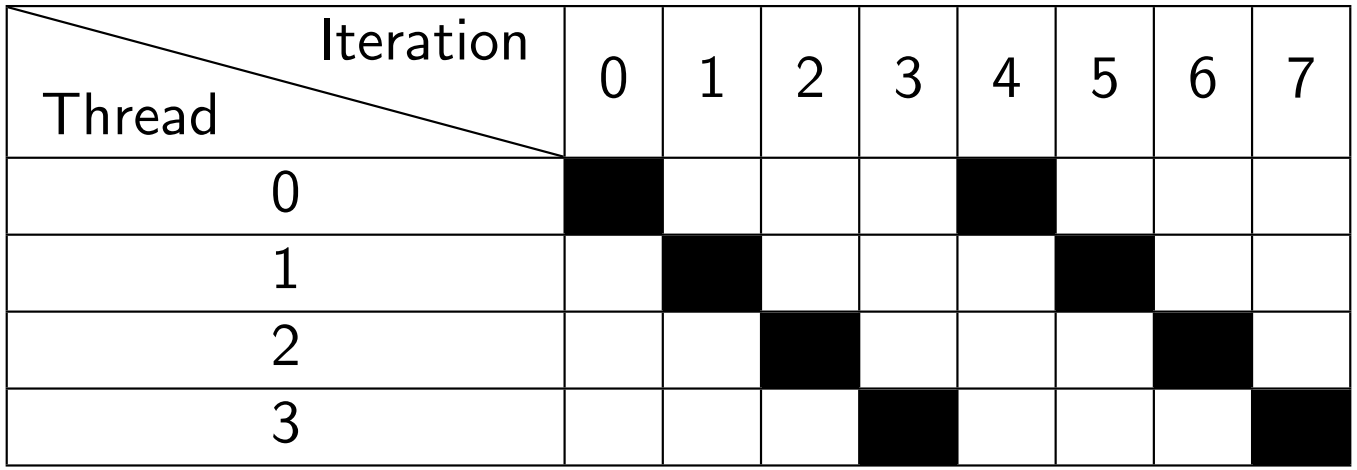
Scheduling kinds: `static`

```
1 ... schedule(static, chunksize)
```

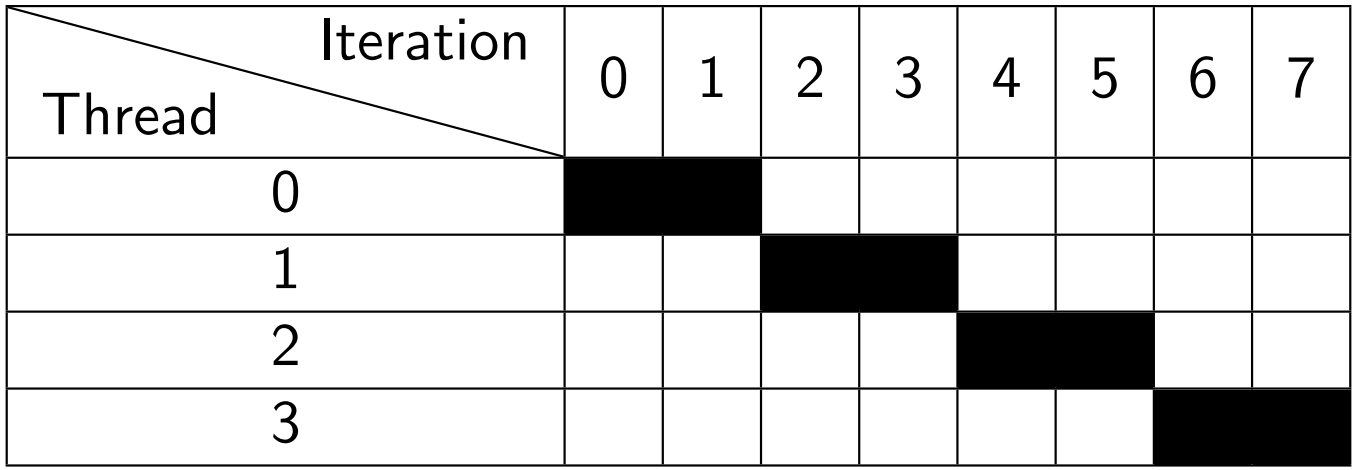
- Packs iterations in chunks of `<chunksize>` consecutive iterations
- Distributes iterations in a round robin fashion.
- Chunk size optional, defaults to 1/n for every thread[5].

[5] approximately

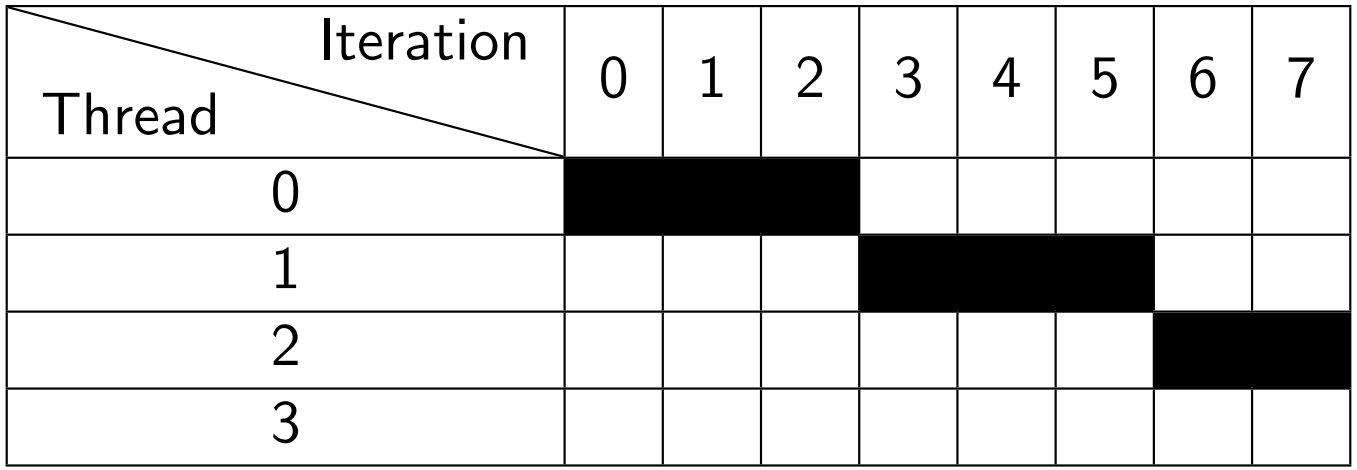
schedule(static,1)



schedule(static,2)



schedule(static,3)



Scheduling kinds: `static`

Pros:

Scheduling kinds: `static`

Pros:

- Every thread knows in advance every iteration it will process in the entire iteration set.

Scheduling kinds: `static`

Pros:

- Every thread knows in advance every iteration it will process in the entire iteration set.
- Has a low overhead.

Scheduling kinds: `static`

Pros:

- Every thread knows in advance every iteration it will process in the entire iteration set.
- Has a low overhead.

Cons:

Scheduling kinds: `static`

Pros:

- Every thread knows in advance every iteration it will process in the entire iteration set.
- Has a low overhead.

Cons:

- Struggles with load imbalance, where iterations may contain different amounts of work.

Scheduling kinds: `static`

Pros:

- Every thread knows in advance every iteration it will process in the entire iteration set.
- Has a low overhead.

Cons:

- Struggles with load imbalance, where iterations may contain different amounts of work.

`schedule(static, 1)` to the rescue!

Using `schedule(static, 1)` would indeed address the lack of load balancing. However, the cache usage would decrease, due to the hindered data locality.

Scheduling kinds: `dynamic`

```
1 ... schedule(dynamic, chunksize)
```

Scheduling kinds: `dynamic`

```
1 ... schedule(dynamic, chunksize)
```

- Distributes the first `n` chunks to the `n` threads.

Scheduling kinds: `dynamic`

```
1 ... schedule(dynamic, chunksize)
```

- Distributes the first `n` chunks to the `n` threads.
- Serving on a first-come-first-served basis afterwards

Scheduling kinds: `dynamic`

```
1 ... schedule(dynamic, chunksize)
```

- Distributes the first `n` chunks to the `n` threads.
- Serving on a first-come-first-served basis afterwards
- Chunk size optional, defaults to 1.

schedule(dynamic,1)

Thread \ Iteration	0	1	2	3	4	5	6	7
0	■							
1		■						
2			■					
3				■				

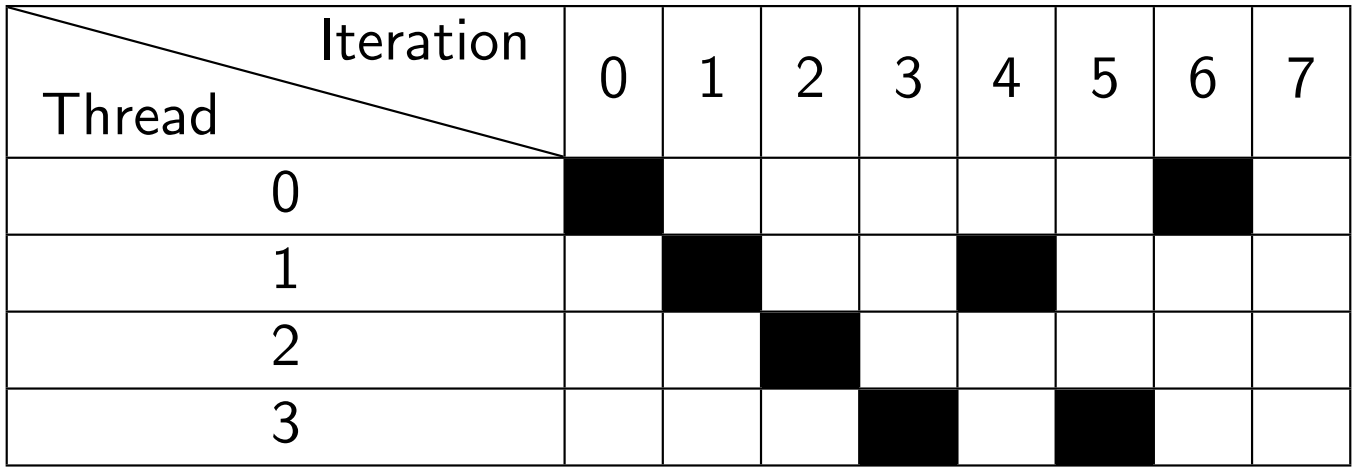
schedule(dynamic,1)

Thread \ Iteration	0	1	2	3	4	5	6	7
0	■							
1		■			■			
2			■					
3				■				

schedule(dynamic,1)

Thread \ Iteration	0	1	2	3	4	5	6	7
0	■							
1		■			■			
2			■					
3				■		■		

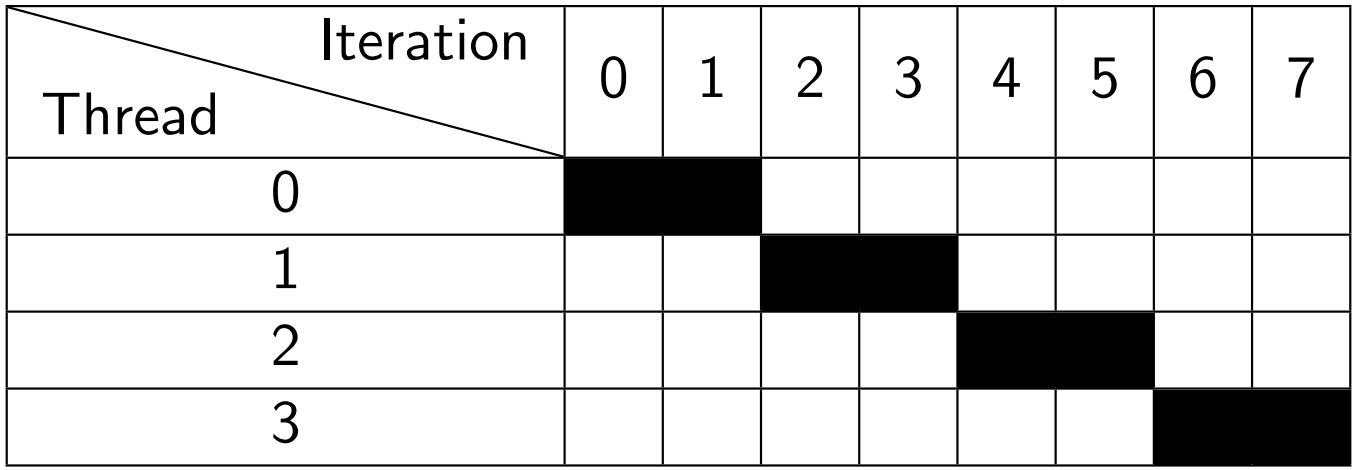
schedule(dynamic,1)



schedule(dynamic,1)

Thread \ Iteration	0	1	2	3	4	5	6	7
0	■						■	■
1		■			■			
2			■					
3				■		■		

schedule(dynamic,2)



Scheduling kinds: `dynamic`

Pros:

Scheduling kinds: `dynamic`

Pros:

- Can more efficiently address load imbalance.

Scheduling kinds: `dynamic`

Pros:

- Can more efficiently address load imbalance.

Cons:

Scheduling kinds: `dynamic`

Pros:

- Can more efficiently address load imbalance.

Cons:

- Has an overhead greater than `static` since it needs to have threads coordinate and synchronise to know who takes the next chunk.
- Must find the right chunk size.

Scheduling kinds: `dynamic`

Pros:

- Can more efficiently address load imbalance.

Cons:

- Has an overhead greater than `static` since it needs to have threads coordinate and synchronise to know who takes the next chunk.
- Must find the right chunk size.

Scheduling kinds: `guided`

```
1 ... schedule(guided, chunksize)
```

[6] The sequentially last chunk might contain fewer than `chunksize` iterations

Scheduling kinds: `guided`

```
1 ... schedule(guided, chunksize)
```

- Packs iterations in chunks of consecutive iterations, using a decreasing `chunksize`.

[6] The sequentially last chunk might contain fewer than `chunksize` iterations

Scheduling kinds: `guided`

```
1 ... schedule(guided, chunksize)
```

- Packs iterations in chunks of consecutive iterations, using a decreasing `chunksize`.
- The `chunksize` is `1/n` of the **remaining** iteration count.

[6] The sequentially last chunk might contain fewer than `chunksize` iterations

Scheduling kinds: `guided`

```
1 ... schedule(guided, chunksize)
```

- Packs iterations in chunks of consecutive iterations, using a decreasing `chunksize`.
- The `chunksize` is `1/n` of the **remaining** iteration count.
- Will decrease, until reaching `<chunksize>`[6].

[6] The sequentially last chunk might contain fewer than `chunksize` iterations

Scheduling kinds: `guided`

```
1 ... schedule(guided, chunksize)
```

- Packs iterations in chunks of consecutive iterations, using a decreasing `chunksize`.
- The `chunksize` is `1/n` of the **remaining** iteration count.
- Will decrease, until reaching `<chunksize>`[6].
- Like dynamic, rest of chunks served on first-come-first-served basis.
- The chunksize is optional, defaults to 1.

[6] The sequentially last chunk might contain fewer than `chunksize` iterations

Scheduling kinds: `guided`

```
1 ... schedule(guided, chunksize)
```

- Packs iterations in chunks of consecutive iterations, using a decreasing `chunksize`.
- The `chunksize` is `1/n` of the **remaining** iteration count.
- Will decrease, until reaching `<chunksize>`[6].
- Like dynamic, rest of chunks served on first-come-first-served basis.
- The chunksize is optional, defaults to 1.

[6] The sequentially last chunk might contain fewer than `chunksize` iterations

Scheduling kinds: `guided`

Pros:

Scheduling kinds: `guided`

Pros:

- Allows more efficient processing of decreasing workloads (upper triangular matrices etc...)

Scheduling kinds: `guided`

Pros:

- Allows more efficient processing of decreasing workloads (upper triangular matrices etc...)

Cons:

Scheduling kinds: `guided`

Pros:

- Allows more efficient processing of decreasing workloads (upper triangular matrices etc...)

Cons:

- Has an overhead greater than `static` since it needs to have threads coordinate and synchronise to know who takes the next chunk.

Scheduling kinds: `guided`

Pros:

- Allows more efficient processing of decreasing workloads (upper triangular matrices etc...)

Cons:

- Has an overhead greater than `static` since it needs to have threads coordinate and synchronise to know who takes the next chunk.

schedule(guided,2)

Thread \ Iteration	0	1	2	3	4	5	6	7	8
0									
1									
2									

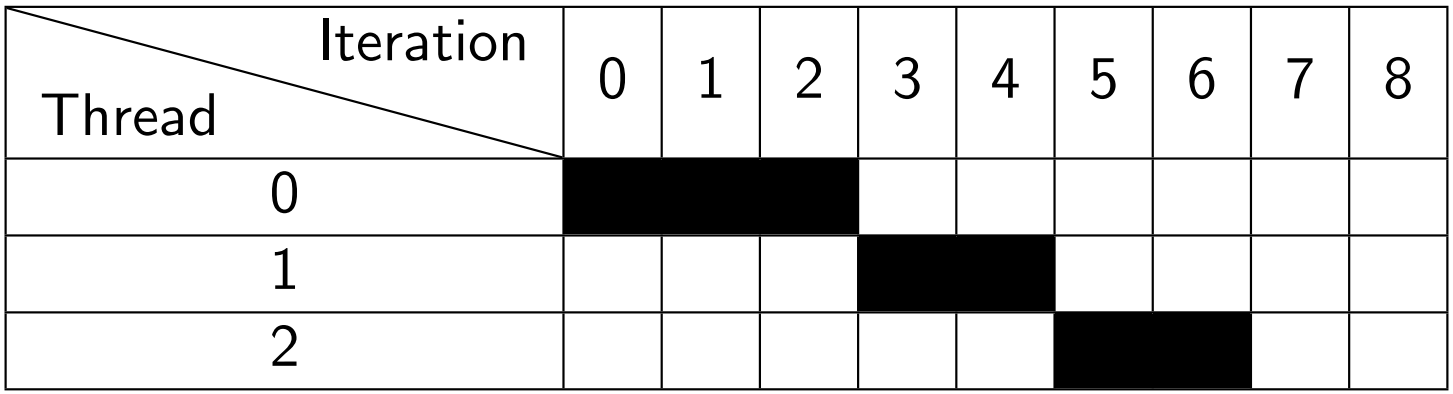
schedule(guided,2)

Thread \ Iteration	0	1	2	3	4	5	6	7	8
0									
1									
2									

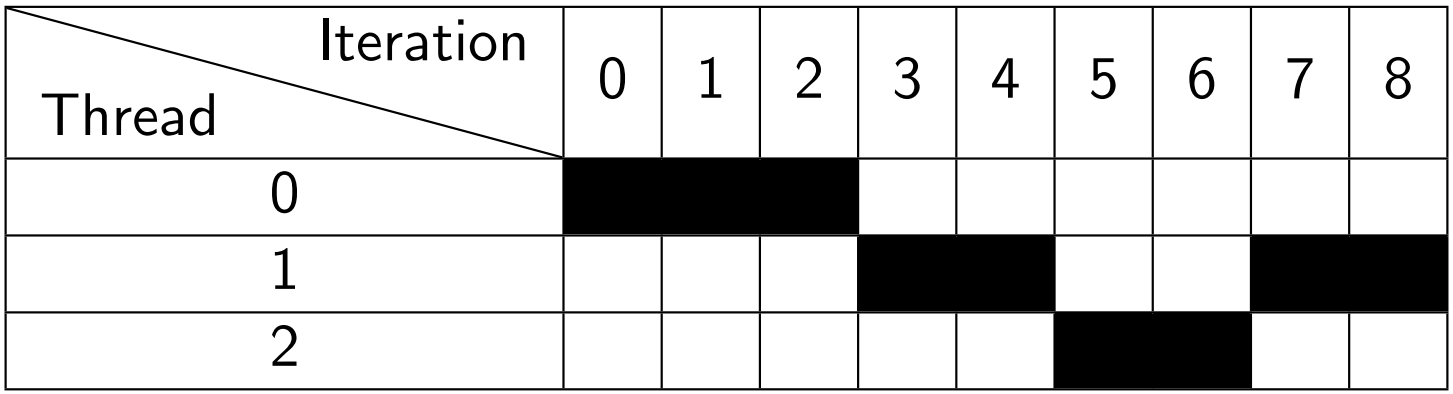
schedule(guided,2)

Thread \ Iteration	0	1	2	3	4	5	6	7	8
0	■	■	■						
1				■	■				
2									

schedule(guided,2)



schedule(guided,2)



Scheduling kinds: `auto`

```
1 ... schedule(auto)
```

- Schedule applied is implementation defined.
- No chunk size.

Scheduling kinds: `auto`

Pros:

Scheduling kinds: `auto`

Pros:

- Easy to trigger.
- Can give access to implementation-specific tricks.

Scheduling kinds: `auto`

Pros:

- Easy to trigger.
- Can give access to implementation-specific tricks.

Cons:

Scheduling kinds: `auto`

Pros:

- Easy to trigger.
- Can give access to implementation-specific tricks.

Cons:

- Have no control to tweak it.

Scheduling kinds: `auto`

Pros:

- Easy to trigger.
- Can give access to implementation-specific tricks.

Cons:

- Have no control to tweak it.

Scheduling kinds: `runtime`

- How to use: `schedule(runtime)`
- Scheduling kind applied is the one in application at runtime, see `omp_set_schedule`.

Scheduling kinds: `runtime`

```
1 void omp_set_schedule(schedule, chunksize);
```

Figure: C binding of `omp_set_schedule`

```
1 PROCEDURE omp_set_schedule(kind, chunk_size)
2     INTEGER(KIND=omp_sched_kind) :: kind
3     INTEGER :: chunk_size
```

Figure: FORTRAN-90 binding of `omp_set_schedule`

Time to practise: `8.Schedules`

You are provided with a source code that has multiple for loops.
The objective is to find the best scheduling kind for each.

Tips

You may need:

- `static`
- `dynamic`
- `guided`
- `auto`
- `runtime`

Table of Contents

There is more to see...

- Task-based parallelism

There is more to see...

- Task-based parallelism
- Collapsing loops

There is more to see...

- Task-based parallelism
- Collapsing loops
- Tiling loops

There is more to see...

- Task-based parallelism
- Collapsing loops
- Tiling loops
- Hyperthreading

There is more to see...

- Task-based parallelism
- Collapsing loops
- Tiling loops
- Hyperthreading
- False-sharing

There is more to see...

- Task-based parallelism
- Collapsing loops
- Tiling loops
- Hyperthreading
- False-sharing
- Asynchrony

There is more to see...

- Task-based parallelism
- Collapsing loops
- Tiling loops
- Hyperthreading
- False-sharing
- Asynchrony
- A lot more...

There is more to see...

- Task-based parallelism
- Collapsing loops
- Tiling loops
- Hyperthreading
- False-sharing
- Asynchrony
- A lot more...

The best place to learn more about OpenMP and how it works, to get the specifications and so on is the OpenMP forum website.

Post-workshop survey

Figure: Link to the post-workshop survey.

Feel free to connect

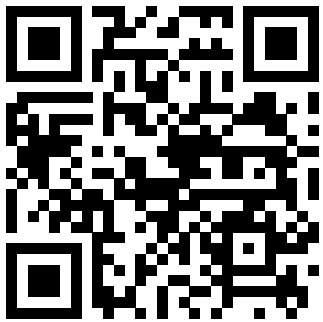


Figure: Link to LinkedIn profile