Introduction to OpenMP for CPUs

International High-Performance Computing Summer School 2023

Ludovic Capelli

EPCC

July 10, 2023

Individuals

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

- 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]

Timetable

First session: Monday

- Topic: CPUs
- From 14:00 until 15:30
- Lunch break
- From 16:00 until 17:30

Timetable

First session: Monday

- Topic: CPUs
- From 14:00 until 15:30
- Lunch break
- From 16:00 until 17:30

Second session: Tuesday

- Topic: GPUs
- From 9:00 until 10:30
- Lunch break
- From 11:00 until 12:30

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 bridges2

```
ssh your_username@bridges2.psc.edu
```

- Clone the repository used in this session

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

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/1.Preamble
```

- Compile the source code in it

```
make
```

- Run the executable.

```
./Preamble
```

Workflow

- Frequent exercises so get your laptop ready.
- If on Bridges2, any module to load?
- Content available in C and FORTRAN.
- Feedback form available, anonymous.

Table of content

1. Motivation
2. How it works
3. The parallel construct
4. Data-sharing attribute clauses
5. Worksharing construct
6. Synchronisation constructs
7. Loops time
8. Reduction
9. Schedule clause
10. Summary

Table of Contents

1 Motivation
2 How it works
3 The parallel construct
4 Data-sharing attribute clauses
5 Worksharing construct
6 Synchronisation constructs
7 Loops time
8 Reduction
9 Schedule clause
10 Summary

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

1 Motivation
2 How it works
3 The parallel construct
4 Data-sharing attribute clauses
5 Worksharing construct
6 Synchronisation constructs
7 Loops time
8 Reduction
9 Schedule clause
10 Summary

Structure of a construct

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

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

Structure of a construct

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

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

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

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

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

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

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

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

Clauses A list of additional features / options to enable.

Structure of a construct

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

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

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

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

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

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

1 Motivation

2 How it works

3 The parallel construct

4 Data-sharing attribute clauses

5 Worksharing construct

6 Synchronisation constructs

7 Loops time

8 Reduction

9 Schedule clause

10 Summary

Hello world example

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

```fortran
! Main thread alone
PRINT *, 'Hello world.'
```

Hello world example

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

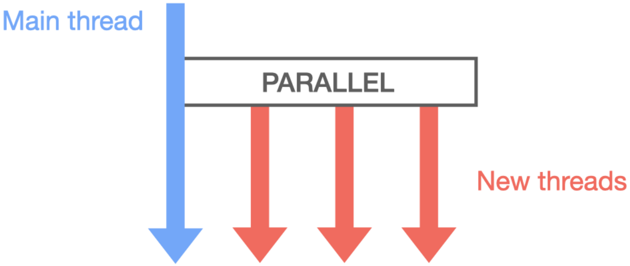
```fortran
! Main thread alone
!$OMP PARALLEL
  ! All threads
  PRINT *, 'Hello world.'
!$OMP END PARALLEL
! Main thread alone
```

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

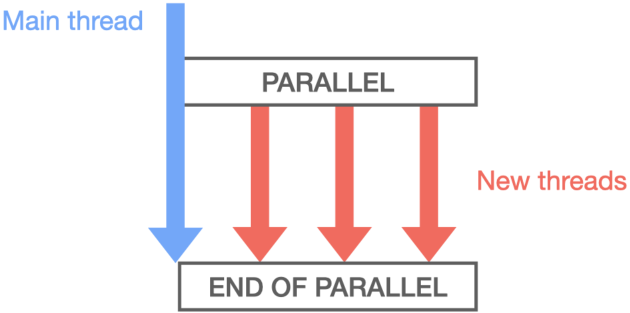
Before the `parallel` construct

Entering `parallel` construct

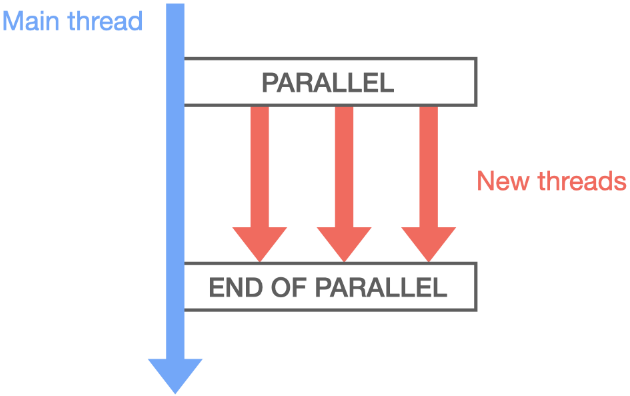
Inside the `parallel` construct



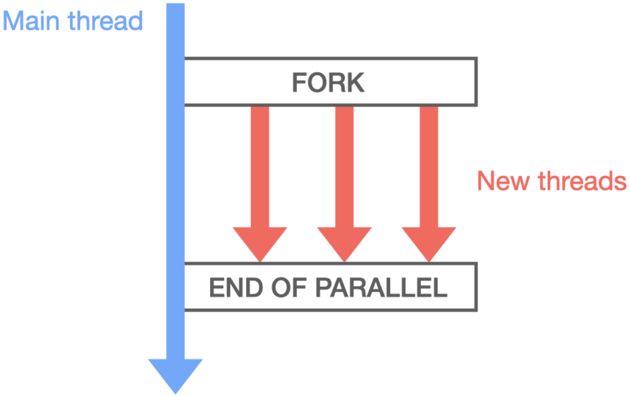
Exiting the `parallel` construct



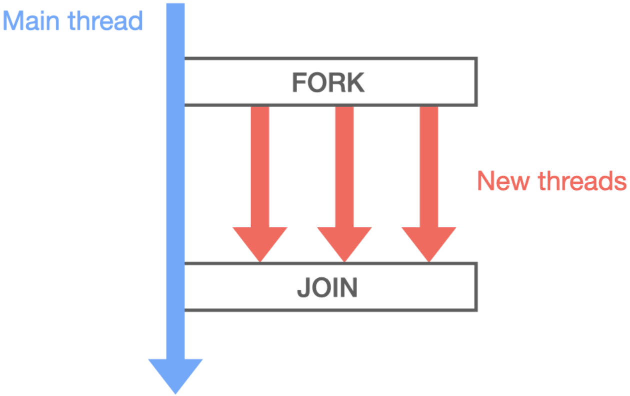
After `parallel` construct



Here comes the fork



Here comes the join



Back to our hello world example, what output?

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

```fortran
! Main thread alone
!$OMP PARALLEL
  ! All threads
  PRINT *, 'Hello world.'
!$OMP END PARALLEL
! Main thread alone
```

Hello world example output.

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

```
Hello world.
Hello world.
Hello world.
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

You can set the number of threads by setting the environment variable `OMP_NUM_THREADS`. Either by specifying it manually for every execution:

```
OMP_NUM_THREADS=4 ./MyProgram
```

or storing it in the corresponding OpenMP environment variable:

```
export OMP_NUM_THREADS=4;
./MyProgram
```

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.f90 -fopenmp`
  - `ifort -o main main.f90 -qopenmp`

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 practice: `2.HelloWorld`

Update the source code provided such that it prints:

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

Tips

You will need:

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

Table of Contents

1 Motivation
2 How it works
3 The parallel construct
4 Data-sharing attribute clauses
5 Worksharing construct
6 Synchronisation constructs
7 Loops time
8 Reduction
9 Schedule clause
10 Summary

What are we talking about?

- A clause in the `parallel` construct.
- Is a variable meant to be shared among all threads?
- Are threads supposed to have their own copy?
- If they have their own copy, does it come pre-initialised?
- 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
```

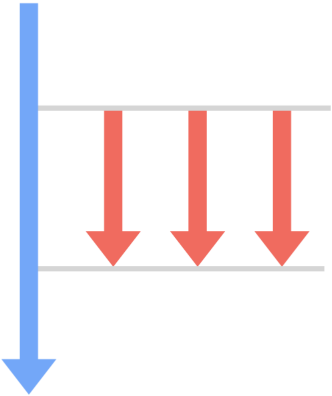
Quite a few choices

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

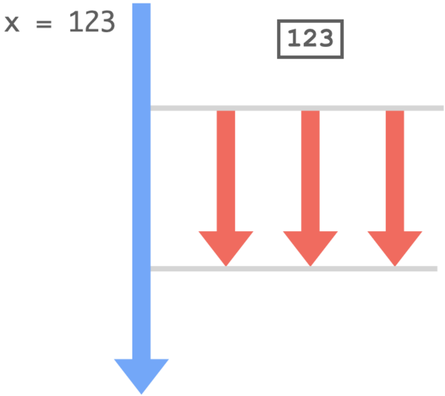
The `shared` clause

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

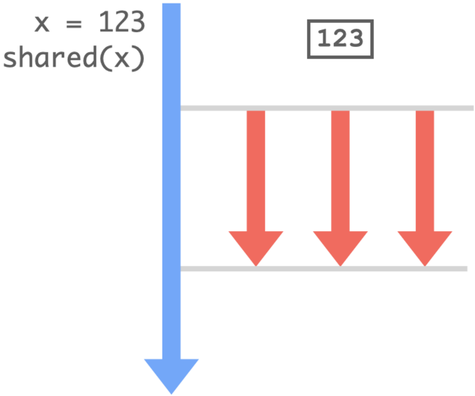
The `shared` clause



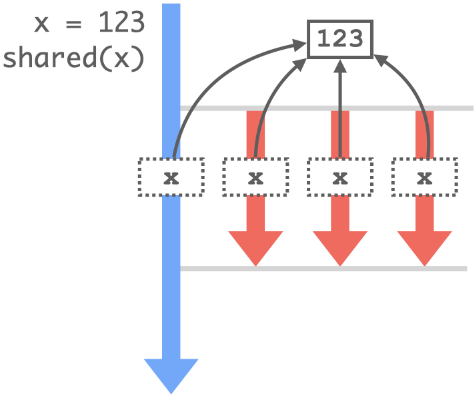
The `shared` clause



The `shared` clause



The `shared` clause



The shared clause

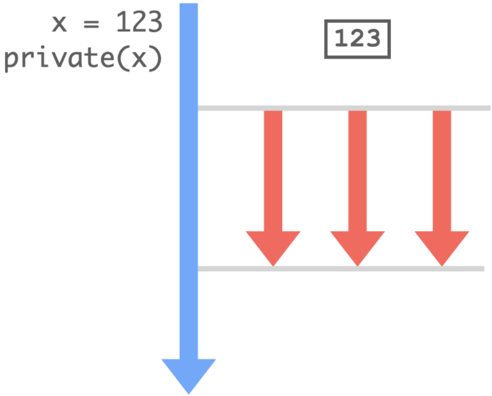
```c
int a = 123;
#pragma omp parallel shared(a)
{
  printf("%d.\n", a);
  a = 456;
}
printf("%d\n", a);
```

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

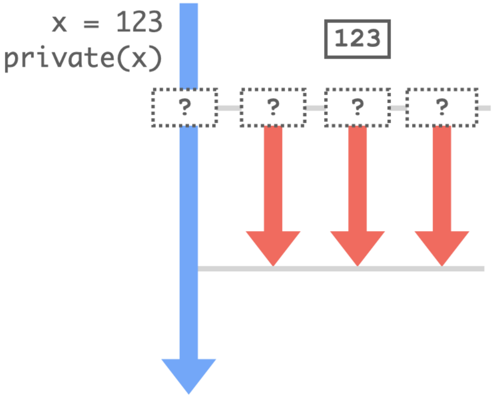
The `private` clause

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

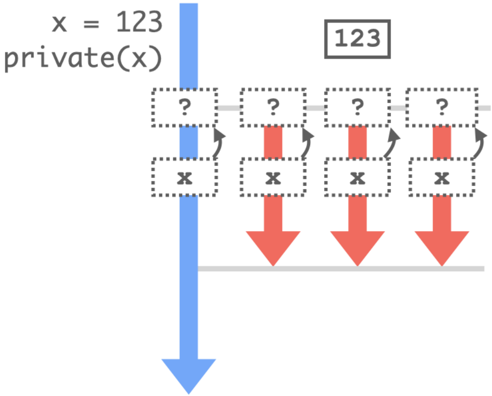
The `private` clause



The `private` clause



The `private` clause



The `private` clause

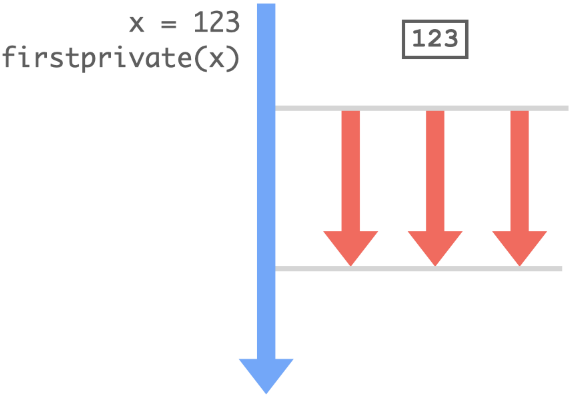
```c
int a = 123;
#pragma omp parallel private(a)
{
  printf("%d.\n", a);
  a = 456;
}
printf("%d\n", a);
```

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

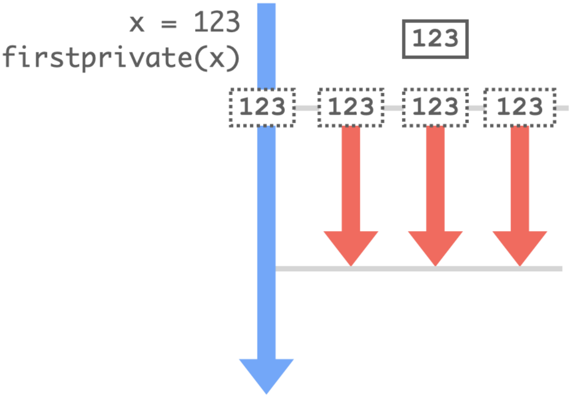
The `firstprivate` clause

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

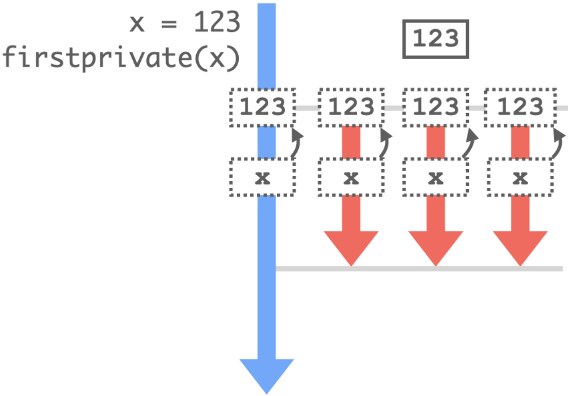
The `firstprivate` clause



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`.
- 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 `firstprivate` clause

```c
int a = 123;
#pragma omp parallel default(none)
{
  // Compiler complains because variable "a" is not
    specified.
  a = 456;
}
```

```fortran
INTEGER :: a := 123
!$OMP PARALLEL DEFAULT(none)
  ! Compiler complains because variable "a" is not
    specified.
  a = 456
!$OMP END PARALLEL
```

Time to practice: `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`

Table of Contents

1 Motivation

2 How it works

3 The parallel construct

4 Data-sharing attribute clauses

5 Worksharing construct

6 Synchronisation constructs

7 Loops time

8 Reduction

9 Schedule clause

10 Summary

Worksharing constructs

- `single`
- `master`[2]

[2] 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` thread is the thread that runs the program and is present outside `parallel` constructs.

The `master` construct - Example

```c
#pragma omp parallel
{
  // All threads
  #pragma omp master
  {
    // Only master thread
  }
  // All threads
}
```

```fortran
!$OMP PARALLEL
  ! All threads
  !$OMP MASTER
    ! Only master thread
  !$OMP END MASTER
  ! All threads
!$OMP END PARALLEL
```

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 practice: `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

1 Motivation

2 How it works

3 The parallel construct

4 Data-sharing attribute clauses

5 Worksharing construct

6 Synchronisation constructs

7 Loops time

8 Reduction

9 Schedule clause

10 Summary

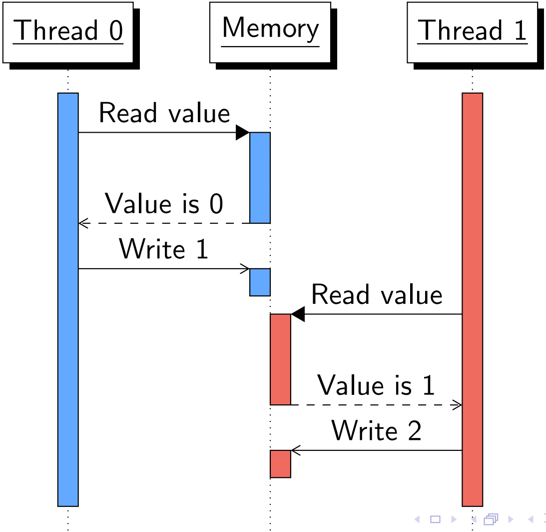
Synchronisation constructs

```
int thread_count = 0;
#pragma omp parallel default(none) shared(thread_count)
{
    thread_count++;
    #pragma omp single
    {
        printf("There are %d threads.\n", thread_count);
    }
}
```

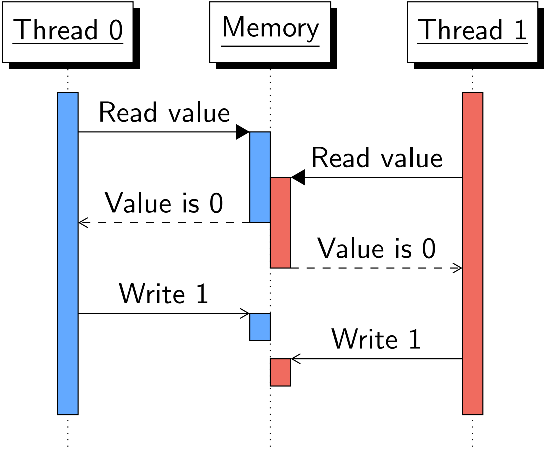
Synchronisation constructs

```fortran
INTEGER :: thread_count := 0
!$OMP PARALLEL DEFAULT(NONE) SHARED(thread_count)
    thread_count = thread_count + 1
    !$OMP SINGLE
        WRITE(*,'A,I0,A') 'There are ', thread_count, '
    threads.'
    !$OMP END SINGLE
!$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.
- 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.
- When several `critical` constructs have the same name, only one thread will be any of these `critical` construct at any one time.
- If a name is not provided, a default (unspecified) name is used.
- In other words, all unnamed `critical` constructs 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 practice: `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

1 Motivation
2 How it works
3 The parallel construct
4 Data-sharing attribute clauses
5 Worksharing construct
6 Synchronisation constructs
7 Loops time
8 Reduction
9 Schedule clause
10 Summary

Time to parallelise loops!

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

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

Time to parallelise loops!

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

```fortran
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

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

What we actually asked for

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

The `for` / `do` constructs

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

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

The for / do constructs

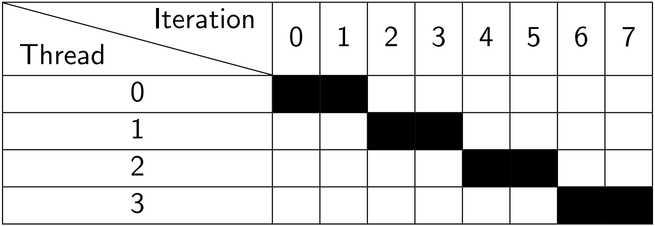
```c
#pragma omp parallel
{
  #pragma omp for
  for(int i=0; i<8; i++)
  {
    a[i] = b[i] + c[i];
  }
}
```

```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

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

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

Time to conquer the world.

```c
int total = 0;
for(int i=0; i<8; i++)
{
    total++;
}
```

```fortran
INTEGER :: total=0
INTEGER :: i=0
DO i=1,8
    total = total + 1
END DO
```

Time to conquer the world.

```c
int total = 0;
#pragma omp parallel for
for(int i=0; i<8; i++)
{
    total++;
}
```

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

Table of Contents

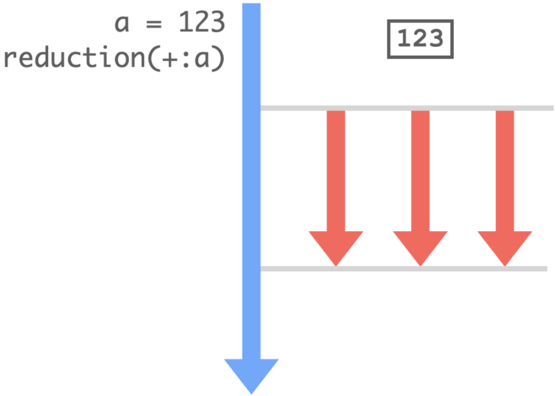
1 Motivation
2 How it works
3 The parallel construct
4 Data-sharing attribute clauses
5 Worksharing construct
6 Synchronisation constructs
7 Loops time
8 Reduction
9 Schedule clause
10 Summary

The `reduction` clause

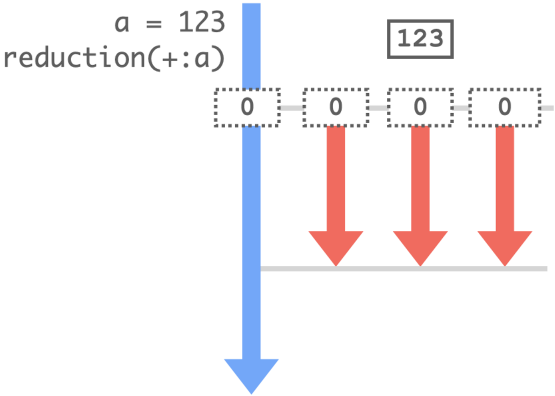
- The `reduction` clause accepts two arguments:
  - The reduction operation
  - The reduced variable
- When encountering a `reduction` clause, each thread makes its own local copy of the reduced variable, before reducing them back into the original variable at the end of the `reduction` region.

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

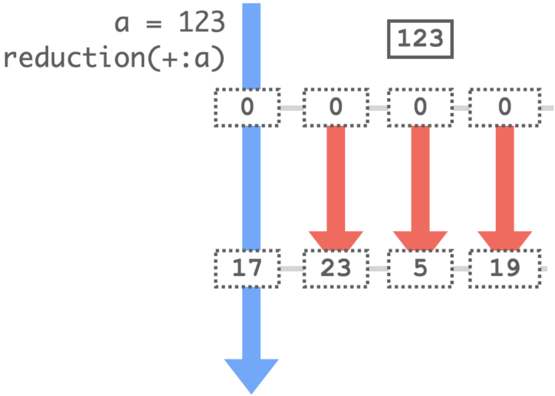
The `reduction` clause



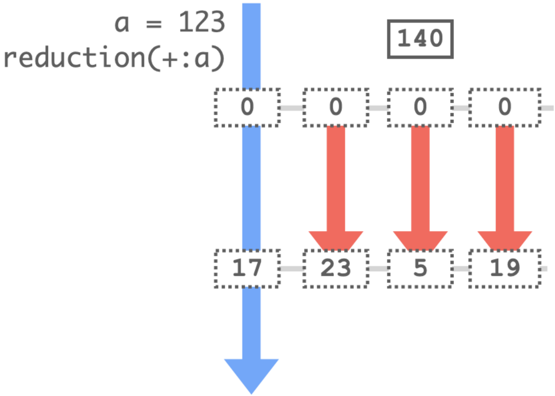
The `reduction` clause



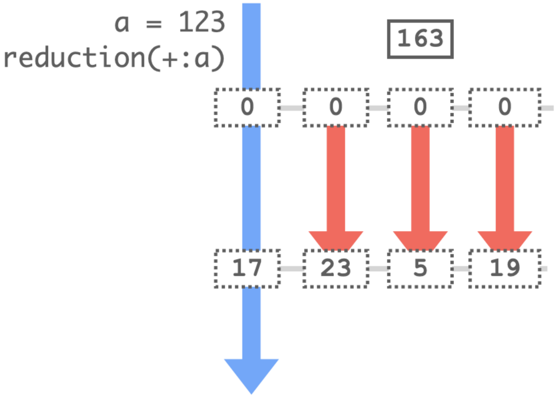
The `reduction` clause



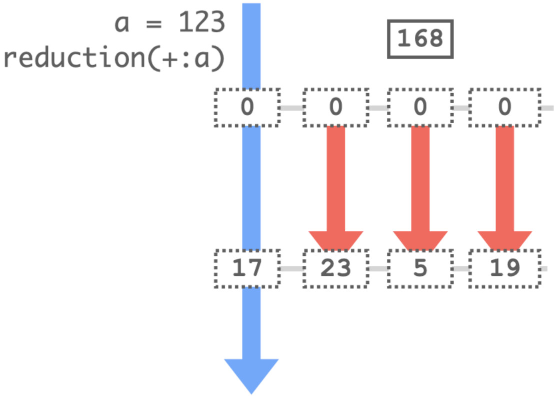
The `reduction` clause



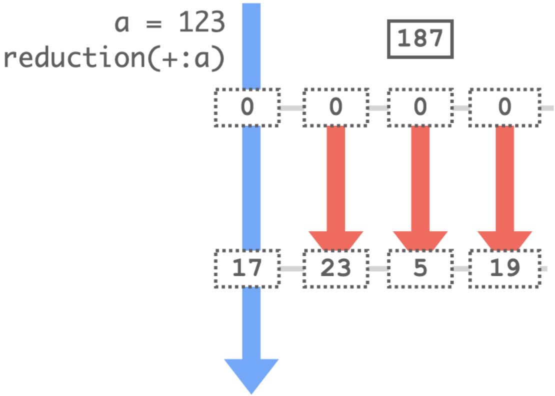
The `reduction` clause



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

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

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

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

The `atomic` construct

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

```
#pragma omp parallel reduction(+:a)
{
    a++;
    #pragma omp barrier
    // The value of a is not updated yet by the reduction since the execution has not reached the end of the parallel construct.
}
```

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 practice: `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

1 Motivation
2 How it works
3 The parallel construct
4 Data-sharing attribute clauses
5 Worksharing construct
6 Synchronisation constructs
7 Loops time
8 Reduction
9 Schedule clause
10 Summary

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

Scheduling kinds: `static`

- How to use: `schedule(static, chunksize)`
- Packs iterations in chunks of `<chunksize>` consecutive iterations
- The chunksize is optional, defaults to 1.
- Distributes the iterations on a cyclic pattern.
- Every thread knows in advance every iteration it will process in the entire iteration set.
- Has a low overhead.
- Although implementation-specific, usually, the `static` scheduling kind is the default.

`schedule(static,1)`

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

`schedule(static,2)`

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

schedule(static,3)

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

Scheduling kinds: `static`

- Illustration of load-imbalance where static performs poorly.
- Different workload per iteration.
- Can always try static 1, problem becomes the data locality we just lost.

Scheduling kinds: `dynamic`

- How to use: `schedule(dynamic, chunksize)`
- Packs iterations in chunks of `<chunksize>` consecutive iterations.
- The chunksize is optional, defaults to 1.
- Distributes the first `n` chunks to the `n` threads.
- Any other chunk will be served on a first-come-first-served basis.
- Provides load-balancing feature.
- Has an overhead greater than `static` since it needs to have threads coordinate and synchronise to know who takes the next chunk.

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)

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,2)

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

Scheduling kinds: `guided`

- How to use: `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>`[3].
- The chunksize is optional, defaults to 1.
- Like dynamic, rest of chunks served on first-come-first-served basis.
- Allows more efficient processing of decreasing workloads (upper triangular matrices etc...)
- Has an overhead greater than `static` since it needs to have threads coordinate and synchronise to know who takes the next chunk.

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

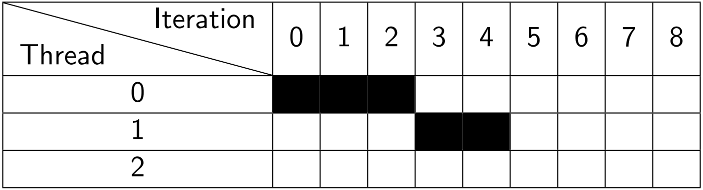
schedule(guided,2)

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)

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						■	■		

Scheduling kinds: `auto`

- How to use: `schedule(auto)`
- Schedule applied is implementation defined.

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

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

```fortran
PROCEDURE omp_set_schedule(kind, chunk_size)
INTEGER(KIND=omp_sched_kind) :: kind
INTEGER :: chunk_size
```

Time to practice: `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

1 Motivation
2 How it works
3 The parallel construct
4 Data-sharing attribute clauses
5 Worksharing construct
6 Synchronisation constructs
7 Loops time
8 Reduction
9 Schedule clause
10 Summary

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.

Your opinion matters

Figure: Link to surveys