# OpenMP
# Shared-Memory Parallel Programming
# Part. I

Orian Louant
orian.louant@uliege.be

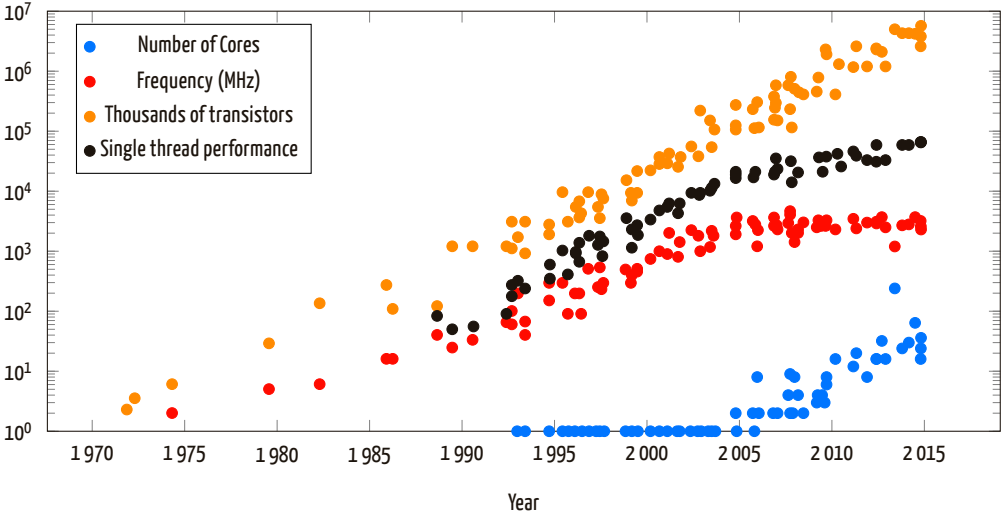
September 29, 2020

# What is OpenMP?

OpenMP is a shared-memory application programming interface which by adding directives to a sequential program describes how the work is shared among threads and order accesses to shared data.

OpenMP hides the low-level details and allows the programmer to describe the parallel code with high-level constructs.

# Motivations for OpenMP



# Motivations for OpenMP

- In the years 2000's the CPU manufacturers have run out of room for boosting CPU performance.
- Instead of driving clock speeds and straight-line instruction throughput higher, they turn to hyperthreading and multicore architectures.

# Motivations for OpenMP

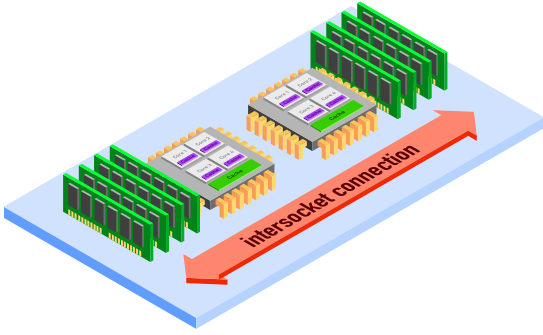
- The need for a parallel programming model for shared memory system then became necessary.
- Fortunately, OpenMP was there with the first release of the specification in 1997 (Fortran) and 1998 (C/C++).

# Motivations for OpenMP

- The need for a parallel programming model for shared memory system then became necessary.
- Fortunately, OpenMP was there with the first release of the specification in 1997 (Fortran) and 1998 (C/C++).

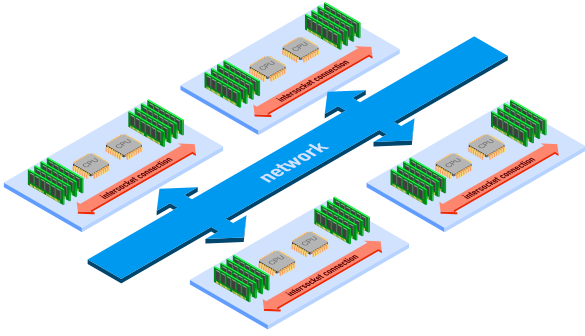
Note: Moore's law is not entirely dead! The number of transistors is still increasing but slower (double every 2.5 years instead of every 2 years)

# Shared-Memory



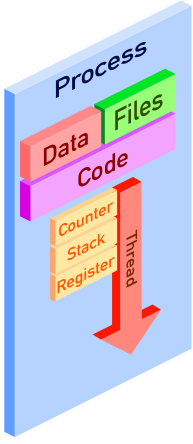

- At least one multi-core CPU
- All CPUs can access a single memory address space
- Systems memory may be physically distributed, but logically shared

# Distributed-Memory



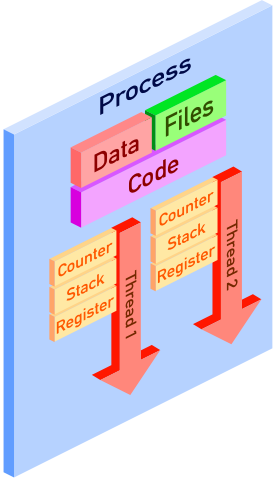

- Multiple nodes
- Interconnected by a high-speed network
- Nodes consist of (a) processor(s) and local memory
- Communication is done via message passing

# Process and threads




- A process is an instance of an application
- A process is executed by at least one thread
- A process is a container describing the state of an application: code, memory mapping, shared libraries, ...

# Process and threads




- A thread is an independent stream of instructions that can be scheduled to run by the operating system.
- Multiple threads can exist within one process, and they share the memory
- A thread only the owns the bare essential resources to exist as executable code: execution counter, stack pointer, registers and thread-specific data

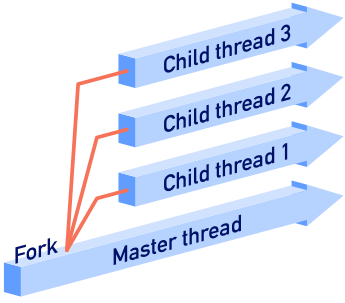
# Why using threads?

The operating system does not need to create a new memory map for a new thread. It increases efficiency on multiprocessor systems.

Shared memory makes it trivial to share data among threads (with potential drawbacks though).

# Fork-Join

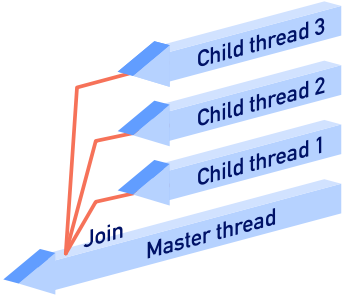
OpenMP use the fork-join model




- The master thread continues after the fork operation
- The children threads begin operation separate from the master thread
- Parallel execution begins

# Fork-Join

OpenMP use the fork join model




- Children join after they finish
- The master thread waits until all the children join
- Join ends parallel execution. Sequential execution of the master thread continues

# Fork-Join

In practice, threads are not created or destroyed for every parallel region.

OpenMP implementations use a thread pool to avoid the cost of threads creation and destruction at each fork and join.

After the join, the children threads go idle.

# OpenMP is using directives

Directives are programming languages constructs that specifies how a compiler should process its input

An OpenMP program is the combination of

- a base language (C, C++ or Fortran)
- annotations with OpenMP directives

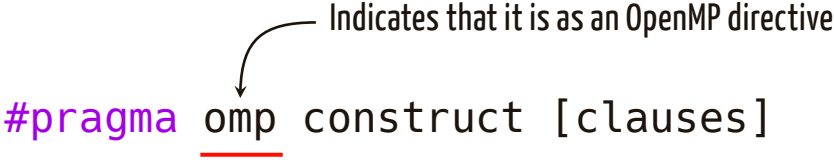
# Anatomy of an OpenMP directive

```
#pragma omp construct [clauses]
```

# Anatomy of an OpenMP directive

```
#pragma omp construct [clauses]
```

# Anatomy of an OpenMP directive



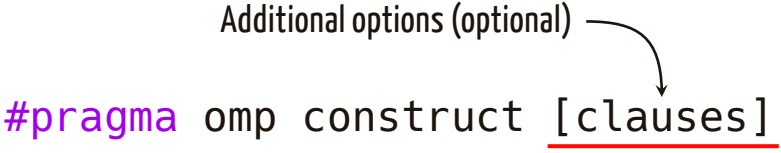
# Anatomy of an OpenMP directive

```
#pragma omp construct [clauses]
```



Give instruction on what to do

# Anatomy of an OpenMP directive



# What the directives do

An OpenMP construct can specify

- the creation of a parallel region
- how to parallelize loops
- whether the variables in the parallel region are private or shared
- how/if the threads are synchronized
- how the work is divided between threads

# The Advantages of Using directive

- Does not modify the serial implementation
- You can still compile and run the program as a serial code.
- Can be added incrementally allowing a gradual parallelization
- Easier to maintain

# Hard work is hidden

Directives hide the actual parallelization work from the programmer

The compiler replaces the directives by the appropriate calls to the OpenMP runtime and library

# Going Parallel

# The Parallel Construct

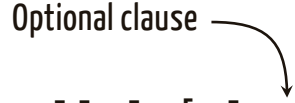
```
#pragma omp parallel [clauses]
  structured-block
```

# The Parallel Construct

Creates a parallel region by spawning a team of threads

```
#pragma omp parallel [clauses]
  structured-block
```

# The Parallel Construct




```
#pragma omp parallel [clauses]
  structured-block
```

# The Parallel Construct

```
#pragma omp parallel [clauses]
  structured-block
```

Block of code

# OpenMP Hello World

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {

  #pragma omp parallel
  {
    printf("Hello, I'm thread %d of %d.\n",
      omp_get_thread_num(),
      omp_get_num_threads());
  }

  return 0;
}
```

# OpenMP Hello World

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {

  #pragma omp parallel
  {
    printf("Hello, I'm thread %d of %d.\n",
      omp_get_thread_num(),
      omp_get_num_threads());
  }

  return 0;
}
```

to gain access to the OpenMP runtime functions

# OpenMP Hello World

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {

  #pragma omp parallel
  {
    printf("Hello, I'm thread %d of %d.\n",
      omp_get_thread_num(),
      omp_get_num_threads());
  }

  return 0;
}
```

create a team of threads

# OpenMP Hello World

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {

  #pragma omp parallel
  {
    printf("Hello, I'm thread %d of %d.\n",
      omp_get_thread_num(),
      omp_get_num_threads());
  }

  return 0;
}
```

get the thread number in the team. Master thread will be 0 and children > 0

# OpenMP Hello World

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {

  #pragma omp parallel
  {
    printf("Hello, I'm thread %d of %d.\n",
      omp_get_thread_num(),
      omp_get_num_threads());
  }

  return 0;
}
```

get the number of threads in the current team

# Compiling the OpenMP Hello World

To compile an OpenMP program, you need to pass a specific flag to the compiler

| | | |
|---|---|---|
| GCC: | gcc | -fopenmp |
| Clang: | clang | -fopenmp |
| Intel: | icc | -qopenmp |

This flag instructs the compiler to consider OpenMP directives

# Compiling the OpenMP Hello World

Compilers with OpenMP are available for all CÉCI clusters.
For example, the folowing modules are available:

```
$ module load <module_name>
```

| | | |
|---|---|---|
| Lemaitre3 | GCC/8.3.0 | intel/2019b |
| Hercules2 | GCC/7.1.0-2.28 | intel/2016b |
| Dragon2 | GCC/8.2.0-2.31.1 | intel/2018b |
| Vega | GCC/9.3.0 | intel/2019b |

# Executing the OpenMP Hello World

```
 $ gcc -fopenmp -o example omp_helloworld.c
 $ OMP_NUM_THREADS=4 ./example
Hello, I'm thread 1 of 4.
Hello, I'm thread 2 of 4.
Hello, I'm thread 3 of 4.
Hello, I'm thread 0 of 4.
```

# Executing the OpenMP Hello World

The `OMP_NUM_THREADS` environment variable allows you to specify the number of threads

```
$ export OMP_NUM_THREADS=4
$ ./example

$ OMP_NUM_THREADS=4 ./example
```

# Executing the OpenMP Hello World

The `OMP_NUM_THREADS` environment variable allows you to specify the number of threads

```
$ export OMP_NUM_THREADS=4
$ ./example
```

4 threads for the duration of the session

```
$ OMP_NUM_THREADS=4 ./example
```

# Executing the OpenMP Hello World

The `OMP_NUM_THREADS` environment variable allows you to specify the number of threads

```
$ export OMP_NUM_THREADS=4
$ ./example

$ OMP_NUM_THREADS=4 ./example
```

← 4 threads for this execution of the program

# Making Things Go Parallel

```
int max_threads = omp_get_max_threads();
int iterations[max_threads];

for(int i = 0; i < max_threads; ++i) iterations[i] = 0;

#pragma omp parallel
{
    for(int i = 0; i < 1000; ++i)
      iterations[omp_get_thread_num()]++;
}

for(int i = 0; i < max_threads; ++i)
  printf("Number of iteration for thread %d: %d\n",
         i, iterations[i]);
```

# Making Things Go Parallel

```
 $ gcc -fopenmp -o example omp_iterations.c
 $ OMP_NUM_THREADS=4 ./example
Number of iteration for thread 0: 1000
Number of iteration for thread 1: 1000
Number of iteration for thread 2: 1000
Number of iteration for thread 3: 1000
```

# Parallel $\neq$ Worksharing

In the previous example, the code in the parallel construct is executed redundantly by each thread.

- Create a team of threads, i.e. fork
- The threads in the team join at the end of the region
- By itself, the parallel construct does not imply any worksharing

# Parallel $\neq$ Worksharing

Executing the same iterations redundantly is not very useful and will certainly not help to reduce the execution time.

We need to share iterations between threads

# Distributing iterations

```
int n = 1000;
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  int nthreads = omp_get_num_threads();

  int low = tid * (n / nthreads);
  int high = (tid + 1) * (n / nthreads);

  for(int i = low; i < MIN(high, n); ++i)
    iterations[tid]++;
}
```

# Distributing iterations

```
int n = 1000;
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  int nthreads = omp_get_num_threads();

  int low = tid * (n / nthreads);
  int high = (tid + 1) * (n / nthreads);

  for(int i = low; i < MIN(high, n); ++i)
    iterations[tid]++;
}
```

lower bound of the iterations for the thread

# Distributing iterations

```
int n = 1000;
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  int nthreads = omp_get_num_threads();

  int low = tid * (n / nthreads);
  int high = (tid + 1) * (n / nthreads);

  for(int i = low; i < MIN(high, n); ++i)
    iterations[tid]++;
}
```

higher bound of the iterations for the thread

# Distributing iterations

```
int n = 1000;
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  int nthreads = omp_get_num_threads();

  int low = tid * (n / nthreads);
  int high = (tid + 1) * (n / nthreads);

  for(int i = low; i < MIN(high, n); ++i)
    iterations[tid]++;
}
```

iterations between the lower and higher bounds

# Distributing iterations

```
 $ gcc -fopenmp -o example omp_iterations.c
 $ OMP_NUM_THREADS=4 ./example
Number of iteration for thread 0: 250
Number of iteration for thread 1: 250
Number of iteration for thread 2: 250
Number of iteration for thread 3: 250
```

# Distributing iterations with a directive

Instead of computing the bounds, we can use the `for` directive.

```
int n = 1000;
#pragma omp parallel
{
  int tid = omp_get_thread_num();

  #pragma omp for
  for(int i = 0; i < n; ++i)
    iterations[tid]++;
}
```

# Distributing iterations with a directive

Instead of computing the bounds, we can use the `for` directive.

```
int n = 1000;
#pragma omp parallel
{
  int tid = omp_get_thread_num();

  #pragma omp for
  for(int i = 0; i < n; ++i)
    iterations[tid]++;
}
```

instructs to distribute the iterations between the threads in the team

# Distributing iterations with a directive

```
 $ gcc -fopenmp -o example omp_for_iters.c
 $ OMP_NUM_THREADS=4 ./example
Number of iteration for thread 0: 250
Number of iteration for thread 1: 250
Number of iteration for thread 2: 250
Number of iteration for thread 3: 250
```

# The Canonical for-loop

The for-loop needs to be in canonical form to be used with the `for` directive

```
#pragma omp for
for ([inttype] var = start; var <  end; ++var
                                <=      var++
                                >       var += incr
                                >=      var = var + incr
                                        var--, ...)
```

- `var`, can not be modified in the loop body. It must be an integer (signed or unsigned), a pointer or random access iterator type
- `start`, `end` and `incr` must be loop invariant expressions, the number of iterations must be computable when the loop begins

# Summary so far

- A `parallel` directive creates a team of threads
- A `for` directive instructs to distribute the iterations of the associated loop between the threads in the team
- The `omp_get_num_threads()` function returns the number of threads in the team
- The `omp_get_thread_num()` function returns the number of the thread within the team

# Data Sharing in a Parallel World

# Hello Again

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {
  int nthreads, tid;

  #pragma omp parallel
  {
    tid = omp_get_thread_num();
    nthreads = omp_get_num_threads();

    printf("Hello, I'm thread %d of %d.\n", tid, nthreads);
  }

  return 0;
}
```

# Hello Again

Most of the time, the program output is what is expected but, ...

```
 $ OMP_NUM_THREADS=4 ./example
Hello, I'm thread 2 of 4.
Hello, I'm thread 2 of 4.
Hello, I'm thread 2 of 4.
Hello, I'm thread 3 of 4.
```

... occasionally, we have imposters pretending to be thread 2.

# What's wrong?

All variables declared outside of the scope of an OpenMP `parallel` construct is shared by all threads by default.

```
int nthreads, tid;
#pragma omp parallel
{
  tid = omp_get_thread_num();
  nthreads = omp_get_num_threads();

  printf("Hello, I'm thread %d of %d.\n",
         tid, nthreads);
}
```

# What's wrong?

All variables declared outside of the scope of an OpenMP `parallel` construct is shared by all threads by default.

declaration outside of the OpenMP construct, theses variables are shared by all threads

```
int nthreads, tid;
#pragma omp parallel
{
  tid = omp_get_thread_num();
  nthreads = omp_get_num_threads();

  printf("Hello, I'm thread %d of %d.\n",
         tid, nthreads);
}
```

# What's wrong?

All variables declared outside of the scope of an OpenMP `parallel` construct is shared by all threads by default.

```
int nthreads, tid;
#pragma omp parallel
{
  tid = omp_get_thread_num(); <------ all threads write to the same location
  nthreads = omp_get_num_threads();

  printf("Hello, I'm thread %d of %d.\n",
         tid, nthreads);
}
```

# What's wrong?

All variables declared outside of the scope of an OpenMP `parallel` construct is shared by all threads by default.

```
int nthreads, tid;
#pragma omp parallel
{
  tid = omp_get_thread_num();
  nthreads = omp_get_num_threads();

  printf("Hello, I'm thread %d of %d.\n",
         tid, nthreads);
}
```

the same here, but, as the value is the same for all threads, it's less likely to go wrong

# What's wrong?

All variables declared outside of the scope of an OpenMP `parallel` construct is shared by all threads by default.

```
int nthreads, tid;
#pragma omp parallel
{
  tid = omp_get_thread_num();
  nthreads = omp_get_num_threads();

  printf("Hello, I'm thread %d of %d.\n",
         tid, nthreads);
}
```

when the values are used here, another thread may have modified them

# What's wrong?

All variables declared outside of the scope of an OpenMP `parallel` construct is shared by all threads by default.

```
int nthreads, tid;
#pragma omp parallel
{
  tid = omp_get_thread_num();
  nthreads = omp_get_num_threads();

  printf("Hello, I'm thread %d of %d.\n",
         tid, nthreads);
}
```

We created a data race

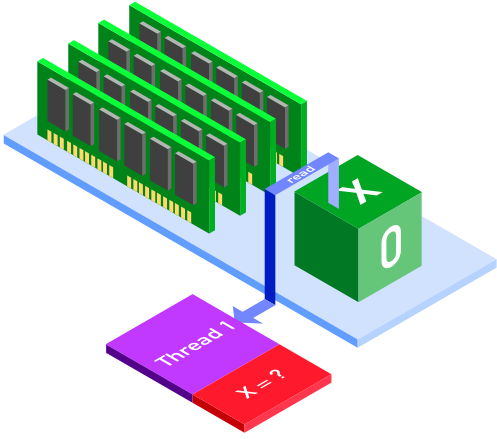
# Data Race

A data race is when one or more threads concurrently access a location in memory or a variable, at least one of which is a write.

As an example, we will consider this simple construct

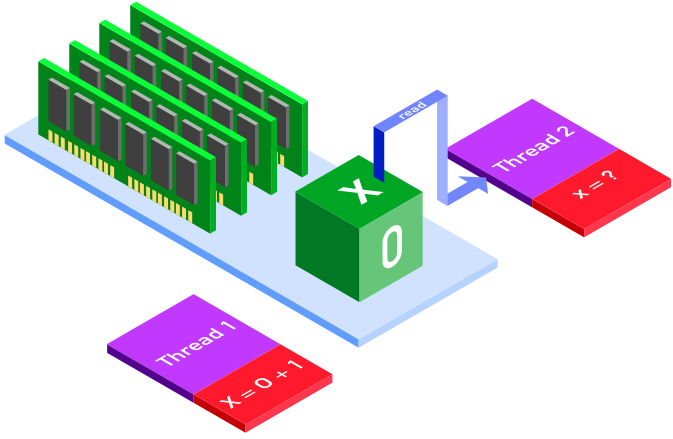
```
int x = 0;
#pragma omp parallel
{
  x = x + 1;
}
```

# Data Race



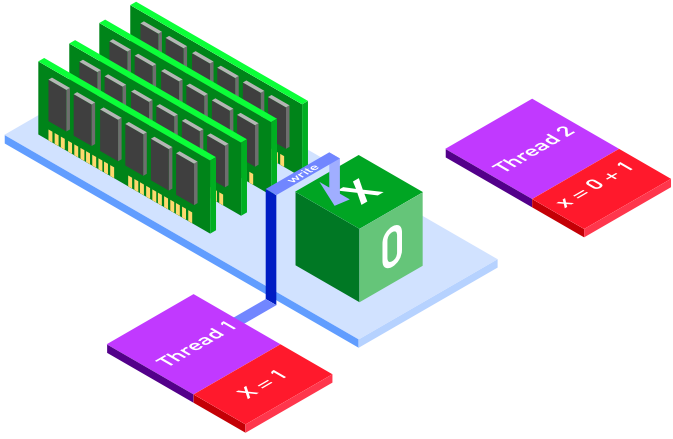

- Thread 1 fetch x from memory and store its value in a register

# Data Race



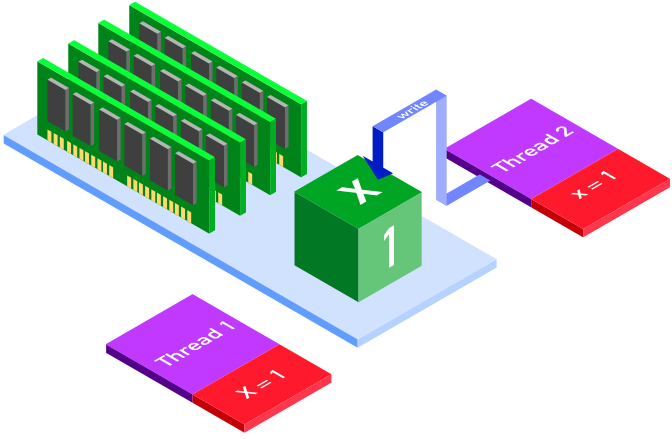

- Thread 1 add one to the value stored in the register
- Thread 2 fetch x from memory and store its value in a register

# Data Race




- Thread 1 store the result of the addition back in the shared memory
- Thread 2 add one to the value stored in the register

# Data Race




- Thread 2 store the result of the addition back in the shared memory

# Data Race

Because of the potential data races in shared-memory parallel programs, extra care is needed as this is not always easy to spot

- with floating-point data, it may be difficult to distinguish from a numerical side-effect
- changing the number of threads can cause the problem to seemingly (dis)appear
- may depend on the load on the system
- may only show up using many threads

# Data Race

In the previous example, we executed `x+1` two times and get `1` as a result (while `2` was expected)

We need a way to prevent data race from happening: only share data that are not modified by other threads.

# Hello Again (Data Race Free)

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {
  int nthreads, tid;

  #pragma omp parallel private(nthreads, tid)
  {
    tid = omp_get_thread_num();
    nthreads = omp_get_num_threads();

    printf("Hello, I'm thread %d of %d.\n", tid, nthreads);
  }

  return 0;
}
```

# Hello Again (Data Race Free)

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {
  int nthreads, tid;

  #pragma omp parallel private(nthreads, tid)
  {
    tid = omp_get_thread_num();
    nthreads = omp_get_num_threads();

    printf("Hello, I'm thread %d of %d.\n", tid, nthreads);
  }

  return 0;
}
```

we added the `private` clause which declares one or more list items to be private to the thread

# Data-Sharing Attributes

```
private( list )
```

The parallel construct can take one or more data-sharing clause. The first one is `private`, which instruct that each thread should have its own instance of the listed variables. The initial value when we enter the parallel region is undefined.

# Data-Sharing Attributes

```
firstprivate( list )
```

If the value of the variable before entering the parallel region matters, we can use `firstprivate` which is the same as `private` but, the variable should be initialised with its value before the `parallel` construct.

# Data-Sharing Attributes

```
shared( list )
```

The third option is to declare a variable as `shared` which indicates that the variables listed should be shared among all threads. This is the default.

# Data-Sharing Attributes

```
default( shared | none )
```

You can change the default data-sharing attribute with the `default` clause. Setting the value to `none` will force you to specify the data-sharing attribute for all your .

# Data-Sharing Attributes Example

```
int x = 1, y = 2, z = 3, a = 4;
#pragma omp parallel private(x) firstprivate(y) shared(z)
{
    x = x + z;
    y = y + z;
    a = a + 1;

    printf("Thread %d: x = %d, y = %d, z = %d\n",
            omp_get_thread_num(), x, y, z);
}

printf("Final: x = %d, y = %d, z = %d, a = %d\n", x, y, z, a);
```

# Data-Sharing Attributes Example

```
 $ OMP_NUM_THREADS=4 ./example
Thread 0: x = 3, y = 5, z = 3
Thread 1: x = 32674, y = 5, z = 3
Thread 3: x = 32674, y = 5, z = 3
Thread 2: x = 32674, y = 5, z = 3
Final: x = 1, y = 2, z = 3
```

# Data-Sharing Attributes Example

```
 $ OMP_NUM_THREADS=4 ./example
Thread 0: x = 4, y = 5, z = 3
Thread 1: x = 32674, y = 5, z = 3
Thread 3: x = 32674, y = 5, z = 3
Thread 2: x = 32674, y = 5, z = 3
Final: x = 1, y = 2, z = 3, a = 7
```

The value of `x` is wrong because using the `private` clause the value of the variable is undefined when entering the `parallel` construct.

# Data-Sharing Attributes Example

```
 $ OMP_NUM_THREADS=4 ./example
Thread 0: x = 4, y = 5, z = 3
Thread 1: x = 32674, y = 5, z = 3
Thread 3: x = 32674, y = 5, z = 3
Thread 2: x = 32674, y = 5, z = 3
Final: x = 1, y = 2, z = 3, a = 7
```

The value of `y`, on the other hand, is correct as the `firstprivate` clause sets the initial value in the `parallel` construct to be the value before entering the construct.

# Data-Sharing Attributes Example

```
 $ OMP_NUM_THREADS=4 ./example
Thread 0: x = 4, y = 5, z = 3
Thread 1: x = 32674, y = 5, z = 3
Thread 3: x = 32674, y = 5, z = 3
Thread 2: x = 32674, y = 5, z = 3
Final: x = 1, y = 2, z = 3, a = 7
```

After the `parallel` region, the values of the private variables are the same as before the region.

# Data-Sharing Attributes Example

```
 $ OMP_NUM_THREADS=4 ./example
Thread 0: x = 4, y = 5, z = 3
Thread 1: x = 32674, y = 5, z = 3
Thread 3: x = 32674, y = 5, z = 3
Thread 2: x = 32674, y = 5, z = 3
Final: x = 1, y = 2, z = 3, a = 7
```

The value of `z` never changes. This variable is shared but never modified.

# Data-Sharing Attributes Example

```
 $ OMP_NUM_THREADS=4 ./example
Thread 0: x = 4, y = 5, z = 3
Thread 1: x = 32674, y = 5, z = 3
Thread 3: x = 32674, y = 5, z = 3
Thread 2: x = 32674, y = 5, z = 3
Final: x = 1, y = 2, z = 3, a = 7
```

We did not specify any data-sharing attribute for a. Thus, this variable has the default attribute and is shared. We see that there is a data race problem as we expected the final value to be 8 with 4 threads.

# Data-Sharing Attribute Rules

Variables with automatic storage duration that are declared in a scope inside the construct are private.

```
#pragma omp parallel
{
  int a = 3; // private
}
```

# Data-Sharing Attribute Rules

Variables with static storage duration that are declared in a scope inside the construct are shared.

```
#pragma omp parallel
{
  static int a; // shared
}
```

# Data-Sharing Attribute Rules

The loop iteration variable(s) in the associated for-loop(s) of a `for` construct is (are) private.

```
int i;
#pragma omp parallel
{
  #pragma omp for
  for(i = 0; i < n; ++i) // i is private
  {
    // ...
  }
}
```

# Data-Sharing Attribute Rules

Objects with dynamic storage duration are shared (allocated by malloc).

```
int* a = (int*)malloc(n * sizeof(int));

#pragma omp parallel
{
  // the array a can not be privatized
  // all threads can read and write the
  // whole array
  a[0] = 3;
}
```

# Good Practices

Set the default data attribute to `none` with `default(none)`.

- You will get a compiler error if you do not explicitly specify the data attribute of your variables
- It forces you to think about the data attribute of your variables

# Loop Carried Data Dependency

The fact that dynamically allocated objects can no be private implies that particular care must be taken when handling arrays.

```
int* a = (int*)malloc(n * sizeof(int));

#pragma omp parallel
{
  #pragma omp for
  for(int i = 0; i < (n-1); ++i)
    a[i] = a[i+1] + b[i];
}
```

# Loop Carried Data Dependency

The fact that dynamically allocated objects can no be private implies that particular care must be taken when handling arrays.

```
int* a = (int*)malloc(n * sizeof(int));

#pragma omp parallel
{
  #pragma omp for
  for(int i = 0; i < (n-1); ++i)
    a[i] = a[i+1] + b[i];
}
```

What if an other thread modify this?

# Loop Carried Data Dependency

A **loop carried data dependency** occurs when a value written in one loop iteration is read or written by another iteration.

| Thread 1 | Thread 2 |
|---|---|
| `a[0] = a[1] + b[0]` | `a[4] = a[5] + b[4]` |
| `a[1] = a[2] + b[1]` | `a[5] = a[2] + b[5]` |
| `a[2] = a[3] + b[2]` | `a[6] = a[3] + b[6]` |
| `a[3] = a[4] + b[3]` | `a[7] = a[4] + b[7]` |

# Loop Carried Data Dependency

A quick test to check whether your loop iterations are independent is to run the loop in reverse order.

```
for(int i = n-2; i >= 0; --i)
  a[i] = a[i+1] + b[i];

int check = 0;
for(int i = 0; i < n; ++i)
  check += a[i];

printf("%d\n", check);
```

Not infallible, but there is few counterexamples.

That’s all for today!

# Next Time

- Synchronization
- Loop scheduling
- Nested and orphaned parallelism
- Performance considerations