OpenMP

Agenda

What is OpenMP I

- OpenMP (Open Multi-Processing) is an application programming interface
- multi-platform shared memory multiprocessing
- support for C, C++ and Fortran on many architectures and operating systems

What is OpenMP I

- OpenMP (Open Multi-Processing) is an application programming interface
- multi-platform shared memory multiprocessing
- support for C, C++ and Fortran on many architectures and operating systems

What is OpenMP I

- OpenMP (Open Multi-Processing) is an application programming interface
- multi-platform shared memory multiprocessing
- support for C, C++ and Fortran on many architectures and operating systems

What is OpenMP II

OpenMP consists of

- compiler directives,
- runtime routines,
- environment variables

What is OpenMP II

OpenMP consists of

- compiler directives,
- runtime routines,
- environment variables

What is OpenMP II

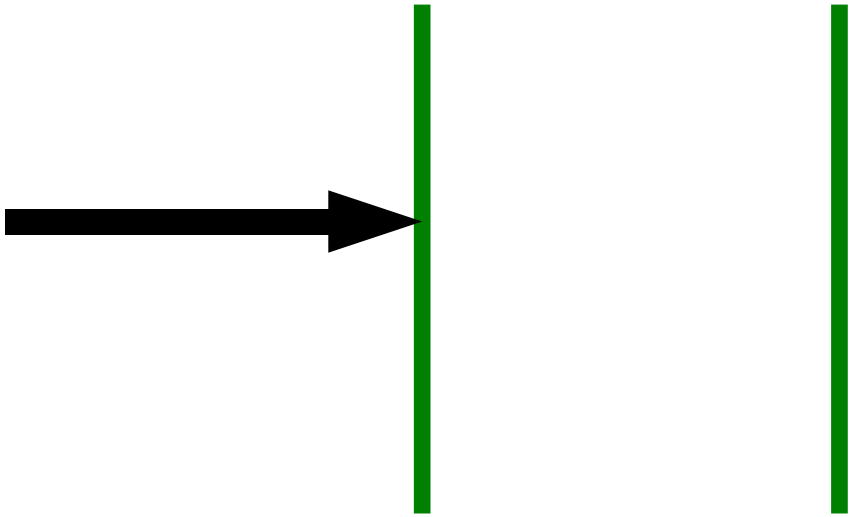
OpenMP consists of

- compiler directives,
- runtime routines,
- environment variables

OpenMP Execution Model I

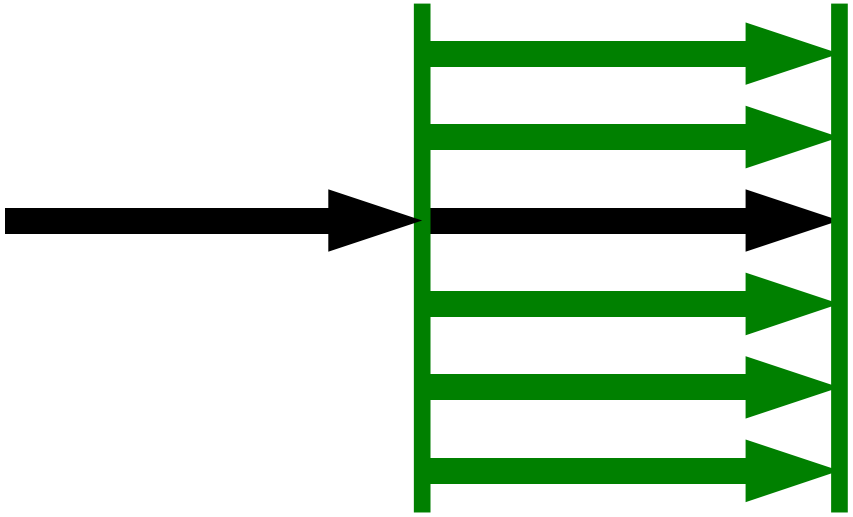
OpenMP understands the world as single threaded...

OpenMP Execution Model II



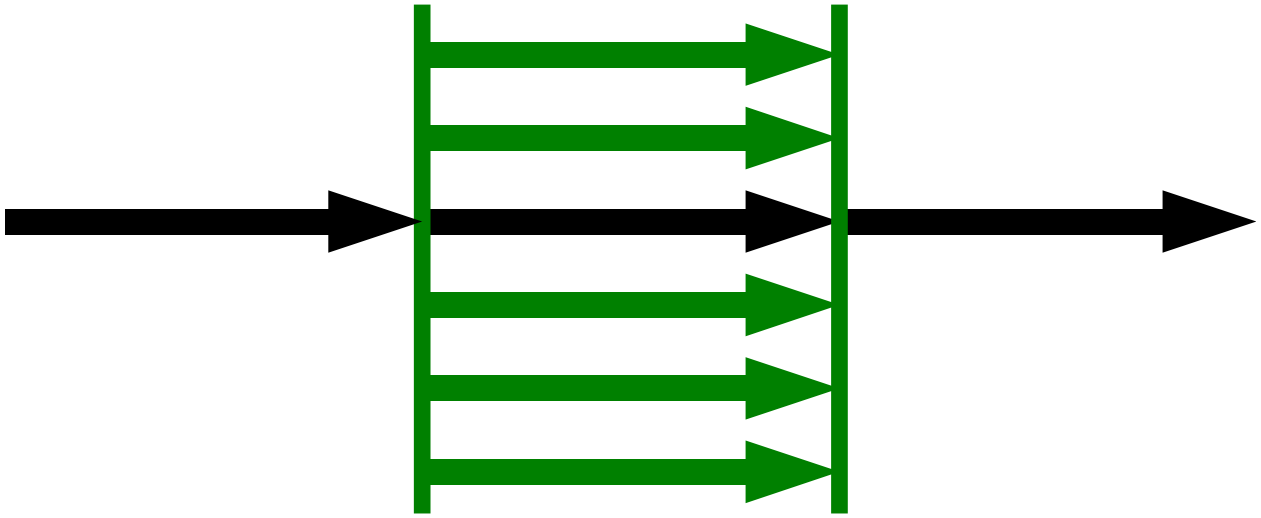
...until this thread hits a region marked for parallel processing...

OpenMP Execution Model III



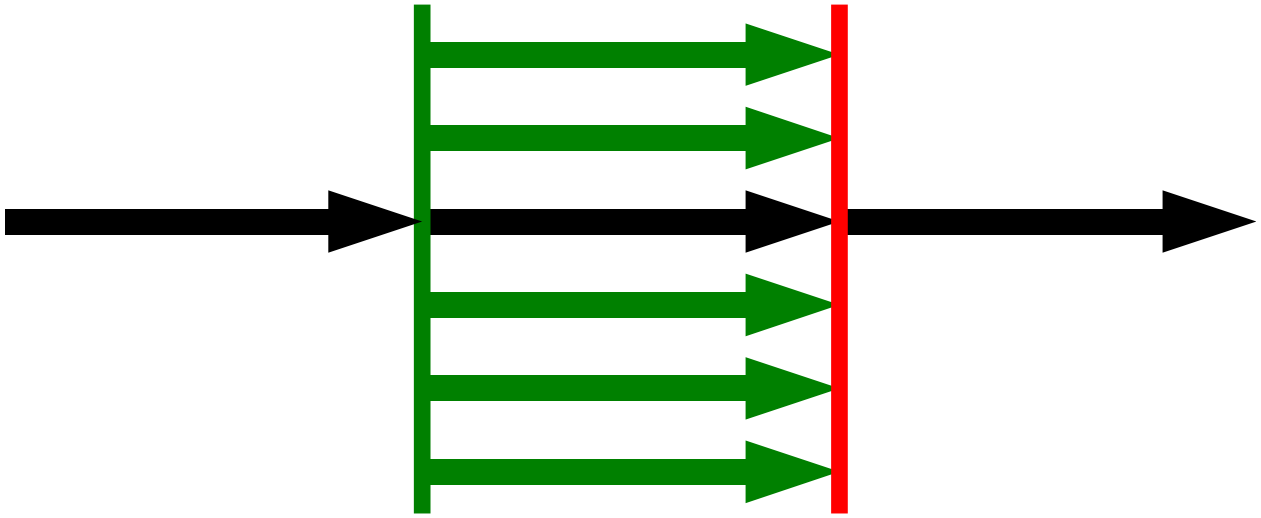
...at which point OpenMP creates slave threads
to do the dirty work...

OpenMP Execution Model IIII



...and after that region reverts to
the single threaded world...

OpenMP Execution Model IIIII



...which of course requires a synchronisation barrier.
This is called a fork-join model.

OpenMP: How does it look like?

```
main() {
  #pragma omp parallel
  {
    printf("Hello world");
  }
}
```

- The #pragma introduces the parallel region
- Forking, Joining, Synchronising is done implicitly
- Compilation: cc -O2 -parallel -omp code.c

OpenMP: How does it look like?

```
main() {
  #pragma omp parallel
  {
    printf("Hello world");
  }
}
```

- The #pragma introduces the parallel region
- Forking, Joining, Synchronising is done implicitly
- Compilation: cc -O2 -parallel -omp code.c

OpenMP: How does it look like?

```
main() {
  #pragma omp parallel
  {
    printf("Hello world");
  }
}
```

- The #pragma introduces the parallel region
- Forking, Joining, Synchronising is done implicitly
- Compilation: cc -O2 -parallel -omp code.c

OpenMP: How does it look like?

```
main() {
  #pragma omp parallel
  {
    printf("Hello world");
  }
}
```

- The #pragma introduces the parallel region
- Forking, Joining, Synchronising is done implicitly
- Compilation: cc -O2 -parallel -omp code.c

OpenMP Library Routines

```
#include <omp.h>

main() {
  #pragma omp parallel
  {
    int n = omp_get_thread_num();
    printf("Greetings from thread %d!", n);
  }
}
```

Possible output:

```
Greetings from thread 2!
Greetings from thread 0!
Greetings from thread 1!
```

OpenMP Library Routines

```
#include <omp.h>

main() {
  #pragma omp parallel
  {
    int n = omp_get_thread_num();
    printf("Greetings from thread %d!", n);
  }
}
```

Possible output:

```
Greetings from thread 2!
Greetings from thread 0!
Greetings from thread 1!
```

OpenMP Library Routines

```
#include <omp.h>

main() {
  #pragma omp parallel num_threads(2)
  {
    int n = omp_get_thread_num();
    printf("Greetings from thread %d!", n);
  }
}
```

Possible output:

```
Greetings from thread 1!
Greetings from thread 0!
```

Better: `OMP_NUM_THREADS=2 ./program`

OpenMP Library Routines

```
#include <omp.h>

main() {
  #pragma omp parallel num_threads(2)
  {
    int n = omp_get_thread_num();
    printf("Greetings from thread %d!", n);
  }
}
```

Possible output:

```
Greetings from thread 1!
Greetings from thread 0!
```

Better: `OMP_NUM_THREADS=2 ./program`

OpenMP Loop Parallelisation I

Much more interesting:

```
#pragma omp parallel
#pragma omp for
for (int i = 0; i < n; i++)
```

A complete loop parallelisation in one line

Terms and Restrictions apply: Start and end values of the loop have to be computable before the loop starts. The loop must have this simple form, including a simple comparison test, a simple increment or decrement expression. Exits with `break`, `goto` or `return` are not allowed. Loops may not contain interesting statements or do anything that may or may not surprise the compiler or the user at any time. Only one loop per customer and compilation procedure. Offer expires at the next full moon. Offer is not valid in California, Asia and mainland Europe. Void where prohibited by law.

OpenMP Loop Parallelisation I

Much more interesting:

```
#pragma omp parallel for
#pragma omp for
for (int i = 0; i < n; i++)
```

A complete loop parallelisation in one line

Terms and Restrictions apply: Start and end values of the loop have to be computable before the loop starts. The loop must have this simple form, including a simple comparison test, a simple increment or decrement expression. Exits with `break`, `goto` or `return` are not allowed. Loops may not contain interesting statements or do anything that may or may not surprise the compiler or the user at any time. Only one loop per customer and compilation procedure. Offer expires at the next full moon. Offer is not valid in California, Asia and mainland Europe. Void where prohibited by law.

OpenMP Loop Parallelisation I

Much more interesting:

```
#pragma omp parallel for
#pragma omp for
for (int i = 0; i < n; i++)
```

A complete loop parallelisation in one line

Terms and Restrictions apply: Start and end values of the loop have to be computable before the loop starts. The loop must have this simple form, including a simple comparison test, a simple increment or decrement expression. Exits with `break`, `goto` or `return` are not allowed. Loops may not contain interesting statements or do anything that may or may not surprise the compiler or the user at any time. Only one loop per customer and compilation procedure. Offer expires at the next full moon. Offer is not valid in California, Asia and mainland Europe. Void where prohibited by law.

OpenMP Loop Parallelisation I

Much more interesting:

```
#pragma omp parallel for
#pragma omp for
for (int i = 0; i < n; i++)
```

A complete loop parallelisation in one line

Terms and Restrictions apply: Start and end values of the loop have to be computable before the loop starts. The loop must have this simple form, including a simple comparison test, a simple increment or decrement expression. Exits with `break`, `goto` or `return` are not allowed. Loops may not contain interesting statements or do anything that may or may not surprise the compiler or the user at any time. Only one loop per customer and compilation procedure. Offer expires at the next full moon. Offer is not valid in California, Asia and mainland Europe. Void where prohibited by law.

Please sign here...

:)

OpenMP Loop Parallelisation II

The Fineprint:

- Start and end values of the loop have to be computable before the loop starts.
- The loop must have the simple form:
  `for (a =` x_1`; a <=>` x_2`; a +=` x_3`)`
- This includes a simple comparison test and a simple increment or decrement expression.
- Exits with `break`, `goto` or `return` are not allowed.
- The loop variable must not be modified within the loop body.

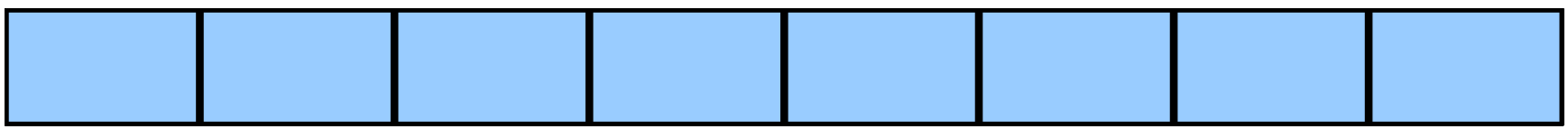
Loop Scheduling I

#pragma openmp parallel for

Given a loop of size x...

Loop Scheduling II

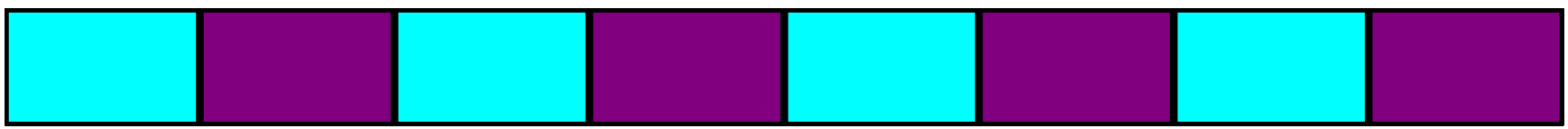
#pragma openmp parallel for schedule(.., x/n)



Given a loop of size x with n=8...

Loop Scheduling III

#pragma openmp parallel for schedule(static, x/n)



The default scheduling is **static**:
Simple round-robin, every thread gets the same amount of work.

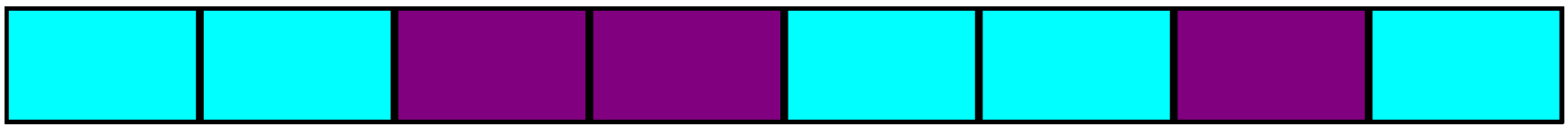
Loop Scheduling IIII

#pragma openmp parallel for schedule(dynamic, x/n)

With **dynamic** scheduling every thread gets one work unit.
Afterwards the work is distributed from a queue.

Loop Scheduling IIIII

#pragma openmp parallel for schedule(guided, x/n)



Guided scheduling is similar to dynamic,
but the size of the units decreases exponentially down to x/n.

OpenMP Data Model I

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < 100; i++) {
  sum = sum + i;
}
```

- Default: All variables are shared
- (except the loop variable)
- After the loop the value of `sum` is undefined.

OpenMP Data Model I

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < 100; i++) {
  sum = sum + i;
}
```

- Default: All variables are shared
- (except the loop variable)
- After the loop the value of `sum` is undefined.

OpenMP Data Model I

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < 100; i++) {
  sum = sum + i;
}
```

- Default: All variables are shared
- (except the loop variable)
- After the loop the value of `sum` is undefined.

OpenMP Data Model I

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < 100; i++) {
  sum = sum + i;
}
```

- Default: All variables are shared
- (except the loop variable)
- After the loop the value of `sum` is undefined.

OpenMP Data Model II

```
int sum = 0;
#pragma omp parallel for private(sum)
for (int i = 0; i < 100; i++) {
  sum = sum + i;
}
```

- Does not work.
- `sum` is undefined at the start of the loop
- But the results are thrown away at the end of the loop

OpenMP Data Model II

```
int sum = 0;
#pragma omp parallel for private(sum)
for (int i = 0; i < 100; i++) {
  sum = sum + i;
}
```

- Does not work.
- sum is undefined at the start of the loop
- But the results are thrown away at the end of the loop

OpenMP Data Model II

```
int sum = 0;
#pragma omp parallel for firstprivate(sum)
for (int i = 0; i < 100; i++) {
  sum = sum + i;
}
```

- Does not work.
- sum is undefined at the start of the loop (solved)
- But the results are thrown away at the end of the loop

OpenMP Data Model II

```
int sum = 0;
#pragma omp parallel for firstprivate(sum)
for (int i = 0; i < 100; i++) {
  sum = sum + i;
}
```

- Does not work.
- `sum` is undefined at the start of the loop (solved)
- But the results are thrown away at the end of the loop

Critical Sections

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < 100; i++) {
  #pragma omp critical
  {
    sum = sum + i;
  }
}
```

- One possible solution is the eliminate the race condition
- However we are now synchronising all the time

Critical Sections

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < 100; i++) {
  #pragma omp critical
  {
    sum = sum + i;
  }
}
```

- One possible solution is the eliminate the race condition
- However we are now synchronising all the time

Reduction

```
int sum = 0;
#pragma omp parallel for reduction(+:  sum)
for (int i = 0; i < 100; i++) {
  sum = sum + i;
}
```

- One synchronisation and reduction step at the end of the loop
- Reduction variable must be shared
- Allowed operators: +, -, *, &, exp, |, ||, &&
- Only a few different operations are allowed (eg. no assignment)

Reduction

```
int sum = 0;
#pragma omp parallel for reduction(+:  sum)
for (int i = 0; i < 100; i++) {
  sum = sum + i;
}
```

- One synchronisation and reduction step at the end of the loop
- Reduction variable must be shared
- Allowed operators: +, -, *, &, exp, |, ||, &&
- Only a few different operations are allowed (eg. no assignment)

Reduction

```
int sum = 0;
#pragma omp parallel for reduction(+:  sum)
for (int i = 0; i < 100; i++) {
  sum = sum + i;
}
```

- One synchronisation and reduction step at the end of the loop
- Reduction variable must be shared
- Allowed operators: +, -, *, &, exp, |, ||, &&
- Only a few different operations are allowed (eg. no assignment)

Reduction

```
int sum = 0;
#pragma omp parallel for reduction(+:  sum)
for (int i = 0; i < 100; i++) {
  sum = sum + i;
}
```

- One synchronisation and reduction step at the end of the loop
- Reduction variable must be shared
- Allowed operators: +, -, *, &, exp, |, ||, &&
- Only a few different operations are allowed (eg. no assignment)

Reasons to use OpenMP in your project

- Simple: No need for message passing, data transport or communication planning
- Data layout and decomposition handled automatically
- Code can be parallelised incrementally and independently, no major rewrites required
- Unified code for both serial and parallel applications: OpenMP constructs degrade gracefully, serial code needs no changes with OpenMP.
- Both coarse-grained and fine-grained parallelism are possible

Reasons to avoid OpenMP

- Requires a compiler that supports OpenMP.
- Scalability is limited by memory architecture.
- No reliable error handling.
- Lacks fine-grained mechanisms to control thread-processor mapping.
- Makes it easy to introduce hard-to-trace bugs.
- Obeys Amdahl's law.

Tanks!

Thank you.
Спасибо.
Dziekuje.
Danke.
謝謝

Tanks!

Thank you.
Спасибо.
Dziekuje.
Danke.
謝謝