Multiprocessors

Feb-Jun 2021

1

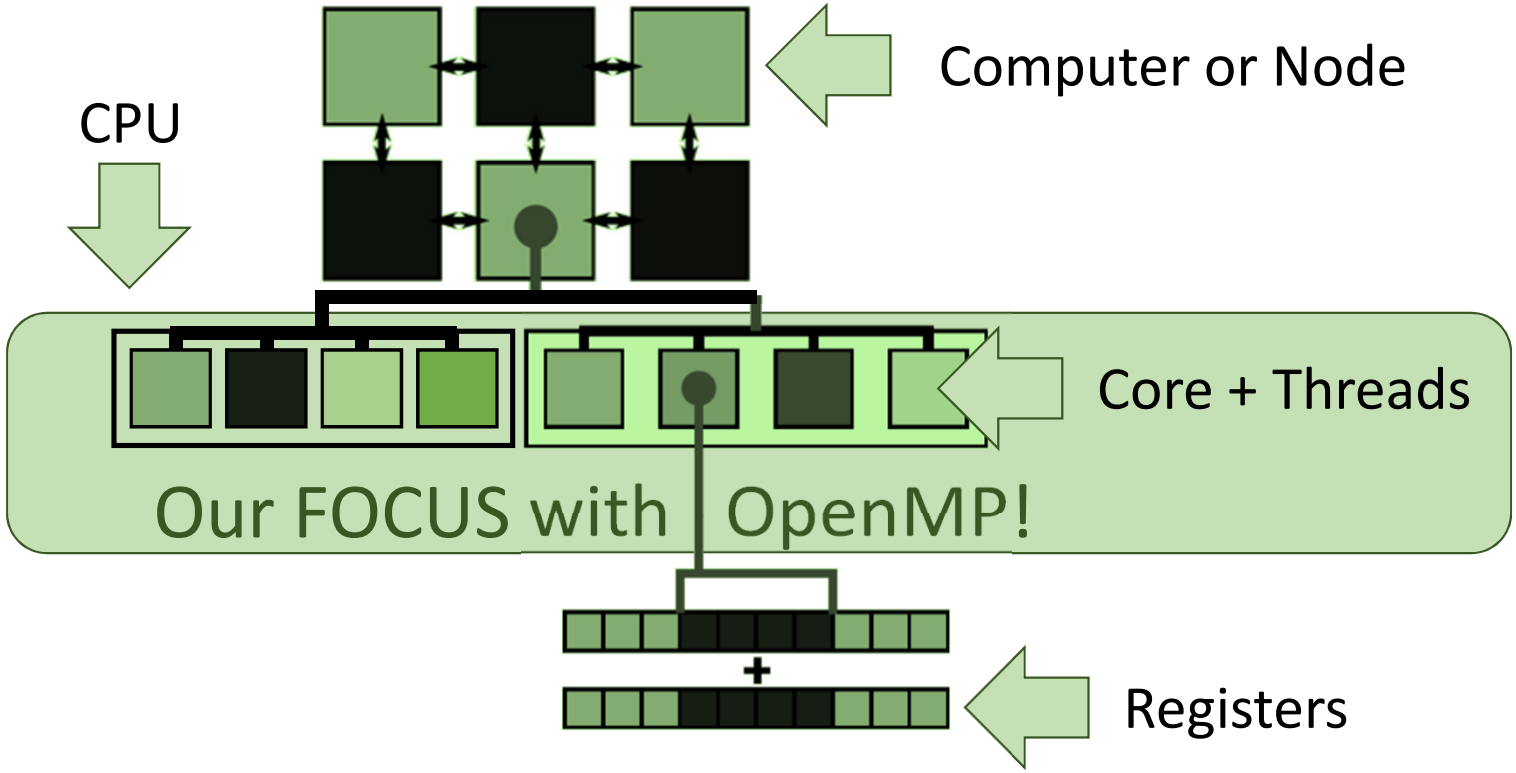
Open Multi-Processing

Alejandro Guajardo Moreno

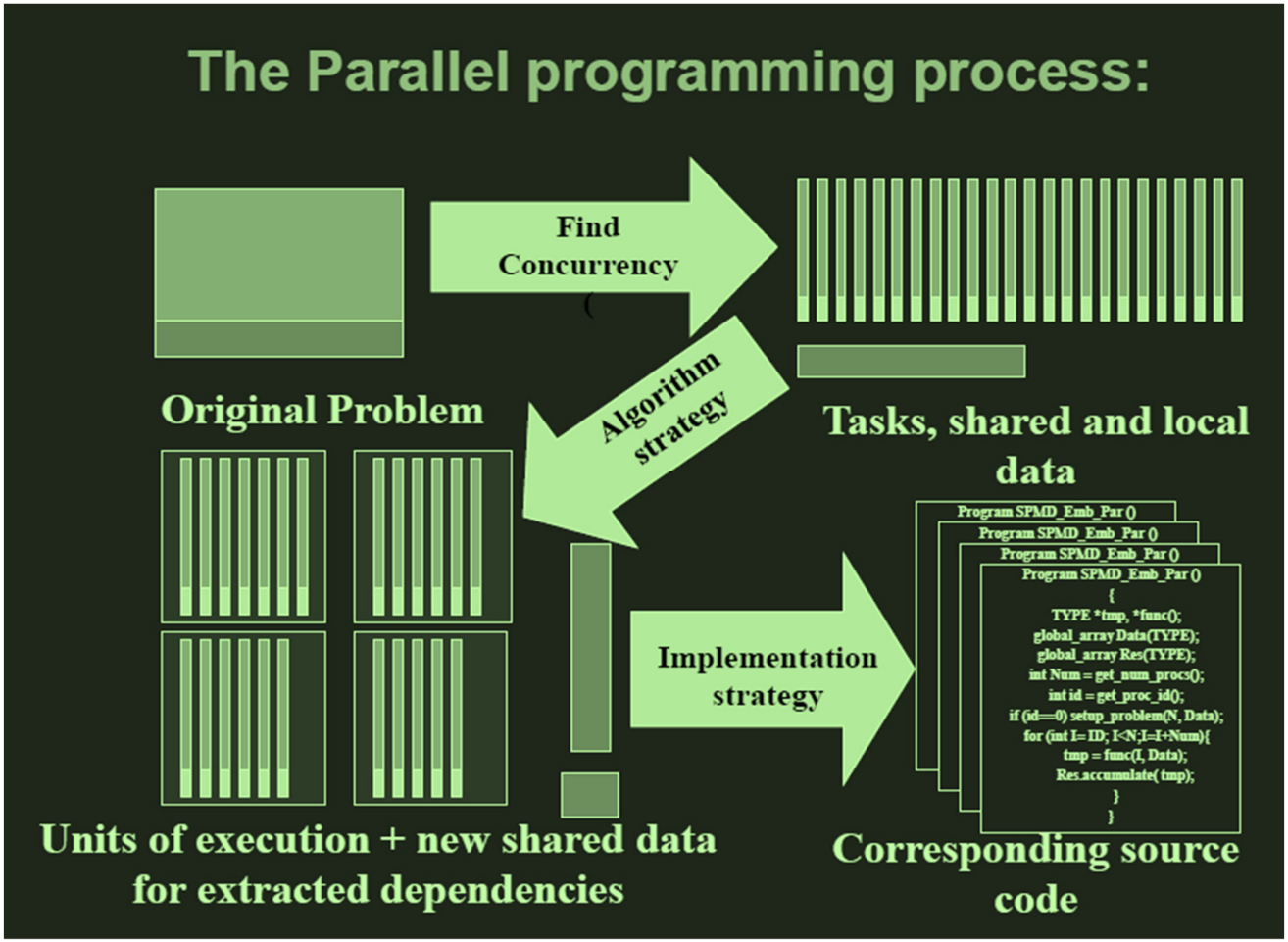
OBJECTIVES

- What is OpenMP?
- When to use OpenMP?
- OpenMP Concepts.
- Enable OpenMP support in compiler.
- Learn how to use OpenMP.

Where we stand



The Parallel programming process



Open Multi-Processing

- Known colloquially as OpenMP.
- Standard API for writing Shared Memory Parallel Applications.
- Consist of compiler directives, library functions and environmental variables.
- It makes possible to combine sequential and parallel code.
- Simplifies writing **multi-threaded** programs in C, C++ and Fortran.
- Standarizes last 22 years of SMP practice.
- Thread based implementation, code translation during compile time.
- Specification maintained by the OpenMP Architecture Review Board.
- https://www.openmp.org/
- Last Stable Version: 5.1 (November 13, 2020)

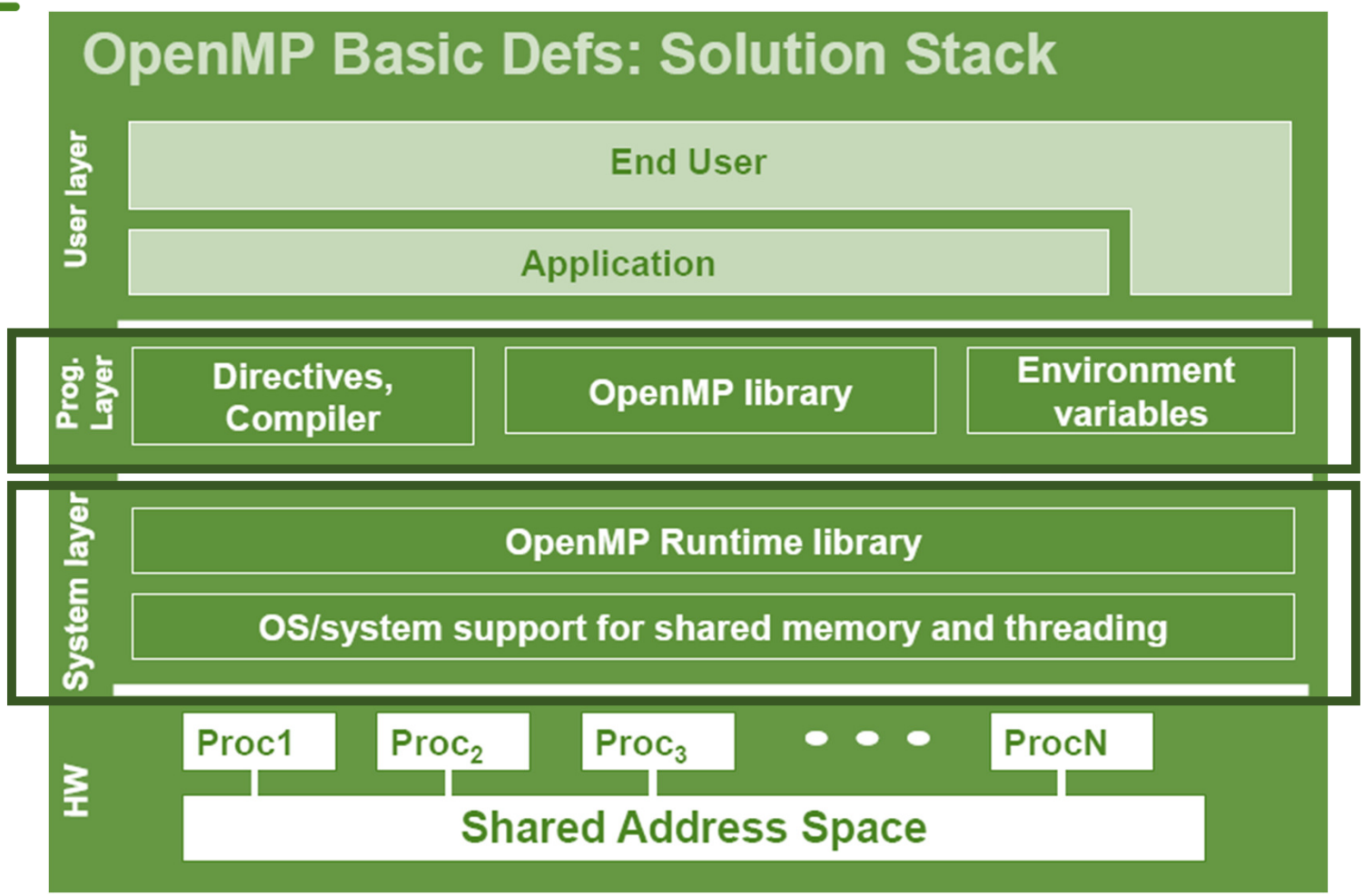
When to consider OpenMP?

- “Always” is a good option.
- Bear in mind… The complier may not be able to parallelize.
  - It cannot find the parallelism.
  - It cannot determine whether it is safe to parallelize.
  - It lacks information.

Advantages of OpenMP

- Performance.
- Scalability.
- Portability.
- Little programming effort.
- Mature standard.
- Ideally suited for multicore architecture.
- Memory and threading model map naturally.
- Allows incremental parallelization.
- Widely available and used.

OpenMP Stack



OpenMP Concept

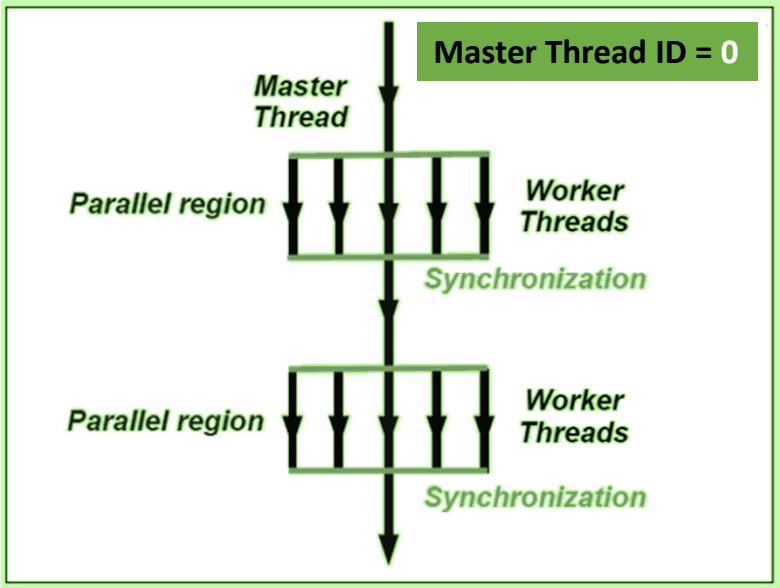
- Fork-Join model.
- Work-sharing constructs
  - data decomposition
  - task decomposition
- Specialized environments for variable scopes.
- Syncronization support.
- Lots of options for fine tunning.

Fork-join model

- The model uses one main thread and forks to multithread execution.
- The model may use several blocks of parallelization.

Fork = Parallel Region

Join = Synchronization



Microsoft Support for OpenMP in Visual Studio

/openmp (Enable OpenMP Support)

04/14/2019 • 3 minutes to read • +1

Causes the compiler to process #pragma omp directives in support of OpenMP.

Syntax

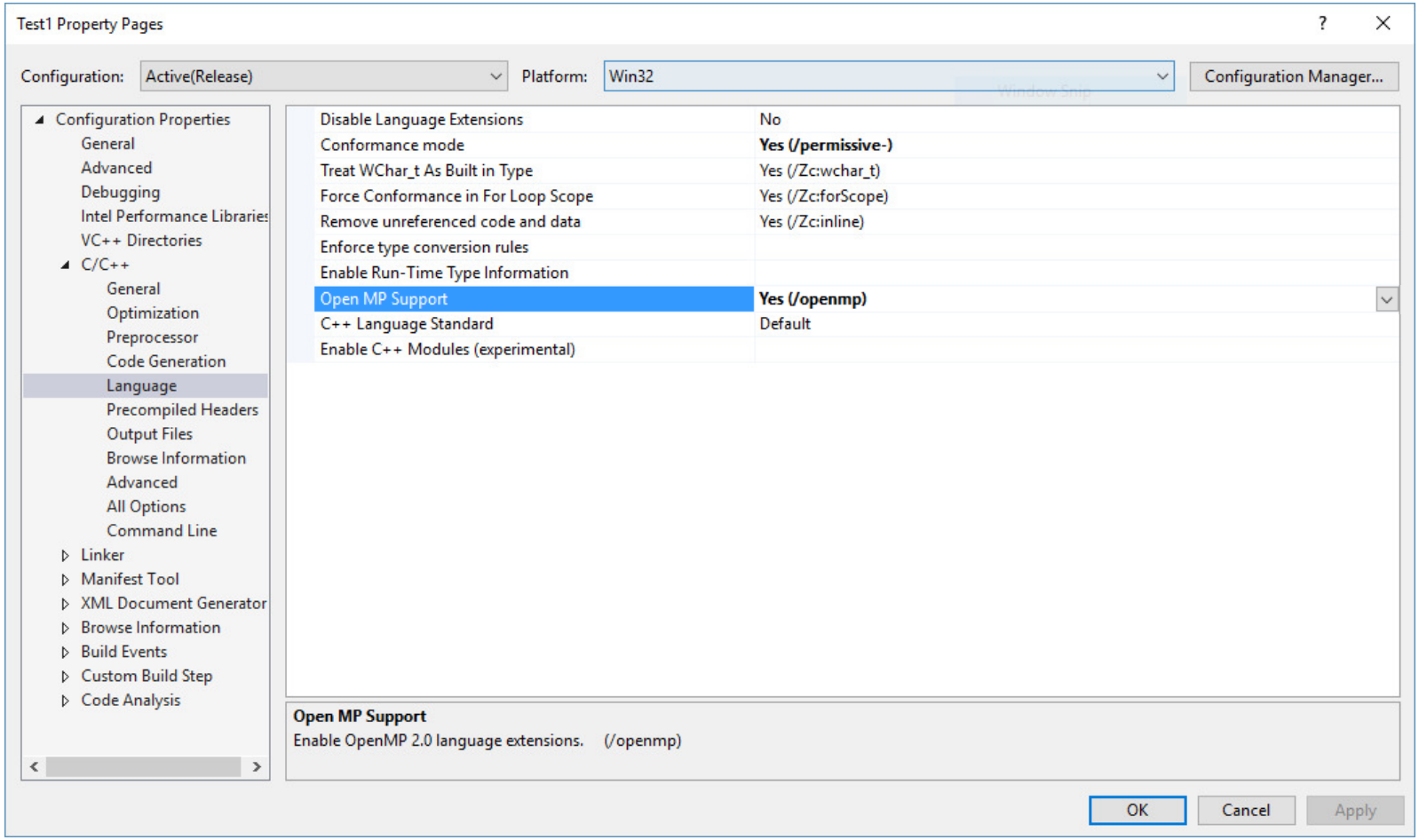
> **/openmp**[**:experimental**]

Remarks

`#pragma omp` is used to specify Directives and Clauses. If **/openmp** isn't specified in a compilation, the compiler ignores OpenMP clauses and directives. OpenMP Function calls are processed by the compiler even if **/openmp** isn't specified.

The C++ compiler currently supports the OpenMP 2.0 standard. However, Visual Studio 2019 also now offers SIMD functionality. To use SIMD, compile by using the **/openmp:experimental** option. This option enables both the usual OpenMP features, and additional OpenMP SIMD features not available when using the **/openmp** switch.

Enable support for OpenMP in Windows Visual Studio



GCC Support for OpenMP C/C++

GCC Version	OpenMP specification implemented
GCC 4.2	OpenMP 2.5
GCC 4.4	OpenMP 3.0
GCC 4.7	OpenMP 3.1
GCC 4.9	OpenMP 4.0
GCC 6	OpenMP 4.5
GCC 9	OpenMP 5.0 (not fully implemented)
GCC 10	OpenMP 5.0 (more features)
GCC 11	OpenMP 5.0 (more features)

Enable support for OpenMP in GCC

-fopenmp

From the man page…

-fopenmp

- Enable handling of OpenMP directives "#pragma omp" in C/C++ and "!$omp" in Fortran.
- When -fopenmp is specified, the compiler generates parallel code according to the OpenMP Application Program Interface v4.5 <http://www.openmp.org/>.
- This option implies -pthread, and thus is only supported on targets that have support for -pthread.
- -fopenmp implies -fopenmp-simd.

```
$ gcc openmp.c -o openmp -fopenmp
```

OpenMP Hello World

```c
#include <stdio.h>

int main() {
   printf("Hello World!...\n");
}
```

```c
#include <stdio.h>

int main() {

   #pragma omp parallel
   {
      printf("Hello OMP World!...\n");
   }
}
```

Exercise

1.- Compile and run the code…

```
#include <stdio.h>

int main() {
   #pragma omp parallel
   {
      printf("Hello OMP World!...\n");
   }
}
```

2.- Enable OpenMP in your compiler.

Windows => Properties => C/C++ => Language => Open MP Support

GCC => -fopenmp

3.- Compile and run the code again.

4.- Answer the following:

- **What happened?**
- **Was "Hello World" printed? How many times?**
- **What kind of "control" you saw?**
- **Any relationship of printed lines with your specific processor?**
- **Any changes to the code?**

So, what happened…

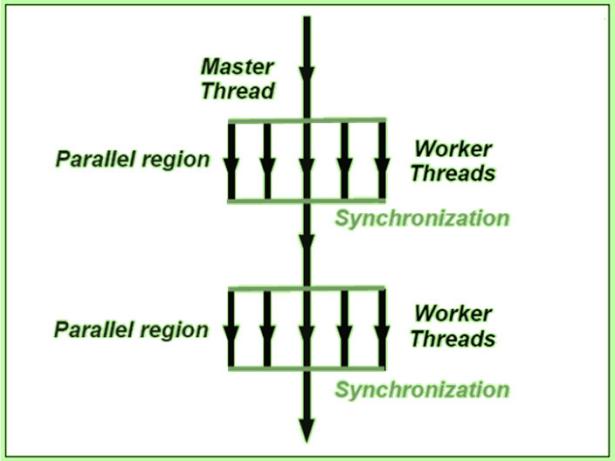
- ✓ We used the basic construct of OpenMP. The DIRECTIVE:

```
#pragma omp parallel
```

- ✓ Parallel region

```
#pragma omp parallel
  { //Parallel region begins
     printf("Hello OMP World!...\n");
  } //Parallel region ends
```

- ✓ We CAN an WILL control execution.



Here's how…

How many threads?

- Environment Variable
- OpenMP Functions
- Directives

How many threads?

- Environment Variable
- OpenMP Functions
- Directives

How many threads? – Environment variables

set OMP_NUM_THREADS=4

export OMP_NUM_THREADS = 4

- ✓ Specifies the default number of threads to use in parallel regions.
- ✓ The value of this variable shall be a positive integers.
- ✓ Nested parallel regions are allowed.
- ✓ In nested parallel regions, the variable shall be a comma-separated list.
- ✓ If undefined, ONE thread per CPU is used.

How many threads?

- Environment Variable
- OpenMP Functions
- Directives

How many threads? – OpenMP Functions

- ✓ They give visibility and control over the creation of threads.
- ✓ Need header file with prototype funcitons.

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

- ✓ Some functions:
  - ✓ `omp_get_thread_num()`

    Returns the thread number of the thread executing within its thread team.
  - ✓ `omp_set_num_threads(nthreads)`

    Sets the number of threads in upcoming parallel regions.

How many threads?

- Environment Variable
- OpenMP Functions
- Directives

How many threads? – OpenMP Functions

- ✓ A clause:

```
num_threads(num_threads)
```

In the DIRECTIVE. Will look like:

```
#pragma omp parallel num_threads(#)
```

Exercise - Individual

1.- Test the different forms to control threads

```
#include <stdio.h>

int main() {
    #pragma omp parallel
    {
        printf("Hello OMP World!...\n");
    }
}
```

```
Environment Variable
set OMP_NUM_THREADS = 2
export OMP_NUM_THREADS = 2

Functions <omp.h>
omp_set_num_threads(nthreads);
omp_get_thread_num()

Directive
#pragma omp parallel num_threads(#)
```

2.- Answer:

- **What happened?**
- **Do you feel now in “control”?**
- **How are the options overridden between them?**

Controlling number of threads from inside…

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

int main()
{
   int nthreads = 4;
   omp_set_num_threads(nthreads);

   #pragma omp parallel
   {
      printf("Hello OMP World!... from thread = %d\n", omp_get_thread_num());
   }
}
```

Controlling number of threads using compiler directive

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

int main()
{
   #pragma omp parallel num_threads(6)
   {
      printf("Hello OMP World!... from thread = %d\n", omp_get_thread_num());
   }
}
```

Parallel region and sync…

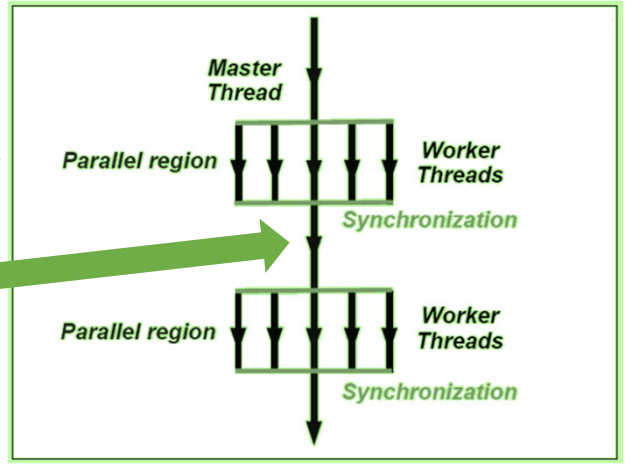
```
#include <stdio.h>

int main() {

   #pragma omp parallel
   { //Parallel region begins
      printf("Hello OMP World!...\n");
   } //Parallel region ends

   printf("This will print after sync...
And only once!...\n");

}
```



What happens behind the curtain…

OpenMP: what the compiler does

```
#pragma omp parallel num_threads(4)
{
    foobar ();
}
```

- The OpenMP compiler generates code logically analogous to that on the right of this slide, given an OpenMP pragma such as that on the top-left
- All known OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for reach parallel region.
- Only three threads are created because the last parallel section will be invoked from the parent thread.

```
void thunk ()
{
    foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
   pthread_create (
        &tid[i],0,thunk, 0);
thunk();

for (int i = 1; i < 4; ++i)
    pthread_join (tid[i]);
```

Mattson

Exercise – Teams of 3

1.- Create a program that do the following:

- **Define 3 Arrays of FLOATS.**
- **16 elements each.**
  - **Array A and B = {0.0, 1.0, 2.0… 15.0}**
  - **Array C is zeroed at the begining.**
- **Using OMP, do:**
  - **C[i] = A[i] + B[i];**

Remember to:
1.- First: find the concurrency.
2.- Second: Define your algorithm strategy.
3.- Third: Define your implementation strategy.

```
OpenMP construct
#pragma omp parallel
{
   //Parallel Region
}

Environment Variable
set OMP_NUM_THREADS = #
Export OMP_NUM_THREADS = #

Functions
omp_set_num_threads(#);
omp_get_thread_num()

Directive
#pragma omp parallel num_threads(#)
```

Adding to FLOAT arrays using OMP & Intrinsics…

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

int main()
{
   float __declspec(align(16)) arrayA[16] = { 0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0 };
   float __declspec(align(16)) arrayB[16] = { 0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0 };
   float __declspec(align(16)) arrayC[16] = { 0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0 };

   int nthreads = 4;
   omp_set_num_threads(nthreads);

   #pragma omp parallel
   {
      int thread_num = omp_get_thread_num();
      int position = thread_num * 4;
      printf("Thread %d running... Working at position %d\n", thread_num, position);
      __m128 a = _mm_load_ps(&arrayA[position]);
      __m128 b = _mm_load_ps(&arrayB[position]);
      __m128 sum = _mm_add_ps(a, b);
      _mm_store_ps(&arrayC[position], sum);;
   }

   for (int i = 0; i < 16; i++) {
      printf("ArrayC[%d]=%6.0lf\n", i, arrayC[i]);
   }
}
```

See you soon…