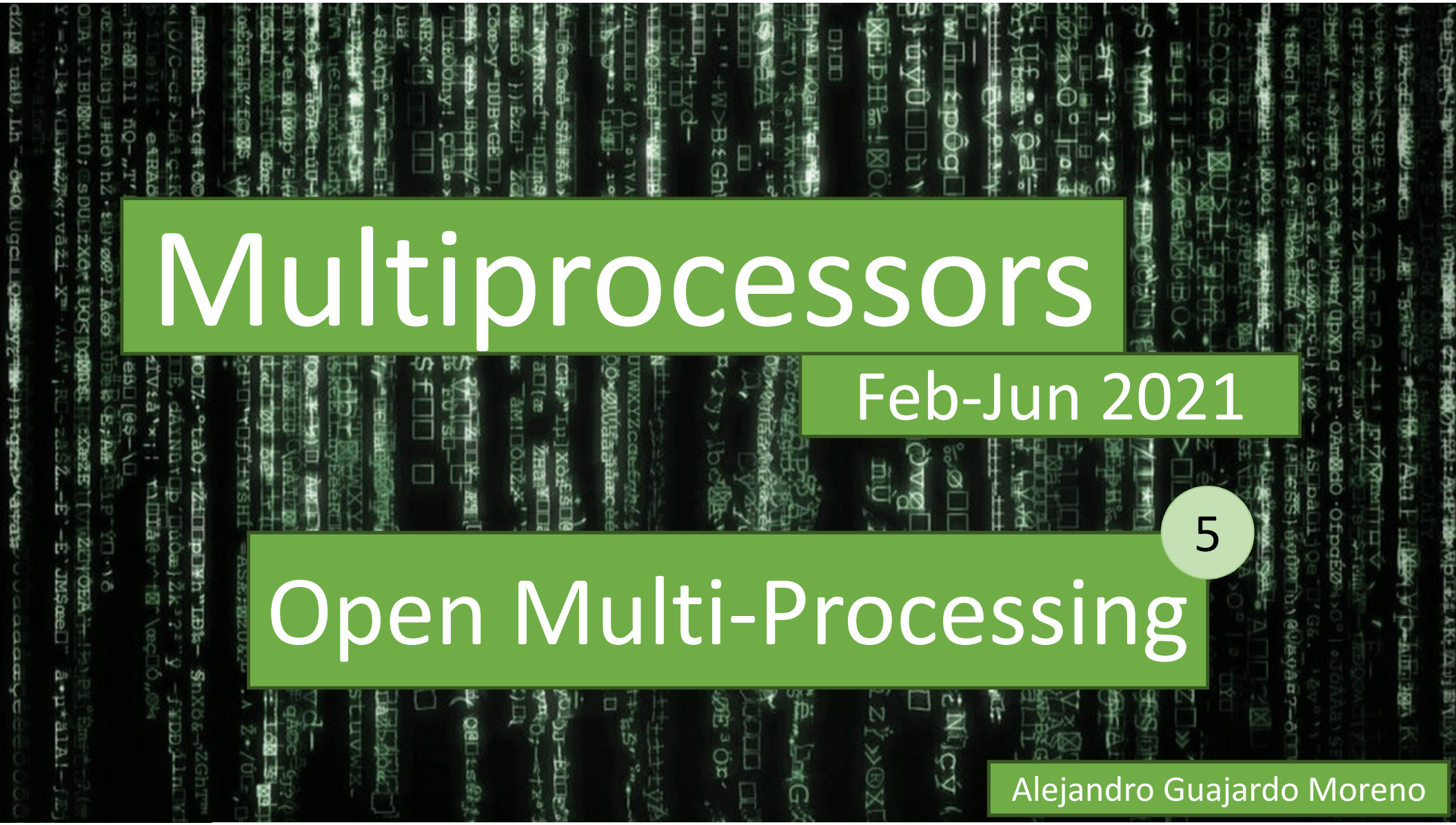


Multiprocessors

Feb-Jun 2021

5

Open Multi-Processing

Alejandro Guajardo Moreno

From Homework…

Create a program using OMP to do task decomposition:

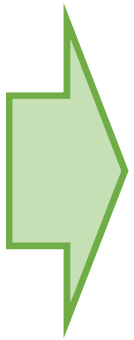
1.- Create a program that do the following:

- **Define 6 Arrays of FLOATS.**
- **16 elements each.**
  - **Array A = {10.0, 11.0, 12.0… 25.0}**
  - **Array B= {1.0, 2.0, 3.0… 16.0}**
  - **Array C, D, E & F are zeroed at the begining.**
- **Using OMP, do task decomposition:**
  - **C[i] = A[i] + B[i];**
  - **D[i] = A[i] - B[i];**
  - **E[i] = A[i] * B[i];**
  - **F[i] = A[i] / B[i];**
- **Print the 4 lines with the results. One line per result.**

2.- Hand in your code and a screenshot of your output.

Can this be improved (1)?

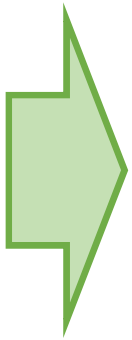
```
#pragma omp parallel num_threads(4)
{
for (int i = 0; i < 16; i++)
   {
      if (omp_get_thread_num() == 0)
         C[i] = A[i] + B[i];

      if (omp_get_thread_num() == 1)
         D[i] = A[i] - B[i];

      if (omp_get_thread_num() == 2)
         E[i] = A[i] * B[i];

      if (omp_get_thread_num() == 3)
         F[i] = A[i] / B[i];
   }
}
```



```
#pragma omp parallel num_threads(4)
{
   if (omp_get_thread_num() == 0)
      for (int i = 0; i < 16; i++)
         C[i] = A[i] + B[i];

   if (omp_get_thread_num() == 1)
      for (int i = 0; i < 16; i++)
         D[i] = A[i] - B[i];

   if (omp_get_thread_num() == 2)
      for (int i = 0; i < 16; i++)
         E[i] = A[i] * B[i];

   if (omp_get_thread_num() == 3)
      for (int i = 0; i < 16; i++)
         F[i] = A[i] / B[i];
}
```

Can this be improved (2)?

```
#pragma omp parallel num_threads(4)
{
   if (omp_get_thread_num() == 0)
      for (int i = 0; i < 16; i++)
         C[i] = A[i] + B[i];

   if (omp_get_thread_num() == 1)
      for (int i = 0; i < 16; i++)
         D[i] = A[i] - B[i];

   if (omp_get_thread_num() == 2)
      for (int i = 0; i < 16; i++)
         E[i] = A[i] * B[i];

   if (omp_get_thread_num() == 3)
      for (int i = 0; i < 16; i++)
         F[i] = A[i] / B[i];
}
```



```
#pragma omp parallel
{
int thread_num = omp_get_thread_num();
switch (thread_num)
{
   case 0: //Add
      for (int i = 0; i < 16; i++) {
         arrayC[i] = arrayA[i] + arrayB[i];
      };
      break;
   case 1: //Substract
      for (int i = 0; i < 16; i++) {
         arrayD[i] = arrayA[i] - arrayB[i];
      };
      break;
   case 2: //Multiply
      for (int i = 0; i < 16; i++) {
         arrayE[i] = arrayA[i] * arrayB[i];
      };
      break;
   case 3: //Divide
      for (int i = 0; i < 16; i++) {
         arrayF[i] = arrayA[i] / arrayB[i];
      };
      break;
}
}
```

Can this be improved (3)?

```
#pragma omp parallel
  {
  int thread_num = omp_get_thread_num();
  switch (thread_num)
  {
     case 0: //Add
        for (int i = 0; i < 16; i++) {
           arrayC[i] = arrayA[i] + arrayB[i];
        };
        break;
     case 1: //Substract
        for (int i = 0; i < 16; i++) {
           arrayD[i] = arrayA[i] - arrayB[i];
        };
        break;
     case 2: //Multiply
        …
     case 3: //Divide
        …
  }
  }
```

```
#pragma omp parallel
{
int thread_num = omp_get_thread_num();
switch (thread_num)
{
   case 0: //Add
      for (int i = 0; i < 16/4; i++) {
         __m128 a = _mm128_load_ps(&A[i*4]);
         __m128 b = _mm128_load_ps(&B[i*4]);
         __m128 sum = _mm128_add_ps(a, b);
         _mm128_store_ps(&C[i*4], sum);;
      };
      break;
   case 1: //Substract
      for (int i = 0; i < 16/4; i++) {
         __m128 a = _mm128_load_ps(&A[i*4]);
         __m128 b = _mm128_load_ps(&B[i*4]);
         __m128 sum = _mm128_sub_ps(a, b);
         _mm128_store_ps(&C[i*4], sum);;
      };          break;
   case 2: //Multiply
      …
   case 3: //Divide
      …
}
```

Can this be improved (4)?

```
#pragma omp parallel
  {
  int thread_num = omp_get_thread_num();
  switch (thread_num)
  {
     case 0: //Add
        for (int i = 0; i < 16/4; i++) {
           __m128 a = _mm128_load_ps(&A[i*4]);
           __m128 b = _mm128_load_ps(&B[i*4]);
           __m128 sum = _mm128_add_ps(a, b);
           _mm128_store_ps(&C[i*4], sum);;
        };
        break;
     case 1: //Substract
        for (int i = 0; i < 16/4; i++) {
           __m128 a = _mm128_load_ps(&A[i*4]);
           __m128 b = _mm128_load_ps(&B[i*4]);
           __m128 sum = _mm128_sub_ps(a, b);
           _mm128_store_ps(&C[i*4], sum);;
        };          break;
     case 2: //Multiply
        …
     case 3: //Divide
        …
  }
```

```
#pragma omp parallel
{
int thread_num = omp_get_thread_num();
switch (thread_num)
{
   case 0: //Add
      for (int i = 0; i < 16/8; i++) {
         __m256 a = _mm256_load_ps(&A[i * 8]);
         __m256 b = _mm256_load_ps(&B[i * 8]);
         __m256 sum = _mm256_add_ps(a, b);
         _mm256_store_ps(&C[i * 8], sum);
      };
      break;
   case 1: //Substract
      for (int i = 0; i < 16/8; i++) {
         __m256 a = _mm256_load_ps(&A[i * 8]);
         __m256 b = _mm256_load_ps(&B[i * 8]);
         __m256 sum = _mm256_sub_ps(a, b);
         _mm256_store_ps(&C[i * 8], sum);
          break;
   case 2: //Multiply
      …
   case 3: //Divide
      …
}
};
```

Can this be improved (5)?

```
#pragma omp parallel
{
int thread_num = omp_get_thread_num();
switch (thread_num)
{
   case 0: //Add
      for (int i = 0; i < 16/8; i++) {
       __m256 a = _mm256_load_ps(&A[i * 8]);
       __m256 b = _mm256_load_ps(&B[i * 8]);
       __m256 sum = _mm256_add_ps(a, b);
       _mm256_store_ps(&C[i * 8], sum);
      };
      break;
   case 1: //Substract
      for (int i = 0; i < 16/8; i++) {
       __m256 a = _mm256_load_ps(&A[i * 8]);
       __m256 b = _mm256_load_ps(&B[i * 8]);
       __m256 sum = _mm256_sub_ps(a, b);
       _mm256_store_ps(&C[i * 8], sum);
      };
      break;
   case 2: //Multiply
      …
   case 3: //Divide
      …
}
```

```
#pragma omp parallel
{
int thread_num = omp_get_thread_num();
switch (thread_num)
{
   case 0: //Add
      {
       __m256 a = _mm256_load_ps(&A[0]);
       __m256 b = _mm256_load_ps(&B[0]);
       __m256 sum = _mm256_add_ps(a, b);
       _mm256_store_ps(&C[0], sum);
       a = _mm256_load_ps(&A[8]);
       b = _mm256_load_ps(&B[8]);
       sum = _mm256_add_ps(a, b);
       _mm256_store_ps(&C[8], sum);
      };
      break;
   case 1: //Substract
      …
   case 2: //Multiply
      …
   case 3: //Divide
      …
}
```

Can this be improved (6)?

```
#pragma omp parallel
{
int thread_num = omp_get_thread_num();
switch (thread_num)
{
   case 0: //Add
      {
       __m256 a = _mm256_load_ps(&A[0]);
       __m256 b = _mm256_load_ps(&B[0]);
       __m256 sum = _mm256_add_ps(a, b);
       _mm256_store_ps(&C[0], sum);
       a = _mm256_load_ps(&A[8]);
       b = _mm256_load_ps(&B[8]);
       sum = _mm256_add_ps(a, b);
       _mm256_store_ps(&C[8], sum);
      };
      break;
   case 1: //Substract
      …
   case 2: //Multiply
      …
   case 3: //Divide
      …
}
```

```
#pragma omp parallel
{
int thread_num = omp_get_thread_num();
switch (thread_num)
{
   case 0: //Add
      {
       __m512 a = _mm512_load_ps(&A[0]);
       __m512 b = _mm512_load_ps(&B[0]);
       __m512 sum = _mm512_add_ps(a, b);
       _mm512_store_ps(&C[0], sum);
      };
      break;
   case 1: //Substract
      …
   case 2: //Multiply
      …
   case 3: //Divide
      …
}
```

Can this be improved (7)?

```
#pragma omp parallel
  {
  int thread_num = omp_get_thread_num();
  switch (thread_num)
  {
     case 0: //Add
        {
         __m512 a = _mm512_load_ps(&A[0]);
         __m512 b = _mm512_load_ps(&B[0]);
         __m512 sum = _mm512_add_ps(a, b);
         _mm512_store_ps(&C[0], sum);
        };
        break;
     case 1: //Substract
        …
     case 2: //Multiply
        …
     case 3: //Divide
        …
  }
```

```
//#pragma omp parallel
//{
//int thread_num = omp_get_thread_num();
//switch (thread_num)
//{
//    case 0: //Add
//       {
         __m512 a = _mm512_load_ps(&A[0]);
         __m512 b = _mm512_load_ps(&B[0]);
         __m512 sum = _mm512_add_ps(a, b);
         __m512 sub = _mm512_sub_ps(a, b);
         __m512 mul = _mm512_mul_ps(a, b);
         __m512 div = _mm512_div_ps(a, b);
         _mm512_store_ps(&C[0], sum);
         _mm512_store_ps(&D[0], sub);
         _mm512_store_ps(&E[0], mul);
         _mm512_store_ps(&F[0], div);
//       };
//       break;
//    case 1: //Substract
//       …
//    case 2: //Multiply
//       …
//    case 3: //Divide
//       …
//}
```

Can this be improved (7)?

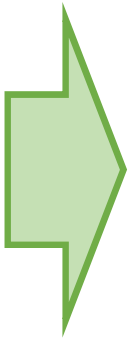
- In this case, consider thread creation costs vs. 6 more intrinsics executed by the main thread…

```
__m512 a = _mm512_load_ps(&A[0]);
__m512 b = _mm512_load_ps(&B[0]);
__m512 sum = _mm512_add_ps(a, b);
__m512 sub = _mm512_sub_ps(a, b); //1
__m512 mul = _mm512_mul_ps(a, b); //2
__m512 div = _mm512_div_ps(a, b); //3
_mm512_store_ps(&C[0], sum);
_mm512_store_ps(&D[0], sub);      //4
_mm512_store_ps(&E[0], mul);      //5
_mm512_store_ps(&F[0], div);      //6
```

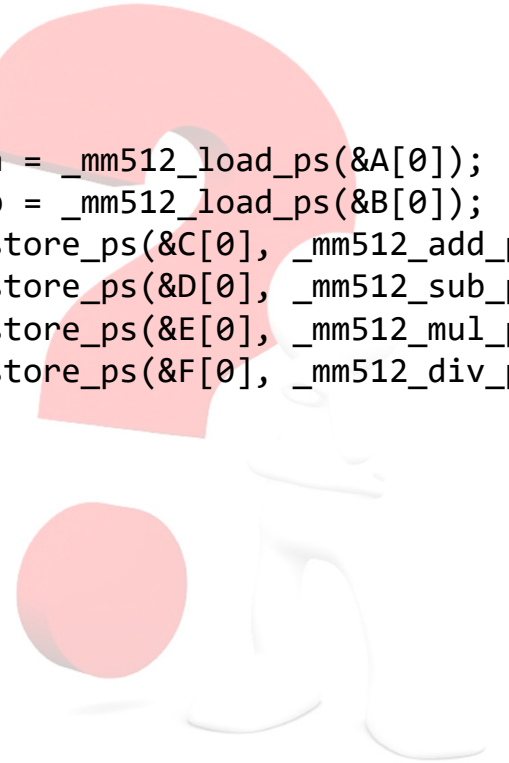
- For low repeat cycles, Intrinsics will be faster.

Can this be improved (8)?

```
__m512 a = _mm512_load_ps(&A[0]);
__m512 b = _mm512_load_ps(&B[0]);
__m512 sum = _mm512_add_ps(a, b);
__m512 sub = _mm512_sub_ps(a, b);
__m512 mul = _mm512_mul_ps(a, b);
__m512 div = _mm512_div_ps(a, b);
_mm512_store_ps(&C[0], sum);
_mm512_store_ps(&D[0], sub);
_mm512_store_ps(&E[0], mul);
_mm512_store_ps(&F[0], div);
```



```
__m512 a = _mm512_load_ps(&A[0]);
__m512 b = _mm512_load_ps(&B[0]);
_mm512_store_ps(&C[0], _mm512_add_ps(a, b));
_mm512_store_ps(&D[0], _mm512_sub_ps(a, b));
_mm512_store_ps(&E[0], _mm512_mul_ps(a, b));
_mm512_store_ps(&F[0], _mm512_div_ps(a, b));
```



Look at the Assembly…

```
case 0:
    {
        __m256 a = _mm256_load_ps(&A[0]);
        __m256 b = _mm256_load_ps(&B[0]);
        __m256 sum = _mm256_add_ps(a, b);
        _mm256_store_ps(&C[0], sum);
    }
    break;
```

```
case 0:
    {
        __m256 a = _mm256_load_ps(&A[0]);
        __m256 b = _mm256_load_ps(&B[0]);
        _mm256_store_ps(&C[0], _mm256_add_ps(a, b));
    }
    break;
```

No more benefit…

```
        case 0:
            {
                __m256 a = _mm256_load_ps(&A[0]);
                __m256 b = _mm256_load_ps(&B[0]);
00B11DEA  mov         eax,dword ptr [B]
00B11DED  pop         esi
                __m256 sum = _mm256_add_ps(a, b);
00B11DEE  vmovups     ymm0,ymmword ptr [eax]
00B11DF2  mov         eax,dword ptr [A]
00B11DF5  vaddps      ymm0,ymm0,ymmword ptr [eax]
                _mm256_store_ps(&C[0], sum);
00B11DF9  mov         eax,dword ptr [C]
00B11DFC  vmovups     ymmword ptr [eax],ymm0
00B11E00  vzeroupper
            }
            break;
```

```
        case 0:
            {
                __m256 a = _mm256_load_ps(&A[0]);
                __m256 b = _mm256_load_ps(&B[0]);
00B21DEA  mov         eax,dword ptr [B]
00B21DED  pop         esi
                _mm256_store_ps(&C[0], _mm256_add_ps(a, b));
00B21DEE  vmovups     ymm0,ymmword ptr [eax]
00B21DF2  mov         eax,dword ptr [A]
00B21DF5  vaddps      ymm0,ymm0,ymmword ptr [eax]
00B21DF9  mov         eax,dword ptr [C]
00B21DFC  vmovups     ymmword ptr [eax],ymm0
00B21E00  vzeroupper
            }
            break;
```

Solution – Task Decomposition using OMP Sections

1.- Using OMP SECTIONS/SECTION....

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

int main()
{
float __declspec(align(16)) arrayA[16] = { 10.0,11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,20.0,21.0,22.0,23.0,24.0,25.0 };
float __declspec(align(16)) arrayB[16] = { 1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0,16.0 };
float __declspec(align(16)) arrayC[16], arrayD[16], arrayE[16], arrayF[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 };

#pragma omp parallel
{
   #pragma omp single
   {
      printf("Total threads in parallel region: %d\n\n", omp_get_num_threads());
   }

   #pragma omp sections
   {
      //Add
      #pragma omp section
      {
         printf("Thread in charge of addition: %d\n", omp_get_thread_num());
         for (int i = 0; i < 16; i++)
         {
            arrayC[i] = arrayA[i] + arrayB[i];
         }
      }

      //Substract
      #pragma omp section
      {
         printf("Thread in charge of substraction: %d\n", omp_get_thread_num());
         for (int i = 0; i < 16; i++)
         {
            arrayD[i] = arrayA[i] - arrayB[i];
         }
      }

      //Multiplication
```

Solution – Task Decomposition using OMP Sections

```
#pragma omp sections
{
    //Add
    #pragma omp section
    {
        printf("Thread in charge of addition: %d\n", omp_get_thread_num());
            for (int i = 0; i < 16; i++)
                arrayC[i] = arrayA[i] + arrayB[i];
    }

    //Substract
    #pragma omp section
    {
        printf("Thread in charge of substraction: %d\n", omp_get_thread_num());
            for (int i = 0; i < 16; i++)
                arrayD[i] = arrayA[i] - arrayB[i];
    }
        printf("Thread in charge of division: %d\n", omp_get_thread_num());
            for (int i = 0; i < 16; i++)
                arrayF[i] = arrayA[i] / arrayB[i];
    }
  }
}
```

From Homework… Any question?

Create a program using OMP to do task decomposition:

1.- Create a program that do the following:

- **Define 6 Arrays of FLOATS.**
- **16 elements each.**
  - **Array A = {10.0, 11.0, 12.0… 25.0}**
  - **Array B= {1.0, 2.0, 3.0… 16.0}**
  - **Array C, D, E & F are zeroed at the begining.**
- **Using OMP, do task decomposition:**
  - **C[i] = A[i] + B[i];**
  - **D[i] = A[i] - B[i];**
  - **E[i] = A[i] * B[i];**
  - **F[i] = A[i] / B[i];**
- **Print the 4 lines with the results. One line per result.**

2.- Hand in your code and a screenshot of your output.

OpenMP – BIG Summary

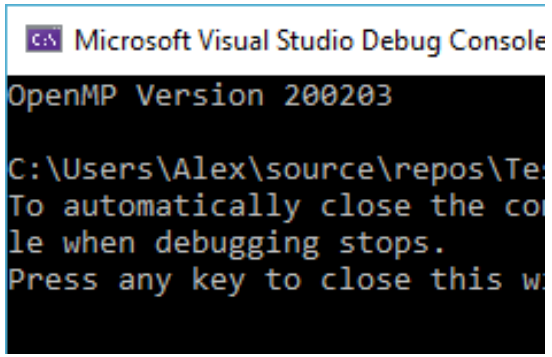
- So, by now, you have a pretty good idea of Open MP.
- We'll put some formallity now.

OpenMP MACRO: _OPENMP

- Is defined by OpenMP-compliant implementations as the decimal constant yyyymm, which will be the year and month of the approved specification.
- How to know which OpenMP specification is implemented in the compiler?
- Just print the constant….

```
mp@multiprocesadores:~/multiprocs$ ./version
OpenMP Version 201511
mp@multiprocesadores:~/multiprocs$
```

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

int main()
{
   printf("OpenMP Version %d\n", _OPENMP);
   return 0;
}
```



OpenMP Specifications for C/C++

	OpenMP Specification
199810	1.0
200203	2.0
200505	2.5
200811	3.0
201109	3.1
201310	4.0
201511	4.5
201811	5.0

OpenMP Conditional Compilation

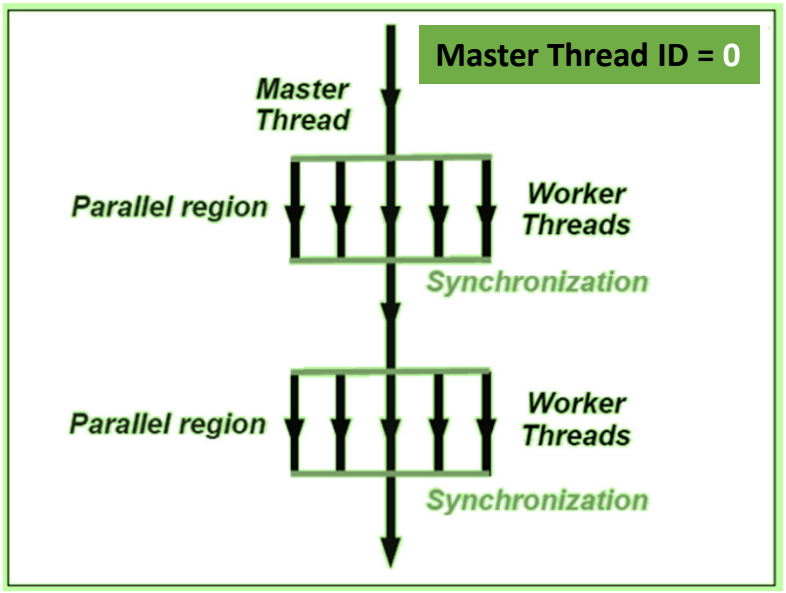
```
#include <omp.h>
#include <stdio.h>

int main()
{
   #ifdef _OPENMP
      int numThreads = omp_get_max_threads();
      printf("Number of threads %d\n\n", numThreads);
      omp_set_num_threads(numThreads - 4);
   #endif

   #pragma omp parallel
   { // Start of parallel region
      printf("Hello OMP World!...\n");
   } // End of parallel region

   return 0;
}
```

Fork-join model

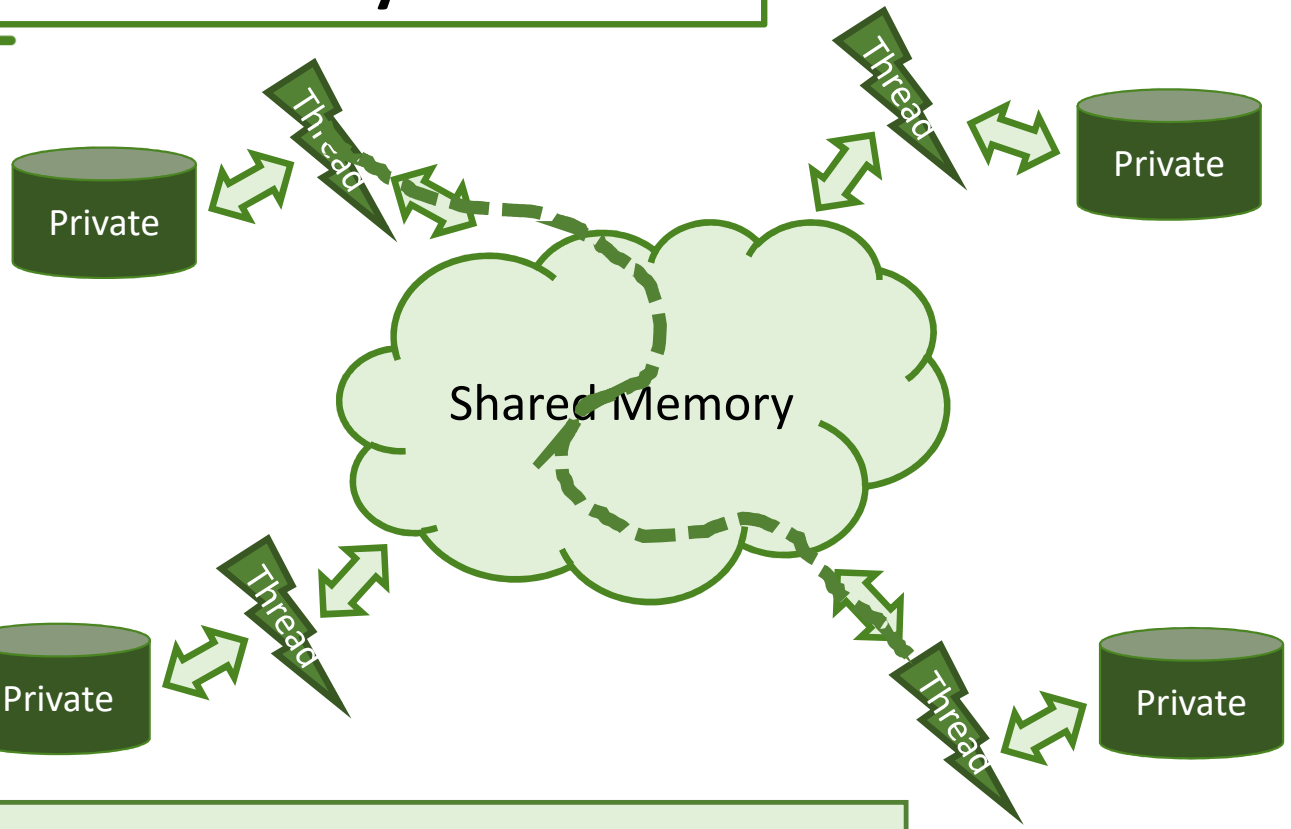
- The model uses one main thread and forks to multithread execution.
- The model may use several blocks of parallelization.

Fork = Parallel Region

Join = Synchronization



OpenMP Memory Model

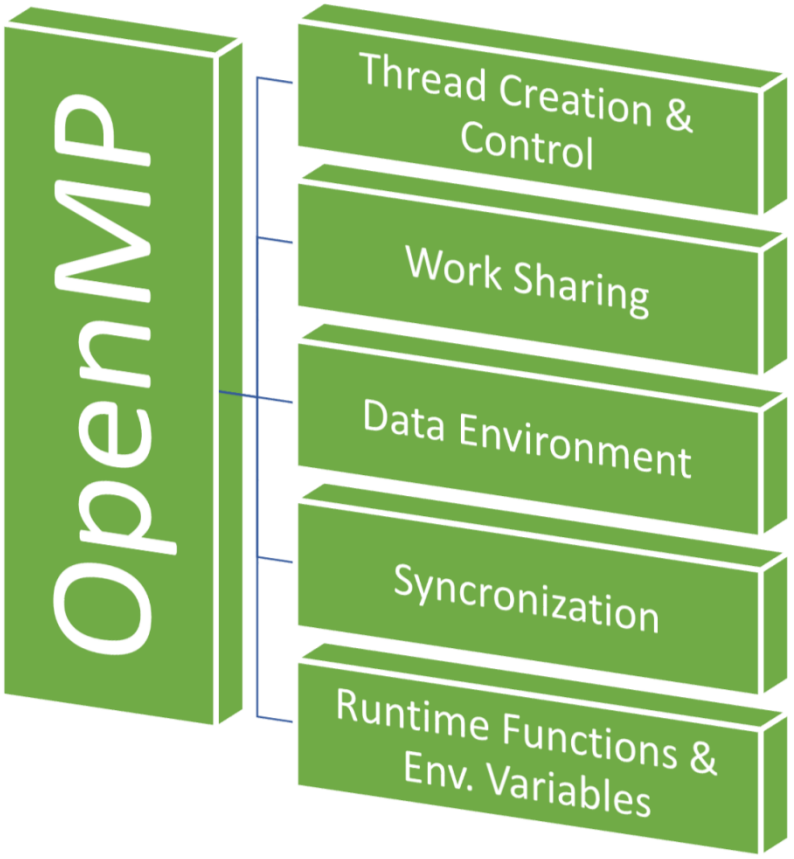


2 types of memory

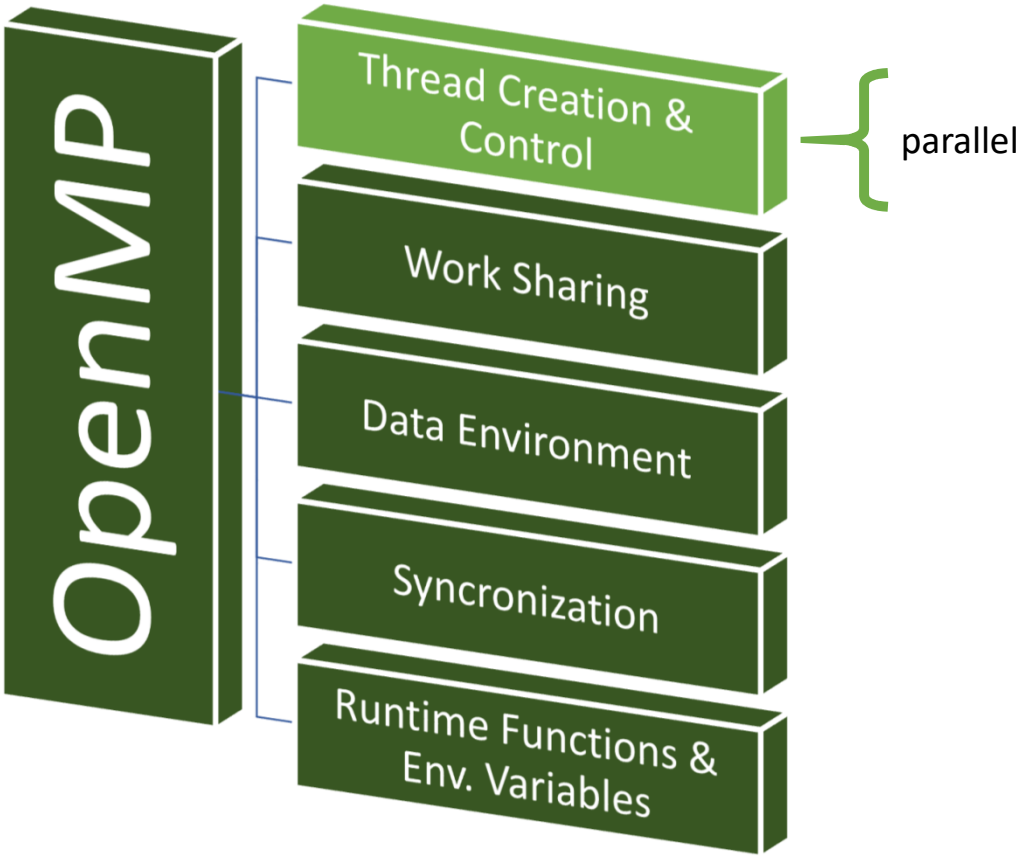
Heap – For all the program (OMP = Shared)

Stack – For just the thread (OMP = Private)

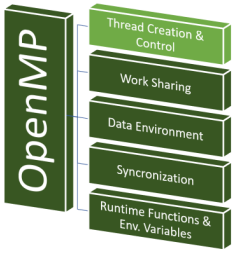
OpenMP Core Elements



OpenMP Core Elements



OpenMP – Thread Creation & Control



parallel construct

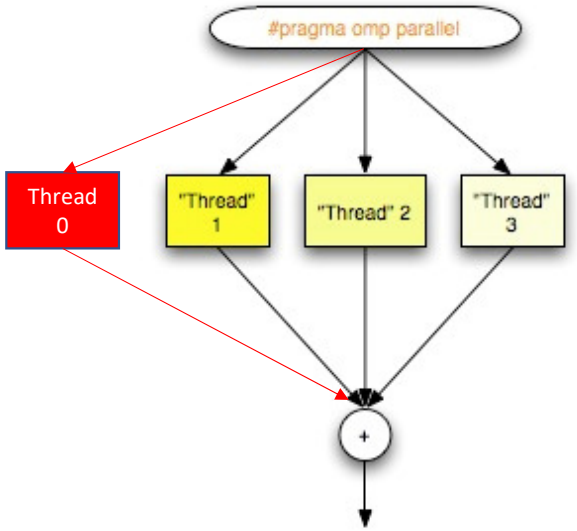
- Defines a parallel region, which is a region of the program that is to be executed by multiple threads in parallel.
- Is the fundamental construct that starts parallel execution.

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line structured-block
```

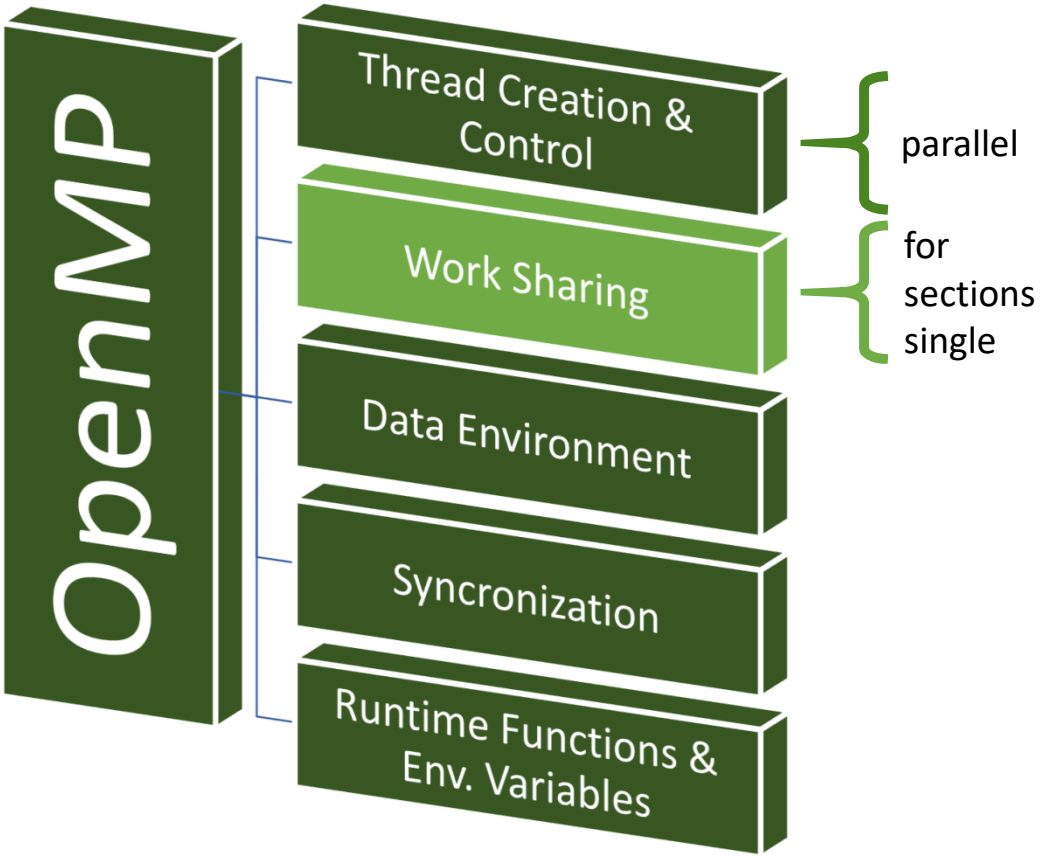
Clauses

if(scalar-expression)
private(variable-list)
firstprivate(variable-list)
default(shared | none)
shared(variable-list)
copyin(variable-list)
reduction(operator: variable-list)
num_threads(integer-expression)

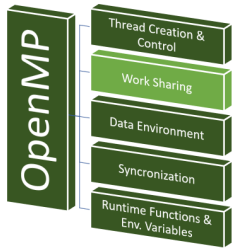
```
#pragma omp parallel
{
  //Parallelized statements
}
```



OpenMP Core Elements



OpenMP – Work Sharing



for construct

- Identifies an iterative work-sharing construct that specifies that the iterations of the associated loop will be executed in parallel.

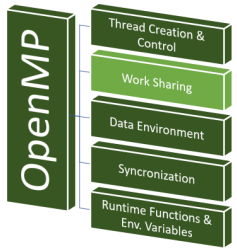
- The iterations of the **`for`** loop are distributed across threads that already exist in the team executing the parallel construct to which it binds.

```
#pragma omp for [clause[[,] clause] ... ] new-line for-loop
```

Clauses

private(variable-list)
firstprivate(variable-list)
lastprivate(variable-list)
reduction(operator: variable-list)
ordered
schedule(kind[, chunk_size])
nowait

OpenMP – Work Sharing



sections construct

- The **`sections`** directive identifies a noniterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team. Each section is executed once by a thread in the team.

```
#pragma omp sections [clause[[,] clause] ...] new-line
{
[#pragma omp section new-line]
structured-block
[#pragma omp section new-line
structured-block ]
...
```

Clauses

private(*variable-list***)**
firstprivate(*variable-list***)**
lastprivate(*variable-list***)**
reduction(*operator*: *variable-list***)**
nowait

Testing OpenMP…

How many threads will execute a “section” section? Which ones?

Write a small program to test it and answer the following questions:

1.- If my program has 4 sections and only 2 threads, which sections will be executed? Explain your answer.

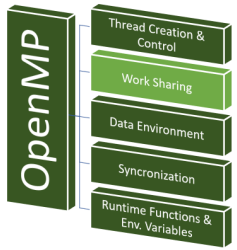
2.- If my program has now 2 sections and 4 threads, what will happen?

3.- Can ThreadID=0 be assigned with a section to process or only ThreadID≥1?

Testing OpenMP… Testing Code

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

void main() {

    #pragma omp parallel num_threads(8)
    {
        #pragma omp single
        printf("Total threads available (%d): %d \n\n", omp_get_thread_num(), omp_get_num_threads());
        #pragma omp sections nowait
     {
         #pragma omp section
         {
             Sleep(10);
             printf("Thread (%d), Hello from section 1\n\n", omp_get_thread_num());
         }
         #pragma omp section
         {
             Sleep(1000);
             printf("Thread (%d), Hello from section 2\n\n", omp_get_thread_num());
         }
         #pragma omp section
         {
             Sleep(1000);
             printf("Thread (%d), Hello from section 3\n\n", omp_get_thread_num());
         }
         #pragma omp section
         {
              Sleep(1000);
              printf("Thread (%d), Hello from section 4\n\n", omp_get_thread_num());
         }
     }
    }
    printf("Thread (%d) outside section\n", omp_get_thread_num());
}
```

OpenMP – Work Sharing



single construct

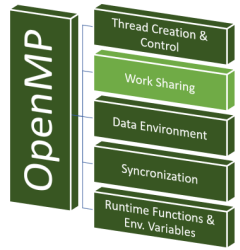
- The **`single`** directive identifies a construct that specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread).

```
#pragma omp single [clause[[,] clause] ...] new-line
structured-block
```

Clauses

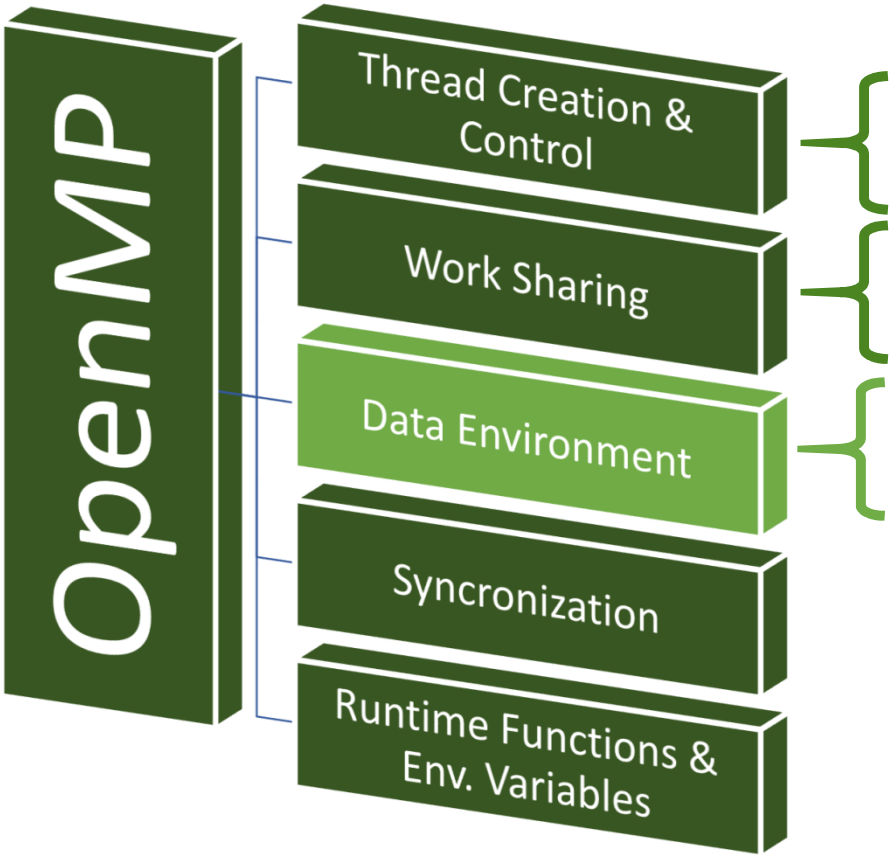
`private`(*variable-list*)
`firstprivate`(*variable-list*)
`copyprivate`(*variable-list*)
`nowait`

OpenMP – NOTES on Work Sharing Constructs



All worksharing constructs have an **implied barrier at the end**.

OpenMP Core Elements

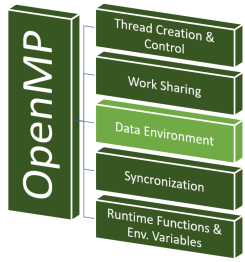


parallel

for
sections
single

threadprivate **private** **firstprivate** **lastprivate** **shared** **default** **reduction** **copyin** **copyprivate**

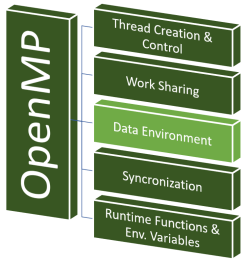
OpenMP – Data Environment



Important notes on variables:

- If a variable is visible when a parallel or work-sharing construct is encountered, and the variable is not specified in a sharing attribute clause or threadprivate directive, then the variable is shared.
- Static variables declared within the dynamic extent of a parallel region are shared.
- Heap allocated memory (for example, using malloc() in C or C++ or the new operator in C++) is shared.
- The pointer to previous point's memory, however, can be either private or shared.

OpenMP – Data Environment



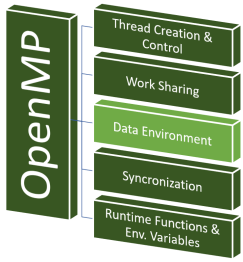
threadprivate directive

- The **threadprivate** directive specifies that variables are replicated, with each thread having its own copy.

```
#pragma omp threadprivate(variable-list)  new-line
```

- A **private** variable is local to a region and will be placed on the stack.
- A **threadprivate** variable on the other hand will be most likely placed in the heap.
- Each copy of an **omp threadprivate** data variable is **initialized once** prior to first use of that copy.
- A threadprivate variable persist across regions (depending on some restrictions).

OpenMP – Data Environment

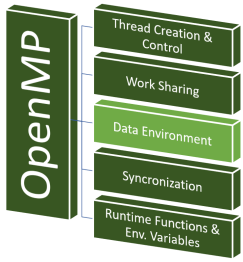


private clause

- The **`private`** clause declares the variables in *variable-list* to be private to each thread in a team.

`private``(`*variable-list*`)`

OpenMP – Data Environment



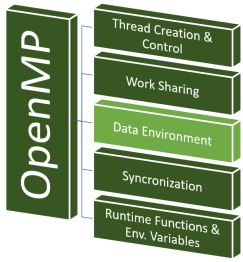
firstprivate clause

- The **`firstprivate`** clause provides a superset of the functionality provided by the **`private`** clause.

`firstprivate``(`*variable-list*`)`

- The main difference versus the **`private`** clause is that the value of the variable **is the value** of the original object that exists immediately prior to the parallel or the work-sharing construct.

OpenMP – Data Environment



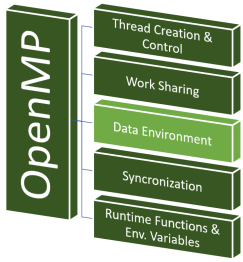
lastprivate clause

- The **`lastprivate`** clause provides a superset of the functionality provided by the **`private`** clause.

`lastprivate``(`*variable-list*`)`

- When a **`lastprivate`** clause appears on the directive that identifies a work-sharing construct, the value of each **`lastprivate`** variable from the sequentially last iteration of the associated loop, or the lexically last section directive, is assigned to the variable's original object.

OpenMP – Data Environment

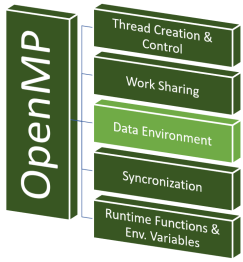


shared clause

- This clause shares variables that appear in the *variable-list* among all the threads in a team.
- All threads within a team access the same storage area for **`shared`** variables.

`shared``(`*variable-list*`)`

OpenMP – Data Environment



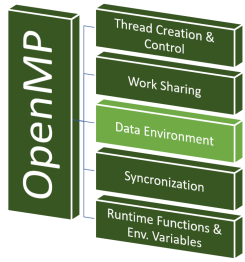
default clause

- The **`default`** clause allows the user to affect the data-sharing attributes of variables.

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

- Specifying **`default(shared)`** is equivalent to explicitly listing each currently visible variable in a **`shared`** clause, unless it is **`threadprivate`** or **`const`** qualified.
- In the absence of an explicit **`default`** clause, the default behavior is the same as if **`default(shared)`** were specified.

OpenMP – Data Environment



reduction clause

- This clause performs a reduction on the scalar variables that appear in *variable-list*, with the operator *op*.

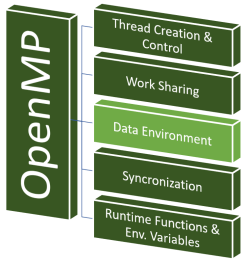
`reduction``(`*op* : *variable-list*`)`

x	= *x* *op* *expr*	
x	*binop*= *expr*	
x	= *expr* *op* *x*	*(except for subtraction)*
x`++`		
`++`*x*		
x`--`		
`--`*x*		

where:

x	One of the reduction variables specified in the *list*.
variable-list	A comma-separated list of scalar reduction variables.
expr	An expression with scalar type that does not reference *x*.
op	Not an overloaded operator but one of `+`, `*`, `-`, `&`, `^`, `|`, `&&`, or `||`.
binop	Not an overloaded operator but one of `+`, `*`, `-`, `&`, `^`, or `|`.

OpenMP – Data Environment

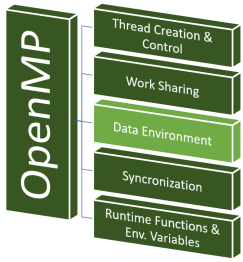


copyin clause

- The **`copyin`** clause provides a mechanism to assign the same value to **`threadprivate`** variables for each thread in the team executing the parallel region.
- For each variable specified in a **`copyin`** clause, the value of the variable in the master thread of the team is copied, as if by assignment, to the thread-private copies at the beginning of the parallel region.

`copyin``(`*variable-list*`)`

OpenMP – Data Environment



copyprivate clause

- The **`copyprivate`** clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members.
- It is an alternative to using a shared variable for the value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level).

`copyprivate``(`*variable-list*`)`

- The **`copyprivate`** clause can only appear on the **`single`** directive.

Exercise: Testing OpenMP… Data Environment

The global variable **var** has a race condition. Use **clauses** to correct the condition. Each thread wants its own copy of **var**, so you should try to make their different values persist from one parallel region to the other.

```
Initial value (0), Var=10

Thread (0), Var=10

Thread (1), Var=20

Thread (0), Var=1020

Thread (1), Var=1020
```

```c
#include <omp.h>
#include <stdio.h>
int var = 10;

void main()
{
    printf("Initial value (%d), Var=%d\n\n", omp_get_thread_num(), var);
    #pragma omp parallel num_threads (2)
    {
        var *= (omp_get_thread_num() + 1);  //Potential race condition.
        printf("Thread (%d), Var=%d\n\n", omp_get_thread_num(), var);
    }
    var += 1000;
    #pragma omp parallel num_threads (2)
    {
        printf("Thread (%d), Var=%d\n\n", omp_get_thread_num(), var);
    }
}
```

threadprivate
private
firstprivate
lastprivate
shared
default
reduction
copyin
copyprivate

Testing OpenMP… Data Environment

```
Initial value (0), Var=10

Thread (0), Var=10

Thread (1), Var=20

Thread (0), Var=1010

Thread (1), Var=20
```

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

int var = 10;
#pragma omp threadprivate (var) //Only valid for global or static data items

void main()
{
    printf("Initial value (%d), Var=%d\n\n", omp_get_thread_num(), var);
    #pragma omp parallel num_threads (2)
    {
        var *= (omp_get_thread_num() + 1);  //Potential race condition.
        printf("Thread (%d), Var=%d\n\n", omp_get_thread_num(), var);
    }
    var += 1000;
    #pragma omp parallel num_threads (2)
    {
        printf("Thread (%d), Var=%d\n\n", omp_get_thread_num(), var);
    }
}
```

Testing OpenMP… Data Environment

threadprivate
private
firstprivate
lastprivate
shared
default
reduction
copyin
copyprivate

In the second parallel region, I want to have the same value of the variable var. How can I do that?

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

int var = 10;
#pragma omp threadprivate (var)  //Only valid for global or static data items

void main()
{
    printf("Initial value (%d), Var=%d\n\n", omp_get_thread_num(), var);
    #pragma omp parallel num_threads (2)
    {
        var *= (omp_get_thread_num() + 1);  //Potential race condition.
        printf("Thread (%d), Var=%d\n\n", omp_get_thread_num(), var);
    }
    var += 1000;
    #pragma omp parallel num_threads (2)
    {
        printf("Thread (%d), Var=%d\n\n", omp_get_thread_num(), var);
    }
}
```

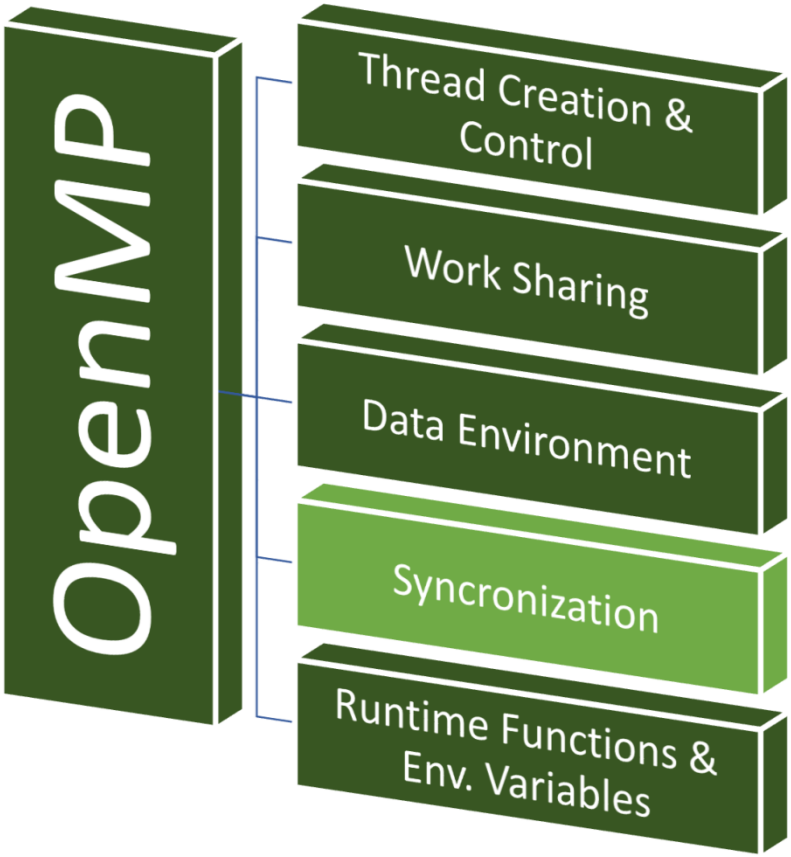
Testing OpenMP… Data Environment

In the second parallel region, I want to have the same value of the variable var. How can I do that?

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

int var = 10;
#pragma omp threadprivate (var)  //Only valid for global or static data items

void main()
{
    printf("Initial value (%d), Var=%d\n\n", omp_get_thread_num(), var);
    #pragma omp parallel num_threads (2)
    {
        var *= (omp_get_thread_num() + 1);  //Potential race condition.
        printf("Thread (%d), Var=%d\n\n", omp_get_thread_num(), var);
    }
    var += 1000;
    #pragma omp parallel copyin (var) num_threads (2)
    {
        printf("Thread (%d), Var=%d\n\n", omp_get_thread_num(), var);
    }
}
```

threadprivate
private
firstprivate
lastprivate
shared
default
reduction
copyin
copyprivate

OpenMP Core Elements



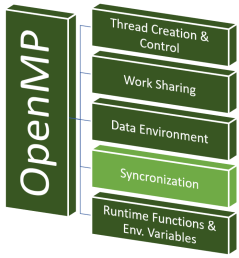
parallel

for
sections
single

threadprivate **private** **firstprivate** **lastprivate** **shared** **default** **reduction** **copyin** **copyprivate**

master atomic
critical flush
barrier ordered

OpenMP – Master and Synchronization



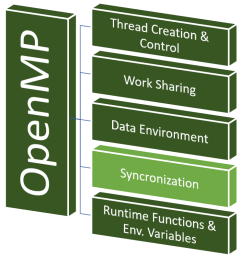
master construct

- The **`master`** directive identifies a construct that specifies a structured block that is executed by the master thread of the team.

```
#pragma omp master new-line
structured-block
```

- There is no implied barrier either on entry to or exit from the master construct.

OpenMP – Master and Synchronization



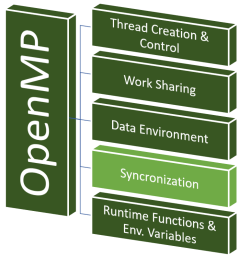
critical construct

- The **`critical`** directive identifies a construct that restricts execution of the associated structured block to a single thread at a time.

`#pragma omp critical` *[(name)] new-line*
structured-block

- An optional *name* may be used to identify the critical region. Identifiers used to identify a critical region have external linkage and are in a name space which is separate from the name spaces used by labels, tags, members, and ordinary identifiers.
- A thread waits at the beginning of a critical region until no other thread is executing a critical region (anywhere in the program) with the same name. All unnamed **`critical`** directives map to the same unspecified name.

OpenMP – Master and Synchronization



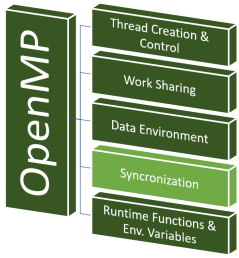
barrier construct

- The **`barrier`** directive synchronizes all the threads in a team. When encountered, each thread in the team waits until all of the others have reached this point.

```
#pragma omp barrier new-line
```

- After all threads in the team have encountered the barrier, each thread in the team begins executing the statements after the barrier directive in parallel

OpenMP – Master and Synchronization

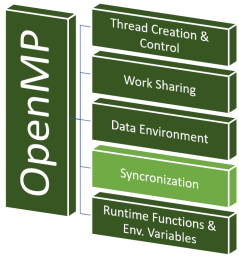


atomic construct

- The **`atomic`** directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

```
#pragma omp atomic new-line
expression-stmt
```

OpenMP – Master and Synchronization



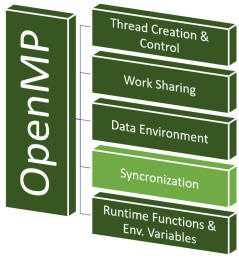
flush construct

- The **`flush`** directive specifies a "cross-thread"sequence point at which the implementation is required to ensure that all threads in a team have a consistent view of certain objects (specified below) in memory.
- Explicit form:

`#pragma omp flush` *[* **`(`***variable-list***`)`** *] new-line*

- A flush directive without a variable-list synchronizes all shared objects except inaccessible objects with automatic storage duration.

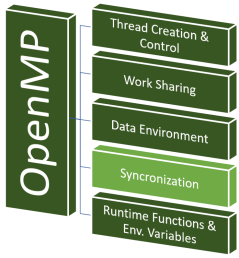
OpenMP – Master and Synchronization



flush construct

- In can also be implied (with no variable list):
  - At **`barrier`**
  - At entry to and exit from **`critical`**
  - At entry to and exit from **`ordered`**
  - At entry to and exit from **`parallel`**
  - At exit from **`for`**
  - At exit from **`sections`**
  - At exit from **`single`**
  - At entry to and exit from **`parallel for`**
  - At entry to and exit from **`parallel sections`**

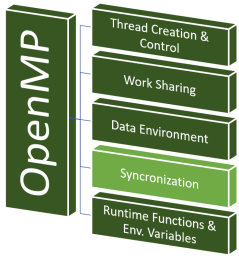
OpenMP – Master and Synchronization



flush construct

- **`flush`** directive **is not implied** for any of the following:
  - If a **`nowait`** clause is present.
  - At entry to **`for`**
  - At entry to or exit from **`master`**
  - At entry to **`sections`**
  - At entry to **`single`**

OpenMP – Master and Synchronization



ordered construct

- The structured block following an **`ordered`** directive is executed in the order in which iterations would be executed in a sequential loop.

```
#pragma omp ordered new-line
structured-block
```

- The **`ordered`** construct sequentializes and orders the execution of ordered regions **while allowing** code outside the region to run in parallel.

Excercise - #pragma omp ordered

Use directive **#pragma omp ordered** and clause **ordered** to print the results in the order they would be printed if executed secuentially.

```
#include <stdio.h>

void main()
{
        int i;
        #pragma omp parallel for num_threads(4)
        for (i = 0; i < 20; i++)
        {

                printf(" %d\n", i);
        }
}
```

```
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
0
1
2
3
4
```

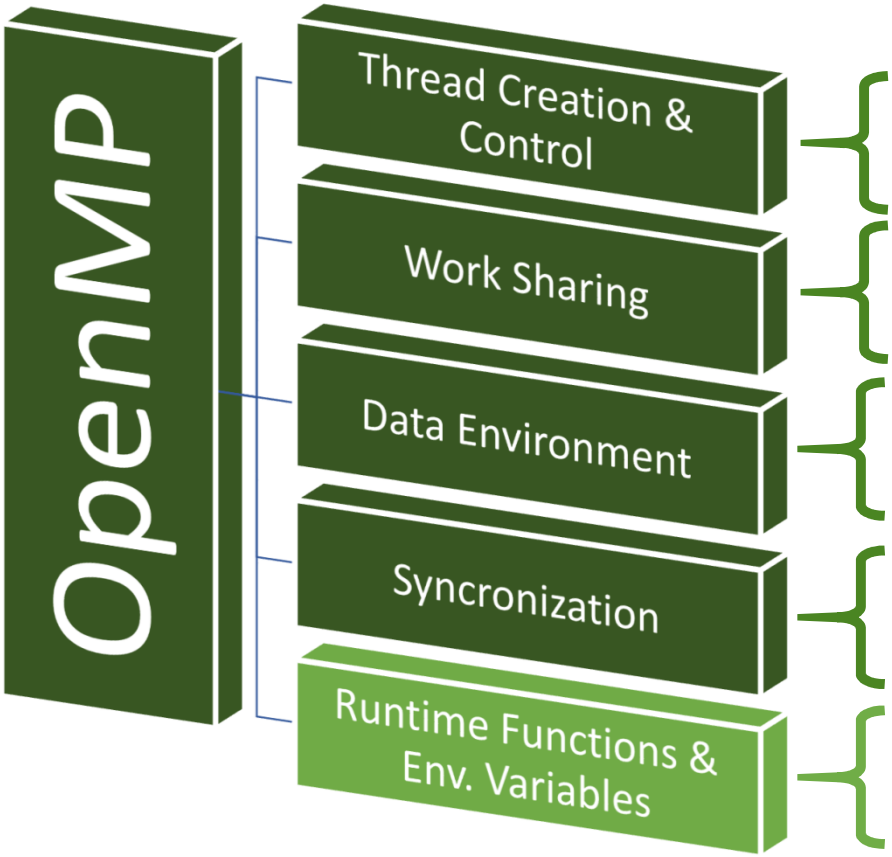
Excercise - #pragma omp ordered

Use directive #pragma omp ordered and clause ordered to print the results in the order they would be printed if executed secuentially.

```
#include <stdio.h>

void main()
{
        int i;
        #pragma omp parallel for ordered num_threads(4)
        for (i = 0; i < 20; i++)
        {
                #pragma omp ordered
                printf(" %d\n", i);
        }
}
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

OpenMP Core Elements



parallel

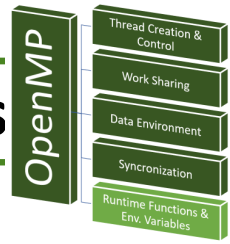
for
sections
single

threadprivate **private** **firstprivate** **lastprivate** **shared** **default** **reduction** **copyin** **copyprivate**

master atomic
critical flush
barrier ordered

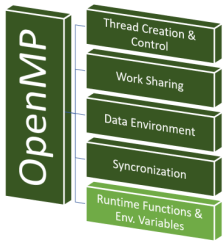
22 functions
4 environment variables

OpenMP – Functions and Environmental Variables



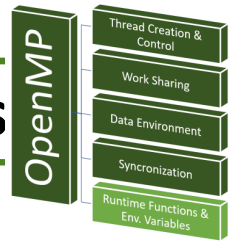
	Function	
Execution Environment		

OpenMP – Functions and Environmental Variables



	Function	
Execution Environment	omp_set_num_threads	sets the default number of threads to use for subsequent parallel regions
	omp_get_num_threads	returns the number of threads currently in the team executing the parallel region from which it is called.
	omp_get_max_threads	returns an integer that is guaranteed to be at least as large as the number of threads that would be used to form a team if a parallel region without a num_threads clause were to be encountered at that point in the code.
	omp_get_thread_num	returns the thread number, within its team, of the thread executing the function. The thread number lies between 0 and omp_get_num_threads()–1, inclusive. The master thread of the team is thread 0.
	omp_get_num_procs	returns the number of processors that are available to the program at the time the function is called.
	omp_in_parallel	returns a nonzero value if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0.
	omp_set_dynamic	enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.
	omp_get_dynamic	returns a nonzero value if dynamic adjustment of threads is enabled, and returns 0 otherwise.
	omp_set_nested	enables or disables nested parallelism.
	omp_get_nested	returns a nonzero value if nested parallelism is enabled and 0 if it is disabled.

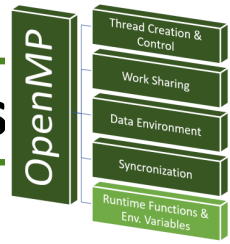
OpenMP – Functions and Environmental Variables



	Function	Nestable Function	Description
Lock			

	Function	Description
Timing		

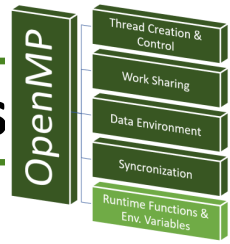
OpenMP – Functions and Environmental Variables



	Function	Nestable Function	Description
Lock	omp_init_lock	omp_init_nest_lock	initializes a simple/nestable lock.
	omp_destroy_lock	omp_destroy_nest_lock	removes a simple/nestable lock.
	omp_set_lock	omp_set_nest_lock	waits until a simple/nestable lock is available.
	omp_unset_lock	omp_unset_nest_lock	releases a simple/nestable lock.
	omp_test_lock	omp_test_nest_lock	tests a simple/nestable lock.

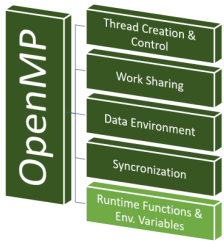
	Function	Description
Timing	omp_get_wtime	returns elapsed wall-clock time.
	omp_get_wtick	returns seconds between successive clock ticks.

OpenMP – Functions and Environmental Variables



Environmental Variable		

OpenMP – Functions and Environmental Variables



Environmental Variable		
OMP_SCHEDULE	sets the run-time schedule type and chunk size.	static,[chunk size] dynamic,[chunk size] guided,[chunk size]
OMP_NUM_THREADS	sets the number of threads to use during execution.	Positive integer
OMP_DYNAMIC	enables or disables dynamic adjustment of the number of threads.	TRUE | FALSE
OMP_NESTED	enables or disables nested parallelism.	TRUE | FALSE (default)

NESTED

Normally DISABLED by default. You can enable it using a function or an environment variable.

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


void main()
{
    //omp_set_nested(1);
    #pragma omp parallel num_threads(2)
    {
        printf("Level 1: number of threads in the team - %d\n", omp_get_num_threads());
        #pragma omp parallel num_threads(2)
        {
            printf("Level 2: number of threads in the team - %d\n", omp_get_num_threads());
            #pragma omp parallel num_threads(2)
            {
                printf("Level 3: number of threads in the team - %d\n", omp_get_num_threads());
            }
        }
    }
}
```

NESTED

Nested disabled…

Microsoft Visual Studio Debug Console

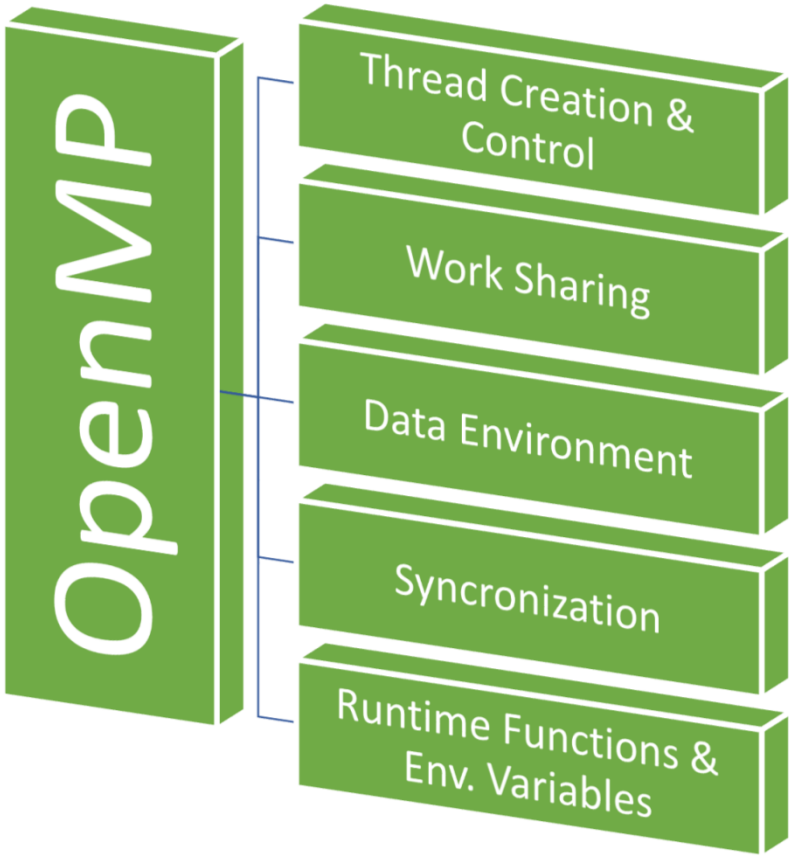
```
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
```

Nested enabled…

Microsoft Visual Studio Debug Console

```
Level 1: number of threads in the team - 2
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
```

OpenMP – Questions



See you soon…