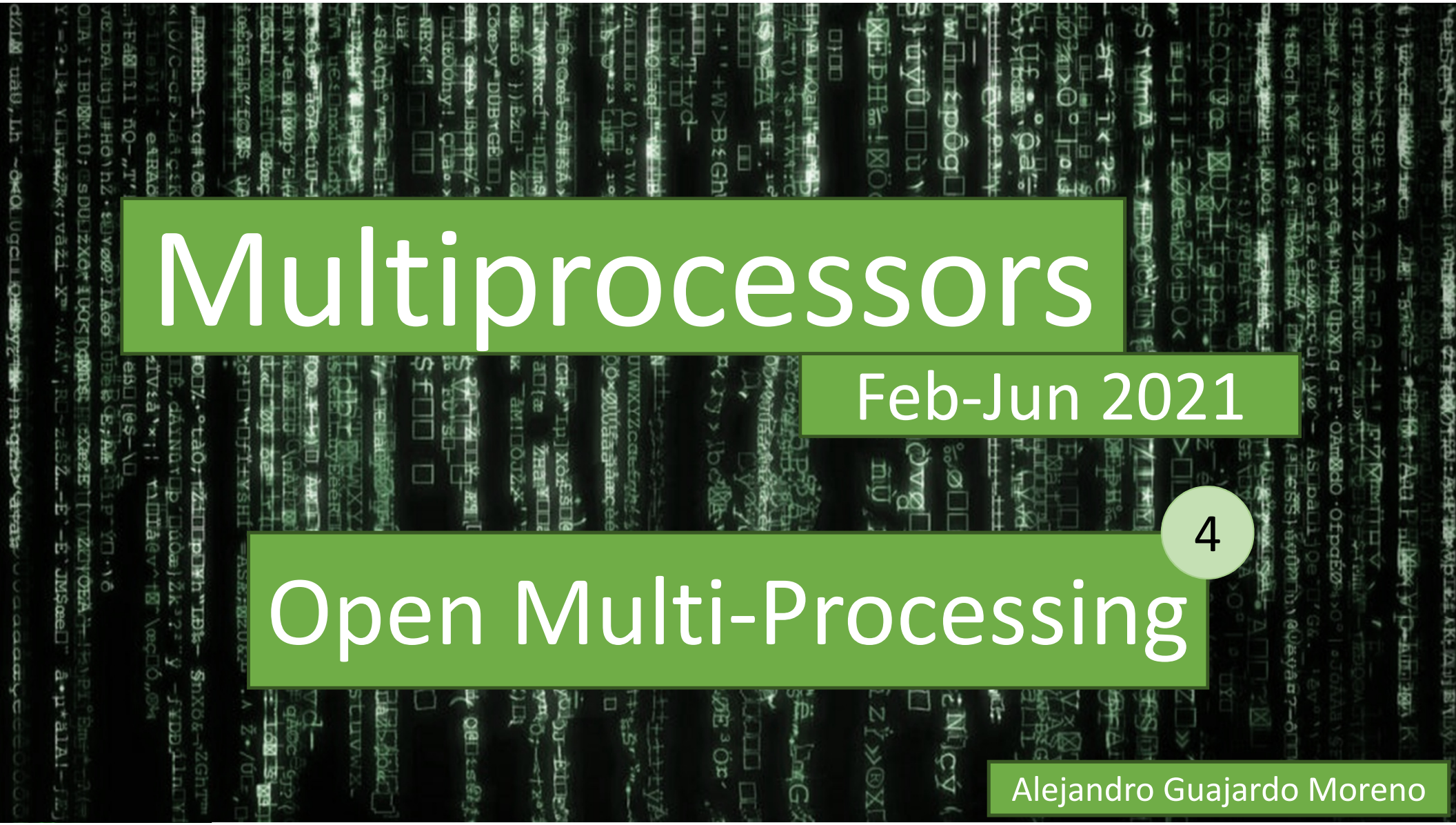


Multiprocessors

Feb-Jun 2021

4

Open Multi-Processing

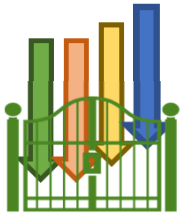
Alejandro Guajardo Moreno

Recap… OpenMP Synchronization

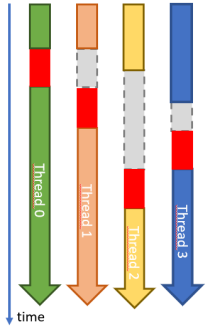
Synchronization - bringing two or more threads to a known and well defined point in their execution.

The 2 most used flavors of synchronization:

- Barrier

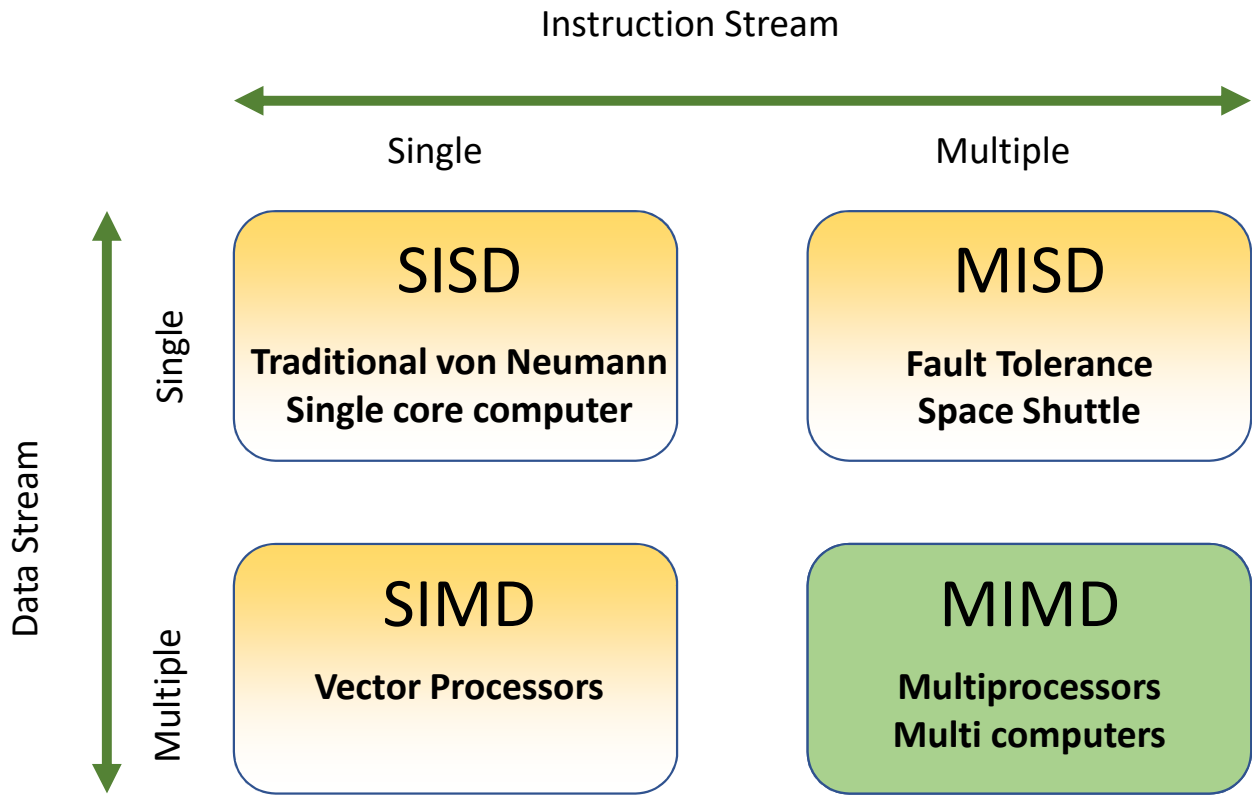


- Mutual Exclusion
  - Critical
  - Atomic



Synchronization is **EXPENSIVE**!

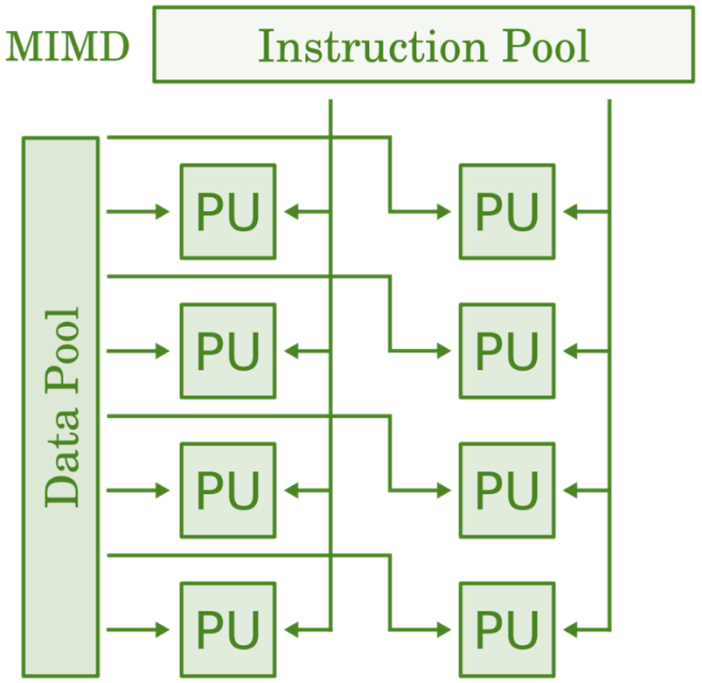
Flynn´s Taxonomy



OpenMP Foucs: **MIMD**

SPMD (MIMD)

- **SPMD (Single Program, Multiple Data)**
- It is a subcategory of MIMD.
- Is the most common style of parallel programming.
- Tasks are split up and run simultaneously on multiple processors.
- Multiple autonomous processors simultaneously execute the same program at independent points
- Use general purpose CPUs.



OpenMP - Worksharing

Single construct shared between the threads.

Worksharing

Loop Construct
Sections/Section Constructs
Single Construct
Task Construct

OpenMP – Worksharing – LOOP CONSTRUCT

```
for (i = 0; i < SIZE; i++)
   A[i] += B[i];
```

Until today, we would write it this way:

Worksharing

Loop Construct

Sections/Section Constructs

Single Construct

Task Construct

```
#pragma omp parallel
{
   int threadID, threadAmount, i, threadStart, threadEnd;
   threadID = omp_get_thread_num();
   threadAmount = omp_get_num_threads();
   threadStart = threadID * SIZE / threadAmount;
   threadEnd = (threadID + 1) * SIZE / threadAmount;
   if (threadID == threadAmount – 1) threadEnd = SIZE;
   for (i = threadStart; i < threadEnd; i++)
         A[i] += B[i];
}
```

OpenMP – Worksharing – LOOP CONSTRUCT

- I want to split iterations of the **for** loop to the different threads.
- **I don´t want to manually distribute the iterations.**

Worksharing

Loop Construct
Sections/Section Constructs
Single Construct
Task Construct

Construct: #pragma omp for

```
for (i = 0; i < SIZE; i++)
   A[i] += B[i];
```

```
#pragma omp parallel
{
     #pragma omp for
        for (i = 0; i < SIZE; i++)
           A[i] += B[i];
}
```

❖ In OpenMP, the loop control index on a parallel loop is **private** to the thread.

OpenMP – Loop carried dependencies

- ✓ Happens when an iteration of a loop uses a value from a previous iteration.

```
int x,y;
float A[SIZE];
y = 10;
for (x = 0; x < SIZE; x++)
{
  y += 5;
  A[x] = calculation(y);
}
```

```
int x;
float A[SIZE];
#pragma omp parallel
{
  #pragma omp for
  for (x = 0; x < SIZE; x++)
  {
    int y = 10 + 5(x + 1);
    A[x] = calculation(y);
  }
}
```

- ✓ Loop carried dependencies are no-go for parallelization.
- ✓ You have to rewrite the loop and eliminate the loop carried dependency.
- ✓ Rewrite the code in terms of the control variable.

OpenMP – Loop carried dependencies

Recognize this?

```
for (i = 0; i < steps; i++)
{
    x = (i + 0.5) * base;
    fdx = 4 / (1 + x * x);
    acum += fdx;
}
```

Loop carried dependency!

These loop carried dependencies have a name:

Reductions!

OpenMP has **built-in** REDUCTION!

Reduction

Syntax:

reduction (op:list)

A **local** copy of each variable in (list) is created and initialized depending on the operation (op).

At the end, all local copies are combined.

OpenMP – Worksharing – LOOP Construct - Reductions

Syntax:

reduction (op:list)

reduction (op:var1,var2)

Initial values for reduction operands.

Mathematically make sense.

Operator	Initialization
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

OpenMP – Worksharing – LOOP Construct - Reductions

```
for (i = 0; i < steps; i++)
{
    x = (i + 0.5) * base;
    fdx = 4 / (1 + x * x);
    acum += fdx;
}
```

```
#pragma omp parallel
{
    #pragma omp for reduction (+:acum)
    for (i = 0; i < steps; i++)
    {
        x = (i + 0.5) * base;
        fdx = 4 / (1 + x * x);
        acum += fdx;
    }
}
```

Exercise

1.- Instead of manually handling the threads, insert an OMP Loop Construct and let the compiler do the work sharing.

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

long cantidadIntervalos = 1000000000;
double baseIntervalo;
double fdx;
double acum = 0;
clock_t start, end;

void main() {
   double x=0;
   long i;
   baseIntervalo = 1.0 / cantidadIntervalos;
   start = clock();
   for (i = 0; i < cantidadIntervalos; i++) {
      x = (i+0.5)*baseIntervalo;
      fdx = 4 / (1 + x * x);
      acum += fdx;
   }
   acum *= baseIntervalo;
   end = clock();
   printf("Result = %20.18lf (%ld)\n", acum, end - start);
}
```

- Use the original serial program.
- Parallelize it with an OMP LOOP Construct.
- You will use:
  - OpenMP PARALLEL directive.
    - #pragma omp parallel
  - FOR clause
    - #pragma omp for
  - REDUCTION on the accumulator.
    - reduction (op:list)
- Minimize the number of changes made.

Exercise – Code used

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


long cantidadIntervalos = 1000000000;
double baseIntervalo;
//double fdx;
double acum = 0;

clock_t start, end;

void main() {

   //double x = 0; //Has to be local to the threads.

   long i;
   baseIntervalo = 1.0 / cantidadIntervalos;
   start = clock();
```

Variable moved. Has to be private to each thread.

Variable moved

Exercise – Code used

```
  #pragma omp parallel
   {
      double fdx; //Has to be local to the threads.
      double x; //Has to be local to the threads.
      #pragma omp for reduction(+:acum)
         for (i = 0; i < cantidadIntervalos; i++) {
            x = (i + 0.5) * baseIntervalo;
            fdx = 4 / (1 + x * x);
            acum += fdx;
         }
      }
      acum *= baseIntervalo;
      end = clock();
      printf("Result = %20.18lf (%ld)\n", acum, end - start);
}
```

Directive Inserted

Variable Moved

Variable Moved

Directive Inserted

OpenMP - Shared and Private Data IMPLICIT

A note on Shared and Private Data…

```
#include <stdio.h>

int main() {

   type global_variable = value;

   #pragma omp parallel
   { //Parallel region begins

      type private_variable = value;

      printf("Hello OMP World!...\n");
   } //Parallel region ends

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

}
```

Shared

Private

OpenMP - Shared and Private Data EXPLICIT

Explicit SHARED

Explicit PRIVATE

```
#include <stdio.h>

int main() {

   int i, n, a, b;

   #pragma omp parallel shared (n, a) private (i, b)
   { //Parallel region begins

      for (i = 0; i < n; i++) {
         b = a + i;
      }
   } //Parallel region ends

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

}
```

OpenMP – Example using PRIVATE

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

long cantidadIntervalos = 1000000000;
double baseIntervalo, fdx, acum = 0; //fdx will be local
clock_t start, end;

void main() {

   double x = 0; //Will be local, using PRIVATE.
   long i;
   baseIntervalo = 1.0 / cantidadIntervalos;
   start = clock();
```

OpenMP – Example using PRIVATE

```
  #pragma omp parallel private (fdx,x)
   {
      #pragma omp for reduction(+:acum)
         for (i = 0; i < cantidadIntervalos; i++) {
            x = (i + 0.5) * baseIntervalo;
            fdx = 4 / (1 + x * x);
            acum += fdx;
         }
      }
      acum *= baseIntervalo;
      end = clock();
      printf("Result= %20.18lf (%ld)\n", acum, end-start);
}
```

fdx, x local to each thread.

OpenMP – #pragma omp parallel & #pragma omp for

You can merge both constructs into one:

```
#pragma omp parallel
{
   #pragma omp for
      for (i = 0; i < 100000; i++)
      A[i] += B[i];
}
```

```
#pragma omp parallel for
   for (i = 0; i < 100000; i++)
      A[i] += B[i];
```

This is true if #pragma omp for is the ONLY thing inside the brackets of #pragma omp parallel.

Exercise

1.- Use only **ONE** pragma.

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

long cantidadIntervalos = 1000000000;
double baseIntervalo;
double fdx;
double acum = 0;
clock_t start, end;

void main() {
   double x=0;
   long i;
   baseIntervalo = 1.0 / cantidadIntervalos;
   start = clock();
   for (i = 0; i < cantidadIntervalos; i++) {
      x = (i+0.5)*baseIntervalo;
      fdx = 4 / (1 + x * x);
      acum += fdx;
   }
   acum *= baseIntervalo;
   end = clock();
   printf("Result = %20.18lf (%ld)\n", acum, end - start);
}
```

- Use the original serial program.
- Parallelize it with an OMP LOOP Construct.
- You will use:
  - OpenMP PARALLEL directive.
  - FOR clause
  - REDUCTION on the accumulator.
- Minimize the number of changes made.

Exercise – Code used

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

long cantidadIntervalos = 1000000000;
double baseIntervalo;
double fdx;
double acum = 0;
clock_t start, end;

void main() {
   double x = 0;
   long i;
   baseIntervalo = 1.0 / cantidadIntervalos;
   start = clock();

   #pragma omp parallel for reduction (+:acum) private (fdx, x)
   for (i = 0; i < cantidadIntervalos; i++) {
      x = (i + 0.5) * baseIntervalo;
      fdx = 4 / (1 + x * x);
      acum += fdx;
   }
   acum *= baseIntervalo;
   end = clock();
   printf("Result = %20.18lf (%ld)\n", acum, end - start);
}
```

OMP Directive + Clauses

OpenMP – Worksharing – LOOP CONSTRUCT - Schedule

- Compilers are “dumb”*.
- Programmer has to help.
- The problem… How to distribute the work the most efficient way possible?
- #pragma omp parallel for **schedule(kind [,chunk size])**

Optional

- If SCHEDULE clause is not present, the default Schedule is implementation-defined.

* Not really, they always play safe.

OpenMP – Worksharing – LOOP CONSTRUCT - Schedule

➢#pragma omp parallel for **schedule(kind [,chunk size])**

Schedule kind:

Kind	Description
static	Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is loop_count/number_of_threads.Set chunk to 1 to interleave the iterations.
dynamic	Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, the chunk size is 1. Be careful when using this scheduling type because of the extra overhead involved.
guided	Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies them minimum size chunk to use. By default the chunk size is approximately loop_count/number_of_threads.

OpenMP – Worksharing – LOOP CONSTRUCT - Schedule

Static vs Dynamic… when to use each?

STATIC

- Pre-determined or known by the programmer.
- Least work at runtime.
- Scheduling done at COMPILE time.

DYNAMIC

- Unpredictable, highly variable work per iteration.
- Most work at runtime.
- Complex scheduling done at RUNTIME.

OpenMP – Worksharing – LOOP CONSTRUCT - Schedule

Another schedule kind: runtime

Kind	Description
static	Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is loop_count/number_of_threads.Set chunk to 1 to interleave the iterations.
dynamic	Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, the chunk size is 1. Be careful when using this scheduling type because of the extra overhead involved.
guided	Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies them minimum size chunk to use. By default the chunk size is approximately loop_count/number_of_threads.
runtime	Uses the OMP_schedule environment variable to specify which one of the three loop-scheduling types should be used. OMP_SCHEDULE is a string formatted exactly the same as would appear on the parallel construct.

No chunk_size

OpenMP – Worksharing – LOOP CONSTRUCT - Schedule

- Can change schedule without recompiling.
- #pragma omp parallel for schedule**(runtime)**
- Uses environment variable **OMP_SCHEDULE**
  - **set OMP_SCHEDULE=kind,chunk**
  - Ej: set OMP_SCHEDULE=static,1000
  - set OMP_SCHEDULE=dynamic
  - set OMP_SCHEDULE=guided,4
  - **export OMP_SCHEDULE=kind,chunk**
  - Ej: export OMP_SCHEDULE=static,1000
  - export OMP_SCHEDULE=dynamic
  - export OMP_SCHEDULE=guided,4

Remarks

The default value in the Visual C++ implementation of the OpenMP standard is `OMP_SCHEDULE=static,0`.

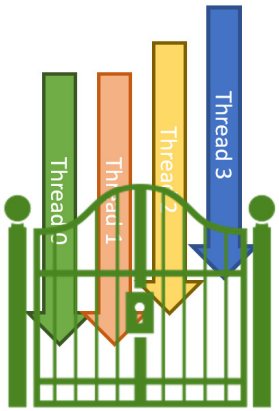
OpenMP – Revisiting Barriers – NOWAIT (EXPLICIT)

OpenMP Synchronization - Barrier

- No thread can proceed past a barrier **until** all the other threads have arrived.
- Syntax C/C++:

```
#pragma omp barrier
```

- Either all threads or none must encounter the barrier: otherwise **DEADLOCK**!!



Allows us to place **EXPLICIT** barriers where needed.

OpenMP – Revisiting Barriers – NOWAIT (IMPLICIT)

```
#pragma omp parallel
{
   int threadID = omp_get_thread_num();
   A[threadID] = functionX(threadID);
   #pragma omp barrier
   #pragma omp for
   for (i = 0; i < SIZE; i++)
      C[threadID] = functionY(threadID,A);
   #pragma omp for nowait
   for (i = 0; i < SIZE; i++)
      B[threadID] = functionZ(threadID,C);
   A[threadID] = functionAA(threadID);
}
```

Explicit Barrier

Implicit Barrier

Implicit Barrier (overridden)

Implicit Barrier at end of parallel region (CANNOT be overridden)

Remember… Synchronization is **EXPENSIVE**!

If we can skip a synchronization, it is worth it!

If we as programmers are sure that our ALGORITHM allows us NOT to do a synchronization in a #pragma omp for:

#pragma omp for nowait

OpenMP – MASTER Construct

The **master** construct:

- Specifies a structured block that is executed by the master thread of the team.
- The syntax of the **master** directive is as follows:

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

- Other threads in the team **do not execute** the associated structured block.
- There is **no implied barrier** either on entry to or exit from the master construct.

OpenMP – MASTER Construct

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

int main() {
int sharedVar = 6;
#pragma omp parallel num_threads(10)
{
   int privateVar = omp_get_thread_num(); //This data is mine!
   sharedVar = omp_get_thread_num(); //I´ll leave my mark!
   Sleep(2);

   #pragma omp master  //Only 1 print of the final value by the Master Thread
   {
   printf("Thread (%d), privateVar = %d, sharedVar= %d \n",
omp_get_thread_num(), privateVar, sharedVar);
   }
}
}
```

OpenMP – SINGLE Construct

The **single** construct:

- Specifies that the associated structured block is executed by only one thread in the team.
- Not necessarily the master thread .
- The syntax of the **single** directive is as follows:

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

- **There is an implicit barrier** after the **single** construct.
- Allows the use of nowait.

OpenMP – SINGLE Construct

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

int main() {
int sharedVar = 6;
#pragma omp parallel num_threads(10)
{
   int privateVar = omp_get_thread_num(); //This data is mine!
   sharedVar = omp_get_thread_num(); //I´ll leave my mark!
   Sleep(2);

   #pragma omp single  //Any thread, print the final value
   {
   printf("Thread (%d), privateVar = %d, sharedVar= %d \n",
omp_get_thread_num(), privateVar, sharedVar);
   }
}
}
```

Exercise

1.- Play with the different scheduling options. What are your conclusions?

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

long cantidadIntervalos = 1000000000;
double baseIntervalo;
double fdx;
double acum = 0;
clock_t start, end;

void main() {
   double x = 0;
   long i;
   baseIntervalo = 1.0 / cantidadIntervalos;
   start = clock();

   #pragma omp parallel for reduction (+:acum) private (fdx, x)
   for (i = 0; i < cantidadIntervalos; i++) {
      x = (i + 0.5) * baseIntervalo;
      fdx = 4 / (1 + x * x);
      acum += fdx;
   }
   acum *= baseIntervalo;
   end = clock();
   printf("Result = %20.18lf (%ld)\n", acum, end - start);
}
```

- Use the OMP code with just 1 directive.
  - Use the SCHEDULE clause with:
    - schedule (static,chunk size)
    - schedule (dynamic,chunk size)
    - schedule (guided,chunk size)
    - schedule (runtime)
  - set OMP_SCHEDULE=kind,chunk

OpenMP – Worksharing – LOOP CONSTRUCT - Schedule

```
#pragma omp parallel for reduction (+:acum) private (fdx, x) schedule (static)

#pragma omp parallel for reduction (+:acum) private (fdx, x) schedule (static, cantidadIntervalos)

#pragma omp parallel for reduction (+:acum) private (fdx, x) schedule (static, cantidadIntervalos/2)

#pragma omp parallel for reduction (+:acum) private (fdx, x) schedule (static, cantidadIntervalos/4)

#pragma omp parallel for reduction (+:acum) private (fdx, x) schedule (static, cantidadIntervalos/8)

#pragma omp parallel for reduction (+:acum) private (fdx, x) schedule (dynamic)

#pragma omp parallel for reduction (+:acum) private (fdx, x) schedule (dynamic, 1)

#pragma omp parallel for reduction (+:acum) private (fdx, x) schedule (dynamic, 2000)

#pragma omp parallel for reduction (+:acum) private (fdx, x) schedule (runtime) |set OMP_SCHEDULE=static,2

#pragma omp parallel for reduction (+:acum) private (fdx, x) schedule (runtime) | set OMP_SCHEDULE=guided,2
```

OpenMP – SECTIONS/SECTION Directive

The **sections/section** directive:

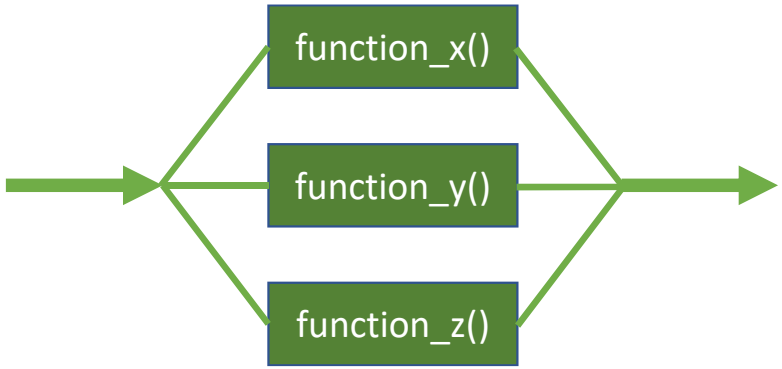
- Identifies a **noniterative** work-sharing construct that specifies a set of constructs that are to be divided among threads in a team.
- Each section is executed once by a thread in the team.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            function_x();
        #pragma omp section
            function_y();
    }
}
```

- There is an implicit barrier after the **sections** construct.
- Allows the use of nowait.

OpenMP – SECTIONS/SECTION Construct Example

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            function_x();
        #pragma omp section
            function_y();
        #pragma omp section
            function_z();
    }
}
```



OpenMP – #pragma omp parallel & #pragma omp sections

You can merge both constructs into one:

```
#pragma omp parallel
{
   #pragma omp sections
   {
      //Add
      #pragma omp section
```

```
#pragma omp parallel sections
{
   //Add
   #pragma omp section
```

This is true if #pragma omp sections is the ONLY thing inside the brackets of #pragma omp parallel.

For NEXT class…

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.

See you soon…


A nice week have you, padawan learners.
Mindful of the future always be, at the expense of the moment never be.