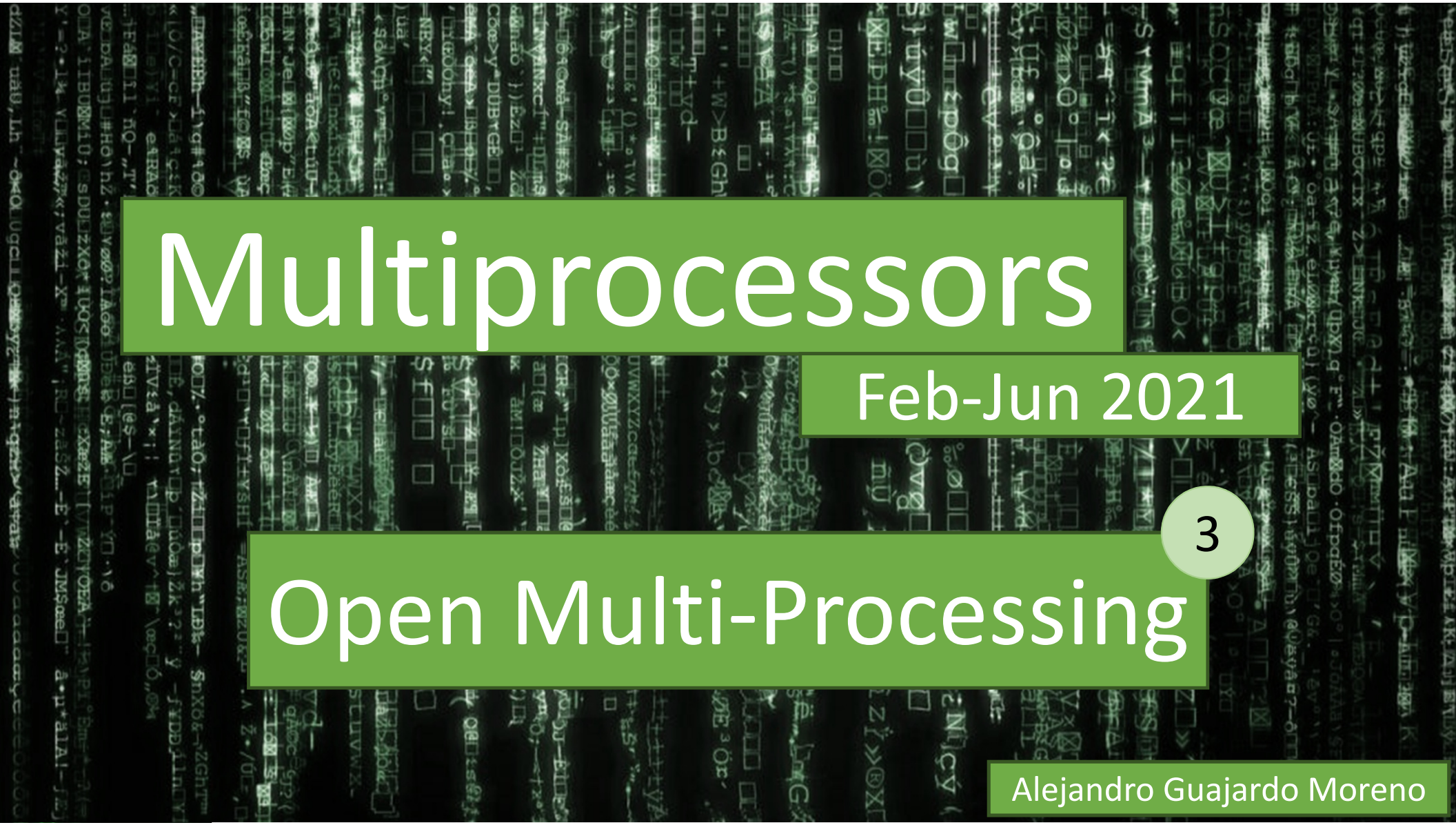


Multiprocessors

Feb-Jun 2021

3

Open Multi-Processing

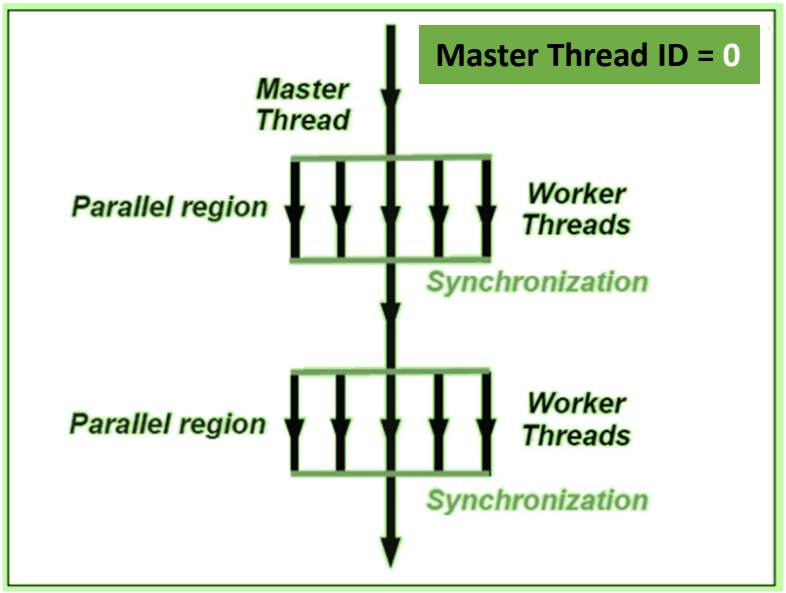
Alejandro Guajardo Moreno

Fork-join model

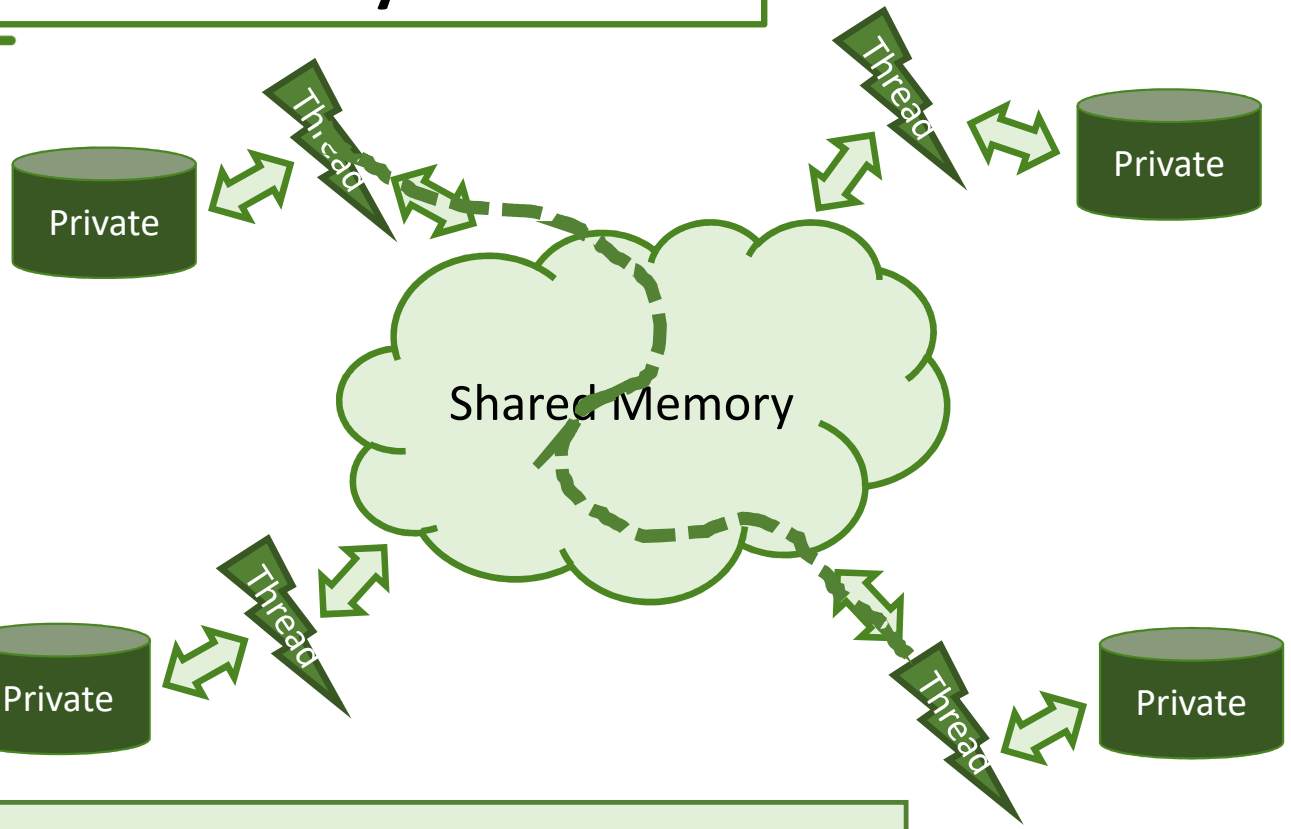
- The model uses one main thread and forks to multithread execution.
- The model may use several blocks of parallelization.

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)

How did you do on your homework?

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

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

   void main() {
      double x;
      long i;
      baseIntervalo = 1.0 / cantidadIntervalos;
      start = clock();
      for (i = 0, x = 0.0; i < cantidadIntervalos; i++) {
         fdx = 4 / (1 + x * x);
         acum = acum + (fdx * baseIntervalo);
         x = x + baseIntervalo;
      }
   end = clock();
   printf("Resultado = %20.18lf (%ld)\n", acum, end - start);
}
```

You only need to use:

Directive
#pragma omp parallel

Functions
omp_set_num_threads
omp_get_num_threads
omp_get_thread_num

Global and private variables

OpenMP PI Calculation: Important notes

```
#include <stdio.h>
#include <time.h>
                    <================ Need to include <omp.h>
   long cantidadIntervalos = 1000000000;
   double baseIntervalo;
   double fdx; <= Global variable has to be converted to private.
   double acum = 0; <= Global variable has to be converted to private or need for a partial sum variable.
   clock_t start, end;

   void main() {
      double x;
      long i;
      baseIntervalo = 1.0 / cantidadIntervalos;
      start = clock();
      for (i = 0, x = 0.0; i < cantidadIntervalos; i++) {
         fdx = 4 / (1 + x * x);
         acum = acum + (fdx * baseIntervalo);
         x = x + baseIntervalo;
      }
   end = clock();
   printf("Resultado = %20.18lf (%ld)\n", acum, end - start);
}
```

<= This needs to change, depending on your algorithm. More on that…

<= b*h1 + b*h2 + b*h3… = b*(h1+h2+h3+..)

OpenMP PI Calculation: Important notes

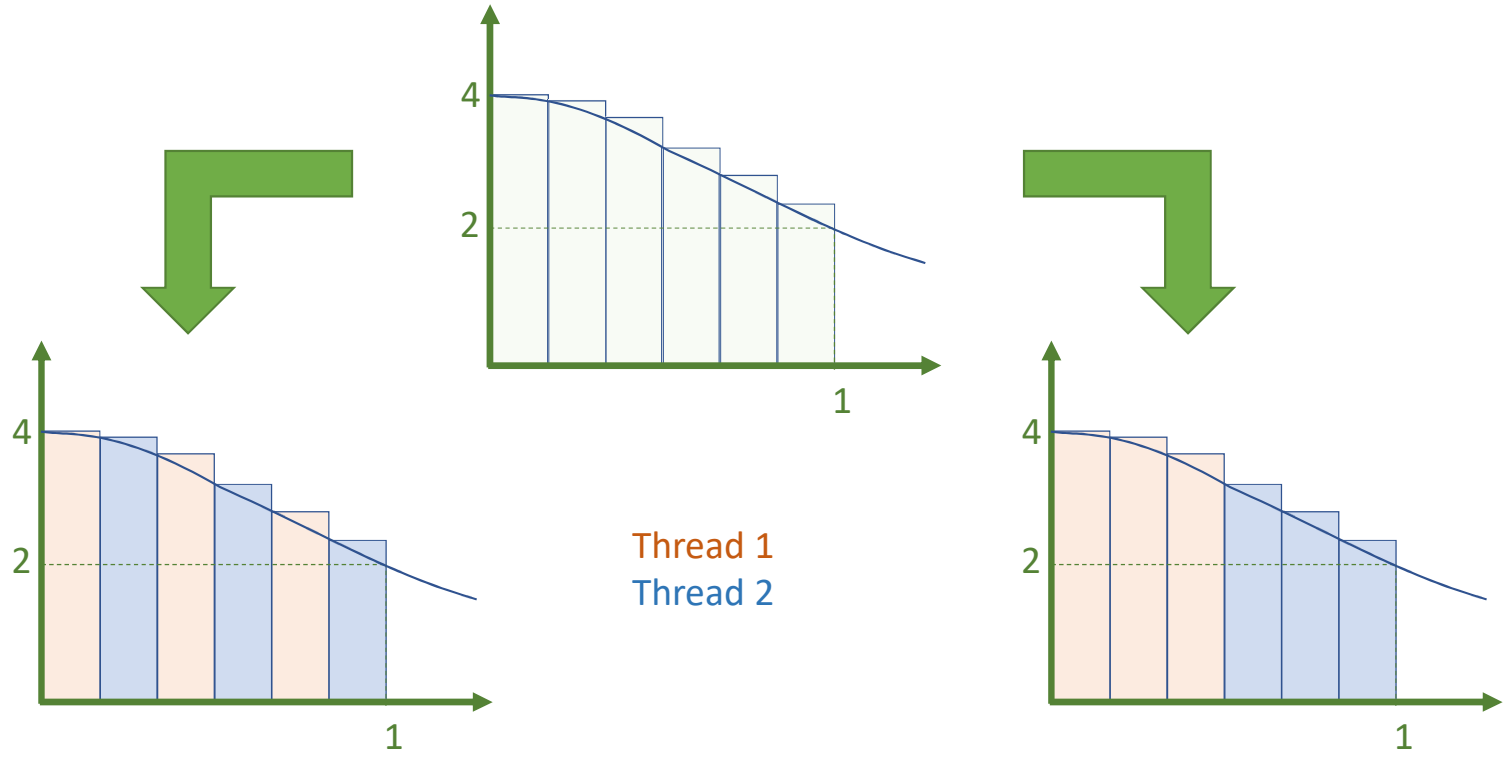
```
#include <stdio.h>
#include <time.h>
#include <omp.h>  //Added include file
#define MAXTHREADS 4 //Defined my desired thread count.
long cantidadIntervalos = 1000000000; // 1 B
double baseIntervalo;
//double fdx; //Can´t be global variable.
//double acum = 0; //Can´t be global variable.
clock_t start, end;

void main() {
   int THREADS = MAXTHREADS;
   //long i;  //Can´t be global variable.
   baseIntervalo = 1.0 / (double)cantidadIntervalos;

   //double x;  //Can´t be global variable.
   double partialSum[MAXTHREADS]; //Defined an array. One element for each thread.
   double totalSum = 0; //Defined my final accumulator.
   omp_set_num_threads(THREADS); //Set my desired amount of threads.
   start = clock();
```

OpenMP PI Calculation: Important notes

```
#pragma omp parallel
   {
   int numThread = omp_get_thread_num();
   double acum = 0; //Was SHARED variable, changed to be a PRIVATE.
   double fdx = 0; //Was SHARED variable, changed to be a PRIVATE.
   double x = 0;  //Can´t be global variable, changed to PRIVATE.
   for (long i = numThread; i < cantidadIntervalos; i += THREADS) {
      x = i * baseIntervalo;
     fdx = 4 / (1 + x * x);
     acum += fdx;
     }
   acum *= baseIntervalo; //Multiply all the heights by the base size.
   partialSum[numThread] = acum;
   printf("Resultado parcial (Thread %d)\nacum = %lf\n", numThread, acum);
   }
end = clock();

for (int c = 0; c < THREADS; c++)
totalSum += partialSum[c];
printf("\nResultado (%d threads) = %20.18lf (%ld)\n", THREADS, totalSum, end - start);
}
```

OpenMP PI Calculation: Important notes (Algorithm)

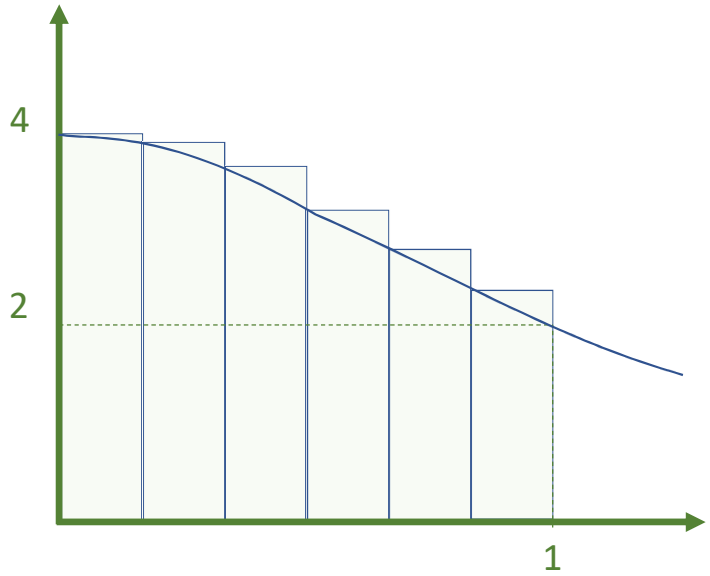
```
for (i = 0, x = 0.0; i < cantidadIntervalos; i++) // Important changes
```

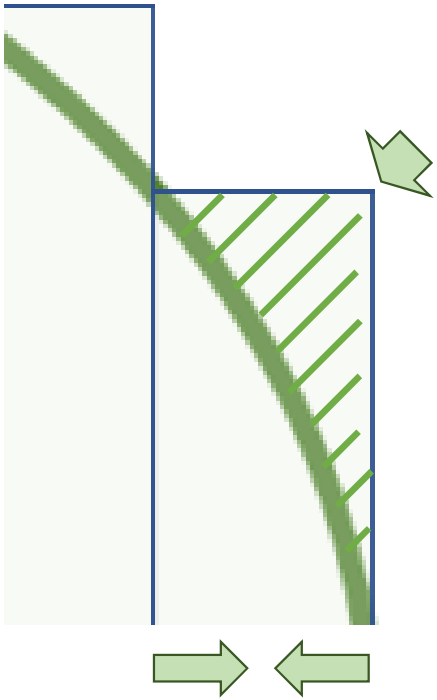


OpenMP PI Calculation: Important notes (Algorithm)

Improvements on **precision & performance**.

Why do we want to increase our **Interval number**?



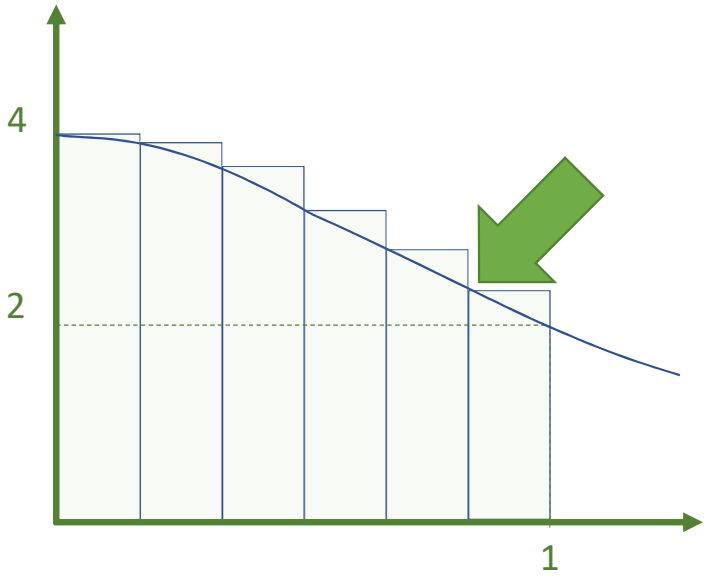


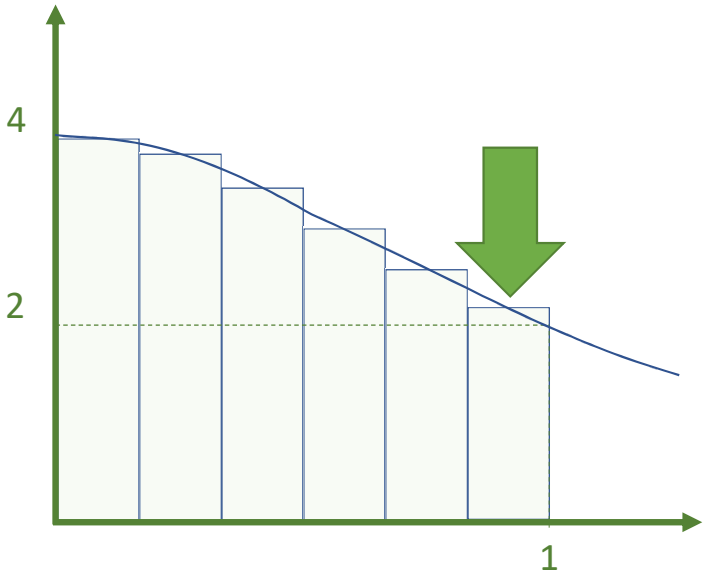
Reduce size of base:

- ✓ Reduce error
- ❖ Increment total number of steps.

OpenMP PI Calculation: Important notes (Algorithm)

Improvements on **precision & performance**.



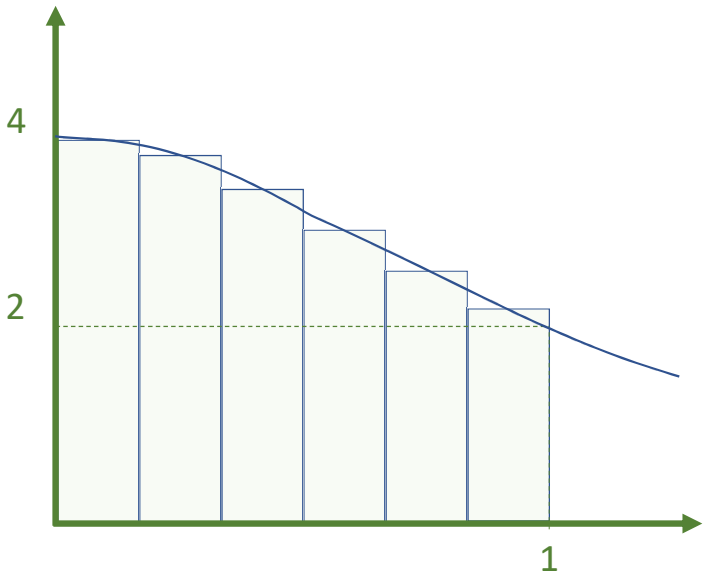


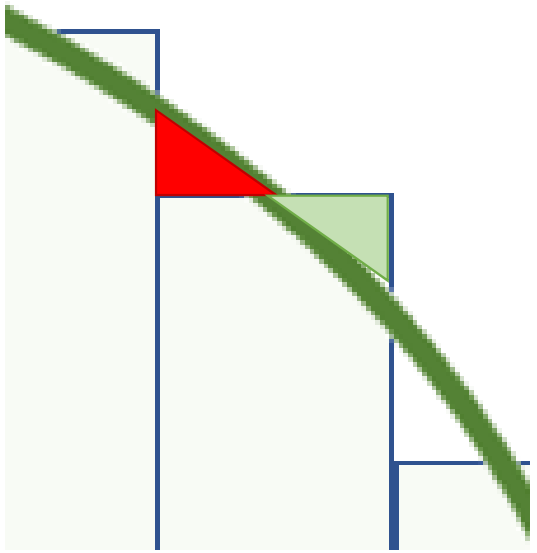
With too few steps, the centered rectangle is much more precise.
With lots of steps, the difference tend to disappear.

OpenMP PI Calculation: Important notes (Algorithm)

Improvements on **precision & performance**.

Modify your algorithm...





Errors almost cancel out.

- ✓ Precision is better at smaller Interval numbers.

Exercise

1.- Run the original code using 1000 steps and 1 thread and record the value of PI.

2.- Improve the precision of your code using centered rectangles.

3.- Using still 1000 steps and 1 thread, compare the PI value of both versions.

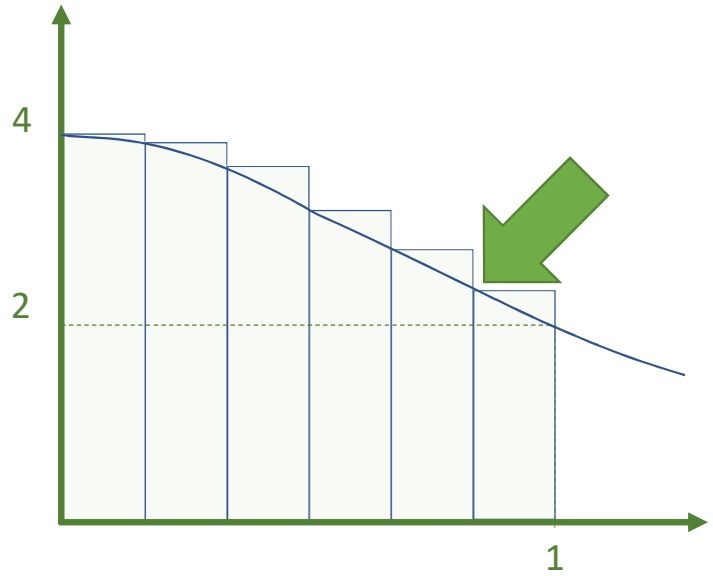
4.- How many steps are needed for each algorithm to get an 8 decimal precision. (Use threads to find out.) Consider PI = 3.14159265.

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

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

   void main() {
      double x;
      long i;
      baseIntervalo = 1.0 / cantidadIntervalos;
      start = clock();
      for (i = 0, x = 0.0; i < cantidadIntervalos; i++) {
         fdx = 4 / (1 + x * x);
         acum = acum + (fdx * baseIntervalo);
         x = x + baseIntervalo;
      }
   end = clock();
   printf("Resultado = %20.18lf (%ld)\n", acum, end - start);
}
```

```c
#include <stdio.h>
#include <time.h>
#include <omp.h>  //Added include file
#define MAXTHREADS 4 //Defined my desired thread count.
long cantidadIntervalos = 1000000000; // 1 B
double baseIntervalo;
//double fdx; //Can´t be global variable.
//double acum = 0; //Can´t be global variable.
clock_t start, end;

void main() {
   int THREADS = MAXTHREADS;
   //long i;  //Can´t be global variable.
   baseIntervalo = 1.0 / (double)cantidadIntervalos;

   //double x;  //Can´t be global variable.
   double partialSum[MAXTHREADS]; //Defined an array. One element for each thread.
   double totalSum = 0; //Defined my final accumulator.
   omp_set_num_threads(THREADS); //Set my desired amount of threads.
   start = clock();



   #pragma omp parallel
   {
   int numThread = omp_get_thread_num();
   double acum = 0; //Was SHARED variable, changed to be a PRIVATE.
   double fdx = 0; //Was SHARED variable, changed to be a PRIVATE.
   double x = 0;  //Can´t be global variable, changed to PRIVATE.
   for (long i = numThread; i < cantidadIntervalos; i += THREADS) {
      x = i * baseIntervalo;
     fdx = 4 / (1 + x * x);
     acum += fdx;
   }
   acum *= baseIntervalo; //Multiply all the heights by the base size.
partialSum[numThread] = acum;
printf("Resultado parcial (Thread %d)\nacum = %lf\n", numThread, acum);
}
end = clock();

for (int c = 0; c < THREADS; c++)
totalSum += partialSum[c];
printf("\nResultado (%d threads) = %20.18lf (%ld)\n", THREADS, totalSum, end - start);
}
```

OpenMP PI Calculation: Important notes (Algorithm)

Improvements on **precision & performance**.

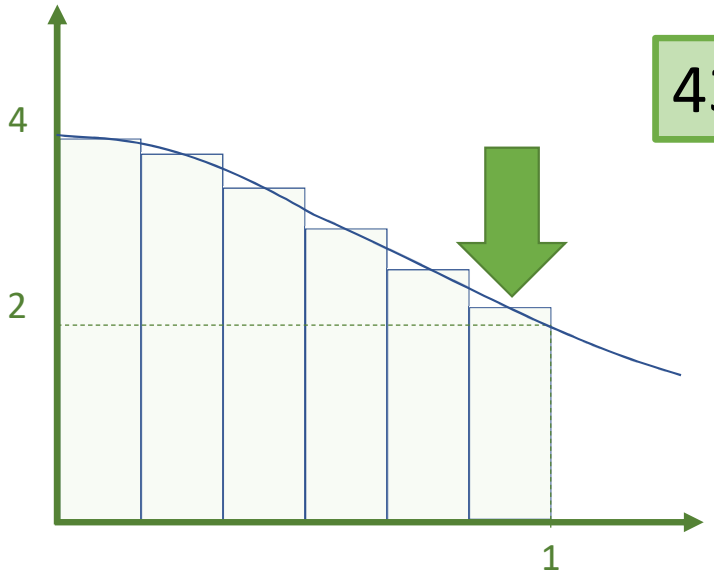


Intervals needed: 155,999,003

```
Resultado (8 threads, 155999000 cantidadIntervalos) = 3.141592660000025905 (271)
Resultado (8 threads, 155999001 cantidadIntervalos) = 3.141592660000015691 (266)
Resultado (8 threads, 155999002 cantidadIntervalos) = 3.141592660000354087 (267)
Resultado (8 threads, 155999003 cantidadIntervalos) = 3.141592659999864257 (267)
Total intervalos requeridos para 8 decimales de precision (8 threads): 155999003 cantidadIntervalos (Tiempo Total: 1072)
```

OpenMP PI Calculation: Important notes (Algorithm)

Improvements on **precision & performance**.



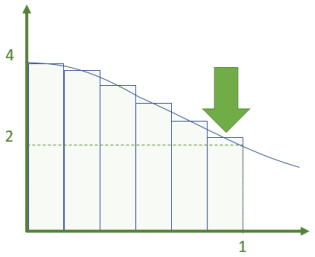
43,250 times less intervals needed!

Intervals needed: 3,607

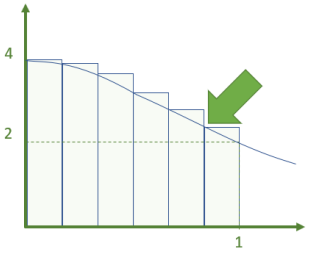
```
Resultado (8 threads, 3603 cantidadIntervalos) = 3.141592660009131066 (0)
Resultado (8 threads, 3604 cantidadIntervalos) = 3.141592660005569471 (0)
Resultado (8 threads, 3605 cantidadIntervalos) = 3.141592660002010984 (0)
Resultado (8 threads, 3606 cantidadIntervalos) = 3.141592659998454273 (0)
Total intervalos requeridos para 8 decimales de precision (8 threads): 3607 cantidadIntervalos (Tiempo Total: 5025)
```

OpenMP PI Calculation: Important notes (Algorithm)

Improvements on **precision & performance**.




```
Resultado (8 threads, 3606 cantidadIntervalos) = 3.141592659998454 73 (0)
```




```
Resultado (8 threads, 155999003 cantidadIntervalos) = 3.1415926599998642 7 (267)
```

At least 267 TIMES faster!

Exercise – Code used

```c
#include <stdio.h>
#include <time.h>
#include <omp.h>  //Added include file
#define MAXTHREADS 8 //Defined my desired thread count.
long cantidadIntervalos = 1000000000; // 1 B
const double PI = 3.14159266;
double baseIntervalo;
//double fdx; //Can´t be global variable.
//double acum = 0; //Can´t be global variable.
clock_t start, end, wholeStart, wholeEnd;

void main() {
   wholeStart = clock();   // The whole program
   double partialSum[MAXTHREADS]; //Defined an array. One element for each thread.
   double totalSum; //Defined my final accumulator.
   int THREADS = MAXTHREADS;
   cantidadIntervalos = 155999000;
//long i;  //Can´t be global variable.
   do
   {
      baseIntervalo = 1.0 / (double)cantidadIntervalos;
      double mitadBaseIntervalo = baseIntervalo * 0.5;
      totalSum = 0.0;
      omp_set_num_threads(THREADS); //Set my desired amount of threads.
      start = clock();
      #pragma omp parallel
      {
      int numThread = omp_get_thread_num();
      double acum = 0; //Was SHARED variable, changed to be a PRIVATE.
      double fdx = 0; //Was SHARED variable, changed to be a PRIVATE.
      double x = 0;  //Can´t be global variable, changed to PRIVATE.
      for (long i = numThread; i < cantidadIntervalos; i += THREADS) {
         x = i * baseIntervalo;  //Rectangles with height at a corner.
         //x = i * baseIntervalo + mitadBaseIntervalo; //Rectangles with height atmiddle.
         fdx = 4 / (1 + x * x);
         acum += fdx;
      }
      acum *= baseIntervalo; //Multipy all heights by the width of base.
      partialSum[numThread] = acum;
      //printf("Resultado parcial (Thread %d)\nacum = %lf\n", numThread, acum);
      }
      end = clock();

      for (int c = 0; c < THREADS; c++)
         totalSum += partialSum[c];
      printf("Resultado (%d threads, %d cantidadIntervalos) = %20.18lf (%ld)\n", THREADS, cantidadIntervalos, totalSum, end - start);
      cantidadIntervalos += 1;
   } while (totalSum >= PI);
   wholeEnd = clock();   // We got to the needed precision.
   printf("Total intervalos requeridos para 8 decimales de precision (%d threads): %d cantidadIntervalos (Tiempo Total: %ld)\n", THREADS, cantidadIntervalos - 1, wholeEnd - wholeStart);
}
```

OpenMP PI Calculation: Conclusions (Algorithm)

- ✓ Think and re-think your algorithm!
- ✓ Performance is NOT only Hardware Parallelism.
- ✓ Define clearly what you want and/or need:
  - ➢ Precision
  - ➢ Performance
  - ➢ Both
- ✓ Test your different options.

OpenMP Synchronization

Remember…

Important concept: Race Condition

- Caused by unintended sharing of data.
- Identified when the program's outcome changes with each run (threads are scheduled differently).
- Synchronization provides a way to control race conditions, by protecting data conflicts.
- Synchronization is very expensive. Use intelligent data access algorithms to minimize the need for synchronization.

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:

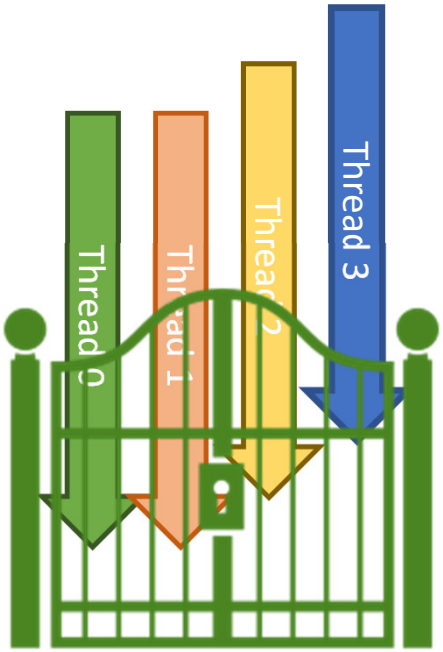
- Barrier
- Mutual Exclusion
  - Critical
  - Atomic

OpenMP Synchronization - Barrier

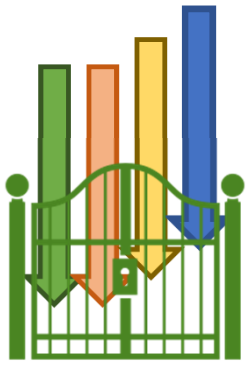
- No thread can proceed past a barrier **until** all the other threads have arrived.
- Syntax C/C++:

```
#pragma omp barrier
```

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

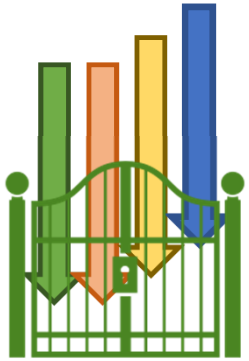


OpenMP Synchronization - Barrier



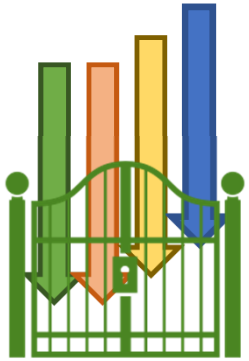
```
#pragma omp parallel
{
    int id = omp_get_thread();

    A[id] = big_multithreaded_calculation(id);

    #pragma omp barrier     // Need barrier to wait
for all threads to finish their calculations.

    B[id]= another_big_calculation(id, A);
}
```

OpenMP Synchronization - Example



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

int main(){
   int x = 1;

   #pragma omp parallel num_threads(4)
   {
      if (omp_get_thread_num() == 0) {
         Sleep(2);
         x = 13;
   }

   // Race condition when reading X.
      printf("Thread (%d): x = %d\n", omp_get_thread_num(), x);

      if (omp_get_thread_num() == 0) {    //No race condition.
         printf("Thread (%d): x = %d\n", omp_get_thread_num(),x );
      } else {
         printf("Thread (%d): x = %d\n", omp_get_thread_num(),x );
      }
   }
   return 0;
}
```

OpenMP Synchronization - Example



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

int main(){
   int x = 1;

   #pragma omp parallel num_threads(4)
   {
      if (omp_get_thread_num() == 0) {
         Sleep(2);
         x = 13;
   }

   #pragma omp barrier   // Race condition has been dealt with.
      printf("Thread (%d): x = %d\n", omp_get_thread_num(), x);
   // If no barrier, worker threads will not wait until ThreadID=0 sets it to 13.
      if (omp_get_thread_num() == 0) {    //No race condition.
         printf("Thread (%d): x = %d\n", omp_get_thread_num(),x );
      } else {
         printf("Thread (%d): x = %d\n", omp_get_thread_num(),x );
      }
   }
   return 0;
}
```

OpenMP Synchronization - Example

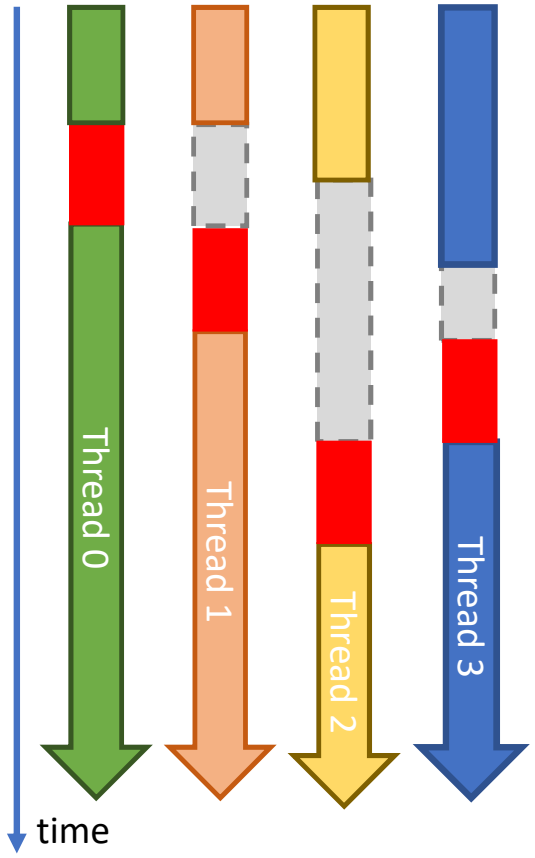
Command Prompt

```
C:\Users\Alex\source\repos\Project12OpenMP\x64\Debug>Project12OpenMP.exe
Thread (1): x = 1
Thread (1): x = 1
Thread (2): x = 1
Thread (2): x = 1
Thread (3): x = 1
Thread (3): x = 1
Thread (0): x = 13
Thread (0): x = 13

C:\Users\Alex\source\repos\Project12OpenMP\x64\Debug>REM Barrier implemented. No more race condition.

C:\Users\Alex\source\repos\Project12OpenMP\x64\Debug>Project12OpenMP.exe
Thread (1): x = 13
Thread (1): x = 13
Thread (3): x = 13
Thread (3): x = 13
Thread (0): x = 13
Thread (0): x = 13
Thread (2): x = 13
Thread (2): x = 13

C:\Users\Alex\source\repos\Project12OpenMP\x64\Debug>
```

OpenMP Synchronization – Mutual Exclusion (CRITICAL)

- Let only one thread execute a block of code.
- Known as "*critical section*".
- Critical sections are commonly used to update variable.
- If another thread needs access to the critical section, it´ll have to wait, wasting resources.
- Critical sections are an easy way to turn an existing code into a correct parallel code.
- There are disadvantages.
- Sometimes rewriting code is needed.
- Syntax C/C++:

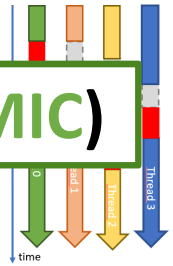
```
#pragma omp critical
{}
```



OpenMP Synchronization – Mutual Exclusion (CRITICAL)


```
int acum;
#pragma omp parallel
{
   int A;
   int number_of_threads = omp_get_num_threads();
   int id = omp_get_thread_num();

   for (int i = id; i < iterations; i += number_of_threads)
   {
      A = big_multithreaded_job(i);
      #pragma omp critical  //Only one thread allowed at the following block.
      {
         some useful code...
         acum += A;
      }
   }
}
```

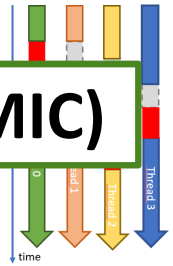
OpenMP Synchronization – Mutual Exclusion (ATOMIC)



- Provides mutual exclusion.
- Only applies to the update of a memory location.
- Used for specific updates, like boolean or increments.
- May try to use hardware available instructions to do so.
- Syntax C/C++:

```
#pragma omp atomic
```

OpenMP Synchronization – Mutual Exclusion (ATOMIC)




```
#pragma omp atomic
```

The statement after the atomic clause is one of:

x++

++x

x--

--x

x binop = expr

- *expr* is an expression with scalar type, and it does not reference the object designated by *x*.
- *binop* is not an overloaded operator and is one of `+`, `*`, `-`, `/`, `&`, `^`, `|`, `<<`, or `>>`.

OpenMP Synchronization – Mutual Exclusion (CRITICAL)

Example:

```
#pragma omp parallel
{
    int acum = 0;
    int tmp;
    int A;
    A = executeSomething();
    tmp = big_multithreaded_code(A);

    #pragma omp atomic  //Only one thread at the following LINE of code.
        acum += tmp;
}
```

Exercise

1.- Run the original code (record execution time).

2.- Use only one global variable to accumulate all the thread results (record execution time).

3.- Use the original code and use OpenMP critical or atomic to do ONLY the final reduction of all the partial accumulations.

```
#include <stdio.h>
#include <time.h>
#include <omp.h>
#define MAXTHREADS 8
long cantidadIntervalos = 10000000; //10 Million
double baseIntervalo;
//double acum = 0; //Can´t be a shared variable
clock_t start, end;

void main() {
int THREADS = MAXTHREADS;
baseIntervalo = 1.0 / (double)cantidadIntervalos;

double x, partialSum[MAXTHREADS], totalSum = 0;
omp_set_num_threads(THREADS);
start = clock();
#pragma omp parallel
{
int numThread = omp_get_thread_num();
double acum = 0; // Can´t be a shared variable, must be private to thread.
double fdx = 0; //Can´t be a shared variable, must be private to thread.
for (long i = numThread; i < cantidadIntervalos; i += THREADS) {
x = i * baseIntervalo;
fdx = 4 / (1 + x * x);
acum += fdx;
}
acum *= baseIntervalo; //Multiply accumulated heights of the rectangles by the base size.
partialSum[numThread] = acum;
printf("Resultado parcial (Thread %d)\nacum = %lf\n", numThread, acum);
}
end = clock();

for (int c = 0; c < THREADS; c++)
totalSum += partialSum[c];
printf("\nResultado (%d threads) = %20.18lf (%ld)\n", THREADS, totalSum, end - start);

}
```

OpenMP Synchronization – Mutual Exclusion (CRITICAL)

If you forget the clause… Race condition guaranteed

```
#include <stdio.h>
#include <time.h>
#include <omp.h>
#define MAXTHREADS 8
long cantidadIntervalos = 10000000; //10 Million
double baseIntervalo;
double acum = 0; //Can´t be a shared variable
clock_t start, end;

void main() {
int THREADS = MAXTHREADS;
baseIntervalo = 1.0 / (double)cantidadIntervalos;

double x, partialSum[MAXTHREADS], totalSum = 0;
omp_set_num_threads(THREADS);
start = clock();
#pragma omp parallel
{
int numThread = omp_get_thread_num();
//double acum = 0; //No puede ser una variable global. Es una variable privada al thread.
double fdx = 0; //No puede ser una variable global. Es una variable privada al thread.
for (long i = numThread; i < cantidadIntervalos; i += THREADS) {
x = i * baseIntervalo;
fdx = 4 / (1 + x * x);
acum += fdx;  //Watch here for a race condition.
}
//acum *= baseIntervalo; //Multiplico todas las altura de los rectangulos acumuladas por el tamaño de la base.
//partialSum[numThread] = acum;
//printf("Resultado parcial (Thread %d)\nacum = %lf\n", numThread, acum);
}
end = clock();

acum *= baseIntervalo; //Move this outside, since acum has all th

//for (int c = 0; c < THREADS; c++)
//totalSum += partialSum[c];
totalSum = acum;
printf("\nResultado (%d threads) = %20.18lf (%ld)\n", THREADS, totalSum, end - start);

}
```

Race condition

```
Resultado (8 threads) = 0.39269900669870672 1 (20)
```

OpenMP Synchronization – Mutual Exclusion (CRITICAL)

Race condition averted! But what is the COST!

```
#include <stdio.h>
#include <time.h>
#include <omp.h>
#define MAXTHREADS 8
long cantidadIntervalos = 10000000; //10 Million
double baseIntervalo;
double acum = 0; //No puede ser una variable global
clock_t start, end;

void main() {
int THREADS = MAXTHREADS;
baseIntervalo = 1.0 / (double)cantidadIntervalos;

double partialSum[MAXTHREADS], totalSum = 0;
omp_set_num_threads(THREADS);
start = clock();
#pragma omp parallel
{
int numThread = omp_get_thread_num();
//double acum = 0; //No puede ser una variable globa
double x, fdx = 0; //No puede ser una variable globa
for (long i = numThread; i < cantidadIntervalos; i += THREADS) {
x = i * baseIntervalo;
fdx = 4 / (1 + x * x);
acum += fdx;  //Watch here for a race condition.
}
//acum *= baseIntervalo; //Multiplico todas las altura de los rectangulos acumuladas por el tamaño de la base.
//partialSum[numThread] = acum;
//printf("Resultado parcial (Thread %d)\nacum = %lf\n", numThread, acum);
}
end = clock();

acum *= baseIntervalo; //Move this outside, since acum ha

//for (int c = 0; c < THREADS; c++)
//totalSum += partialSum[c];
totalSum = acum;
printf("\nResultado (%d threads) = %20.18lf (%ld)\n", THREADS, totalSum, end - start);

}
```

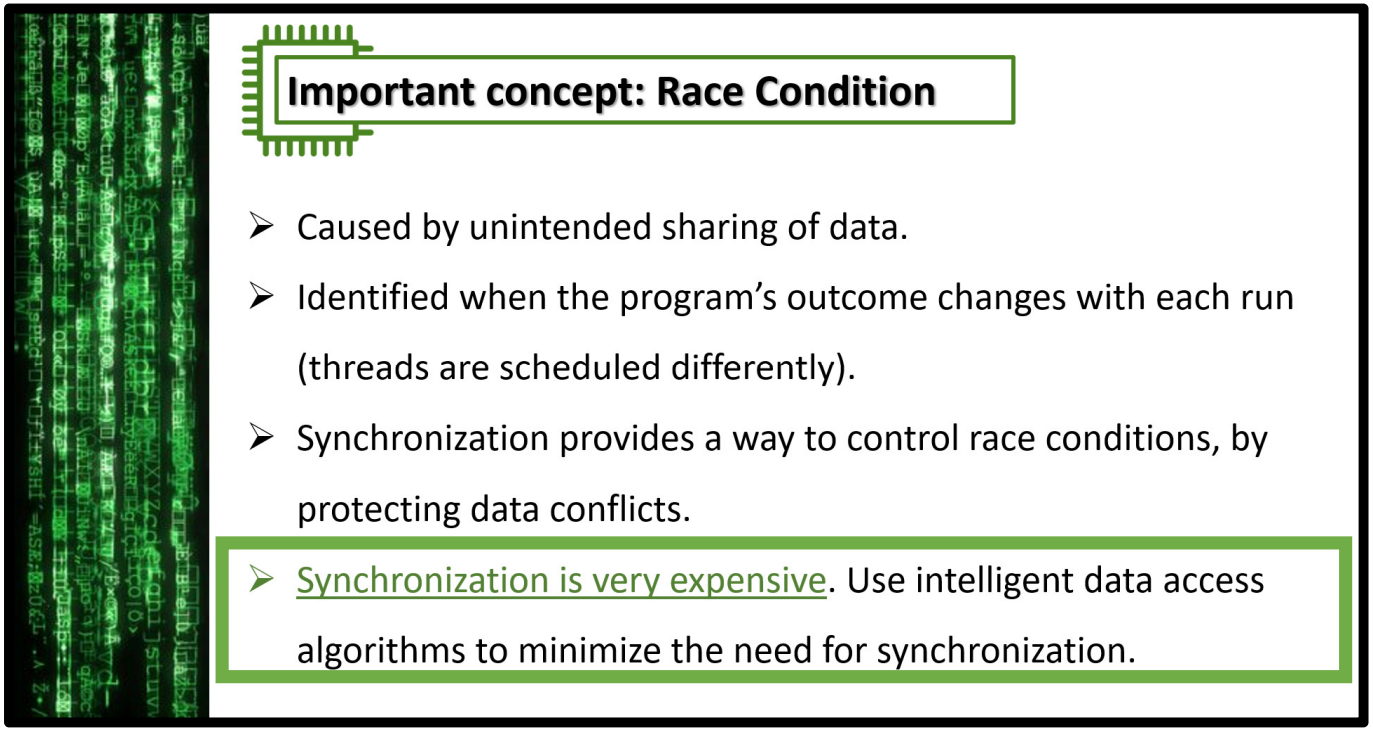
```
#pragma omp parallel
    {
        int numThread = omp_get_thread_num();
        //double acum = 0; //No puede ser una variable global. Es una variable privada al thread.
        double fdx = 0; //No puede ser una variable global. Es una variable privada al thread.
        for (long i = numThread; i < cantidadIntervalos; i += THREADS) {
            x = i * baseIntervalo;
            fdx = 4 / (1 + x * x);
            #pragma omp atomic
            acum += fdx;  //Watch here for a race condition.
        }
        //acum *= baseIntervalo; //Multiplico todas las alturas de los rectangulos acumuladas por el tamaño de la base.
        //partialSum[numThread] = acum;
        //printf("Resultado parcial (Thread %d)\nacum = %lf\n", numThread, acum);
    }
    end = clock();
```

Race condition

```
Resultado (8 threads) = 3.14159275359001766l (1988)
```

OpenMP Synchronization

Remember…



Important concept: Race Condition

- Caused by unintended sharing of data.
- Identified when the program's outcome changes with each run (threads are scheduled differently).
- Synchronization provides a way to control race conditions, by protecting data conflicts.
- Synchronization is very expensive. Use intelligent data access algorithms to minimize the need for synchronization.

OpenMP Synchronization – Mutual Exclusion (CRITICAL)

Using synchronization wisely…

```
#include <stdio.h>
#include <time.h>
#include <omp.h>
#define MAXTHREADS 8
long cantidadIntervalos = 10000000; //10 Million
double baseIntervalo;
//double acum = 0; //No puede ser una variable global
clock_t start, end;

void main() {
   int THREADS = MAXTHREADS;
   baseIntervalo = 1.0 / (double)cantidadIntervalos;

   double partialSum[MAXTHREADS], totalSum = 0;
   omp_set_num_threads(THREADS);
   start = clock();
   #pragma omp parallel
   {
      int numThread = omp_get_thread_num();
      double acum = 0; //No puede ser una variable global. Es una variable privada al thread.
      double x, fdx = 0; //No puede ser una variable global. Es una variable privada al thread.
      for (long i = numThread; i < cantidadIntervalos; i += THREADS) {
         x = i * baseIntervalo;
         fdx = 4 / (1 + x * x);
         acum += fdx;
      }
      acum *= baseIntervalo; //Multiplico todas las altura
      //partialSum[numThread] = acum; //Instead of partial
      #pragma omp critical
         totalSum += acum;
      printf("Resultado parcial (Thread %d)\nacum = %lf\n"
   }
   end = clock();
   //for (int c = 0; c < THREADS; c++) //No longer needed
   //totalSum += partialSum[c];
   printf("\nResultado (%d threads) = %20.18lf (%ld)\n", THREADS, totalSum, end - start);
}
```

```
Resultado (8 threads) = 3.141592753589807163 (21)
```

vs. synchronizing to a global accumulator.

```
Resultado (8 threads) = 3.141592753590017661 (1988)
```

```
acum *= baseIntervalo; //Multiplico todas las alturas de los rectangulos a
//partialSum[numThread] = acum; //Instead of partial accumulation over an
#pragma omp critical
   totalSum += acum;
printf("Resultado parcial (Thread %d)\nacum = %lf\n", numThread, acum);
```

See you soon…