Parallel Programming with OpenMP

OpenMP Intro

Loop Parallelism

Scheduling policies for iterations

Bibliography

- [Pacheco]: Peter Pacheco, Matthew Malensek, Introduction to Parallel Programming, 2nd Edition, Morgan Kaufmann Publisher, March 2020, **Chapter 5**
- OpenMP Specifications: https://www.openmp.org/specifications/

OpenMP: a Standard for Directive Based Parallel Programming

- **Open M**ulti-**P**rocessing
- OpenMP is a **directive-based API** that can be used with different languages for programming shared address space machines
  - It is not a new language, but programs in existing languages can be annotated with omp directives in order to guide their parallel execution
  - The OpenMP API is *standardized*. Specifications are maintained by an industrial consortium
  - https://www.openmp.org/specifications/
  - Current version is V.5.2 (2021). V.6.0 under construction planned for end of 2024
  - The OpenMP API is specified for C/C++ and Fortran
  - There are many implementations in compilers from different providers

OpenMP Programming Model

- OpenMP directives in C and C++ are based on the `#pragma omp` compiler directives.
- Pragmas are special preprocessor instructions
- Compilers that do not support the pragmas just ignore them !

- A directive consists of a directive name followed by clauses

```
#pragma omp directive [clause list]
/* structured block */
```

- Each directive applies to the succeeding statement, which can be be a structured block

- Parallelization with OpenMP can be as simple as taking a serial program and inserting compiler directives !

#pragma omp parallel

- OpenMP programs execute serially until they encounter the **#pragma omp parallel directive** that makes ***a parallel region (or parallel section)***

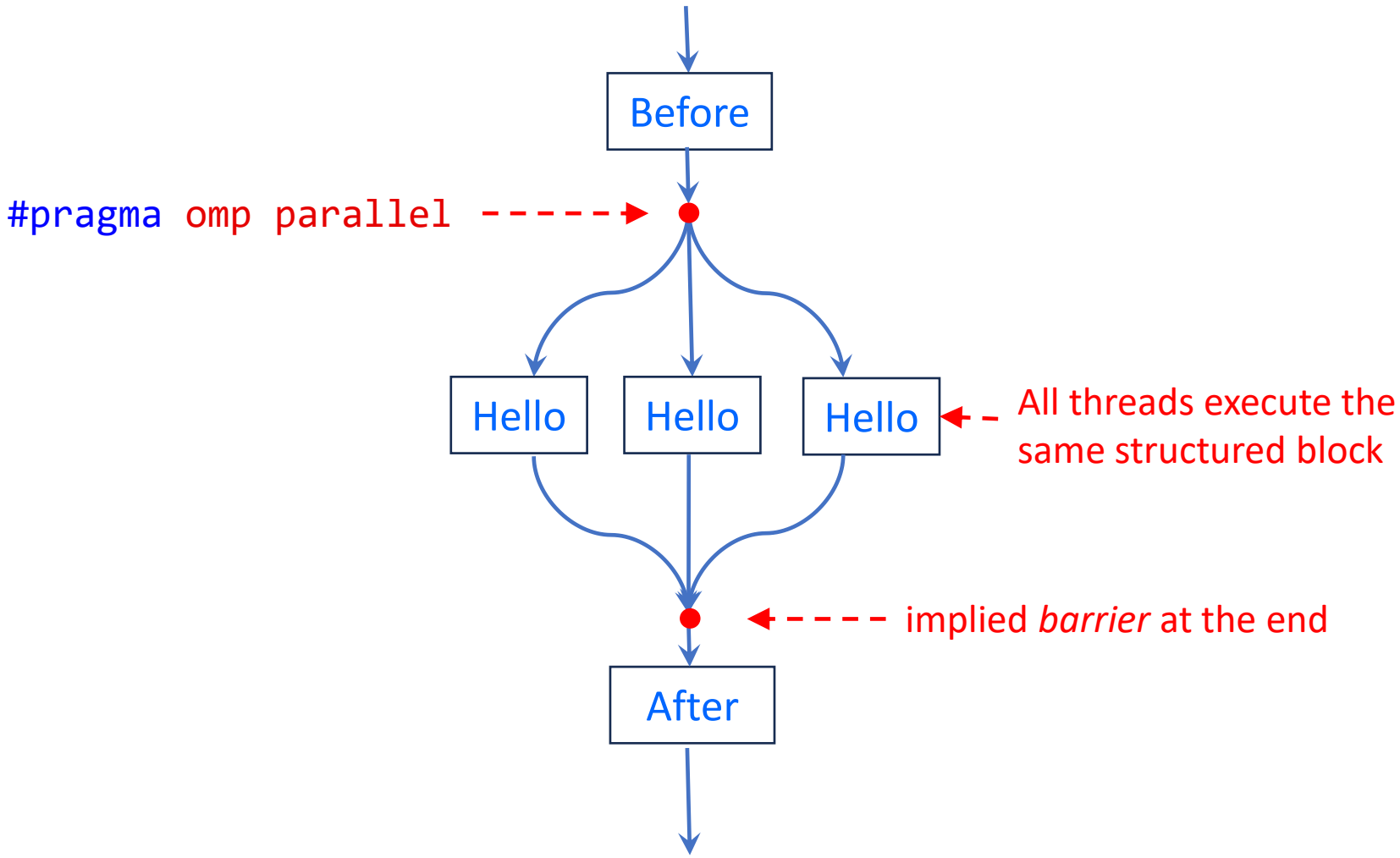
```
#pragma omp parallel [clause list]
/* structured block */
```

- Creates a group of threads (a *team* of threads)
- **Every thread in the team executes the code of the given *structured block***
- The original thread is called the *master* thread and has thread number 0 in its team
- There is an implied *barrier* at the end of a parallel section.
- By default (with no additional clauses) the number of threads created is determined by the run-time system(detects the maximum physical level of parallelism)

First Hello World Example

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

int main(int argc, char** argv) {

    printf("This is BEFORE parallel \n");

#pragma omp parallel
    {
        printf("Hello World from thread=%d out of %d\n",
                  omp_get_thread_num(), omp_get_num_threads());
    }

    printf("This is AFTER parallel \n");

    return 0;
}
```

Hello World threads



Compiling and Running

- OpenMP is supported by various compilers
- In all cases, ***compilation must be done with a flag*** that explicitly ***enables OpenMP***
- Without this flag the compiler does NOT process #pragma omp directives and *the program is compiled as a simple program without parallelism*
- GNU: **`gcc -fopenmp`**
- MinGW: **`gcc -fopenmp`**
- VisualC++: **`cl /openmp`**
  - VisualC++ supports only OpenMP standard 2.0
- https://www.openmp.org/resources/openmp-compilers-tools/

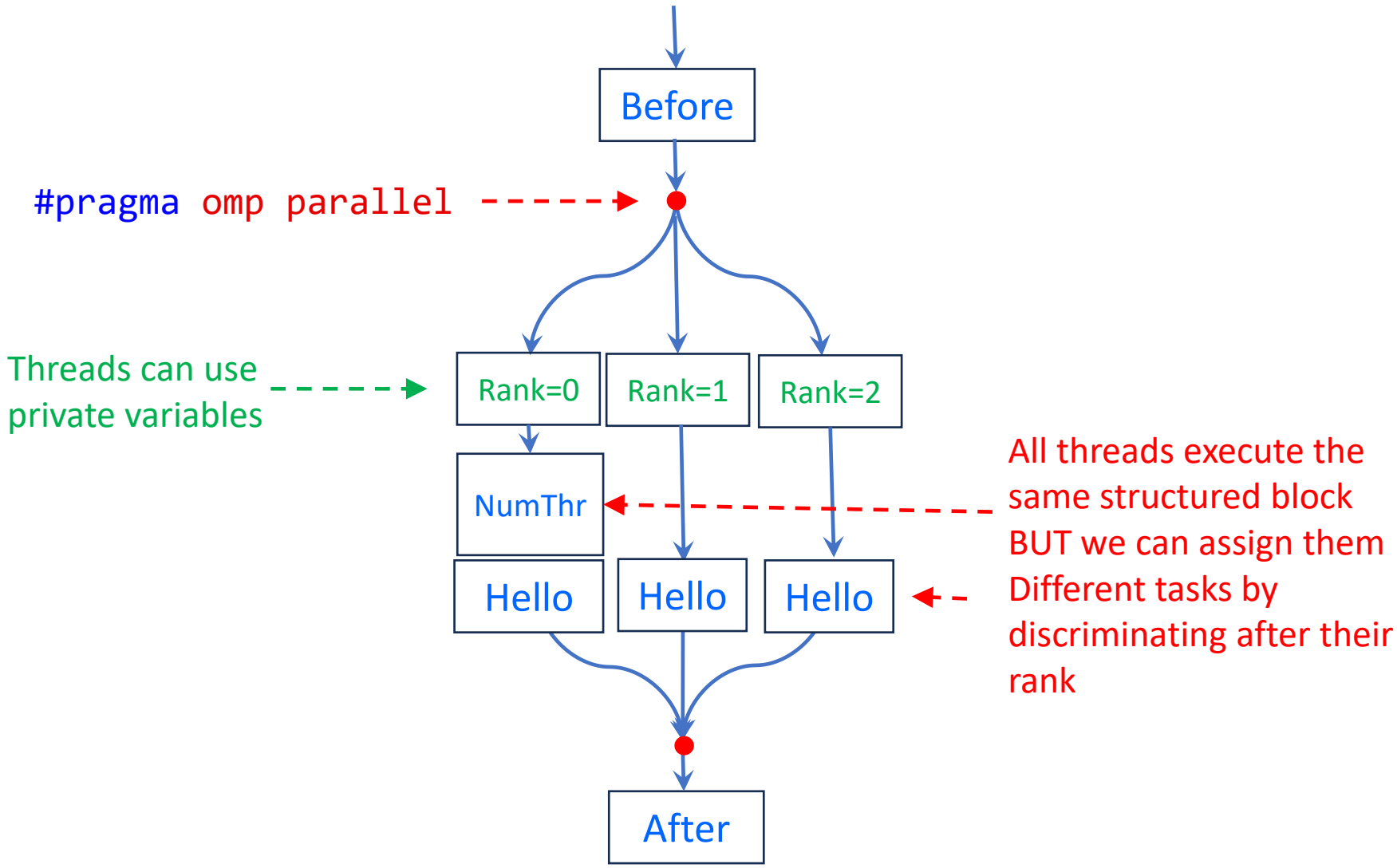
OpenMP functions

- OpenMP is mainly based on pragmas (preprocessor directives) but there is also a small API containing a number of functions
- If you want to use the functions, have to include <omp.h> (If you only use pragmas, you do not have to include it)
- Some OpenMP functions:
- `omp_get_thread_num()`
- `omp_get_num_threads()`
- `omp_get_wtime()`

Second HelloWorld Example

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

int main(int argc, char **argv)
{
    int my_rank;

#pragma omp parallel num_threads(20) private(my_rank)
    {
        my_rank = omp_get_thread_num();
        if (my_rank == 0)
            printf("Hello from thread=%d. There are %d threads\n",
                           my_rank, omp_get_num_threads());
        else
            printf("Hello World from thread=%d\n", my_rank);
    }

    return 0;
}
```

Hello World threads V2



SPMD Program Models

- SPMD (Single Program, Multiple Data) is the model that applies to OpenMP *parallel regions*
  - All threads of the parallel region execute the same code
  - Each thread has unique ID
- Use the thread ID to diverge the execution of the threads
  - Different threads can follow different paths through the same code : `if (my_rank==…)`
- SPMD is the most commonly used pattern for structuring parallel programs - MPI, OpenMP, CUDA, etc

#pragma omp parallel clauses

- `#pragma omp parallel [clause list]`
- **Clauses are used to specify:**
  - **Degree of Concurrency:** `num_threads(integer)` specifies the number of threads that are created
  - **Data Handling:**
    - `private (variable list)` indicates variables local to each thread.
    - `firstprivate (variable list)` is similar to the `private`, except values of variables are ***initialized*** to corresponding values before the parallel directive.
    - `shared (variable list)` indicates that variables are shared across all the threads
  - **Conditional Parallelization:** `if (expression)`
    - determines whether the parallel construct results in creation of threads.

Example parallel with clauses

```
#pragma omp parallel if (is_parallel== 1) num_threads(8) \
  private (a) shared (b) firstprivate(c) {
 /* structured block */
}
```

- If the value of the variable is_parallel equals one, then eight threads are created.
- Each of these threads gets private copies of variables a and c, and shares a single value of variable b.
- The value of each copy of c is initialized to the value of c before the parallel directive.
- The default state of a variable can be specified by the clause default (shared) or default (none).
- **If there is no default clause present and no private or shared clauses, then the variables are considered shared**
  - **If you need threads to have private variables, must explicitly specify it!**

OpenMP Programming Model

```
int a, b;
main() {
  [ // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
      [ // parallel segment
    }
  [ // rest of serial segment
}
```

Sample OpenMP program

```
int a, b;
main() {
  [ // serial segment
    for (i = 0; i < 8; i++)
        pthread_create (....., internal_thread_fn_name, ...);

    for (i = 0; i < 8; i++)
        pthread_join (.....);
  [ // rest of serial segment

}

void *internal_thread_fn_name (void *packaged_argument) [
    int a;
  [ // parallel segment
}
```

Code inserted by the OpenMP compiler

Corresponding Pthreads translation

Critical Sections with OpenMP

- # pragma omp critical
  - If a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region
- # pragma omp atomic
  - The ATOMIC directive specifies that a specific memory location must be updated atomically
  - This directive provides a mini-critical section that contains only one instruction

Critical Section Counterexample

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

int main(int argc, char** argv) {

    int counter = 0;

#pragma omp parallel num_threads(20) shared(counter)
    {
            counter++;  // WRONG!! RACE CONDITION!
    }

    printf("counter=%d \n", counter);
    return 0;
}
```

Critical Section Example

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

int main(int argc, char** argv) {

    int counter = 0;

#pragma omp parallel num_threads(20) shared(counter)
    {
        #pragma omp critical
        {
           counter++;
           printf("thread  %d has incremented counter to %d \n",
                                  omp_get_thread_num(), counter);
        }
    }

    printf("Final value of counter=%d \n", counter);
    return 0;
}
```

Other Synchronization Constructs

- Synchronization constructs that applies to one structured block to be executed by one thread only (one thread in total):
  - #pragma omp single
  - #pragma omp master

- Synchro construct that specifies that applies to a structured block that must be executed in sequential order (combined with parallel for)
  - #pragma omp ordered

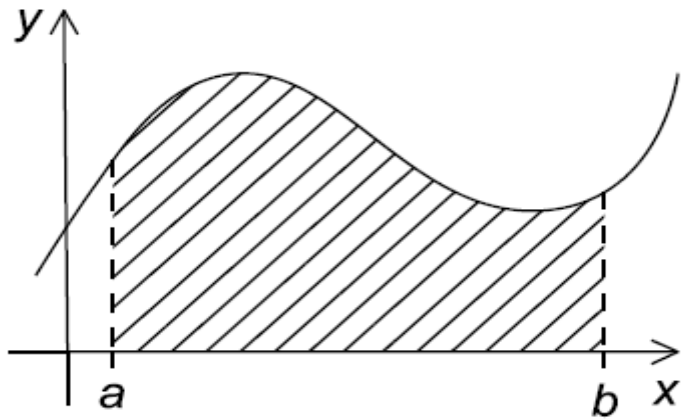
- Directive that defines a synchronization point
  - #pragma omp barrier

Code Examples

- hello_omp.c
- hello_omp2.c
- criticalsection_omp.c

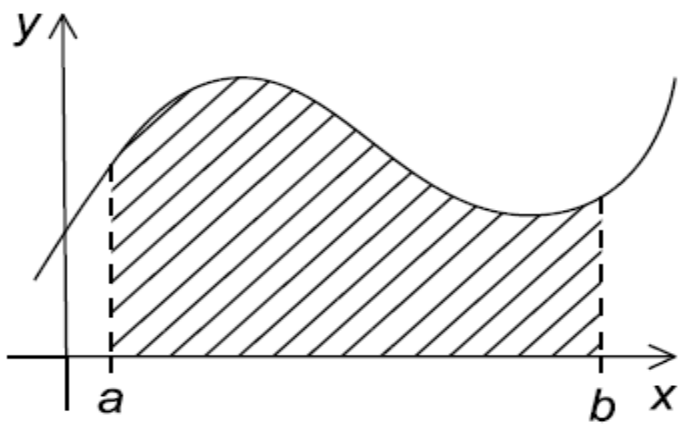
A Parallelization Problem: Estimating the area under a curve by the trapezoidal rule

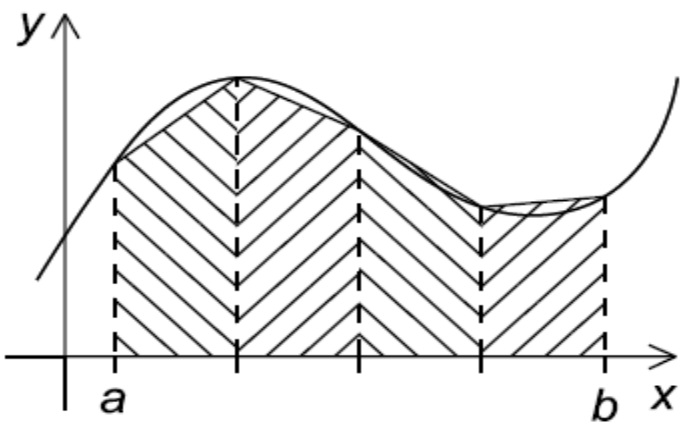
- Approximate the area between the graph of a function, $y = f(x)$, two vertical lines at x=a and x=b, and the x-axis.
- It is a method for ***numerical integration***



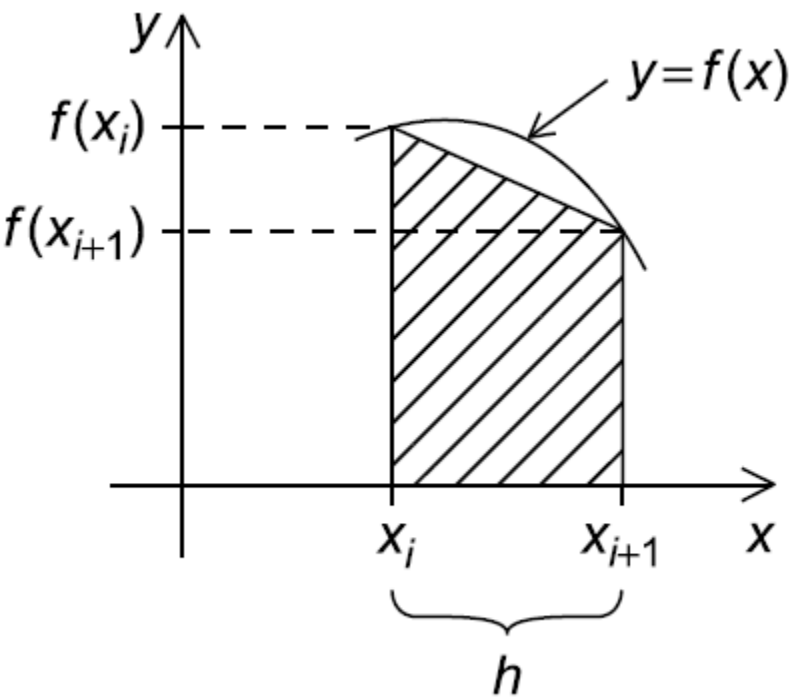
The trapezoidal rule method

- The basic idea:
  - divide the interval on the *x*-axis into ***n*** equal subintervals
  - approximate the area lying between the graph and each subinterval by a trapezoid
  - sum up the ***n*** trapezoid areas
- In order to have a good precision, ***n*** must be very big!





The trapezoidal rule




```
Trapez[i] = (f(Xi)+f(Xi+1))*h/2

Integral= Sum(Trapez[i]), i=0..n-1

S = Sum ((f(Xi)), i=1..n-1

Integral= ((f(a)+f(b))/2 + S) * h
```

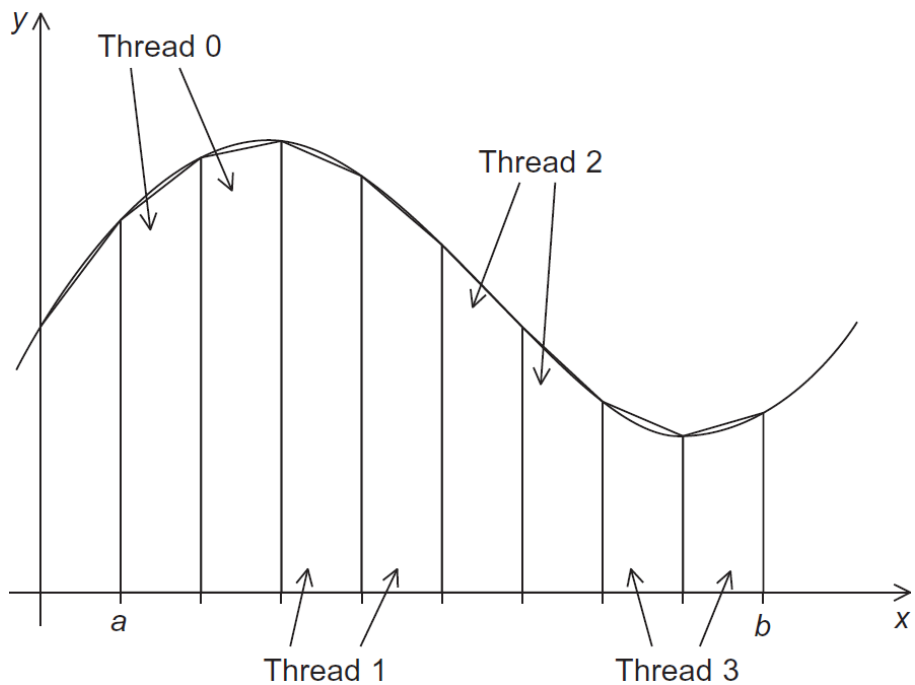
```
h = (b - a) / n

Xi = a + i*h  (X0=a, Xn=b)
```

Trapezoidal Rule - Serial version

```
double serial_integral(double a, double b, int n)
{
   double integral;
   double h = (b - a) / n;
   int i;
   integral = (f(a) + f(b)) / 2.0;
   for (i = 1; i <= n - 1; i++)
   {
      integral += f(a + i * h);
   }
   integral = integral * h;
   return integral;
}
```

Trapezoidal Rule - Parallel v1 – by data partitioning

- Input data partitioning: partition the interval [*a, b*] into subintervals, and each thread computes trapezes and sums them up in its subinterval
- At the end, all partial results from subintervals (threads) must be added



Data parallel version

- [a,b] with ***n*** *intervals*
- thread_count threads
- Each thread has:
  - n_local=n/thread_count *local intervals*
  - a_local, b_local calculated according to my_rank
  - my_result
- Final result global_result is the sum of all partial results

Trapezoidal - Parallel v1

```
double parallel_integral_v1(double a, double b, int n, int
thread_count)
{
   double global_result;

#pragma omp parallel num_threads(thread_count)
   {
      double h, x, my_result;
      double local_a, local_b;
      int i, local_n, rest;
      int my_rank = omp_get_thread_num();

      h = (b - a) / n;
      local_n = n / thread_count;
      if (my_rank == thread_count - 1)
         rest = n % thread_count;
      else
         rest = 0;
      local_a = a + my_rank * local_n * h;
      local_b = local_a + (local_n + rest) * h;
```

Continues on next slide...

Follow-up from previous slide…

```
      my_result = (f(local_a) + f(local_b)) / 2.0;
      for (i = 1; i <= local_n + rest -1; i++)
      {
         x = local_a + i * h;
         my_result += f(x);
      }
      my_result = my_result * h;

#pragma omp critical
      global_result += my_result;
   }

   return global_result;
}
```

Trapezoidal Rule - Parallel v2 – use Reduction

- In parallel version v1 every thread computes a partial result, and at the end, using a critical section, adds the partial result into the global result
- This is a frequently occurring pattern that threads compute partial results that must be combined at the end – it is called a ***reduction***
- A ***reduction*** is a computation that repeatedly applies the same ***reduction operator*** to a sequence of operands in order to get a single result
- The ***reduction operator*** must be ***associative***, because we do not know the order of applying it to the operands in the sequence
- There is a special clause in OpenMP for reduction
- We can achieve the same semantic by using a reduction clause instead of the explicit critical section where partial results are summed up

Reduction Clause in OpenMP

- Typical usage: Sum of N numbers, without reduction

```
int s = 0;
#pragma omp parallel default(none) shared(s)
  {
     int rank = omp_get_thread_num();
     int local_n = N / omp_get_num_threads();
     int start = rank * local_n;
     int end = (rank+1) * local_n-1;

     int local_s = 0;
     for (int i = start; i <= end; i++)
        local_s = local_s + i;
  #pragma omp critical
     s = s + local_s;
  } // end parallel section

  printf(" s=%d \n", s);
```

Every thread works on its local copy of s

Every thread merges its local s into global shared s. Must use **critical section**

Reduction Clause in OpenMP

- Typical usage: Sum of N numbers, with reduction

```
int s;
#pragma omp parallel default(none) num_threads(2) reduction(+:s)
  {
     int rank = omp_get_thread_num();
     int local_n = N / omp_get_num_threads();
     int start = rank * local_n;
     int end = (rank+1) * local_n – 1;

     s=0;
     for (int i = start; i <= end; i++)
        s = s + i;
  } // end parallel section

  printf("s=%d \n", s);
```

Every thread works on its local copy of s

After the parallel region, s contains the sum of the local copies. Merging in a critical section was implicit done by reduction!

Reduction Clause in OpenMP

- The **`reduction`** clause specifies how multiple local copies of a variable (the *reduction variable*) at different threads are combined into a single copy at the master when threads exit.
- The reduction performs the combination in a *safe* mode (it uses an implicit critical region)
- The usage of the `reduction` clause is

  `reduction(operator: variable list)`
- The variables in the list are implicitly specified as being private to threads.
- The `operator` must be *associative*. It can be one of `+`, `*`, `-`, `&`, `|`, `^`, `&&`, and `||`.

Trapezoidal - Parallel v2

```
double parallel_integral_v2(double a, double b, int n, int thread_count)
{
   double result;

#pragma omp parallel num_threads(thread_count) reduction(+ : result)
   {
      double h, x;
      double local_a, local_b;
      int i, local_n, rest;
      int my_rank = omp_get_thread_num();

      h = (b - a) / n;
      local_n = n / thread_count;
      if (my_rank == thread_count - 1)
         rest = n % thread_count;
      else
         rest = 0;

      local_a = a + my_rank * local_n * h;
      local_b = local_a + (local_n + rest) * h;
```

Continues on next slide...

Follow-up from previous slide...

```
  result = (f(local_a) + f(local_b)) / 2.0;

  for (i = 1; i <= local_n + rest -1; i++)
      {
         x = local_a + i * h;
         result += f(x);
      }
   result = result * h;

   } /* end parallel region */

   return result;
}
```

Every thread works on its local copy of result

After the parallel region, result contains the sum of the local copies

Specifying Concurrent Tasks in OpenMP

- The **`parallel`** directive by default creates ***SPMD*** type of parallelism
- OpenMP provides two directives - **`for`** and **`sections`** - to use **together with** the `parallel` directive in order to specify different types of parallelism:
  - The **`sections`** directive is used to specify different blocks to be executed by different threads. This is a form of ***task parallelism***.
  - The **`for`** directive is used to split parallel iteration spaces across threads. This is a very specific form of data parallelism = ***loop parallelism***

Parallel sections Example

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

int main(int argc, char **argv) {
    int my_rank;

#pragma omp parallel sections private(my_rank)
    {
#pragma omp section
        {
            my_rank = omp_get_thread_num();
            printf("I am thread %d and do AAAAAA \n");
        }
#pragma omp section
        {
            my_rank = omp_get_thread_num();
            printf("I am thread %d and do something else BBBBB \n");
        }
    }

    return 0;
}
```

SPMD vs Worksharing

- SPMD (Single Program Multiple Data) :
  - #pragma omp parallel creates a SPMD type of behaviour, *each thread executes the same code*
  - Still we can have some threads doing something different if the code contains explicit if (my_rank==XXX) statements
- Worksharing or task parallelism: *threads are assigned different code*
  - #pragma omp sections explicitly assigns different code blocks to different threads

Loop parallelism

Parallel for

- The **`for` directive** is used to ***split parallel iteration spaces across threads***.
- This is a very specific form of data parallelism = loop parallelism

- Can be used as:

```
#pragma omp parallel for [clauses]
```

- Or as:

```
#pragma omp parallel
#pragma omp for [clauses]
```

Attention! fct1() vs fct2()

```
#define NTHREADS 4
#define N 5

void fct1(void)
{
#pragma omp parallel for num_threads(NTHREADS)
   for (int  i = 0; i <N; i++)
   {
      printf("hello ");
   }
}

void fct2(void)
{
#pragma omp parallel num_threads(NTHREADS)
   for (int  i = 0; i <N; i++)
   {
      printf("hello ");
   }
}
```

fct1() prints hello **5** times

fct2() prints hello **20** times

Parallelizable for statements

```
                                          index++
                                          ++index
                      index < end         index--
                      index <= end        --index
for  index = start  ; index >= end  ;     index += incr
                      index > end         index -= incr
                                          index = index + incr
                                          index = incr + index
                                          index = index - incr
```

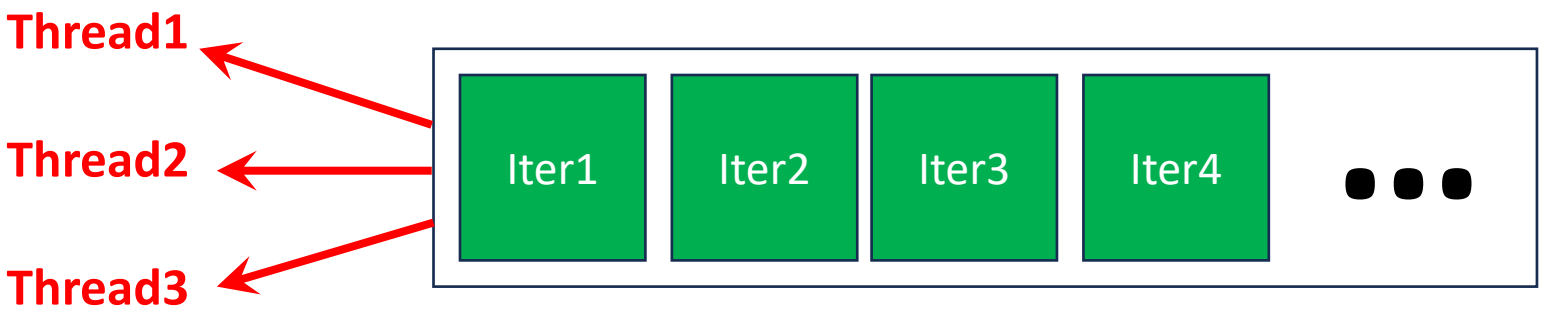
- The variable index must have integer or pointer type (it can't be a float)
- The expressions start, end, and incr must not change during execution of the loop
- During execution of the loop, the variable index can only be modified by the "increment expression" in the for statement

Example: Non-paralellizable loop

```
found = 0;
i = 0;
while ((i < N) && (!found))
{
    if (a[i] == X)
        found = 1;
    else
        i++;
}
```

Parallel for – How it works

- **Threads are NOT created for every iteration!**
- The thread group is created BEFORE the iterations start
- **The iterations are similar to jobs that need to be done and will be picked up by the existing threads**
- By default, iterations are assigned to threads according to the default cyclic (round-robin) schedule, but this can be changed with the schedule clause



Parallel for

- Sum of N numbers, with parallel for

```
int s = 0;

#pragma omp parallel for shared(s)
{
   for (int i = 0; i < N; i++) {
   #pragma omp critical
      s = s + i;
    }
} //end parallel

 printf("s=%d \n", s);
```

Every thread executes a part of the iterations and contributes to global shared s
Must use critical section!

Correct code, but
Bad performance

Parallel for

- Sum of N numbers, with parallel for

```
s = 0;

#pragma omp parallel shared(s)
  {
   int local_s = 0;

  #pragma omp for
     for (int i = 0; i < N; i++)
        local_s = local_s + 1;

  #pragma omp critical
     s = s + local_s;
  } //end parallel

  printf("s=%d \n", s);
```

Threads are created only by the #pragma omp parallel ! The #pragma omp for just distributes iterations on existing threads

Every thread has local_s and contributes to global shared s Must use critical section!

Correct code, OK performance, but **unnecessary complicated**

Parallel for

- **Typical usage**: Sum of N numbers, with **parallel for and reduction**

```
 int  s = 0;
#pragma omp parallel for reduction(+:s)
   for (int i = 0; i < N; i++)
      s = s + i;

   printf("for critical s=%d \n", s);
```

Every thread gets a number of iterations and computes its local partial sum in s

After the parallel region, s contains the sum of the local copies. Merging in a critical section was implicit done by reduction!

Parallel for – Data dependencies

- OpenMP parallelizes the for loop by dividing the iterations of the loop among the threads
- *Important precondition: Before putting a parallel for directive,* **the programmer** *must ensure that the iterations can be executed in any order!*
- If you put a parallel for directive on a loop in which the results of one or more iterations depend on other iterations, you have a race condition and incorrect results

Data dependencies - Counterexample

```
void main()
{
    int fibo[N];

    fibo[0] = fibo[1] = 1;

#pragma omp parallel for num_threads(4)
    for (int i = 2; i < N; i++)
        fibo[i] = fibo[i - 1] + fibo[i - 2];

    for (int i = 0; i < N; i++)
        printf("%d ", fibo[i]);
}
```

Iteration i depends on iteration i-1 and i-2 !

The correct output should be:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

Different runs of this incorrect parallel program produce various outputs!

Trapezoidal - Parallel v3 – use parallel for

```
double parallel_integral_v3(double a, double b, int n, int thread_count)
{
   double integral;
   double h = (b - a) / n;
   int i;
   integral = (f(a) + f(b)) / 2.0;

#pragma omp parallel for num_threads(thread_count) reduction(+ : integral)
   for (i = 1; i <= n - 1; i++)
   {
      integral += f(a + i * h);
   }
   integral = integral * h;
   return integral;
}
```

Code Examples

- omp_trapezoids.c (V1, V2 and V3)

Scheduling policies for iterations

The Schedule Clause

- #pragma omp parallel for schedule(type, chunksize)
- Type can be:
  - static: the iterations are assigned to the threads before the loop is executed. Assignment is done taking into account iteration number and thread id
  - dynamic or guided: the iterations are assigned to the threads while the loop is executing. Threads that are free at the moment get iterations assigned
  - auto: the compiler and/or the run-time system determine the schedule.
  - runtime: the schedule is determined at run-time.
- The chunksize (a positive integer) specifies how many iterations are scheduled together as a “package”.

Test the effect of different scheduling types

```
#define NTHREADS 4
#define POLICY static // or dynamic
#define CHUNKSIZE 2   // or 1,  3
#define N 10


void test(void)
{
#pragma omp parallel for num_threads(NTHREADS) schedule(POLICY, CHUNKSIZE)
   for (int  i = 0; i <N; i++)
   {
      printf("iteration %d done by thread %d \n", i, omp_get_thread_num());
   }
}
```

The effect of different scheduling types

	Static, Chunk=1	Static, Chunk=2	Dynamic, Chunk=1	Dynamic, Chunk=2
Iteration 0	Thread 0	Thread 0	Thread 3	Thread 1
Iteration 1	Thread 1	Thread 0	Thread 0	Thread 1
Iteration 2	Thread 2	Thread 1	Thread 1	Thread 0
Iteration 3	Thread 3	Thread 1	Thread 2	Thread 0
Iteration 4	Thread 0	Thread 2	Thread 3	Thread 3
Iteration 5	Thread 1	Thread 2	Thread 3	Thread 3
Iteration 6	Thread 2	Thread 3	Thread 2	Thread 2
Iteration 7	Thread 3	Thread 3	Thread 1	Thread 2
Iteration 8	Thread 0	Thread 0	Thread 0	Thread 1
Iteration 9	Thread 1	Thread 0	Thread 0	Thread 1

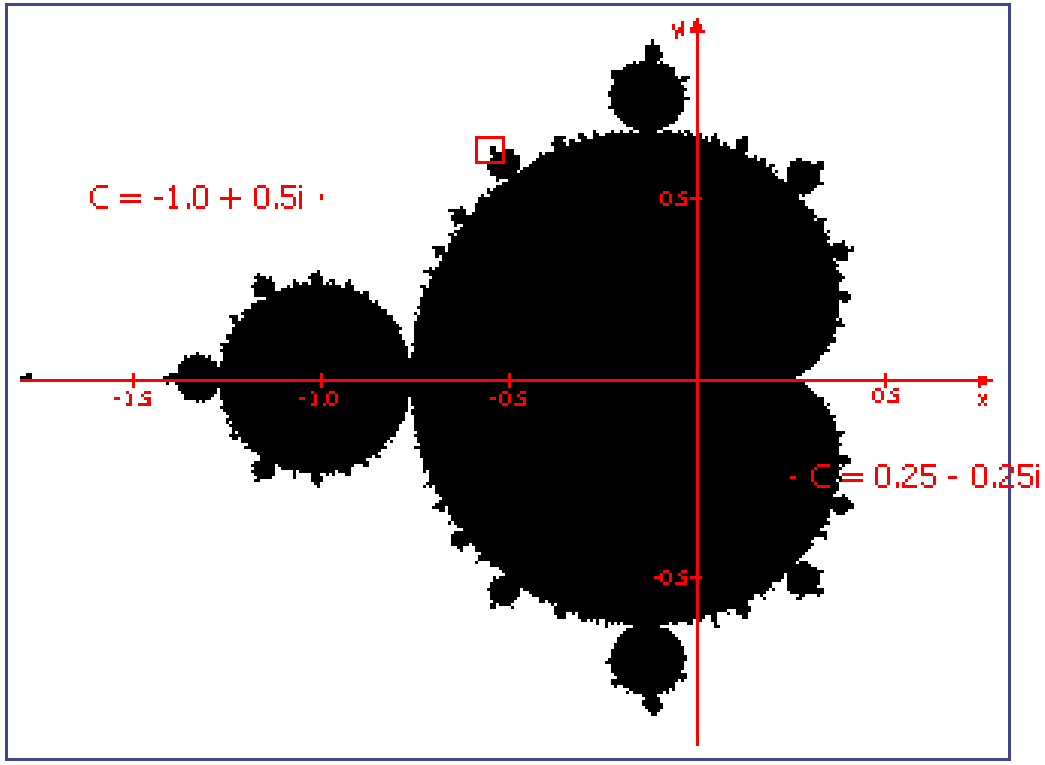
Always the same

Every time different

Using the Schedule Clause

- If all iterations perform ***the same amount*** of computation: ***use static scheduling***
  - In this case dynamic scheduling would just introduce an unnecessary overhead
- If each iteration performs a ***different amount*** of computation: ***use dynamic scheduling as a form of load balancing***
  - With dynamic scheduling of iterations: tasks are known in advance (the iterations), but task scheduling is done dynamically (deciding at runtime which thread gets which iteration)

Example: The Mandelbrot Set



- C is a complex number
- Use C in the iterative formula of the series Z:

$$Z_0 = 0$$

$$Z_{n+1} = Z_n^2 + C$$

- Does the point C belong to the Mandelbrot set?
  - Yes, if Z converges
  - No, if Z does not converge

http://matthiasbook.de/papers/parallelfractals/mandelbrot.html

Problem: Compute the Area of The Mandelbrot Set

- This problem has no known analytical solution
  - the area of the Mandelbrot set is approximately 1.506484, with a 95% confidence interval from 1.506480 to 1.506488 [article]
- Numeric solution:
- adaptation of a MonteCarlo solution (avoids using random()):
  - Consider a rectangle that is known to frame the Mandelbrot set
  - Generate *all* points inside the rectangle (generate *a grid* of points inside the rectangle, the more points the better)
    - For every point, test if it belongs to Mandelbrot set (test convergence of Z)
    - If yes, increment counter insidepoints
  - An approximation of the area is given by:

    Area = insidepoints/total grid points * rectanglearea

```c
void compute_serial(double *area, double *error) {
    int numinside = 0, numoutside = 0;

    for (int i = 0; i < NPOINTS; i++)
        for (int j = 0; j < NPOINTS; j++) {
            // generate grid of points C in the rectangle
            struct complex c;
            c.r = -2.0 + 2.5 * (double)(i) / (double)(NPOINTS);
            c.i = 1.125 * (double)(j) / (double)(NPOINTS);

            struct complex z;
            z = c; // start computing series z for c
            for (int iter = 0; iter < MAXITER; iter++)  {
                double temp = (z.r * z.r) - (z.i * z.i) + c.r;
                z.i = z.r * z.i * 2 + c.i;
                z.r = temp;
                if ((z.r * z.r + z.i * z.i) > 4.0) { // z diverges
                    numoutside++;
                    break;
                }
            }
        }
    numinside = NPOINTS * NPOINTS - numoutside;
    *area = 2.0 * 2.5 * 1.125 * (double)(numinside) / (double)(NPOINTS * NPOINTS)
    *error = *area / (double)NPOINTS;
}
```

```
void compute_parallel_v1(double *area, double *error){
    int numinside = 0, numoutside = 0;
    #pragma omp parallel for default(shared) schedule(static) \
                 reduction(+ : numoutside) num_threads(NUMTHREADS)
    for (int i = 0; i < NPOINTS; i++)
        for (int j = 0; j < NPOINTS; j++)  {
            // generate grid of points C in the rectangle
            struct complex c;
            c.r = -2.0 + 2.5 * (double)(i) / (double)(NPOINTS);
            c.i = 1.125 * (double)(j) / (double)(NPOINTS);
            struct complex z;
            z = c; // start computing series z for c
            for (int iter = 0; iter < MAXITER; iter++) {
                double temp = (z.r * z.r) - (z.i * z.i) + c.r;
                z.i = z.r * z.i * 2 + c.i;
                z.r = temp;
                if ((z.r * z.r + z.i * z.i) > 4.0) { // z diverges
                    numoutside++;
                    break;
                }
            }
        }
    numinside = NPOINTS * NPOINTS - numoutside;
    *area = 2.0 * 2.5 * 1.125 * (double)(numinside) / (double)(NPOINTS * NPOINTS)
    *error = *area / (double)NPOINTS;
}
```

- V1:

```
#pragma omp parallel for default(shared) schedule(static) \
                reduction(+ : numoutside) num_threads(NUMTHREADS)
```

- V2:

```
#pragma omp parallel for default(shared) schedule(dynamic) \
                reduction(+ : numoutside) num_threads(NUMTHREADS)
```

- V3:

```
#pragma omp parallel for default(shared) schedule(static, CHUNKSIZE) \
                reduction(+ : numoutside) num_threads(NUMTHREADS)
```

- V4:

```
#pragma omp parallel for default(shared) schedule(dynamic, CHUNKSIZE) \
                reduction(+ : numoutside) num_threads(NUMTHREADS)
```

Nested loops

- It is possible to parallelize the inner loop, but in most cases it is better to parallelize the outer loop!

```
#pragma omp parallel for
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            dowork
```

```
    for (int i = 0; i < N; i++)
#pragma omp parallel for
        for (int j = 0; j < N; j++)
            dowork
```

A team of threads is created and destroyed at every iteration on i!
This is a lot of **overhead** for very finegrained tasks

Nested parallelism

- Putting a pragma parallel for on several nested loops:
- Not all openmp implementations support the feature
- Even for these that support, you have to explicitly turn the feature on by setting OMP_NESTED or omp_set_nested
- It is worth doing it only if the hardware efficiently supports such a big number of threads

```
#pragma omp parallel for
    for (int i = 0; i < N; i++)
#pragma omp parallel for
        for (int j = 0; j < N; j++)
                dowork
```

Every thread created at for i spawns a new team of threads

The Collapse Clause

- Collapse(n) -Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause

```
#pragma omp parallel for collapse(2)
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
               dowork
```