



The Abdus Salam
International Centre
for Theoretical Physics



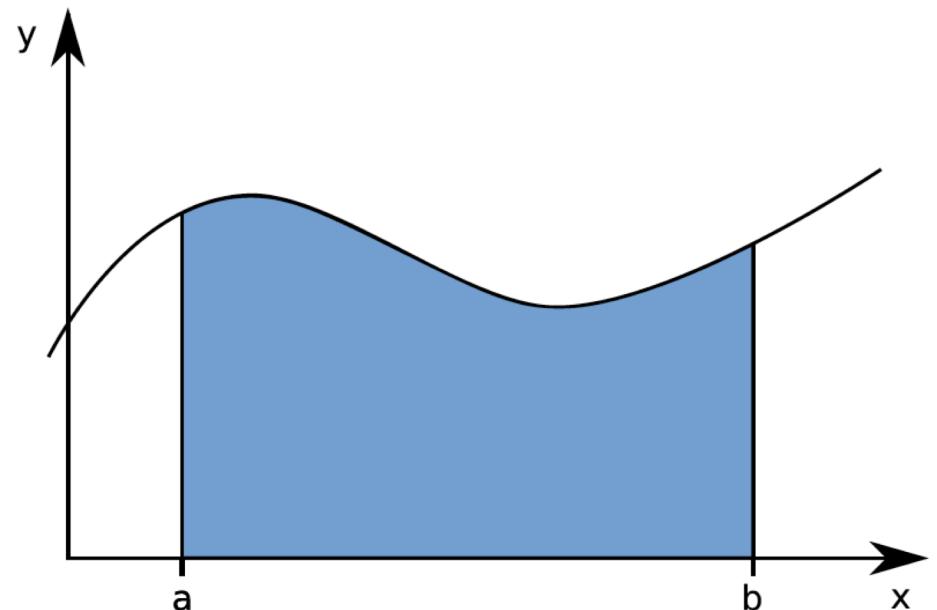
Parallel Programming 101

<https://github.com/igirotto/DSSC>

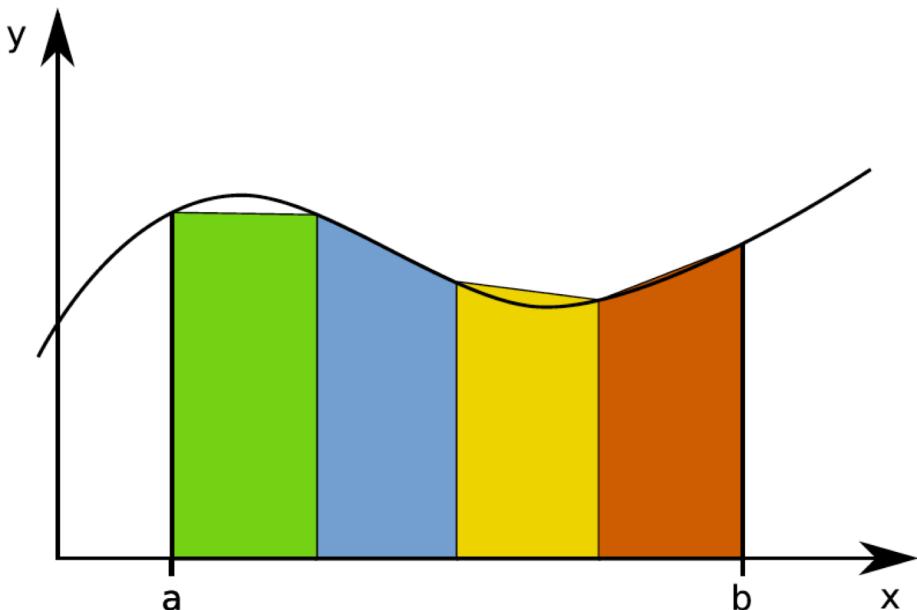
Ivan Girotto – igirotto@ictp.it

International Centre for Theoretical Physics (ICTP)

Synchronization



(a)



(b)

Figure: Approximating the area using the trapezoidal rule

Synchronization

$$\int_a^b f(x) dx \approx \underbrace{\frac{(b-a)}{n}}_h \left[\frac{f(x_0) + f(x_1)}{2} + \frac{f(x_1) + f(x_2)}{2} + \dots + \frac{f(x_{n-2}) + f(x_{n-1})}{2} + \frac{f(x_{n-1}) + f(x_n)}{2} \right]$$

$$\int_a^b f(x) dx \approx \underbrace{\frac{(b-a)}{n}}_h \left[\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \dots + f(x_{n-1}) + \frac{f(x_n)}{2} \right]$$

```
double h = (b - a) / n;
double approx = (f(a) + f(b)) / 2.0;

for(int i = 1; i <= n-1; ++i) {
    double x_i = a + i*h;
    approx += f(x_i);
}
approx = h * approx;
```

Synchronization

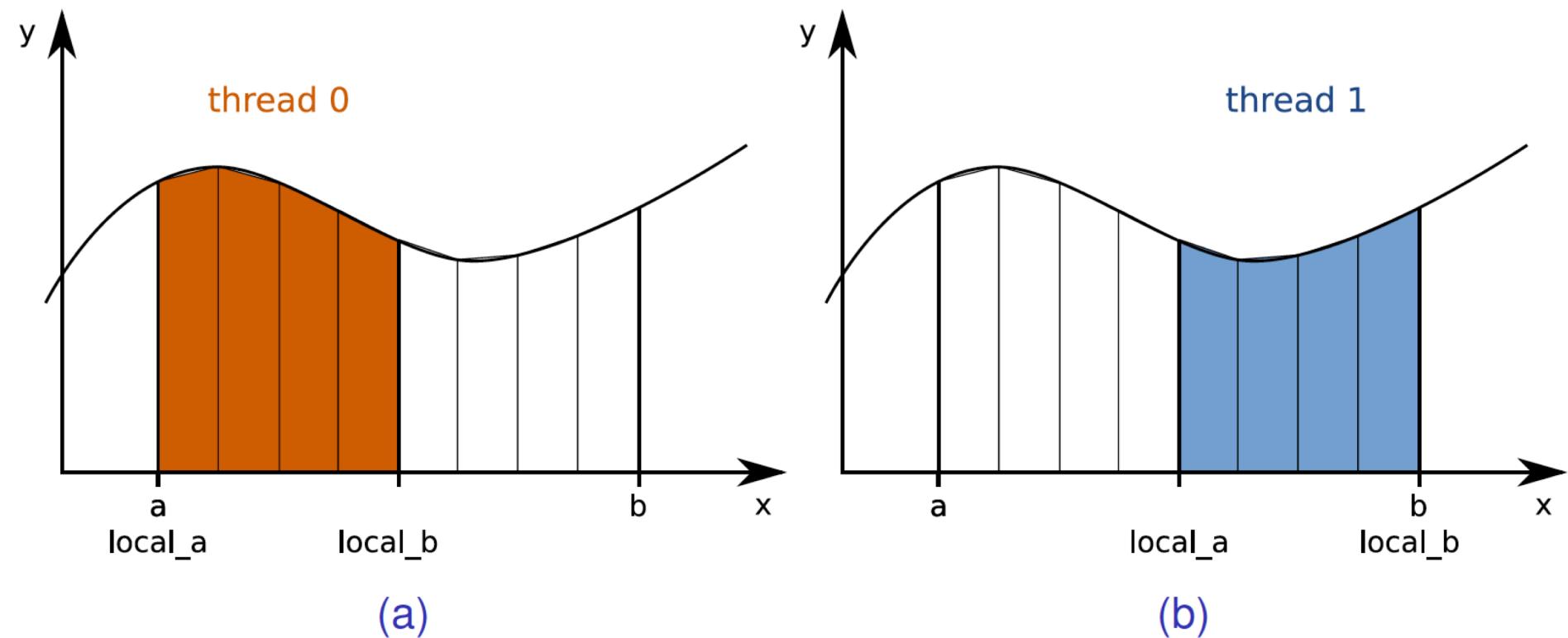
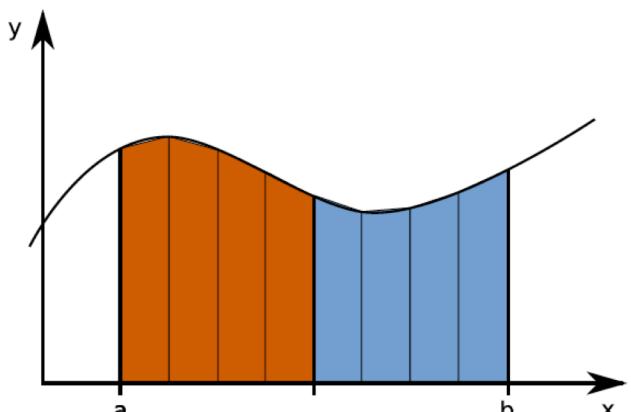
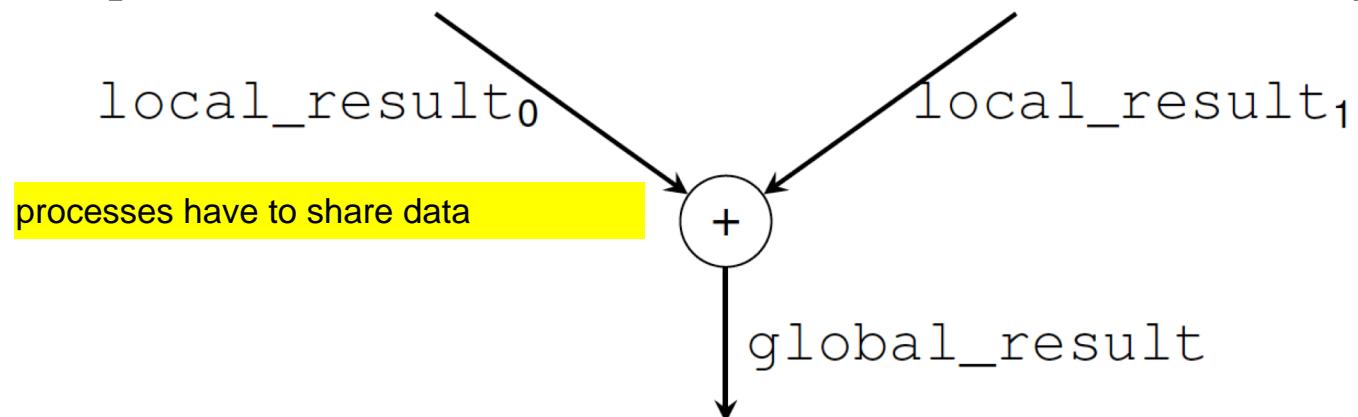


Figure: Splitting the work between two threads

Synchronization



```
double global_result = 0.0;

#pragma omp parallel
{
    double h = (b - a) / n;
    int local_n = N/N_threads
    double local_a = Local_n*Th_id
    double local_b = local_n+start

    double local_result = (f(local_a) + f(local_b)) / 2.0;
    for(int i = 1; i <= local_n-1; ++i) {
        double x_i = local_a + i*h;
        local_result += f(x_i);
    }
    local_result = h * local_result;

    ...
}
```

A race condition exists when the following is true:

1. one or more threads read the same data location
2. at least one of them is writing that data location

A block of code that causes a race condition is called a **critical section**.

global result= local result.we're both reading global result (which has to be loaded in register) and writing.so PROTEC

Synchronization: critical directive

- ▶ ensures that threads have mutually-exclusive access to a block of code
- ▶ only one thread can enter a critical section
- ▶ effectively serializes execution of this block of code
- ▶ **all unnamed critical blocks in the same parallel block are treated as one**
- ▶ **expensive!** because is the O.S. that schedule threads in run time-> VERY SLOW. Better to be used only for b

```
#pragma omp critical
global_result += local_result;
```

```
#pragma omp critical
{
    global_result += local_result;
}
```

Synchronization: Named critical blocks



- ▶ Non overlapping critical sections can run in parallel
- ▶ Useful to minimize blocking if not needed

```
if(...) {
    // threads that go this way...
#pragma omp critical(a)
{
    // modify shared resource a
}
} else {
    // ... don't block threads that
    // go this way.
#pragma omp critical(b)
{
    // modify shared resource b
}
}
```

Synchronization: atomic directive

- ▶ atomic enables mutual exclusion for some simple operations
- ▶ these are converted into special hardware instructions if supported
- ▶ however, it only protects the read/update of the target location

hardware-> fast, but very easy instructions

acceptable operations

- ▶ $x++$
- ▶ $x--$
- ▶ $++x$
- ▶ $--x$
- ▶ $x \text{ binop} = \text{expr}$
- ▶ $x = x \text{ binop expr}$
- ▶ $x = \text{expr binop } x$

where binop is one of

+ * - / & ^ | << >>

```
#pragma omp parallel
{
    // compute my_result

    #pragma omp atomic
    x += my_result;
}
```

```
#pragma omp parallel
{
    #pragma omp atomic
    x += func(); // warning func() is not atomic!
}
```

it makes the add noncritical, but you can have problems in func(). Atomic guarantees sequential access only

Synchronization: barrier directive



This directive synchronizes the threads in a team by causing them to wait until all of the other threads have reached this point in the code.

```
#pragma omp parallel
{
    // do something in parallel using your team of threads

    #pragma omp barrier
    // wait until all threads reach this point

    // continue computation in parallel
}
```

Timing



```
double tstart = omp_get_wtime();
// do work
double duration = omp_get_wtime() - tstart;
```

```
#pragma omp parallel
{
    double tstart = omp_get_wtime();
    // do part 1

    #pragma omp barrier

    double part1_duration = omp_get_wtime() - tstart;

    // continue with part 2
}
```

Timing



```
double tstart = omp_get_wtime();
// do work
double duration = omp_get_wtime() - tstart;
```

```
#pragma omp parallel
{
    double tstart = omp_get_wtime();
    // do part 1

    #pragma omp barrier

    double part1_duration = omp_get_wtime() - tstart;

    // continue with part 2
}
```

Define a function which does the local sum in range $[local_a, local_b]$

```
double local_sum(double local_a, double local_b, int local_n, double h) {  
    double local_result = (f(local_a) + f(local_b)) / 2.0;  
    for(int i = 1; i <= local_n-1; ++i) {  
        double x_i = local_a + i*h;  
        local_result += f(x_i);  
    }  
    local_result = h * local_result;  
    return local_result;  
}
```

Parallel Integral



```
double global_result = 0.0;

#pragma omp parallel
{
    double h = (b - a) / n;

    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();

    int local_n = n / nthreads;
    double local_a = a + tid * local_n * h;
    double local_b = local_a + local_n * h;

    // what is wrong with this code? critical is slow
    #pragma omp critical
    global_result += local_sum(local_a, local_b, local_n, h);
}
```

Parallel Integral



```
double global_result = 0.0;

#pragma omp parallel
{
    double h = (b - a) / n;

    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();

    int local_n = n / nthreads;
    double local_a = a + tid * local_n * h;
    double local_b = local_a + local_n * h;

    double local_result = local_sum(local_a, local_b, local_n, h);

    #pragma omp atomic
    global_result += local_result;
}
```

Synchronization: reduction clause



- ▶ creates a private variable for each thread
- ▶ each thread works on private copy
- ▶ finally all thread results are accumulated using operator
- ▶ allowed operators: +, -, *, &, |, ^, &&, ||, min, max
- ▶ each operator has a default initialization value (e.g. 0 for addition, 1 for multiplication)

```
double global_result = 0.0;

#pragma omp parallel reduction(+:global_result)
{
    double h = (b - a) / n;
    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    int local_n = n / nthreads;
    double local_a = a + tid * local_n * h;
    double local_b = local_a + local_n * h;

    global_result += local_sum(local_a, local_b, local_n, h);
}
```

Synchronization: nowait clause



Work sharing constructs have an implicit barrier at their end. With nowait you can allow them to continue after they finish their part.

```
#pragma omp parallel
{
    #pragma omp for
    for(...) {
    }
    // implicit barrier

    #pragma omp for
    for(...) {
    }
}
```

```
#pragma omp parallel
{
    #pragma omp for nowait
    for(...) {
    }
    // threads can continue

    #pragma omp for
    for(...) {
    }
}
```

Synchronization: master clause



```
#pragma omp parallel
{
    #pragma omp master
    {
        // only master thread should execute this
        // useful for I/O or initialization
        // there is NO implicit barrier!
    }

    // add explicit barrier if needed
    #pragma omp barrier
    ...
}
```

Synchronization: single clause

```
#pragma omp parallel
{
    #pragma omp single
    {
        // only one thread will execute this block
        // all others wait until it completes
        // implicit barrier!
    }
}
```

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
        // only one thread will execute this block
        // others will go right past it
    }
}
```

<https://computing.llnl.gov/tutorials/openMP/>