Introduction to Parallel Computing

Shared-memory programming with OpenMP

Table of Contents

Introduction

- OpenMP is an API for shared-memory Multiple Instruction Multiple Data (MIMD) programming.
- OpenMP is designed for systems in which each thread/process has access to all available memory.

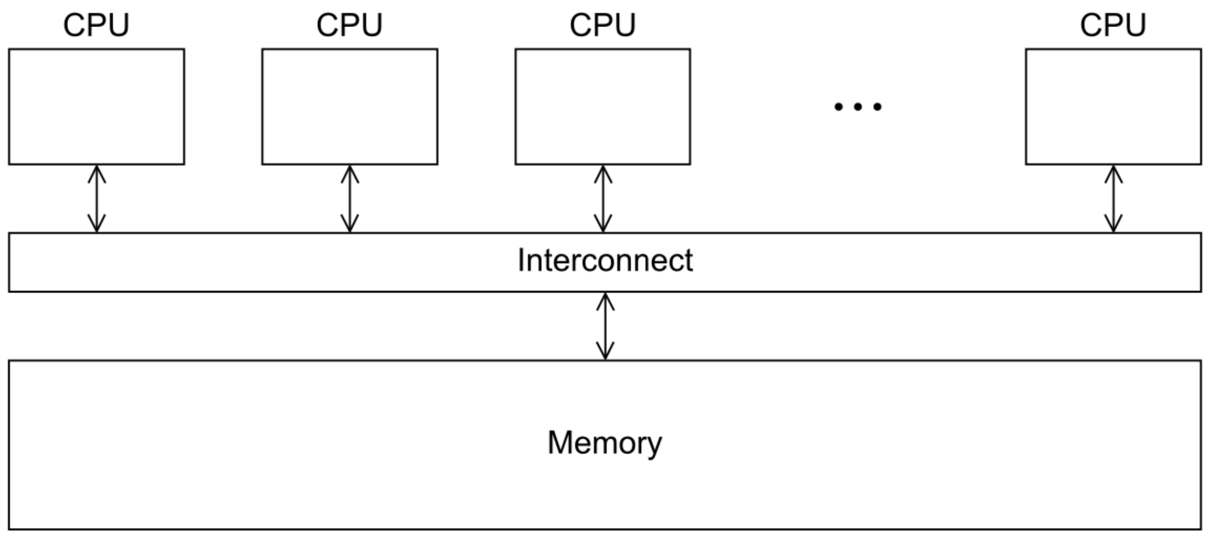


FIGURE 5.1

A shared-memory system.

- It allows the programmer to **state that a block of code should be executed in parallel**, and the precise determination of the tasks and which thread should execute them is left to the **compiler and the run-time system**.
- Some compilers and languages that support OpenMP:
    - GNU: GCC – C/C++/Fortran
    - LLVM: Clang – C/C++
    - Microsoft: MSVC – C/C++
    - Nividia: C/C++/Fortran
- OpenMP API is directive-based.
    - ***Directives*** are special preprocessor instructions called ***pragma***.
- The $\#\boldsymbol{pragma}$ directive is a special purpose directive that is used to turn on or off some features.

Program 5.1: A "hello, world" program that uses OpenMP.

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

void hello(void);

int main(int argc, char *argv[]) {
    int thread_count = strtol(argv[1], NULL, 10);

#pragma omp parallel num_threads(thread_count)
    hello();

    return 0;
}

void hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
}
```

- $strtol$ is a function in the $stdlib.h$ header that converts strings to long integer. It's used to convert the command line argument $argv[1]$ into a string. The first parameter is string value to be converted, the second is a reference to the end of the string, the third is the base of long value.
- $omp\ parallel\ num\_threads()$ tells the compiler that next line will be executed in parallel by a specific number of threads.
- $omp\_get\_thread\_num()$: a function returns the number (rank) of the current thread.
- $omp\_get\_num\_threads()$: a function returns the total number of threads that are assigned to the process.
- To compile and run the previous program:
  - For Clang and Apple M1

```
clang -Xpreprocessor -fopenmp -I/opt/homebrew/opt/libomp/include -
L/opt/homebrew/opt/libomp/lib -lomp -o my_program <filename.c>
```

  - For GCC on Windows, Linux, and Mac

```
gcc-14 -o my_program -fopenmp <filename.c>
```

  - Or

```
gcc -o my_program -fopenmp <filename.c>
```

  - Then, run using

```
./my_program 4
```

- The program output is different from a run to another. This is because each thread is competing for access to $stdout$.

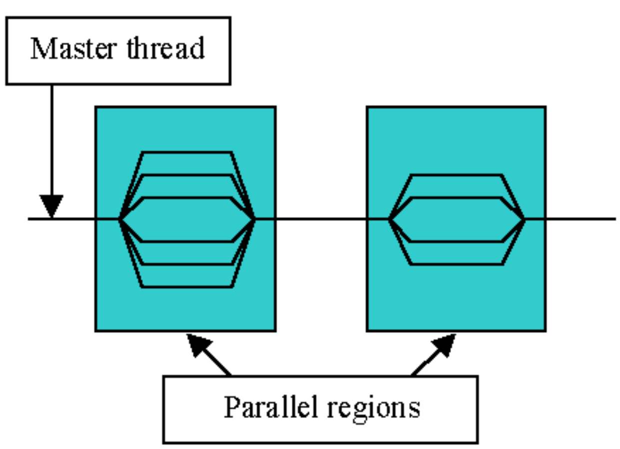
```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

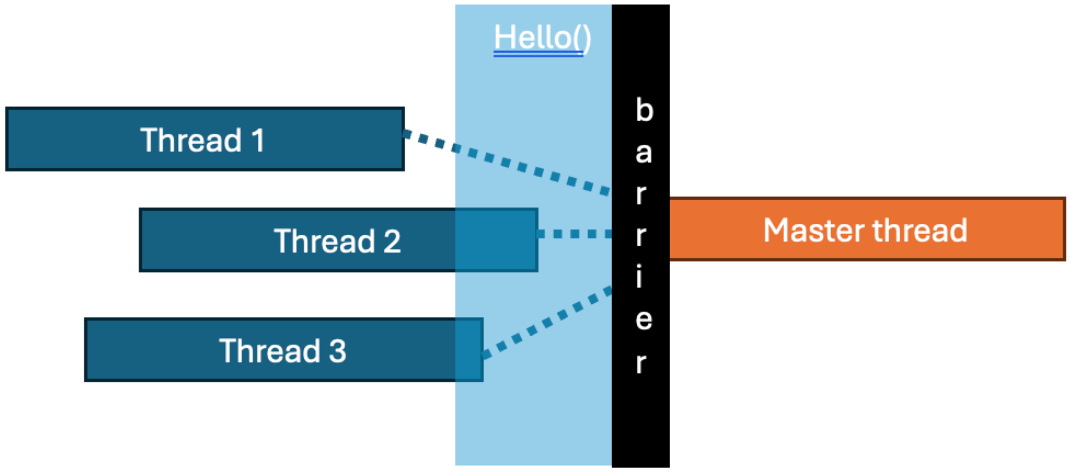
or

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

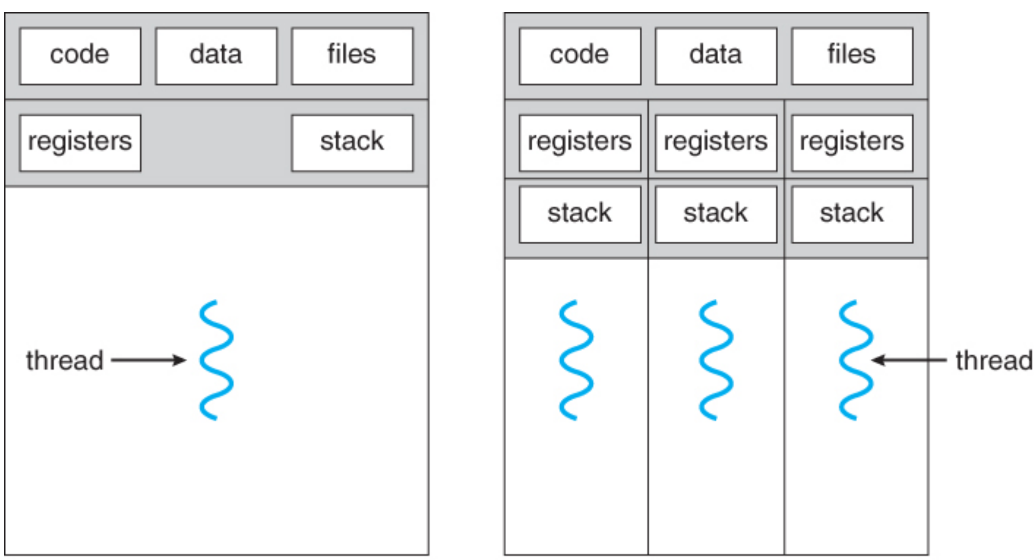
- When the program **reaches the $\boldsymbol{parallel}$ directive**, the original thread **forks** $thread\_count - 1$ additional threads.
- The collection of threads executing the $parallel$ block – the original thread and the new threads – is called a **team**.
    - **Master** thread: the first thread of execution, thread 0
    - **Parent** thread: thread that encountered a $parallel$ directive and started a team of threads.
        - In many cases, the parent is also the master thread.
    - **Child** thread: each thread started by the parent.



- **Implicit barrier**: a thread that has completed the parallel block of code will wait for all the other threads in the team to complete the block.



- Each thread has its own **stack**; so, a thread executing the $Hello$ function will create its own private, local variables in the function.
- Since $\boldsymbol{stdout}$ is shared among the threads, each thread can execute the $printf()$ to print its rand and the number of threads.
  - There is no **scheduling** of access to $stdout$, so the actual order in which the threads print their results is **nondeterministic**.



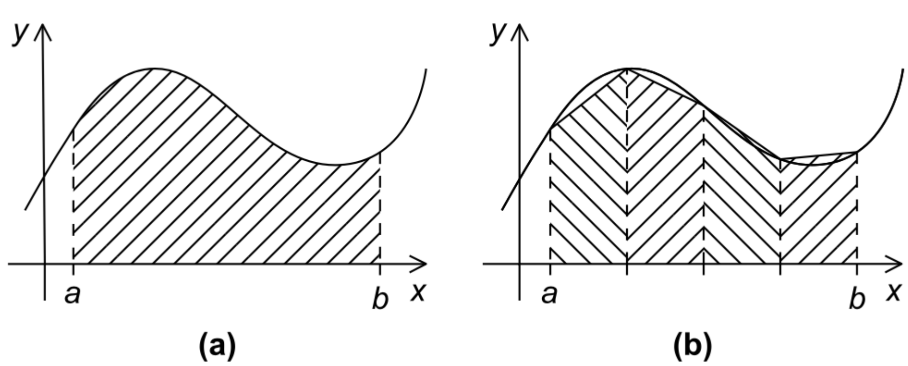
single-threaded process multithreaded process

- To handle missing *omp* or if a compiler doesn't support *omp*, we can modify the previous program as follows:

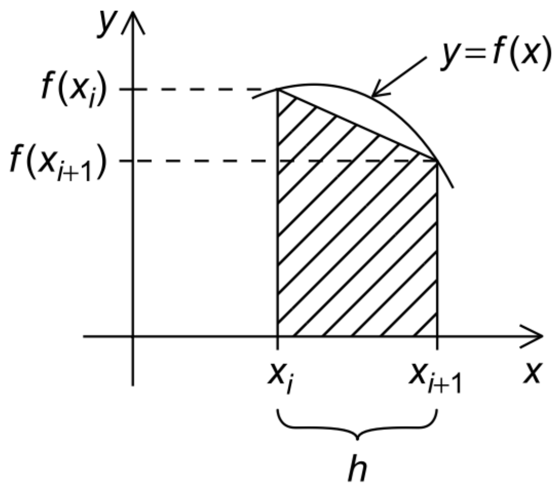
```
#include <stdio.h>
#include <stdlib.h>

#ifdef _OPENMP

#include <omp.h>

#endif

void hello(void);

int main(int argc, char *argv[]) {
    int thread_count = strtol(argv[1], NULL, 10);

#pragma omp parallel num_threads(thread_count)
    hello();

    return 0;
}

void hello(void) {
#ifdef _OPENMP
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
#else
    int my_rank = 0;
    int thread_count = 1;
#endif

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
}
```

The trapezoidal rule

- The trapezoidal rule is used to **estimate the area under the curve**.
- To estimate the area between the graph $f(x)$, the vertical lines $x = a$ and $x = b$ and the x-axis, divide the interval $[a, b]$ into n subintervals and approximating the area over each subinterval by the area of a trapezoid.



- The area of each subinterval is calculated using: $\frac{h}{2}\left(f(x_i) + f(x_{i+1})\right)$



  - $h = x_{i+1} - x_i$ → the length of each subinterval
  - $f(x_i)$ and $f(x_{i+1})$ is the length of the vertical segments.

- Since we are dividing the area into subintervals, we let
  - n is number of subintervals
  - b and a are the limits of the intervals
  - $h = (b - a)/n$, the length of each subinterval
  - $x_i = a + ih$ for $i = 0, 1, \ldots, n$

- Thus, if we call the leftmost endpoint x_0, and the rightmost endpoint x_n, we have that:
  $x_0 = a, \quad x_1 = a + h, \quad x_2 = a + 2h\ , \ldots , \quad x_{n-1} = a + (n-1)h, \quad x_n = b$

- The sum of the areas = $h\left(\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{f(x_n)}{2}\right)$
- Then the approximation is

```
/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

```c
#include <stdio.h>

double f(double x);     /* Function we're integrating */
double Trap(double a, double b, int n, double h);

int main(void) {
    double integral;   /* Store result in integral   */
    double a, b;       /* Left and right endpoints   */
    int n;          /* Number of trapezoids       */
    double h;

    printf("Enter a, b, and n\n");
    scanf("%lf", &a);
    scanf("%lf", &b);
    scanf("%d", &n);

    h = (b - a) / n;
    integral = Trap(a, b, n, h);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.15f\n", a, b, integral);

    return 0;
}  /* main */

double Trap(double a, double b, int n, double h) {
    double integral = (f(a) + f(b)) / 2.0;
    for (int k = 1; k <= n - 1; k++) {
        integral += f(a + k * h);
    }
    integral = integral * h;

    return integral;
}  /* Trap */

double f(double x) {
    return x * x;
}  /* f */
```

Parallelizing the trapezoidal rule

Partition the problem solution into tasks

- Find the area of a single trapezoid
- Computing the sum of these areas

Identify communication channels

- Join the task of computing the area of each trapezoid with the task of computing the final sum

Aggregate tasks into composite tasks

- Assign a contiguous block of trapezoids to each thread.
- Partition the interval [a,b] into smaller subintervals.

Map composite tasks to cores

- Each thread applies the serial trapezoidal rule to its subinterval
- Each thread will be executed on a signle core

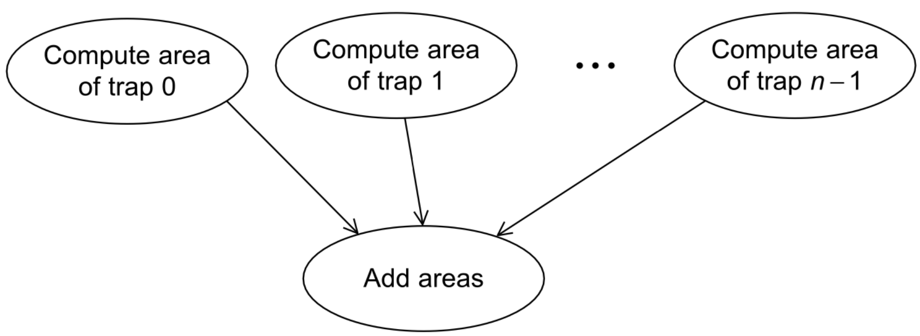
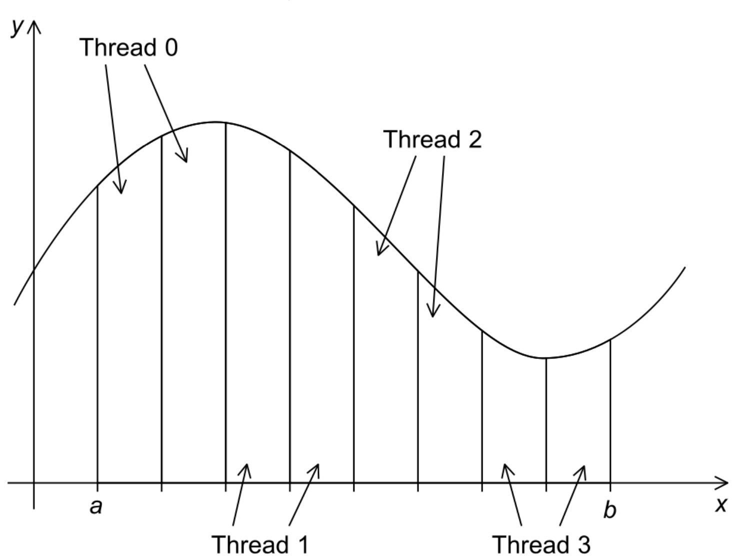


FIGURE 3.5

Tasks and communications for the trapezoidal rule.



Parallelizing the problem:

- Assume we have $a = 0,\ b = 500, n = 100$ and the number of threads = 4

Compute the length of each subinterval

- $h = (b - a)/n$ ==> $h = (500 - 0)/100$=5

Compute the number of intervals assigned to each thread

- $local_n = n/num\_threads$ ==> $local_n = 100/4$=25

Compute the start and the end of sub-areas assigned to each thread

$local_a = \ a + rank * local_n * h \qquad local_b = local_a + local_n * h$

- $local_a = 0 + 0 \ * 25 \ * 5 = 0$, $local_b = 0 + 25 \ * 5 = 125$
- $local_a = 0 + 1 \ * 25 \ * 5 = 125$, $local_b = 125 + 25 \ * 5 = 250$
- $local_a = 0 + 2 \ * 25 \ * 5 = 250$, $local_b = 250 + 25 \ * 5 = 375$
- $local_a = 0 + 3 \ * 25 \ * 5 = 375$, $local_b = 375 + 25 \ * 5 = 500$

[0:125]	[125: 250]	[250:375]	[375:500]
(f(0) + f(125)) /2	(f(125) + f(250)) /2	(f(250) + f(375)) /2	(f(375) + f(500)) /2

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

Final result = my_result + my_result + my_result + my_result

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

double f(double x);     /* Function we're integrating */
void trap(double a, double b, int n, double *global_result_p);

int main(int argc, char *argv[]) {
    double global_result = 0.0;  /* Store result in global_result */
    double a, b;                 /* Left and right endpoints      */
    int n;                   /* Total number of trapezoids    */

    int thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);

#  pragma omp parallel num_threads(thread_count)
    trap(a, b, n, &global_result);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n", a, b, global_result);
    return 0;
}  /* main */

double f(double x) {
    return x * x;
}  /* f */

void trap(double a, double b, int n, double *global_result_p) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    double x;

    double h = (b - a) / n;
    int local_n = n / thread_count;
    double local_a = a + my_rank * local_n * h;
    double local_b = local_a + local_n * h;
    double my_result = (f(local_a) + f(local_b)) / 2.0;
    for (int i = 1; i <= local_n - 1; i++) {
        x = local_a + i * h;
        my_result += f(x);
    }
    my_result = my_result * h;

#  pragma omp critical
    *global_result_p += my_result;
}  /* trap */
```

- The program reads the number of threads to fork as a command line argument.
- Next, the program asks the user to enter**: a, the start of the interval**, **b, the end of the interval, and n the number of subintervals** (the smaller trapezoids).

- Notice that when we call the $trap$ function, we pass $\boldsymbol{a, b, n}$ **as values**, while the $\boldsymbol{global\_result}$ **is passed as an address**.
- Inside $trap()$:
  - The program gets the **rank** (ID) of the current thread. This used to compute the start and the end of the interval that is assigned to the current thread.
  - $\boldsymbol{thread\_count}$ is used to store the total number of threads used to execute the function. This is used to compute how many subintervals (smaller trapezoids) to each thread.
  - $\boldsymbol{h}$ **is the length** value of each trapezoid, which is computed by dividing the size of the interval $(b - a)$ by the total number of trapezoids (n).
  - $\boldsymbol{local\_n}$ is a variable that computes how many trapezoids are assigned to each thread. It is computed by dividing the total number of trapezoids by the total number of threads available.
  - $\boldsymbol{local\_a}$ and $\boldsymbol{local\_b}$ store the limits of each subinterval (trapezoid) assigned to each thread.
  - $\boldsymbol{my\_result}$ **is local integral value** of each thread. It's computed using the $local\_a$ and $local\_b$ instead of the limits of the entire interval $[a, b]$.
  - $\#$ $\boldsymbol{pragma\ omp\ critical}$ is a directive that tells the compiler to make the next statement to have **mutually exclusive** access. This means that the statement $* \, global\_result += my\_result$ will be executed by only one thread. If more than one thread is executing the same statement at the same time, the result will be incorrect.
  - A code that includes access to a shared resource and causes a **race condition**, is called a **critical section**.
    - **Race condition**: multiple threads are attempting to access a shared resource, at least one of the accesses is an update, and the accesses can result in an error.
- **Notice that the variables $\boldsymbol{local\_a}$, $\boldsymbol{local\_b}$, and $\boldsymbol{my\_result}$ are different for each thread.**
- If number of total trapezoids, n, is **not divisible** by the number of available threads, $thread\_count$, then we will result in **inaccurate** results and we must include error handling mechanisms.
- For example, if $\boldsymbol{n = 14}$ and $\boldsymbol{thread\_count = 4}$, then each thread will compute

  $$\boldsymbol{local\_n \ = \ n/thread\_count \ = \ 14/4 \ = \ 3}$$

  Thus, each thread will use 3 trapezoids and $global\_result$ will be computed with $4 \times 3 = 12$ instead of 14.
- We can handle this error by including an if condition to check that n is divisible by $thread\_count$.

```
if ( n % thread_count != 0) {
fprintf (stderr ,"n must be evenly divisible by thread_count\n" );
exit (0) ;
}
```

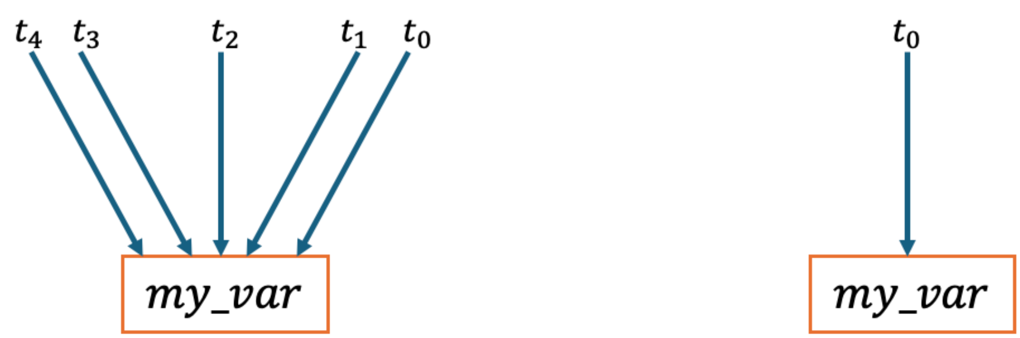
- During the execution of the program, the $local\_a = a + my\_rank * local\_n * h;$ will be as following:

```
thread 0:   a + 0*local_n*h
thread 1:   a + 1*local_n*h
thread 2:   a + 2*local_h*h

        . . .
```

  - So, the value of my_rank is replaced by the rank of the thread.
  - The same applies for the $local\_b = local\_a + local\_n * h;$

Scope of variables

- In serial programming, the **scope** of a variable consists of those parts of a program in which the variable can be used.
- In OpenMP, the **scope** of a variable refers to the set of threads that can access the variable in a parallel block.



Shared variable
all team threads can access it

Private variable
Only one thread can access it

- In the $trapezoidal$ program:
  - The variables declared in the $\boldsymbol{main}$ **function** ($a, b, n, global\_result, thread\_count$) are **shared;** they are declared before the $parallel$ directive.
  - The variables inside the $\boldsymbol{trap}$ function (h, x, my_rank, etc.) are **private**; they are declared within the scope of parallel directive.
  - The variable $\boldsymbol{global\_result\_p}$ in the $trap$ function is **private**. However, it refers to the **shared** variable $\boldsymbol{global\_result}$ that is defined in the main function. So, $\boldsymbol{global\_result\_p}$ is treated as a **shared variable**.
  - If $\boldsymbol{global\_result\_p}$ were **private** to each thread, there would be no need for the $critical$ directive. It must be shared to compute the final result computed by individual threads.

- In summary, variables that have been declared before a $parallel$ directive have **shared** scope, while variables declared in the block (e.g., local variables in functions) have **private** scope.