Introduction to Parallel Computing

Shared-memory programming with OpenMP

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.

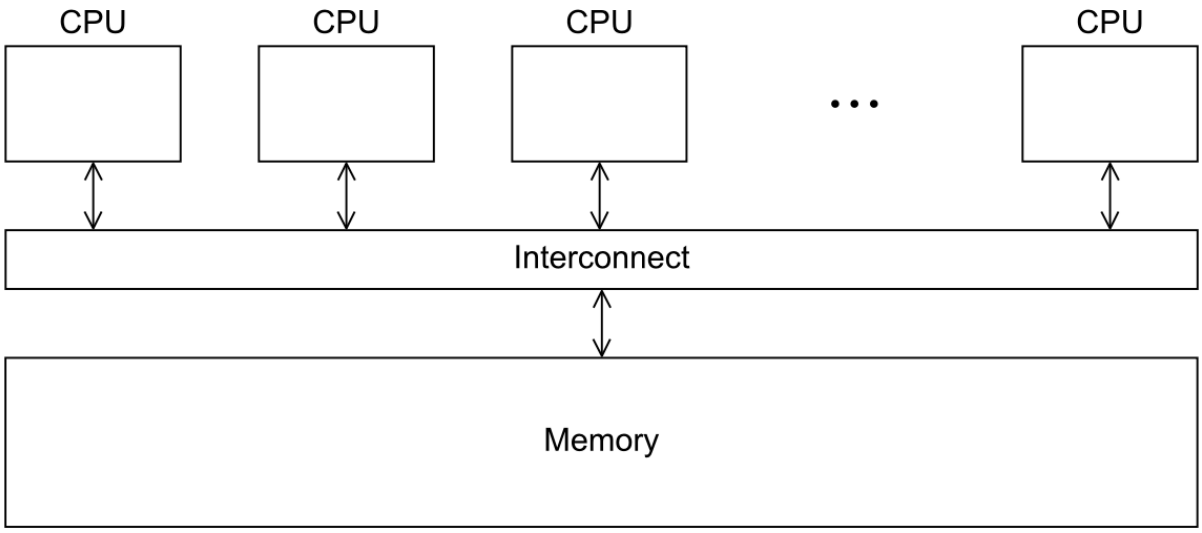


FIGURE 5.1

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 $\#pragma$ 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.

```c
#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 $stdout$.

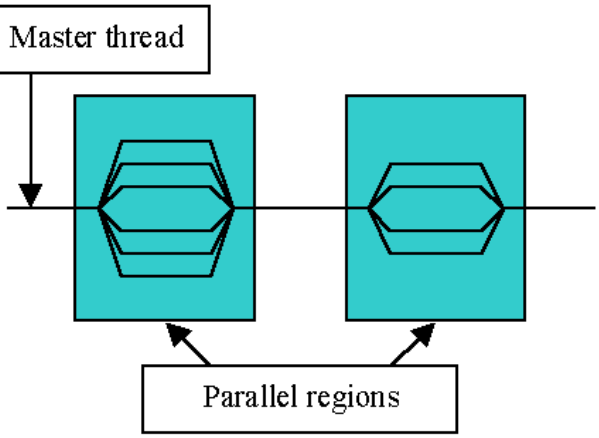
```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

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

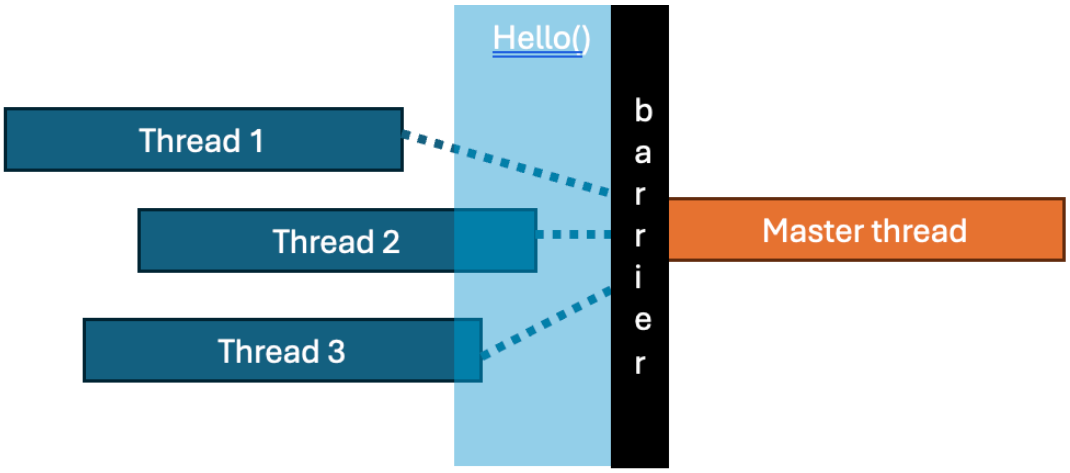
or

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

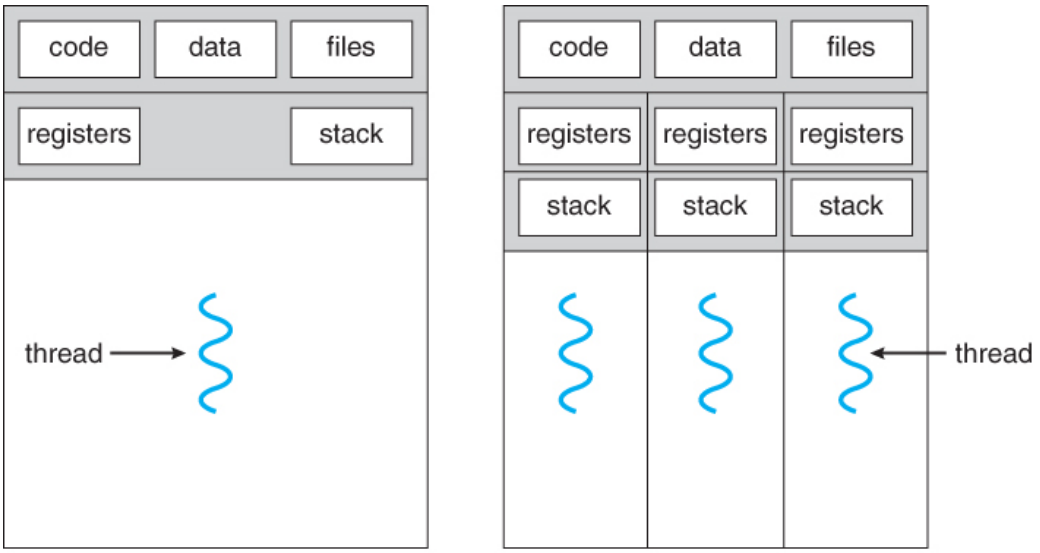
- When the program reaches the $parallel$ directive, the original thread forks $thread\_count - 1$ additional threads.
- The collection of threads executing the $parallel$ block – the original thread and the new threads – is called a **team**.
    - **Master** thread: the first thread of execution, thread 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 $stdout$ 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**.



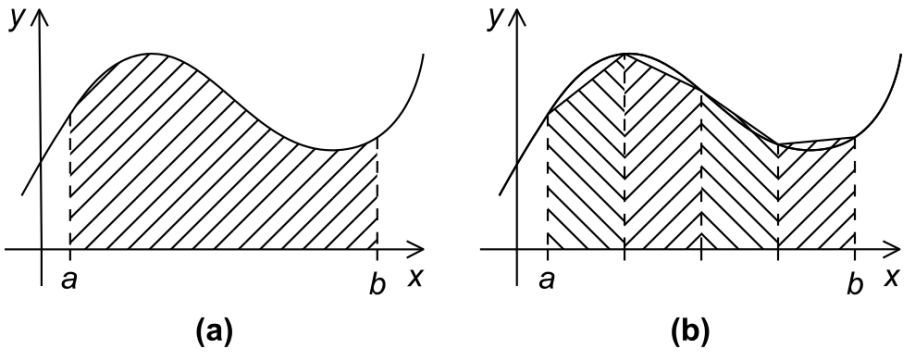
single-threaded process multithreaded process

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

```c
#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.



- Let
  - n is number of subintervals
  - b and a are the limits of the intervals
  - $h = (b - a)/n$
  - $x_i = a + ih$ for $i = 0, 1, \ldots, n$
- 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;          /* Height of trapezoids       */

    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 larger 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

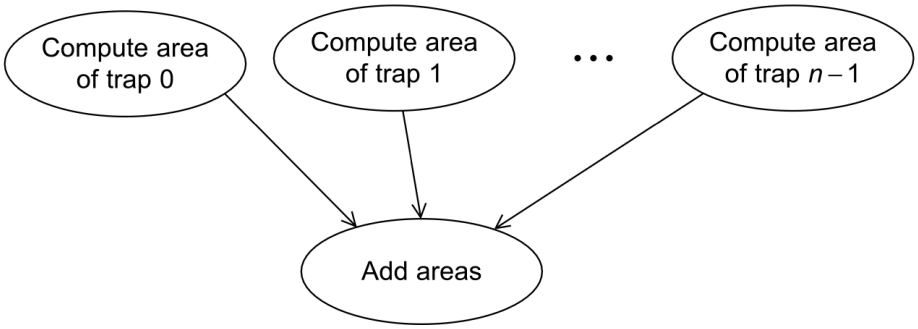
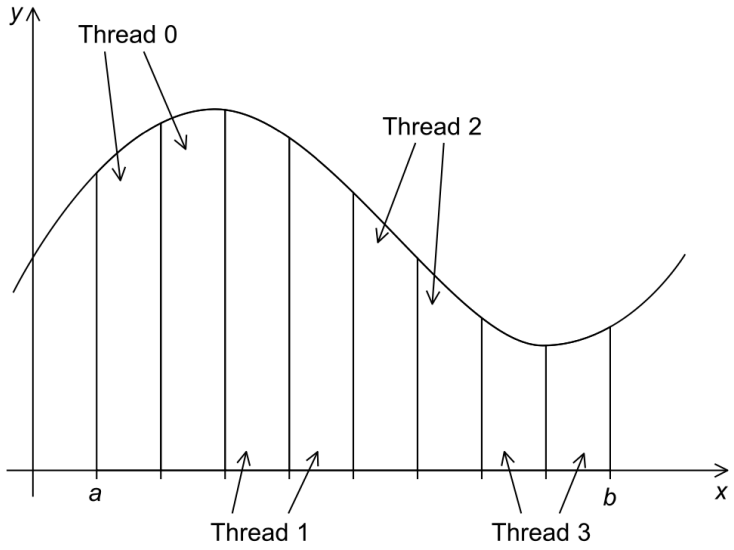


FIGURE 3.5

Tasks and communications for the trapezoidal rule.



```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 a, b, n as values, while the $global\_result$ 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.
  - $thread\_count$ 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.
  - h is the heigh value of each trapezoid, which is computed by dividing the size of the interval $(b - a)$ by the total number of trapezoids (n).
  - $local\_n$ 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.
  - $local\_a$ and $local\_b$ store the limits of each subinterval (trapezoid) assigned to each thread.
  - 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 $[a, b]$.
  - $\#\ pragma\ omp\ critical$ 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 $local\_a$, $local\_b$, and 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 have to include error handling mechanisms.
- For example, if $n = 14$ and $thread\_count = 4$, then each thread will compute

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

  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) ;
}
```

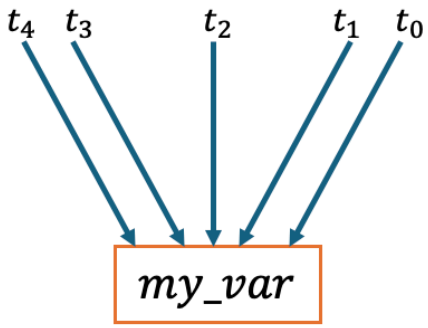
- During the execution of the program, the $local\_a = a + my\_rank * local\_n * h;$ will be as following:

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

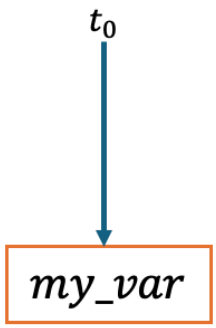
  - So, the value of my_rank is replaced by the rank of the thread.
  - The same applies for the $local\_b = local\_a + local\_n * h;$

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 ***main*** **function** ($a, b, n, global\_result, thread\_count$) are **shared;** they are declared before the $parallel$ directive.
  - The variables inside the ***trap*** function (h, x, my_rank, etc.) are **private**; they are declared within the scope of parallel directive.
  - The variable ***global_result_p*** in the $trap$ function is **private**. However, it refers to the **shared** variable ***global_result*** that is defined in the main function. So, ***global_result_p*** is treated as a **shared variable**.
  - If ***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.

The reduction clause

- What is wrong with this code? (it prints the correct result)

```
//gcc-14 -o main -fopenmp p5_5_trapz_parallel_bad.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

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

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)
    {
# pragma omp critical
        global_result += local_trap(a, b, n);
    }

    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 */

double local_trap(double a, double b, int n) {
    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;
    return my_result;
}  /* local_trap */
```

- The call to $local\_trap$ can only be executed by one thread at a time.
- Effectively, we're forcing the threads to execute the trapezoidal rule sequentially.

- How to fix the previous code?
  - Just move the critical section after the function call.

```c
//gcc-14 -o main -fopenmp p5_6_trapz_parallel_fix.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

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

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)
    {
        double my_result = 0.0;     /* private */
        my_result += local_trap(a, b, n);
# pragma omp critical
        global_result += my_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 */

double local_trap(double a, double b, int n) {
    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;
    return my_result;
} /* local_trap */
```

- Now the call to $local\_trap$ is outside the critical section, and the threads can execute their calls simultaneously.
- Note that my_result is **private** because it is declared in the parallel block.

- OpenMP provides a cleaner method to avoid serializing the execution of $local\_trap$ by using the **reduction clause**.
- A **reduction clause** consists of two elements:

Reduction operator

- An associative binary operation (+, *)

Reduction variable

- The variable in which the results of an operation (+, *) are stored

Reduction

- A computation that repeatedly applies the same **reduction operator** to a **sequence of operands** to get a single result

- For example, if A is an array of n integers, the computation

```
int sum = 0;
for (i = 0; i < n; i++)
    sum += A[i];
```

is a reduction in which the **reduction operator** is **+** and the **reduction variable** is $\boldsymbol{sum}$.

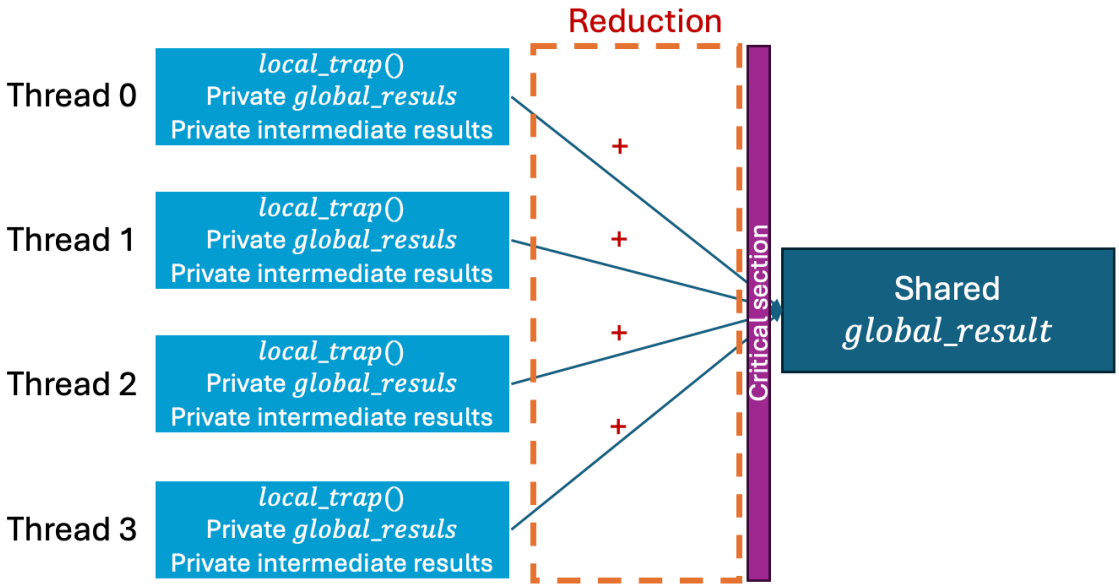
- Reduction clause in OpenMP is defined as follows

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count) \
       reduction(+: global_result)
    global_result += Local_trap(double a, double b, int n);
```

  - $< parallel\ directive > \ reduction(< operator >: < variable\ list >)$
  - Reduction variable → $global\_result$
  - Reduction operator → +

```c
//gcc-14 -o main -fopenmp p5_7_trapz_parallel_reduction.c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

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

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;

    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) \
      reduction(+: global_result)
    global_result += local_trap(a, b, n);

    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) {
    double return_val;

    return_val = x * x;
    return return_val;
}  /* f */

double local_trap(double a, double b, int n) {
    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;

    return my_result;
}  /* Trap */
```

- Behind the scenes,
  - OpenMP creates a private variable for each thread
  - The run-time system stores each thread's result in this private variable
  - OpenMP creates a critical section, and the values stored in the private variables are added in this critical section.



- A reduction operator can be $+, -, *, \&, |, \hat{}, \&\&\ and\ ||$.
  - Why is **division** not supported? Because it is **not commutative** or **associative**.
  - Although **subtraction** is not commutative or associative, it's handled in a different way in OpenMP.
- $float$ and $double$ reduction variables may cause the results to be slightly different.
  - Floating point arithmetic isn't associative.
  - For example, if a, b, and c are floats, then $(a + b) + c \neq a + (b + c)$.
- The private variables of each thread are initialized to a default value according to the datatype of the reduction variable

Table 5.1 Identity values for the various reduction operators in OpenMP.

Operator	Identity Value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

The *parallel for* directive

- Instead of using explicit parallelization, we can use the *parallel for* directive
- It's placed immediately before the *for* loop.
  - Serial program vs parallel program

```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```

```
   h = (b-a)/n;
   approx = (f(a) + f(b))/2.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
   for (i = 1; i <= n-1; i++)
      approx += f(a + i*h);
   approx = h*approx;
```

- The system divides the iterations of the *for* loop among the threads.
- The loop variable i has a default **private** scope; so each thread has its own copy of i.

```
#include <stdio.h>

double f(double x);
double Trap(double a, double b, int n, double h);

int main(void) {
    double integral;
    double a, b;
    int n;
    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;

#pragma omp parallel for num_threads(4) reduction(+:integral)
    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 */
```

- Note the following:
  1. OpenMP only parallelize for loop – no parallelization for $while$ and $do - while$ loops
  2. The number of iterations of the for loop must be determined
     a. Infinite loops cannot be parallelized

```
for ( ; ; ) {
    . . .
}
```

     b. for loops with $break$ cannot be parallelized

```
for (i = 0; i < n; i++) {
    if ( . . . ) break;
    . . .
}
```

- Notice the error in this code

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

int search(int key, int arr[], int n) {
    int i;
# pragma omp parallel for
    for (i = 0; i < 5; i++) {
        if (key == arr[i])
            return i;
    }
    return -1;

}

int main(int argc, char *argv[]) {
    int n;
    int arr[] = {1, 2, 3, 4, 5};
    int key = 4;

    scanf("%d", &n);
    int res = search(key, arr, n);
    printf("%d\n", res);
    return 0;
}
```

  - The error is at the $return\ i$; statement.
  - OpenMP cannot **exit** a loop that is parallelized.

- Try this Fibonacci program without and with parallelization:

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

int main(int argc, char *argv[]) {
    int fib[20];
    fib[0] = fib[1] = 1;

#pragma omp parallel for
    for (int i = 2; i < 20; ++i) {
        fib[i] = fib[i - 1] + fib[i - 2];
    }

    for (int i = 0; i < 20; ++i) {
        printf("%d ", fib[i]);
    }
    return 0;
}
```

  - When parallelization is used, the output is unpredictable.
  - This is because the Fibonacci program includes **dependencies** – each iteration is dependent on the previous iterations. Hence, **we cannot compute the value at a specific index without computing the value at lower indices**.
- In summary, OpenMP don't check for dependences among iterations in a loop.

Estimating π

- To compute a numerical approximation to π, we use the following formula

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}.$$

- The serial program

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    double factor = 1.0;
    double sum = 0.0;
    for (int i = 0; i < 1000; ++i) {
        sum += factor / (2 * i + 1);
        factor = -factor;
    }
    double pi = 4 * sum;
    printf("PI = %lf", pi);
    return 0;
}
```

- The parallel program

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    double factor = 1.0;
    double sum = 0.0;
#pragma omp parallel for
    for (int i = 0; i < 1000; ++i) {
        sum += factor / (2 * i + 1);
        factor = -factor;
    }
    double pi = 4 * sum;
    printf("PI = %lf", pi);
    return 0;
}
```

  - This program has a problem: there is a **data dependency** between $factor$ and sum, thus it gives **incorrect** results.

- We can fix the previous program by replacing

```
sum += factor/(2*k+1);
factor = -factor;
```

  by

```
if (k % 2 == 0)
    factor = 1.0;
else
    factor = -1.0;
sum += factor/(2*k+1);
```

  or

```
factor = (k % 2 == 0) ? 1.0 : -1.0;
sum += factor/(2*k+1);
```

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    double factor = 1.0;
    double sum = 0.0;
#pragma omp parallel for
    for (int i = 0; i < 1000; ++i) {
        factor = (i % 2 == 0) ? 1.0 : -1.0;
        sum += factor / (2 * i + 1);
    }
    double pi = 4 * sum;
    printf("PI = %lf", pi);
    return 0;
}
```

  - This program gives correct results, but not accurate!

- To fix the previous program:
  1. We must tell OpenMP that sum is a **reduction** variable, thus the final result is accumulated into it
  2. The $factor$ variable is shared because it's defined before the $parallel\ for$ directive. We must tell OpenMP that $factor$ is private.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    double factor = 1.0;
    double sum = 0.0;
#pragma omp parallel for reduction(+:sum) private(factor)
    for (int i = 0; i < 1000; ++i) {
        factor = (i % 2 == 0) ? 1.0 : -1.0;
        sum += factor / (2 * i + 1);
    }
    double pi = 4 * sum;
    printf("PI = %lf", pi);
    return 0;
}
```

- Instead of letting OpenMP to choose the scope of the variables, we can explicitly specify the scope using the ***default*** clause

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    double factor = 1.0;
    double sum = 0.0;
    int i;
    int n = 1000;
#pragma omp parallel for default(none) reduction(+:sum)\
        private(i, factor) shared(n)
    for (i = 0; i < n; ++i) {
        factor = (i % 2 == 0) ? 1.0 : -1.0;
        sum += factor / (2 * i + 1);
    }
    double pi = 4 * sum;
    printf("PI = %lf", pi);
    return 0;
}
```

Sorting

Bubble sort

- The serial bubble sort algorithm:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int n = 10;
    int a[] = {10, -5, 9, 1, 0, 2, 4, 3, 6, 11};

    for (int i = n-1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (a[j] > a[j + 1]) {
                int tmp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = tmp;
            }}}

    for (int i = 0; i < n; i++) {
        printf("a[%d] = %d\n", i, a[i]);
    }
    return 0;
}
```

  - The outer loop handles all the n-element array
  - The inner loop handles the first n-1 element array
  - The inner loop compares consecutive pairs of elements in the current list. When a pair is out of order ($a[i] > a[i+1]$) it swaps them.
- **The inner loop depends on the outer loop.**
- **The inner loop itself also has dependence on previous iterations**:
  - Say an iteration j will swap the values at index $a[j]$ and $a[j+1]$, this will impact iteration $j+1$.
  - So, iteration $j+1$ cannot decide whether to swap the elements or not, until iteration j finishes.
- We cannot remove the **loop-carried dependence** without completely rewriting the algorithm.

Odd-even transposition sort

- A sorting algorithm similar to bubble sort but can be parallelized.
- The serial algorithm is as follows

```c
for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0)
        for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
    else
        for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

Table 5.2 Serial odd-even transposition sort.

Phase	Subscript in Array							
	0		1		2		3	
0	9	⇔	7		8	⇔	6	
	7		9		6		8	
1	7		9	⇔	6		8	
	7		6		9		8	
2	7	⇔	6		9	⇔	8	
	6		7		8		9	
3	6		7	⇔	8		9	
	6		7		8		9	

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int phase, i, tmp;
    int n = 10;
    int a[] = {10, -5, 9, 1, 0, 2, 4, 3, 6, 11};

    for (phase = 0; phase < n; phase++) {
        if (phase % 2 == 0) {
            for (i = 1; i < n; i += 2) {
                if (a[i - 1] > a[i]) {
                    tmp = a[i - 1];
                    a[i - 1] = a[i];
                    a[i] = tmp;
                }}
        } else {
            for (i = 1; i < n - 1; i += 2) {
                if (a[i] > a[i + 1]) {
                    tmp = a[i + 1];
                    a[i + 1] = a[i];
                    a[i] = tmp;
           }}}}
    for (i = 0; i < n; i++) {
        printf("a[%d] = %d\n", i, a[i]);}
    return 0;}
```

- There is a **carried dependence** in the outer loop; two iterations cannot be executed simultaneously.
- The inner loops do not have dependence; each of the inner loops can be executed simultaneously.
    - This because, say the even-phase for loop, compares two adjacent elements. So, for two distinct values of i, say $i = j$ and $i = k$, the paris $\{j-1, j\}$ and $\{k-1, k\}$ will be disjoint.
    - Hence, the comparison and possible swaps of the pairs $(a[j-1], a[j])$ and $(a[k-1], a[k])$ can proceed simultaneously.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int phase, i, tmp;
    int n = 10;
    int a[] = {10, -5, 9, 1, 0, 2, 4, 3, 6, 11};

    for (phase = 0; phase < n; phase++) {
        if (phase % 2 == 0) {
#pragma omp parallel for num_threads(4) shared(a, n) private(i, tmp)
            for (i = 1; i < n; i += 2) {
                if (a[i - 1] > a[i]) {
                    tmp = a[i - 1];
                    a[i - 1] = a[i];
                    a[i] = tmp;
                }
            }
        } else {
#pragma omp parallel for num_threads(4) shared(a, n) private(i, tmp)
            for (i = 1; i < n - 1; i += 2) {
                if (a[i] > a[i + 1]) {
                    tmp = a[i + 1];
                    a[i + 1] = a[i];
                    a[i] = tmp;
                }
            }
        }
    }
    for (i = 0; i < n; i++) {
        printf("a[%d] = %d\n", i, a[i]);
    }
    return 0;
}
```

- Compare the performance of the bubble sort vs the odd-even sort on an array of 100,000 integers.
    - *Comment either the bubble sort or the odd-even sort block, when testing.*
- You will notice that the parallel sort algorithm is almost 2x faster than the serial one.

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

#define ARRAY_SIZE 100000

void generate_random_array(int *array, int size) {
    // Seed the random number generator
    srand(time(NULL));

    for (int i = 0; i < size; i++) {
        array[i] = rand();
    }}

void bubble_sort(int *array, int size) {
    for (int i = size - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (array[j] > array[j + 1]) {
                // Compare adjacent elements
                int tmp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = tmp;
            }}}}

void odd_even_sort(int *array, int size) {
    int phase, i, tmp;

    for (phase = 0; phase < size; phase++) {
        if (phase % 2 == 0) {
#pragma omp parallel for num_threads(4) shared(array, size) private(i, tmp)
            for (i = 1; i < size; i += 2) {
                if (array[i - 1] > array[i]) {
                    tmp = array[i - 1];
                    array[i - 1] = array[i];
                    array[i] = tmp;
                }
            }
        } else {
#pragma omp parallel for num_threads(4)shared(array, size) private(i, tmp)
            for (i = 1; i < size - 1; i += 2) {
                if (array[i] > array[i + 1]) {
                    tmp = array[i + 1];
                    array[i + 1] = array[i];
                    array[i] = tmp;
                }}}}}

int main(int argc, char *argv[]) {
    int array[ARRAY_SIZE];
    // Generate the random array
    generate_random_array(array, ARRAY_SIZE);

    time_t start, end;

    time(&start);
    bubble_sort(array, ARRAY_SIZE);
    time(&end);
    printf("%f\n", difftime(end, start));

    time(&start);
    odd_even_sort(array, ARRAY_SIZE);
    time(&end);
    printf("%f\n", difftime(end, start));

    return 0;
}
```

- **The previous parallel odd-even sort has one issue: omp forks the threads at the beginning of the first for loop and joins them at the end. The team of threads is forked at the beginning of the second for loop and joined again at the end.**
- This fork-join process costs time.
- We can solve this issue by forking the threads once at the beginning of the outer loop and only join them at the end of the second inner loop.

```
#define ARRAY_SIZE 200000

void enhanced_odd_even_sort(int *array, int size) {
    int phase, i, tmp;
#pragma omp parallel num_threads(4) shared(array, size)
private(i, tmp, phase)
    for (phase = 0; phase < size; phase++) {
        if (phase % 2 == 0) {
#pragma omp for
            for (i = 1; i < size; i += 2) {
                if (array[i - 1] > array[i]) {
                    tmp = array[i - 1];
                    array[i - 1] = array[i];
                    array[i] = tmp;
                }
            }
        } else {
#pragma omp for
            for (i = 1; i < size - 1; i += 2) {
                if (array[i] > array[i + 1]) {
                    tmp = array[i + 1];
                    array[i + 1] = array[i];
                    array[i] = tmp;
                }
            }
        }
    }
```

- Run this enhanced odd-even sort in comparison with the basic odd-sort function on an array of size 200,000.
- You will notice that this new function is slightly faster than the basic function.
- In the enhanced function:
  1. We defined the outer loop as a parallel structure, but without using $parallel\ for$ directive. So, this will fork the threads.
  2. At the beginning of each of the inner loops, we used the for directive. This will let the loops to execute in parallel using the forked threads, without forking another team of threads.

Scheduling loops

- The $parallel\ for$ directive automatically applies **block portioning**: if the serial loop has n iterations, then thread 0 handles the first $n/thread\_count$ iterations, thread 1 handles the second $n/thread\_count$ iterations, and so on.
- This could be **less optimal**.
- For example, say we want to parallelize the serial loop

```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

  Where the time required to compute f is **proportional** to the size of the argument i.
  - Then, block portioning will assign much more work to the last thread, $thread\_count - 1$, than it will assign to thread 0
- It's better to use **cyclic partitioning** for better **work-balance** between threads: the iterations are assigned, one at a time, in a "**round-robin**" fashion to the threads.
- Suppose $t = thread\_count$. Then a **cyclic partitioning** will assign the iterations as follows

Thread	Iterations
0	$0,\ n/t,\ 2n/t,\ \ldots$
1	$1,\ n/t+1,\ 2n/t+1,\ \ldots$
$\vdots$	$\vdots$
$t-1$	$t-1,\ n/t+t-1,\ 2n/t+t-1,\ \ldots$

- Example:

```
double f(int i) {
   int j, start = i*(i+1)/2, finish = start + i;
   double return_val = 0.0;

   for (j = start; j <= finish; j++) {
      return_val += sin(j);
   }
   return return_val;
}  /* f */
```

  - Run the next three programs and compare the performance.

```
gcc-14 -o prog -fopenmp <filename.c>; ./prog
```

- The serial program

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

double Sum(long n);
double f(long i);

int main(int argc, char* argv[]) {
   double  global_result;
   double  start, finish;
   long n = 100000;

   start = omp_get_wtime();
   global_result = Sum(n);
   finish = omp_get_wtime();

   printf("Result = %.14e\n", global_result);
   printf("Elapsed time = %f seconds\n", finish-start);

   return 0;
}

double f(long i) {
   long j;
   long start = i*(i+1)/2;
   long finish = start + i;
   double return_val = 0.0;

   for (j = start; j <= finish; j++) {
      return_val += sin(j);
   }
   return return_val;
}

double Sum(long n) {
   double approx = 0.0;
   long i;

   for (i = 0; i <= n; i++) {
     approx += f(i);
   }
   return approx;
}
```

- The parallel program

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

double Sum(long n, int thread_count);
double f(long i);

int main(int argc, char* argv[]) {
    double  global_result;
    double  start, finish;

    int thread_count = 2;
    long n = 100000;

    start = omp_get_wtime();
    global_result = Sum(n, thread_count);
    finish = omp_get_wtime();

    printf("Result = %.14e\n", global_result);
    printf("Elapsed time = %f seconds\n", finish-start);

    return 0;
}

double f(long i) {
    long j;
    long start = i*(i+1)/2;
    long finish = start + i;
    double return_val = 0.0;

    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
}

double Sum(long n, int thread_count) {
    double approx = 0.0;
    long i;

#  pragma omp parallel for num_threads(thread_count) \
reduction(+: approx)
    for (i = 0; i <= n; i++) {
        approx += f(i);
    }
    return approx;
}
```

- The **scheduled** parallel program

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

double Sum(long n, int thread_count);
double f(long i);

int main(int argc, char *argv[]) {
    double global_result;
    double start, finish;

    int thread_count = 2;
    long n = 100000;

    start = omp_get_wtime();
    global_result = Sum(n, thread_count);
    finish = omp_get_wtime();

    printf("Result = %.14e\n", global_result);
    printf("Elapsed time = %f seconds\n", finish - start);

    return 0;
}

double f(long i) {
    long j;
    long start = i * (i + 1) / 2;
    long finish = start + i;
    double return_val = 0.0;

    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
}

double Sum(long n, int thread_count) {
    double approx = 0.0;
    long i;

#  pragma omp parallel for num_threads(thread_count) \
reduction(+: approx) schedule(dynamic)
    for (i = 0; i <= n; i++) {
        approx += f(i);
    }
    return approx;
}
```

- **Scheduling** in OpenMP: assigning iterations to threads.
- The ***schedule* clause** can be used to assign iterations to threads:

$$schedule(< type > [, chunk\ size])$$

- **Chunk**: a block of iterations that would be executed consecutively in the serial loop.
  - *chunk size* determines the **number of iterations in a block**. It's **optional**.

- Types of scheduling:

static

- The iterations are assigned to the threads before the loop is executed

dynamic/guided

- The iterations are assigned to the threads while the loop is executing
- After a thread completes its current set of iterations, it requests more from the run-time system

auto

- The compiler and/or the run-time system determine the schedule
- *Does not specify chunk size*

runtime

- The schedule is determined at run-time based on an environment variable

The *static* schedule type

- The system assigns **chunks** of ***chunksize*** **iterations** to each thread in a round-robin fashion.
- Example: suppose we have 12 iterations, 0, 1,…,11, and three threads.
  - For $schedule(static, 1)$, the iterations will be assigned as

$$\begin{aligned} &\text{Thread } 0: \quad 0, 3, 6, 9 \\ &\text{Thread } 1: \quad 1, 4, 7, 10 \\ &\text{Thread } 2: \quad 2, 5, 8, 11 \end{aligned}$$

  - For $schedule(static, 2)$, the iterations will be assigned as

$$\begin{aligned} &\text{Thread } 0: \quad 0, 1, 6, 7 \\ &\text{Thread } 1: \quad 2, 3, 8, 9 \\ &\text{Thread } 2: \quad 4, 5, 10, 11 \end{aligned}$$

  - For $schdule(static, 4)$, the iterations will be assigned as

$$\begin{aligned} &\text{Thread } 0: \quad 0, 1, 2, 3 \\ &\text{Thread } 1: \quad 4, 5, 6, 7 \\ &\text{Thread } 2: \quad 8, 9, 10, 11 \end{aligned}$$

  - The **default** schedule is defined by a particular implementation of OpenMP, but in most cases it is equivalent to the clause:

$$schedule(\ \boldsymbol{static}\ , total\_iterations\ /\ thread\_count)$$

- The $static$ schedule is a good choice when each loop iteration takes roughly the **same amount of time** to compute.

The $dynamic$ and $guided$ schedule type

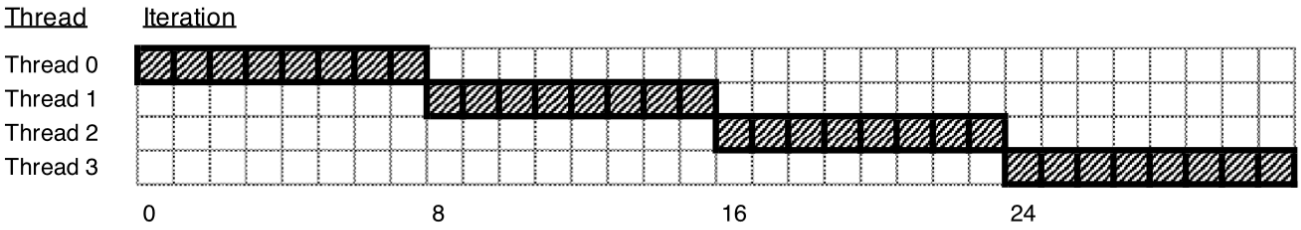
- $dynamic$ schedule: the iterations are also **broken up into chunks** of $chunksize$ consecutive iterations.
    - Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system.
- The **default $chunksize$** is 1.
- The **difference** between $\boldsymbol{static}$ and $\boldsymbol{dynamic}$ schedules:
    - $dynamic$ schedule assigns ranges to threads on a **first-come, first-served** basis.
        - Good choice if loop iterations **do not take a uniform** amount of time to compute.
        - There is some **overhead** associated with assigning them dynamically at run-time.
    - $static$ schedule assigns a range of threads **fixedly** at the start of the loop
        - Used when loop iterations **take a uniform** amount of time.
- With **larger** chunk sizes, **fewer** dynamic assignments will be made.
- In a $\boldsymbol{guided}$ **schedule**, as chunks are completed, the size of the new chunks **decreases**.
- If no $chunksize$ is specified, the size of the chunks decreases down to **1**. If $chunksize$ is specified, it decreases down to $\boldsymbol{chunksize}$.

The $runtime$ schedule type

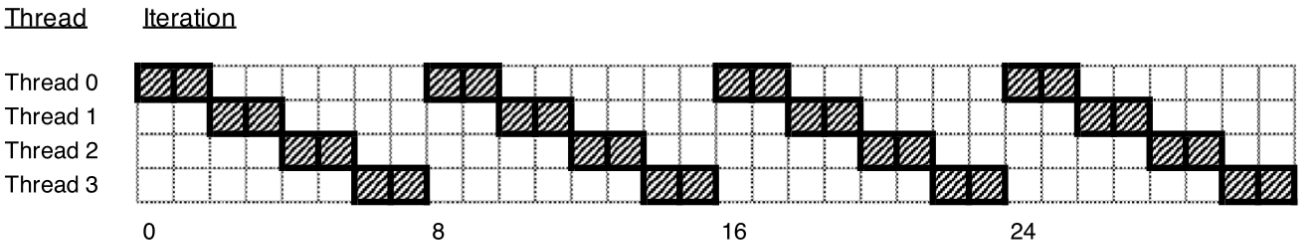
- When $schedule(runtime)$ is specified, the system uses the **environment variable** **`OMP_SCHEDULE`** to determine at run-time how to schedule the loop.
    - Environment variables are **named values** that can be accessed by a running program.
    - Examples: $PATH, HOME, SHELL$
- The **`OMP_SCHEDULE`** environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.
    - `$ export OMP_SCHEDULE="static ,1"`

- Scheduling **visualization** for the **static**, **dynamic**, and **guided** schedule types with **4 threads and 32 iterations**.

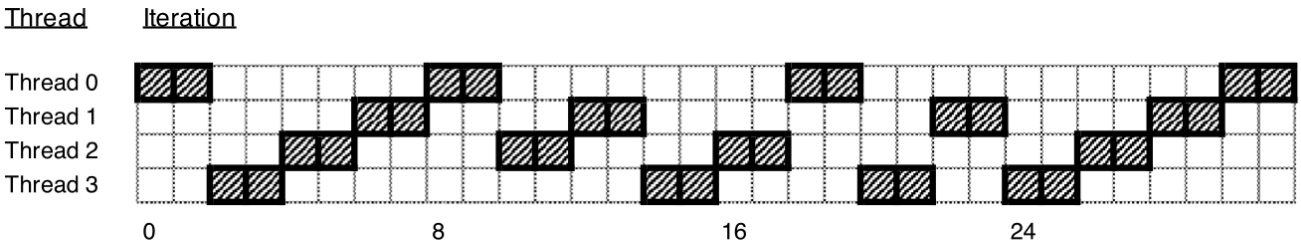
```
schedule(static)
```



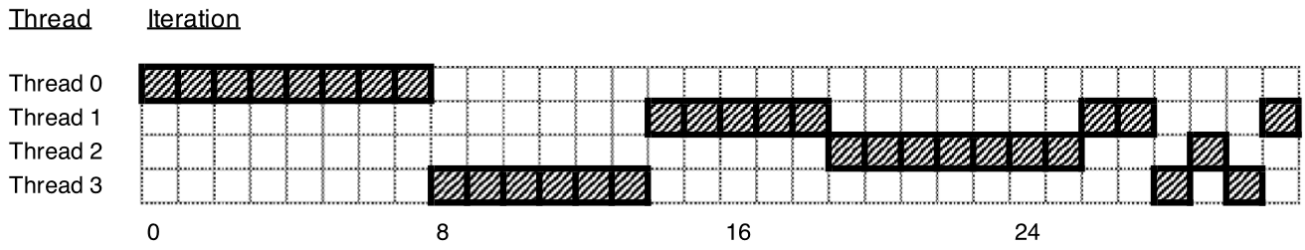

```
schedule(static, 2)
```




```
schedule(dynamic, 2)
```




```
schedule(guided)
```

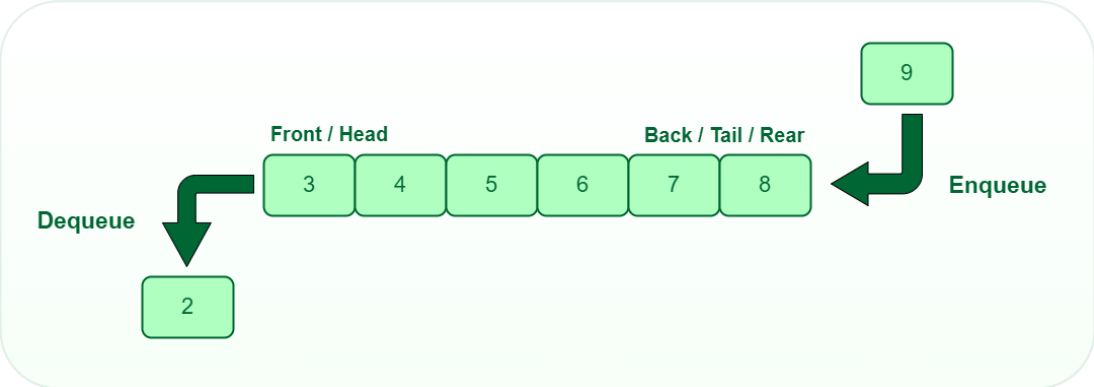


Which schedule type to choose?

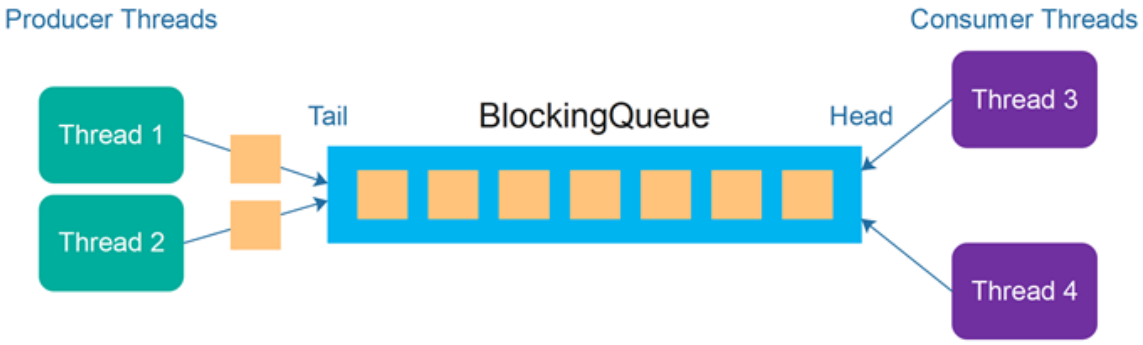
- It depends on the problem and the nature of the loop.
- There is some **overhead** associated with the use of a **schedule clause**.
  - $static$ overhead **less than** $dynamic$ overhead **less than** $guided$ overhead
- Practically, you should **try different options** until you hit a satisfactory performance.
  - Make use of the $schedule(runtime)$ to test different settings.

Producers and consumers

- A **queue** is a data structure in which elements are inserted at the rear of the queue and removed from the front of the queue.
  - Adding elements → enqueue
  - Removing elements → dequeue



- Queues appear in many **applications**:
  - Print spooling
  - Operating systems scheduling/task scheduling
  - Data buffers in networking
  - Load balancing in web servers
- Queues are fundamental in **producer-consumer** applications:
  - Part the of the application is **producing** data, and another part is **consuming** the data



- Let's implement a **message-passing** application that is based on the **producer-consumer** pattern.
  - Each thread has a **shared-message queue**.
  - Each thread generates random (integer values) messages and random destination for the messages.
  - A **producer** thread **enqueues** the message in the queue.
  - A **consumer** thread checks if the queue received a message, and then **dequeues** it.
  - Each thread **alternates** between sending and receiving messages.
  - The user will specify the **number of messages** each thread should send.
  - When a thread is **done sending messages**, it receives messages until all the threads are done, at which point all the threads **quit**.
- Pseudocode for each thread might look something like this:

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
   Send_msg();
   Try_receive();
}

while (!Done())
   Try_receive();
```

- Core segments/functions of the program:
  - Sending messages → $Send\_msg()$
  - Receiving messages → $Try\_receive()$
  - Terminating the program → $Done()$

Sending messages

- Accessing a **message queue** to **enqueue** a message is a **critical section**.
  - We need to regulate how threads access the queues to avoid dropping messages.
- To successfully **enqueue** a message, we need a pointer to the **rear** of the queue.
  - When we enqueue a message, we'll update the rear pointer.

```
   mesg = random();
   dest = random() % thread_count;
#  pragma omp critical
   Enqueue(queue, dest, my_rank, mesg);
```

  - The $Enqueue$ functions handles updating the rear pointer.
  - Note that this allows a thread to send a message to itself.

Receiving messages

- **Only the owner of the queue (that is, the destination thread) will dequeue from a given message queue.**
- If there are at least two messages in the queue, a call to $Dequeue$ **can't conflict** with any calls to $Enqueue$.
- Before $Dequeue$ing the queue, **we must check its size**; if 0, return none. If the size is >=1, then return the element at the front.
    - We store two variables: $enqueued$, which is the number of added elements, and $dequeued$, which is the number of removed elements.

```
    queue_size = enqueued - dequeued;
    if (queue_size == 0) return;
    else if (queue_size == 1)
#       pragma omp critical
        Dequeue(queue, &src, &mesg);
    else
        Dequeue(queue, &src, &mesg);
    Print_message(src, mesg);
```

    - The $critical\ section$ is added to prevent other threads from updating the queue when the owner thread is $Dequeue$ing it when it has only one element.

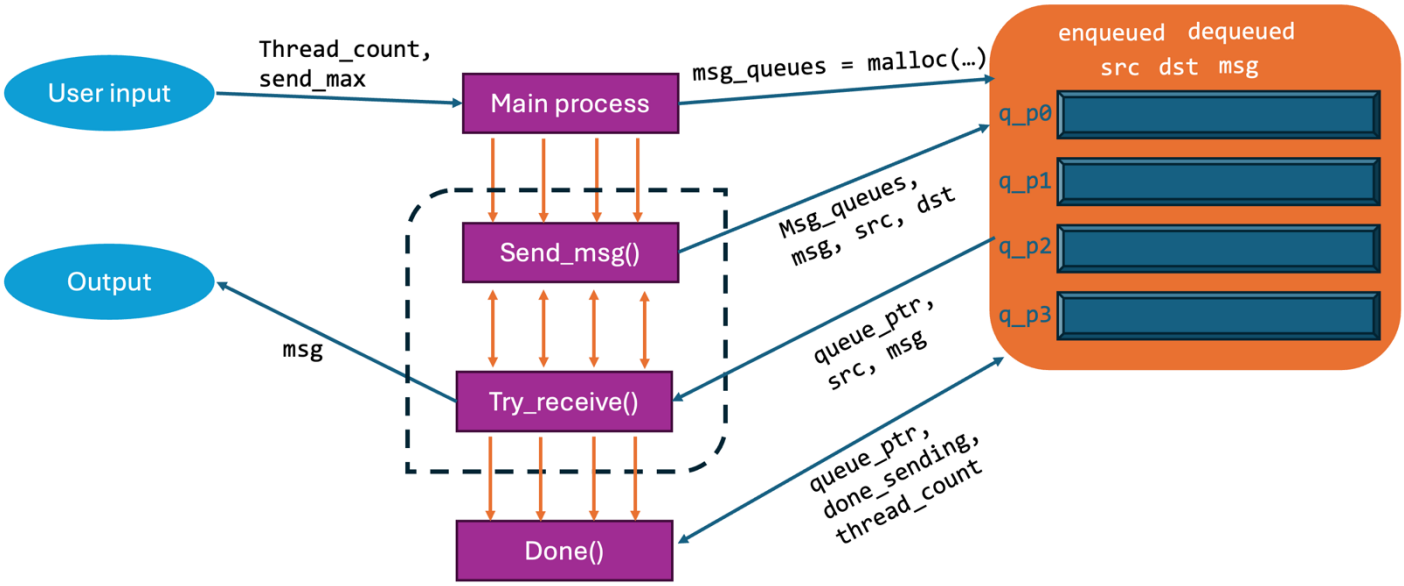
Terminating the program

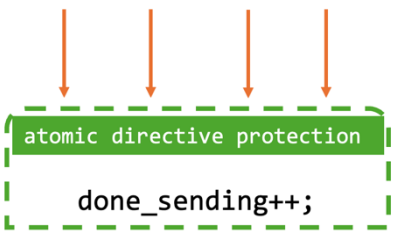
- The **naïve way** to terminate the program (implementing $Done()$) is as follows:

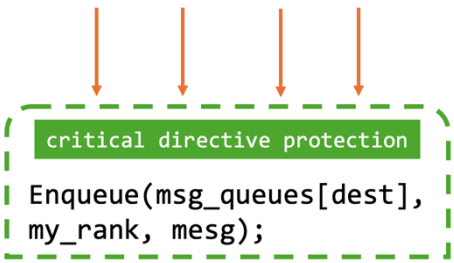
```
queue_size = enqueued - dequeued;
if (queue_size == 0)
   return TRUE;
else
   return FALSE;
```

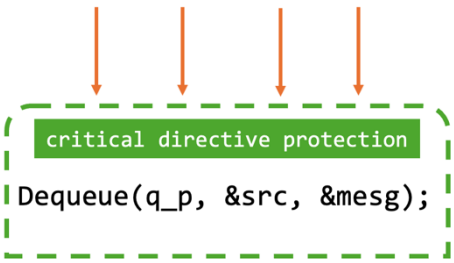
- The problem with this way is that it is possible that the owner thread compute $queue\_size = 0$, while another thread is adding an element into the queue.
    - Thus, the added message will never be received by the owner thread.
- Instead, we can include $done\_sending$ as a counter variable, so it serves as a flag when all threads have no more messages to send.

```
queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
   return TRUE;
else
   return FALSE;
```


User input
Thread_count,
send_max
Main process
msg_queues = malloc(…)
enqueued dequeued
src dst msg
q_p0
q_p1
q_p2
q_p3
Send_msg()
Msg_queues,
msg, src, dst
Output
msg
Try_receive()
queue_ptr,
src, msg
queue_ptr,
done_sending,
thread_count
Done()


atomic directive protection
done_sending++;


critical directive protection
Enqueue(msg_queues[dest],
my_rank, mesg);


critical directive protection
Dequeue(q_p, &src, &mesg);

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include "queue.h"
// gcc-14 -o prog -fopenmp -DDEBUG omp_msgps.c queue.c
const int MAX_MSG = 10000;

void Usage(char* prog_name);
void Send_msg(struct queue_s* msg_queues[], int my_rank, int
thread_count, int msg_number);
void Try_receive(struct queue_s* q_p, int my_rank);
int Done(struct queue_s* q_p, int done_sending, int thread_count);

int main(int argc, char* argv[]) {
   int thread_count;
   int send_max;
   struct queue_s** msg_queues;
   int done_sending = 0;

   if (argc != 3) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   send_max = strtol(argv[2], NULL, 10);
   if (thread_count <= 0 || send_max < 0) Usage(argv[0]);

   msg_queues = malloc(thread_count*sizeof(struct queue_node_s*));

#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(thread_count, send_max, msg_queues,
done_sending)
   {
      int my_rank = omp_get_thread_num();
      int msg_number;
      srandom(my_rank);
      msg_queues[my_rank] = Allocate_queue();

#     pragma omp barrier

      for (msg_number = 0; msg_number < send_max; msg_number++) {
         Send_msg(msg_queues, my_rank, thread_count, msg_number);
         Try_receive(msg_queues[my_rank], my_rank);
      }
#     pragma omp atomic
      done_sending++;
#     ifdef DEBUG
      printf("Thread %d > done sending\n", my_rank);
#     endif

      while (!Done(msg_queues[my_rank], done_sending, thread_count))
         Try_receive(msg_queues[my_rank], my_rank);

      Free_queue(msg_queues[my_rank]);
      free(msg_queues[my_rank]);
   }  /* omp parallel */

   free(msg_queues);
   return 0;
}  /* main */
```

```
void Usage(char *prog_name) {
   fprintf(stderr, "usage: %s <number of threads> <number of
messages>\n",prog_name);
   fprintf(stderr, "   number of messages = number sent by each
thread\n");
   exit(0);
}  /* Usage */

void Send_msg(struct queue_s* msg_queues[], int my_rank, int
thread_count, int msg_number) {
   int mesg = -msg_number;
   int dest = random() % thread_count;
#  pragma omp critical
   Enqueue(msg_queues[dest], my_rank, mesg);
#  ifdef DEBUG
   printf("Thread %d > sent %d to %d\n", my_rank, mesg, dest);
#  endif
}  /* Send_msg */

void Try_receive(struct queue_s* q_p, int my_rank) {
   int src, mesg;
   int queue_size = q_p->enqueued - q_p->dequeued;

   if (queue_size == 0) return;
   else if (queue_size == 1)
#     pragma omp critical
      Dequeue(q_p, &src, &mesg);
   else
      Dequeue(q_p, &src, &mesg);
   printf("Thread %d > received %d from %d\n", my_rank, mesg, src);
}   /* Try_receive */

int Done(struct queue_s* q_p, int done_sending, int thread_count) {
   int queue_size = q_p->enqueued - q_p->dequeued;
   if (queue_size == 0 && done_sending == thread_count)
      return 1;
   else
      return 0;
}   /* Done */
```

- When the program begins execution, the master thread will get command-line arguments and allocate an array of message queues, one for each thread.
- This array needs to be shared among the threads, since any thread can send to any other thread, and hence any thread can enqueue a message in any of the queues.
- We use *omp barrier* to add an explicit barrier so that no thread starts enqueuing or dequeuing until all threads have constructed their queues to avoid errors.
- After completing its sends, each thread increments *done_sending* before proceeding to its final loop of receives.
    - Incrementing *done_sending* is a **critical section**, and we could protect it with a ***critical*** **directive**.
    - Instead, we use **a higher performance directive**, *omp atomic*

- $omp\ atomic$ can only **protect** statements with **a single assignment**.
- For example, ***omp atomic*** protect statements of the following form:

```
x <op>= <expression>;
x++;
++x;
x--;
--x;
```

  - $< expression >$ **must not reference** x
  - $< op >$ can be + , $*$,–, / , &, ^ , | , <<, or > >
- $atomic$ directive has special for $\boldsymbol{load - modify - store}$ instructions in modern processors, which is **more efficient** than $critical\ section$

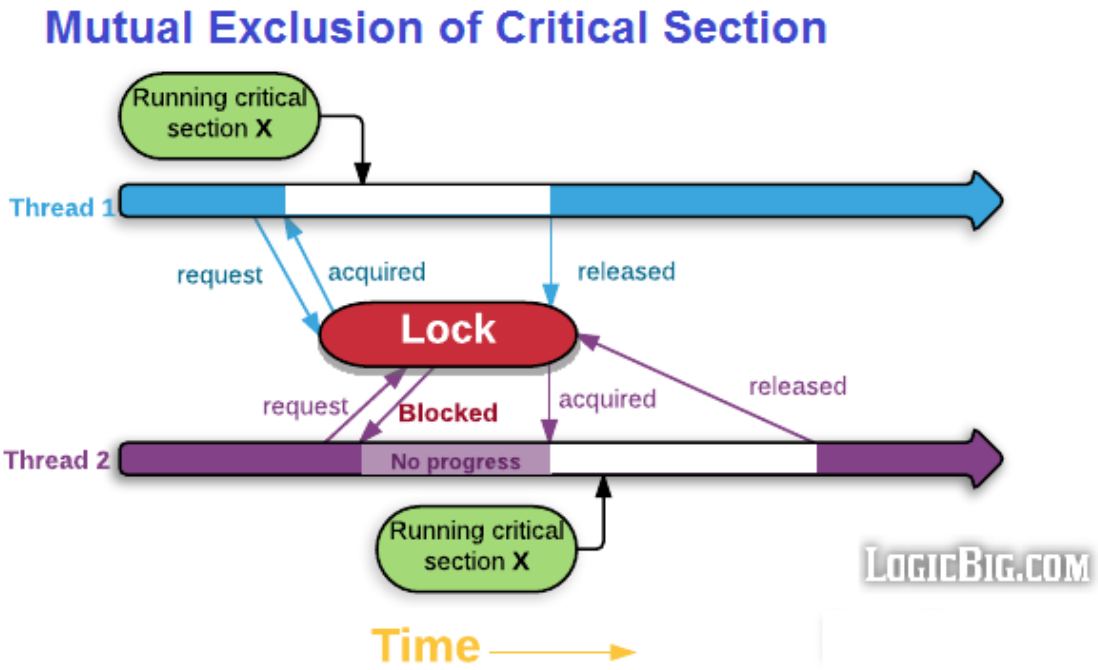
Locks

- As an alternative to $critical$ sections, we can use $\boldsymbol{Locks}$.
- A $lock$ consists of a **data structure** and **functions** enforce mutual exclusion to data or resources.
- Defining locks follows this pseudocode:

```
/* Executed by one thread */
Initialize the lock data structure;
. . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure;
Critical section;
Unlock or unset the lock data structure;
. . .
/* Executed by one thread */
Destroy the lock data structure;
```

  - The lock data structure is **shared among the threads** that will execute the critical section.
  - **One** of the threads will **initialize** the lock.
  - **One** of the threads will **destroy** the lock.
  - A thread entering the critical section will ***set*** the lock.
  - When the thread finishes the critical section, it will ***releases*** or ***unset*** the lock.

- How locks achieve mutual exclusion:



- Locks in OpenMP:

Action	Function
Initialize a lock	`void omp_init_lock (omp_lock_t* lock_p);`
Acquire a lock	`void omp_set_lock (omp_lock_t* lock_p);`
Release a lock	`void omp_unset_lock (omp_lock_t* lock_p);`
Remove a lock	`void omp_unset_lock (omp_lock_t* lock_p);`

Using locks in the message-passing program

- In the previous program, we used ***critical*** **directive** to enforce a mutually exclusive access to a **shared resource**.
  - `done_sending++;`
  - `Enqueue(q_p, my_rank, mesg);`
  - `Dequeue(q_p, &src, &mesg);`
- There is one efficiency issue with this critical section: the *critical* directive **blindly** allows only one thread to **Enqueue** or **Dequeue**.
- But we don't need to block threads that **don't conflict with each other**: for instance, it's safe for **thread 0** to enqueue a message in **thread 1's queue** at the same time that **thread 1** is enqueuing a message in **thread 2's queue**.
  - **So, the *critical* directive allows only one thread do *enqueue*ing at a time regardless the process causes conflicts or not.**
- To **overcome** this, we use *locks* and instead of making the **whole function** of *enqueue* or *dequeue* a **critical section**, we will make the **corresponding queues only critical sections.**

- So, we can do the following:

```
# pragma omp critical
/* q_p = msg_queues[dest] */
Enqueue(q_p, my_rank, mesg);
```

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```

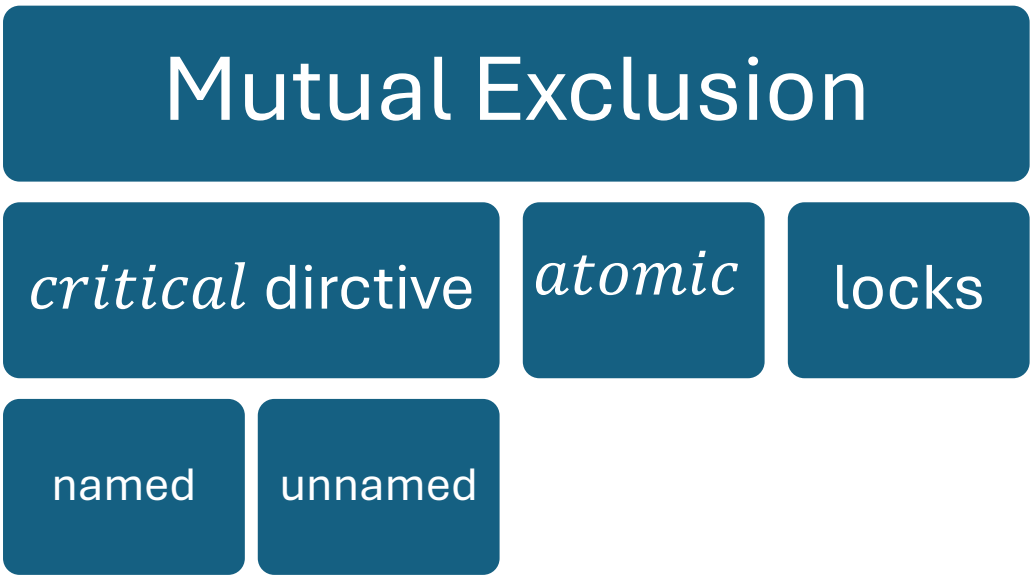
```
# pragma omp critical
/* q_p = msg_queues[my_rank] */
Dequeue(q_p, &src, &mesg);
```

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &mesg);
omp_unset_lock(&q_p->lock);
```

- **Check the source code** in *queue_lk.h*, *queue_lk.c*, and *omp_msglk.c*.
  - **The *lock* is defined as a data member in the *queue* struct.**
- Now when a thread tries to send or receive a message, it can only be **blocked** by a thread accessing the **same message queue**, since **different message queues have different locks**.
- In previous implementation, **only one thread** could **send** at a time, regardless of the destination.

Critical directives, *atomic* directives, or locks?

- So far, we introduced **three** basic mutual exclusion mechanisms:



Mechanism	Properties	Syntax	Group regions under one directive?

atomic	Simple and the fastest	`#pragma omp atomic`
		`x <op>= <expression>;`
		`x++;`
		`--x;`
			Yes

critical **(unnamed)**	Easier than locks	`#pragma omp critical`
		`{`
		`…`
		`}`
			Yes

critical **(named)**	-	`#pragma omp critical(name)`
		`{`
		`…`
		`}`
			No

Locks	Better for data structures	`omp_set_lock(lock);`
		`…`
		`omp_unset_lock(lock);`
			No

- Critical regions specified by $atomic$ or $critcal$ (unnamed) directives **<u>may</u>** treat all their regions as one block.
- For example:

```
#pragma omp parallel
{
    #pragma omp atomic
    var1 = var1 + 1;  // Atomic operation on var1

    #pragma omp atomic
    var2 = var2 + 1;  // Atomic operation on var2
}
```

  - **In one implementation**, the runtime might enforce exclusive access, meaning that the operations on $var1$ and $var2$ cannot happen at the same time, even though they operate on different variables.
  - **In another implementation**, the runtime might allow these operations to run concurrently if they access different variables, as there's no dependency or conflict between the two operations.

- The same applies if using (**unnamed**) $critical$ directive:

```
#pragma omp parallel
{
    #pragma omp critical
    {
        var1 = var1 + 1;
    }

    #pragma omp critical
    {
        var2 = var2 + 1;
    }
}
```

- **Named** *critical* directive and **locks** avoid that.

```
#pragma omp parallel
{
    #pragma omp critical(section1)
    {
        var1 = var1 + 1;  // Atomic operation on var1
    }

    #pragma omp critical(section2)
    {
        var2 = var2 + 1;  // Atomic operation on var2
    }
}
```

Some caveats

1. **Don't mix** the different types of mutual exclusion for a **single** critical section.
   - The code below uses two mutual exclusive mechanisms for the variable x.

```
#  pragma omp atomic
   x += f(y);

#  pragma omp critical
   x = g(x);
```

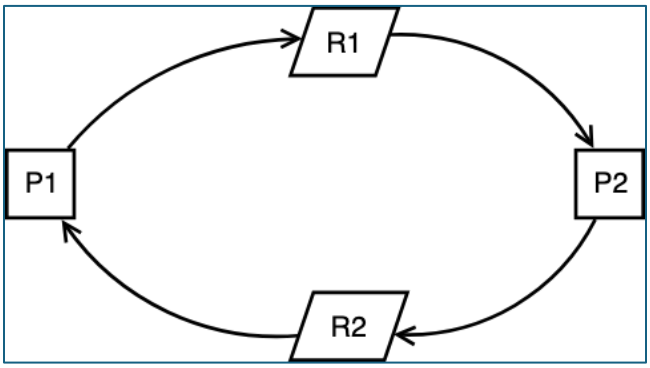
   - It's possible that the two sections get executed **concurrently**, leading to **incorrect** results.
   - You can either: **use** *critical* directive **for both sections** or **rewrite** $g()$ to have the **form** required by the *atomic* directive.
2. There is no **guarantee of fairness** in mutual exclusion constructs.
   - For example, the code below may allow one thread to **always** be executing the function g, thus preventing other threads from accessing it.

```
   while(1) {
      . . .
#     pragma omp critical
      x = g(my_rank);
      . . .
   }
```

   - This won't happen if the *while* loop **terminates**.
3. It can be dangerous to "**nest**" mutual exclusion constructs.

```
#  pragma omp critical
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical
      z = g(x);  /* z is shared */
      . . .
   }
```

- This will cause **deadlock**: a situation in which a group of threads are waiting for each other thread to finish.



- If a thread is executing the first block, it won't be able to **enter the second block**. At the same time, it **will not leave the first block** until it proceeds to the second block.
- One possible solution is use **named** *critical* sections

```
#  pragma omp critical(one)
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical(two)
      z = g(x);  /* z is global */
      . . .
   }
```

- But this is not the **ultimate** solution, as deadlocks can occur as following:

Time	Thread *u*	Thread *v*
0	Enter crit. sect. `one`	Enter crit. sect. `two`
1	Attempt to enter `two`	Attempt to enter `one`
2	Block	Block

Caches, cache coherence, and false sharing