MODULE-4

Shared-memory programming with OpenMP – OpenMP pragmas and directives, the trapezoidal rule, scope of variables, the reduction clause, loop carried dependency, scheduling, producers and consumers, caches, cache coherence and false sharing in OpenMP, tasking, and thread safety.

Introduction to OpenMP

OpenMP is an **API (Application Programming Interface)** designed specifically for **shared-memory MIMD (Multiple Instruction, Multiple Data)** programming. The term **"MP" stands for multiprocessing**, aligning with the shared-memory computing paradigm. In systems targeted by OpenMP, **multiple autonomous CPU cores** can access a **common main memory**, as typically illustrated in a shared-memory architecture diagram (like Fig. 5.1).

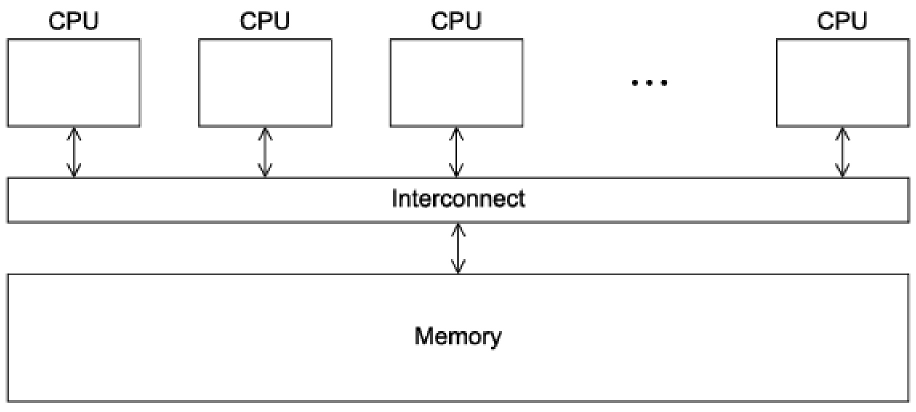


FIGURE 5.1

A shared-memory system.

Comparison with Pthreads

Although both **OpenMP and Pthreads** serve the purpose of shared-memory programming, they differ in their approach and complexity:

- **Pthreads is low-level** and requires **explicit thread management**. The programmer defines the behavior of each thread manually.
- **OpenMP offers a higher-level abstraction**. The programmer can simply indicate that a block of code should run in parallel. The **OpenMP compiler and runtime system automatically manage thread behavior**.

This difference makes **OpenMP easier to use** for certain types of problems, especially those involving structured parallelism.

- **Pthreads is a library-based API**, and any standard C compiler can compile Pthreads code, provided the system supports the library.

- **OpenMP requires compiler support**. Not all compilers support OpenMP, which means you may encounter compilers that cannot handle OpenMP directives.

The dual existence of OpenMP and Pthreads highlights a **trade-off**:

- **Pthreads offers full control** and is suitable for complex, custom thread behaviors.
- **OpenMP is simpler to use** but may **limit low-level control** over thread interactions.

This makes OpenMP suitable for rapid development and easy parallelization of certain serial codes, while Pthreads is preferred for **fine-grained, customized concurrency control**.

OpenMP was developed by a consortium of programmers and researchers who aimed to **simplify high-performance parallel programming**. Writing large-scale programs using lower-level APIs like Pthreads or MPI can be challenging and error-prone. OpenMP was designed to **allow incremental parallelization** of existing serial programs, a feature not possible with MPI and only partly feasible with Pthreads.

4.1 Getting started

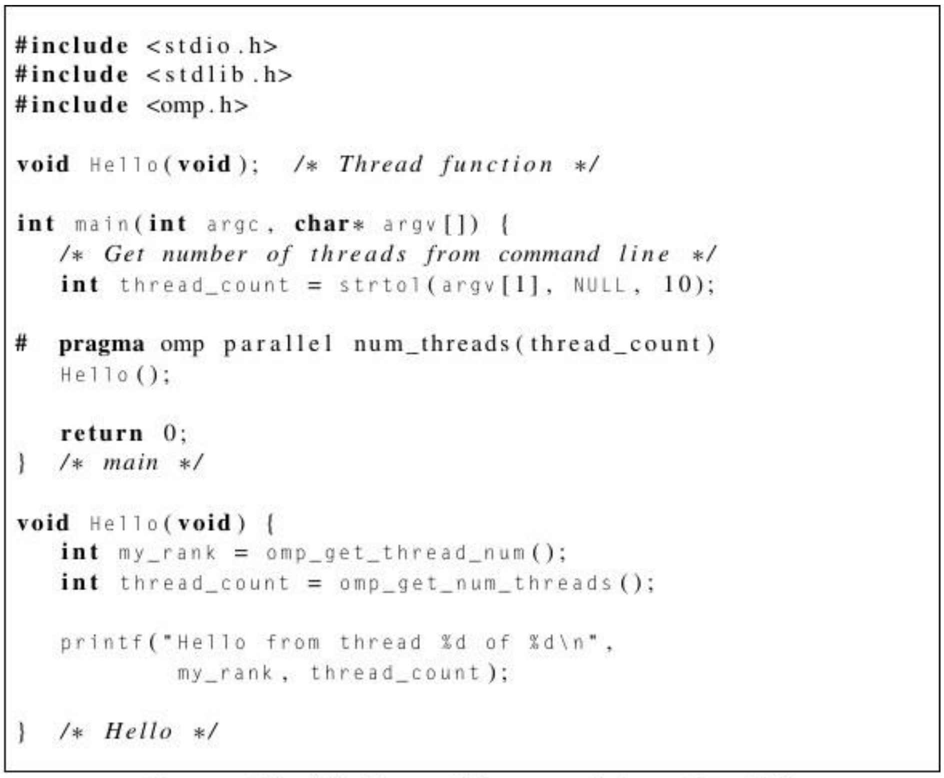
OpenMP uses a **directives-based shared-memory API**, primarily through **compiler directives called pragmas** in C and C++. **Pragmas** are special preprocessor instructions that **enable behavior beyond the standard C language**, and **compilers that don't support them simply ignore them**. This makes OpenMP programs **backward compatible—**they can still be compiled and run sequentially on systems without OpenMP support, although without parallelism.

In C/C++, **OpenMP pragmas begin with**

#pragma,

where the # must appear at the **start of the line**, and the rest of the directive should be **indented consistently** with the code. By default, **pragmas are single-line instructions**, so if a directive needs to span multiple lines, **the newline must be escaped using a backslash** \.

This design makes OpenMP flexible and safe for integration into existing C programs. For instance, a simple OpenMP "hello, world" program (see **Program 5.1**) can include a #pragma omp parallel directive, and **if OpenMP is not supported**, the compiler will ignore the directive and compile the code as a regular sequential program.



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

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

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

   return 0;
}  /* main */

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);

}  /* Hello */
```

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

4.1.1 Compiling and running OpenMP programs

To compile an OpenMP program using **GCC**, we must include the **-fopenmp option**. For example, a typical compilation command would be:

```
$ gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
```

This tells the compiler to enable OpenMP support during compilation. When running the program, we can specify the **number of threads** we want to use by passing it as a command-line argument. For instance, to run with **four threads**, we use:

```
$ ./omp_hello 4
```

The output might look like:

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

However, since all threads are writing to **stdout concurrently**, the **output order is not guaranteed**. The messages may appear in any **permutation of thread ranks**, such as:

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

To test the program with **just one thread**, we can run:

```bash
$ ./omp_hello 1
```

And the expected output will be:

```
Hello from thread 0 of 1
```

4.1.2 The program

In OpenMP programs, in addition to using **directives**, we also include a library of functions and macros by adding the header file:

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

This is essential for calling OpenMP-specific functions like omp_get_thread_num() and omp_get_num_threads().

Like Pthreads, OpenMP programs often **take the number of threads as a command-line argument**. This is typically done using the strtol() function, which converts the argument from string to integer format. The syntax of this function is:

```
long strtol(const char* str, char** endptr, int base);
```

In most cases, we pass NULL as the second parameter and use 10 as the base for decimal numbers.

When a C program begins execution, it runs in a **single-threaded process**. However, when the OpenMP **parallel directive** is encountered, the compiler spawns **multiple threads**, each of which can execute a **structured block** (a single entry and exit point). The simplest directive is:

```
#pragma omp parallel
```

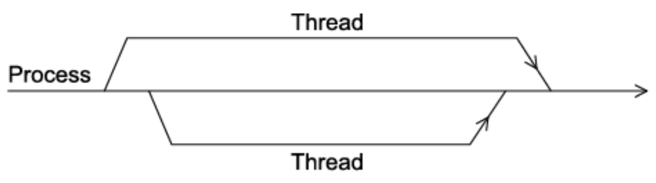


FIGURE 5.2

A process forking and joining two threads.

This causes the runtime system to start a **team of threads**, including the **master thread** (thread 0) and other threads it spawns (child threads). The number of threads used is determined by the system, unless the programmer overrides it using a clause:

```
#pragma omp parallel num_threads(thread_count)
```

The actual number of threads started may vary depending on system limits, but most systems support **hundreds or thousands of threads**.

Each thread in the team runs the specified block of code. For instance, if the block calls a function like Hello(), each thread executes it. When all threads complete this block, an **implicit barrier** ensures that they wait for each other before the parent thread proceeds. The **parent thread** is the one that encountered the parallel directive and started the team. In most cases, it is also the master thread.

Each thread has its own **private stack**, meaning it will create its **own local variables** when executing the function. To retrieve its **thread ID** and the **total number of threads**, a thread can call:

```
int omp_get_thread_num(void);
int omp_get_num_threads(void);
```

These functions return the thread's rank (from 0 to thread_count−1) and the total number of active threads, respectively.

All threads share access to **stdout**, so each thread can print its message using printf. However, because output is not synchronized, the **order of printed lines is nondeterministic**. This behavior demonstrates **OpenMP's implicit parallelism**, where we achieve substantial multi-threaded execution with minimal code, contrasting with the more verbose and explicit thread management required in Pthreads

4.1.3 Error checking

In the provided OpenMP example program, **no error checking is performed** to keep the code simple and compact. However, in real-world scenarios, this is **risky** and **not recommended**. Good programming practice dictates that developers must anticipate and handle **possible errors**, especially those involving **user input and system support**.

A primary check involves ensuring that a **valid command-line argument** is supplied. After retrieving the number of threads using strtol(), we should verify that the value is **positive**, as negative or zero threads would be invalid. Furthermore, although not critical in simple programs, one could also confirm whether the **actual number of threads** created by OpenMP matches the user-supplied count.

Another source of potential errors is the **compiler's support for OpenMP**. If OpenMP is **not supported**, the compiler may silently **ignore the #pragma omp parallel directive**, but still generate **errors** for #include <omp.h> or calls to OpenMP functions like omp_get_thread_num() and omp_get_num_threads().

To handle this, we can **conditionally compile** OpenMP-specific code by checking whether the **macro _OPENMP is defined.** This macro is automatically defined by compilers that support OpenMP. For example, instead of directly including the header:

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

We use:

```c
#ifdef _OPENMP
# include <omp.h>
#endif
```

Similarly, function calls that depend on OpenMP can be wrapped as:

```c
#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
```

This fallback ensures that **even if OpenMP is not supported**, the program can still **execute sequentially** with a **single thread**, assuming rank 0 and a thread count of 1.

The authors note that a more robust version of the sample program (with these checks) is available on the book's website. However, in textbook examples, they typically **omit error checks** for clarity, and **assume** that OpenMP support is available.

4.2 The Trapezoidal Rule

The **trapezoidal rule** is a **numerical method** for approximating the **area under a curve** y=f(x), between the interval [a,b]. It works by **dividing the interval into n subintervals** and estimating the area over each subinterval using a **trapezoid.**

If $h = \frac{b-a}{n}$ and $x_i = a + i \cdot h$, then the approximation of the area is given by:

$$\text{Area} \approx h\left[\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{f(x_n)}{2}\right]$$

This rule provides a simple way to **estimate definite integrals**, especially when the **analytical integration is complex or not possible**.

```
/* 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;
```

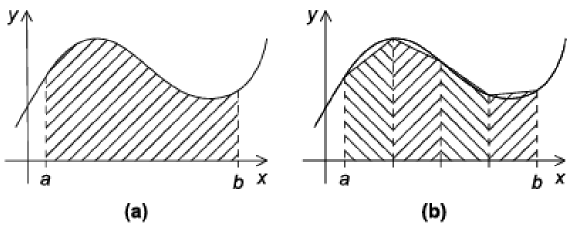


FIGURE 5.3

The trapezoidal rule.

4.2.1 A First OpenMP Version

1. **Identified two types of jobs**:
   a. Computation of the areas of individual trapezoids
   b. Adding the areas of trapezoids
2. **No communication** among jobs in the first collection (1a), but each job in (1a) communicates with job (1b).
3. **Assumption**: Number of trapezoids (n) is much larger than the number of cores. So, we **aggregated jobs** by assigning a **contiguous block of trapezoids** to each thread. This partitions the interval [a,b][a, b][a,b] into **subintervals**, and each thread applies the **serial trapezoidal rule** to its subinterval (refer **Figure 5.4**).

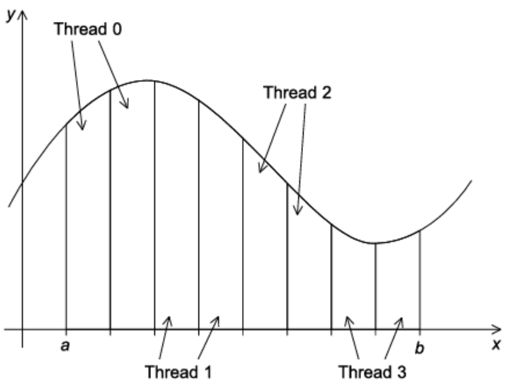


FIGURE 5.4

Assignment of trapezoids to threads.

A simple idea is to use a shared variable global_result and let each thread update it like this:

```
global_result += my_result;
```

But this leads to a **race condition** — if multiple threads execute this statement simultaneously, the result will be **unpredictable**.

Example scenario (with my_result = 1 for thread 0 and my_result = 2 for thread 1):

- Both threads load global_result = 0
- Each adds their my_result
- Final global_result ends up being 2, instead of expected 3 (See the **timetable table** in the text for detailed illustration)

This is a **race condition**:

- Multiple threads access a **shared resource**
- At least one of the accesses is an **update**
- Accesses **interleave** in a way that leads to incorrect results

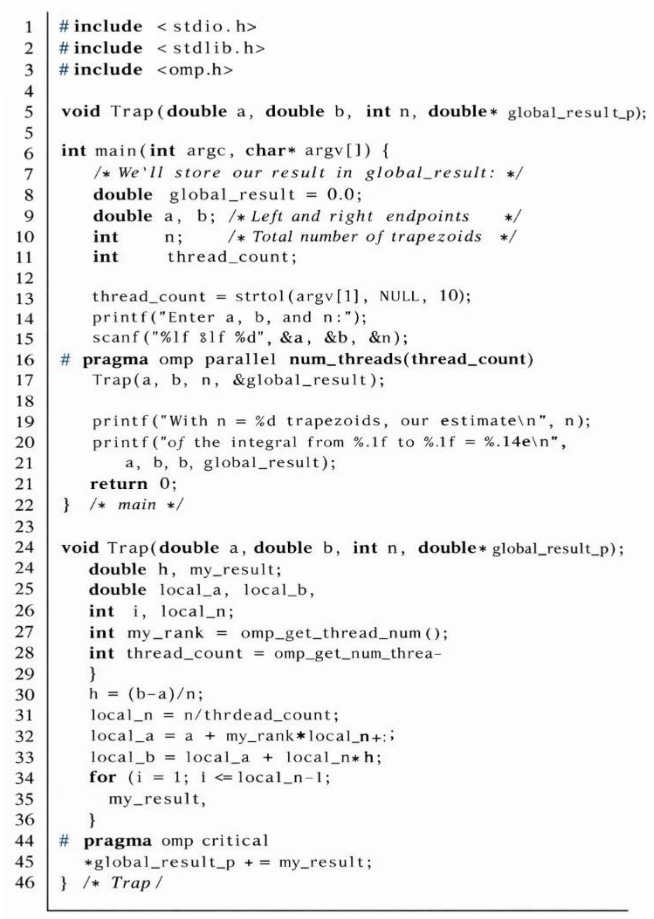
The statement global_result += my_result is a **critical section —** code that updates a shared resource and must be executed by **only one thread at a time**.

In OpenMP, the **critical directive** ensures that only one thread at a time executes a particular block of code, providing **mutual exclusion** among threads. This prevents data races when multiple threads try to update shared variables simultaneously.

In **Program 5.2**, the **main function** first executes as a single thread to read the inputs — the number of threads, and the values of *a*, *b*, and *n*. The **parallel directive** at Line 17 specifies that the function **Trap** should be executed by *thread_count* threads in parallel. After the parallel region completes, all additional threads are terminated, and only the main thread continues to print the final result.

Within the **Trap function**, each thread obtains:

1. Its **rank** and the **total number of threads**,
2. The **base length** of the trapezoids,
3. The **number of trapezoids** assigned to it,
4. The **left and right endpoints** of its interval, and
5. Its **local contribution** to the shared variable **global_result**.



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

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
    /* We'll store our result in global_result: */
    double  global_result = 0.0;
    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:");
    scanf("%lf 8lf %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 %.1f to %.1f = %.14e\n",
        a, b, b, global_result);
   return 0;
}  /* main */

void Trap(double a, double b, int n, double* global_result_p);
   double h, my_result;
   double local_a, local_b,
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threa-
   }
   h = (b-a)/n;
   local_n = n/thrdead_count;
   local_a = a + my_rank*local_n+:;
   local_b = local_a + local_n*h;
   for (i = 1; i <=local_n-1;
     my_result,
   }
#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap /
```

Program 5.2: First OpenMP trapezoidal rule program.

Each thread computes its own **partial area** using local variables such as *local_a* and *local_b*, which differ for each thread. These local results are then added to **global_result** inside a **critical section**, ensuring that only one thread updates it at a time.

If *n* (the total number of trapezoids) is not **evenly divisible** by *thread_count*, the program will use fewer trapezoids than requested. For example, if *n = 14* and *thread_count = 4*, each thread gets *local_n = 14 / 4 = 3* trapezoids, resulting in only *4 × 3 = 12* trapezoids being used. Therefore, a check is included in the code to ensure that *n % thread_count == 0*.

Since each thread handles a block of *local_n* trapezoids, the **interval width** for each thread is *local_n × h*. Thus, the left and right endpoints for each thread are computed as:

local_a = a + my_rank × local_n × h

local_b = local_a + local_n × h

This means, for example:

- Thread 0: a + 0 × local_n × h

- Thread 1: a + 1 × local_n × h
- Thread 2: a + 2 × local_n × h, and so on.

4.3 Scope Variable

the **scope of variables** decides which threads can access them during parallel execution.

In serial programming, a variable declared inside a function has *function scope*, while a variable declared outside all functions has *file scope*. Similarly, in OpenMP, variables can have **shared** or **private** scope depending on where they are declared.

A **shared variable** can be accessed by all threads in a team. Only one copy exists, and all threads operate on it. Variables declared **before** a parallel directive are shared by default. For example, in the trapezoidal rule program, variables like a, b, n, global_result, and thread_count are shared since they are declared in the main function before the parallel region.

A **private variable** is unique to each thread. Each thread has its own copy, and any modification by one thread does not affect others. Variables declared **inside** a parallel block or inside a function executed in parallel are private by default. For example, in the Trap function:

*global_result_p += my_result;

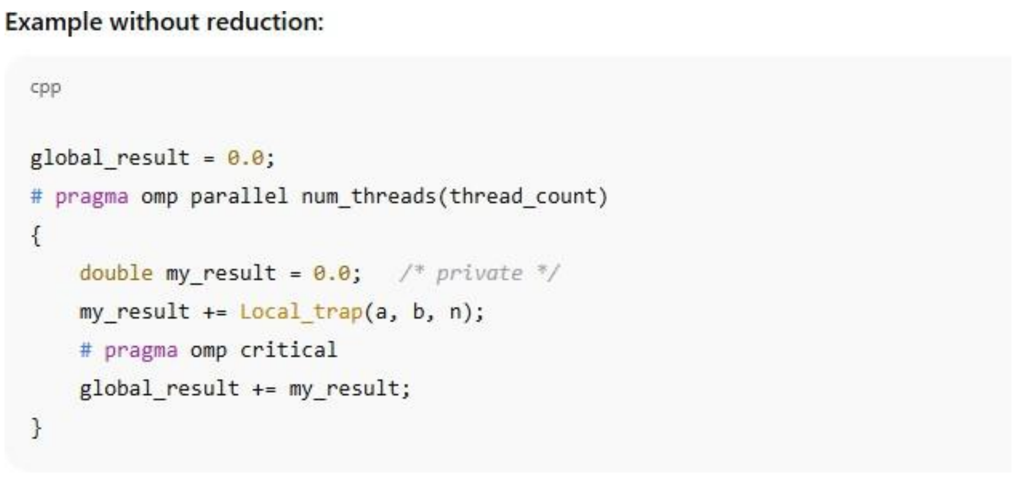
Here, global_result_p is a private pointer for each thread, but it refers to the **shared** variable global_result declared in main. Therefore, all threads update the same shared variable, and this requires synchronization using a critical section.

If global_result were private, each thread would have its own copy, and the final result in main would be incorrect.

4.4 The Reduction Clause

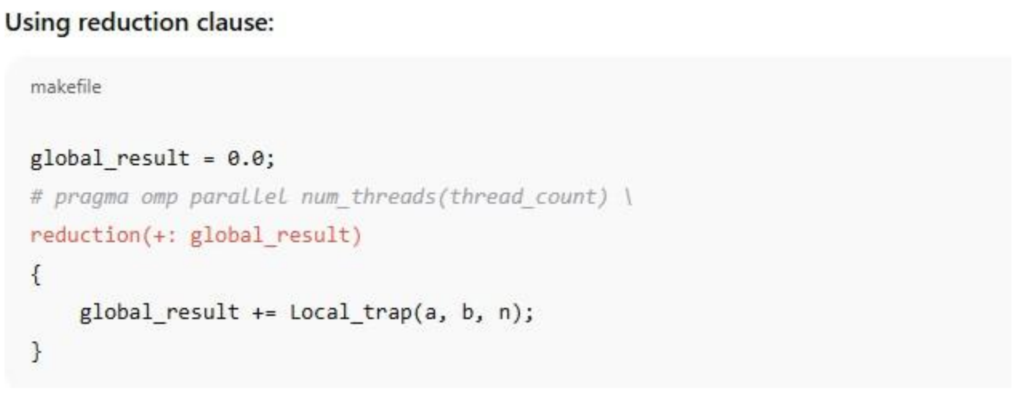
In parallel programs, when multiple threads update a shared variable (like global_result), we must ensure updates happen correctly without making the code sequential. Using a **critical section** ensures correctness but reduces parallelism since only one thread executes the section at a time.

To solve this, OpenMP provides the **reduction clause**, which allows each thread to have its own **private copy** of a variable and combine the results automatically after the parallel region ends.


Example without reduction:

cpp

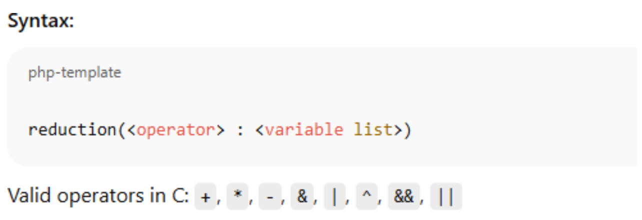
```cpp
global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
{
    double my_result = 0.0;    /* private */
    my_result += Local_trap(a, b, n);
    # pragma omp critical
    global_result += my_result;
}
```


Using reduction clause:

makefile

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

Here, global_result is declared as a reduction variable, and the operator + specifies that all threads' results are added together.


Syntax:

php-template

```
reduction(<operator> : <variable list>)
```

Valid operators in C: `+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`

OpenMP creates a **private copy** of the reduction variable for each thread.

Each thread updates its copy during execution.

At the end of the parallel block, OpenMP automatically **combines** all private results using the specified operator and stores the final value in the **shared variable**.

.

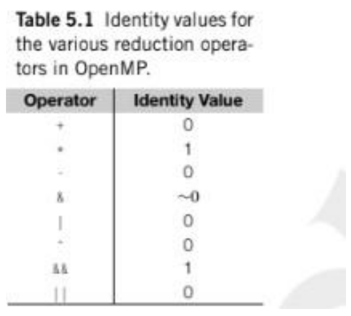


Table 5.1 Identity values for the various reduction operators in OpenMP.

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

4.5 The *Parallel for* Directive

- The #pragma omp parallel for directive is used to automatically parallelize loops.
- It divides loop iterations among threads without needing manual work distribution**.**

```
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 **parallel for** directive forks multiple threads to execute loop iterations in parallel.
- The **system divides iterations** (usually block-wise) among available threads.
- The **loop variable (i)** is **private** by default.
- Other variables are **shared** unless declared otherwise.
- The **reduction(+: approx)** clause ensures each thread keeps a private copy of approx, and all partial results are added safely at the end.
- Without reduction, approx would be a shared variable, causing incorrect results due to race conditions

4.5.1 Caveats

The #pragma omp parallel for directive makes it easy to parallelize loops — sometimes by simply adding a single line before a loop. However, there are a few important **limitations** to keep in mind:

- **Only for loops can be parallelized** — OpenMP does not support while or do-while loops directly.
  (These can be rewritten as equivalent for loops if needed.)
- **Number of iterations must be known before loop execution.**
  OpenMP can only parallelize loops when:
    - The iteration count can be determined **from the for statement itself**, and
    - **Before** the loop starts. or loops with break statements inside cannot be parallelized.

```
for (;;)
```

- **Loop must be in canonical form.**
  The loop should follow a standard pattern:

```
for (index = start; index < end; index += incr)
```

Restrictions:

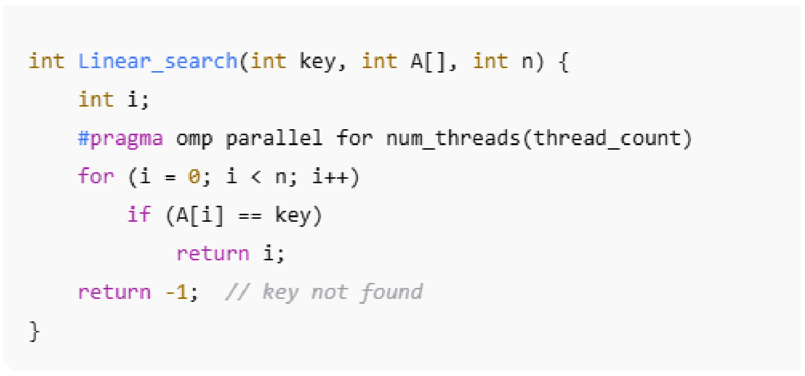
    - index must be an integer or pointer.
    - start, end, and incr must be compatible and constant during execution.
    - index can only be modified by the increment part of the for statement.

- **Structured block requirement.**
  The loop should not have multiple exit points (like extra break or goto).

- **Exception:**
  The only allowed early exit is through a call to exit() inside the loop.

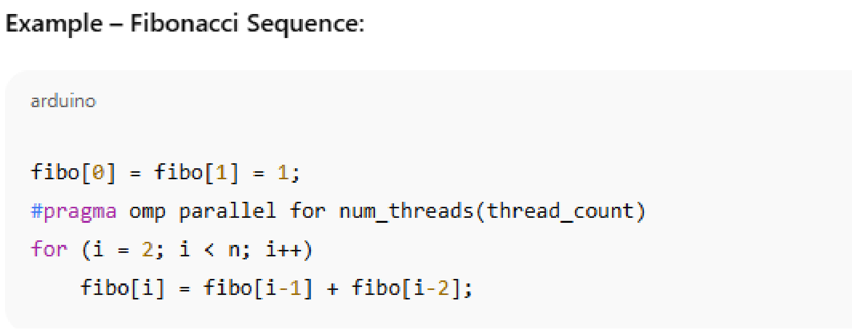
4.5.2 Data Dependences

When a for loop does not meet OpenMP's rules, the compiler may reject it. For example, consider the following linear search code:



```
int Linear_search(int key, int A[], int n) {
    int i;
    #pragma omp parallel for num_threads(thread_count)
    for (i = 0; i < n; i++)
        if (A[i] == key)
            return i;
    return -1;  // key not found
}
```

- This produces an error:
  error: invalid exit from OpenMP structured block
  because return causes an **invalid exit** from the parallel region.
- A more serious issue occurs when **iterations depend on previous iterations**, known as **data dependence**.



Example – Fibonacci Sequence:

arduino

```
fibo[0] = fibo[1] = 1;
#pragma omp parallel for num_threads(thread_count)
for (i = 2; i < n; i++)
    fibo[i] = fibo[i-1] + fibo[i-2];
```

Although the compiler accepts it, the output becomes **unpredictable** when run with multiple threads.
This is because the computation of fibo[i] depends on the values of fibo[i-1] and fibo[i-2], which may not be computed yet by other threads.

For instance:

- One thread may compute fibo[2] to fibo[5]
- Another may start from fibo[6] before the first thread finishes
  → leading to incorrect results such as 1 1 2 3 5 8 0 0 0 0.

1. OpenMP **does not automatically detect data dependences** — the programmer must ensure loops are independent.
2. Loops where one iteration **depends on results from another** (called **loop-carried dependences**) **cannot be safely parallelized** using parallel for.
3. Such loops may need **sequential execution** or advanced OpenMP features like the **Tasking API** for correct parallelization.

4.5.3 Finding Loop-Carried Dependences

When using the **parallel for** directive, we only need to check for **loop-carried dependences** — not all kinds of data dependences.

Example :

```cpp
for (i = 0; i < n; i++) {
    x[i] = a + i * h;
    y[i] = exp(x[i]);
}
```

Here, there is a data dependence between x[i] (line 2) and y[i] (line 3), but since both refer to the **same iteration**, it is **not loop-carried**.
Hence, the parallel version is valid:

```
#pragma omp parallel for num_threads(thread_count)
for (i = 0; i < n; i++) {
    x[i] = a + i * h;
    y[i] = exp(x[i]);
}
```

Each thread works on different values of i, so no conflicts occur.

To find **loop-carried dependences**, we focus on **variables that are updated (written)** inside the loop.
If a variable is **written in one iteration** and **read or written in another**, it creates a **loop-carried dependence**, preventing safe parallelization.

4.5.4 Estimating π

The formula for π is:

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

Serial version:

c

```c
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
    sum += factor / (2 * k + 1);
    factor = -factor;
}
pi_approx = 4.0 * sum;
```

Here, factor must be a **double** to ensure floating-point accuracy during division.

If we directly parallelize this with OpenMP:

```c
#pragma omp parallel for num_threads(thread_count) reduction(+:sum)
for (k = 0; k < n; k++) {
    sum += factor / (2 * k + 1);
    factor = -factor;
}
```

the output becomes **incorrect** because factor depends on its previous value — this creates a **loop-carried dependence**.
Different threads updating factor simultaneously cause data inconsistency.

To fix it, calculate factor independently for each iteration:

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

However, factor is still **shared** among threads by default.
We must make it **private** to avoid interference between threads:

```
double sum = 0.0;
#pragma omp parallel for num_threads(thread_count) \
reduction(+:sum) private(factor)
for (k = 0; k < n; k++) {
    if (k % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
    sum += factor / (2 * k + 1);
}
pi_approx = 4.0 * sum;
```

- reduction(+:sum) ensures each thread keeps a local sum and combines them safely.
- private(factor) ensures each thread has its own copy of factor.

Variables with private scope are uninitialized at the start and undefined after the parallel block.

```
int x = 5;
#pragma omp parallel num_threads(thread_count) private(x)
{
    int my_rank = omp_get_thread_num();
    printf("Thread %d > before init, x = %d\n", my_rank, x);
    x = 2 * my_rank + 2;
    printf("Thread %d > after init, x = %d\n", my_rank, x);
}
printf("After parallel block, x = %d\n", x);
```

4.5.5 More on Scope

In OpenMP, correctly specifying the **scope of variables** inside a parallel or parallel for block is crucial to ensure correct execution and avoid data conflicts.

The issue we saw with the variable factor while estimating π is quite common.
To handle such cases properly, programmers should **explicitly declare the scope** of all variables instead of relying on OpenMP's default behavior.

The default(none) Clause

By adding:

```
default(none)
```

to a parallel directive, OpenMP **forces the programmer** to explicitly specify the scope of every variable that is declared **outside** the parallel block.

- Variables declared **inside** the block are **automatically private** (each thread gets its own copy).
- Variables declared **outside** the block must be explicitly declared as shared, private, or reduction

Example – π Calculation

```c
double sum = 0.0;
#pragma omp parallel for num_threads(thread_count) \
default(none) reduction(+:sum) private(k, factor) shared(n)
for (k = 0; k < n; k++) {
    if (k % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
    sum += factor / (2 * k + 1);
}
```

In this code:

- sum → **reduction variable** (partially private, then combined)
- factor, k → **private** (each thread has its own copy)
- n → **shared** (same value for all threads)

4.6 More about loops in OpenMP: sorting

4.6.1 Bubble Sort

The **Bubble Sort** algorithm repeatedly compares adjacent elements and swaps them if they are in the wrong order. This process continues until the entire list is sorted in ascending order.

```
for (list_length = n; list_length >= 2; list_length--)
    for (i = 0; i < list_length - 1; i++)
        if (a[i] > a[i + 1]) {
            tmp = a[i];
            a[i] = a[i + 1];
            a[i + 1] = tmp;
        }
```

Here, a is an array of n integers.
The **outer loop** finds the largest element in each pass and places it at its correct position. After the first pass, the largest element is at a[n−1]; after the second, the next largest is at a[n−2], and so on.

The **inner loop** performs pairwise comparisons and swaps elements to "bubble up" the largest value in the current pass.

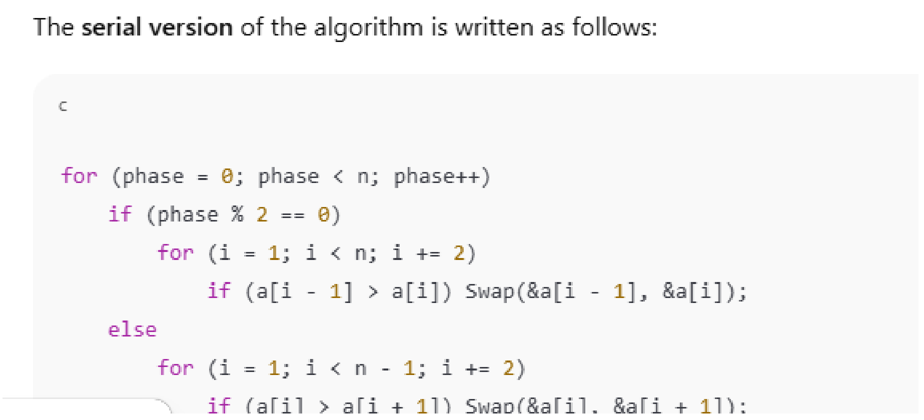
However, **loop-carried dependencies** exist in both loops:

- The **outer loop** depends on results of previous iterations (each pass works on a partially sorted list from the prior pass).
- The **inner loop** also depends on results of earlier comparisons (a swap affects the next comparison).

Due to these dependencies, **Bubble Sort is difficult to parallelize** without rewriting the algorithm. The parallel for directive in OpenMP cannot be directly applied to these dependent loops.

4.6.2 Odd–Even Transposition Sort

Odd–even transposition sort is a **variation of bubble sort** that offers more opportunities for **parallelism**. It repeatedly compares and swaps adjacent elements, but alternates between **odd** and **even phases** to gradually bring the list into order.



The **serial version** of the algorithm is written 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]);
```

Here, a is a list of n integers sorted in **increasing order** after n phases.
During an **even phase** (phase % 2 == 0), the algorithm compares and swaps the pairs (a[0], a[1]), (a[2], a[3]), and so on.
During an **odd phase**, it compares and swaps (a[1], a[2]), (a[3], a[4]), etc.
After all phases are completed, the list becomes sorted.

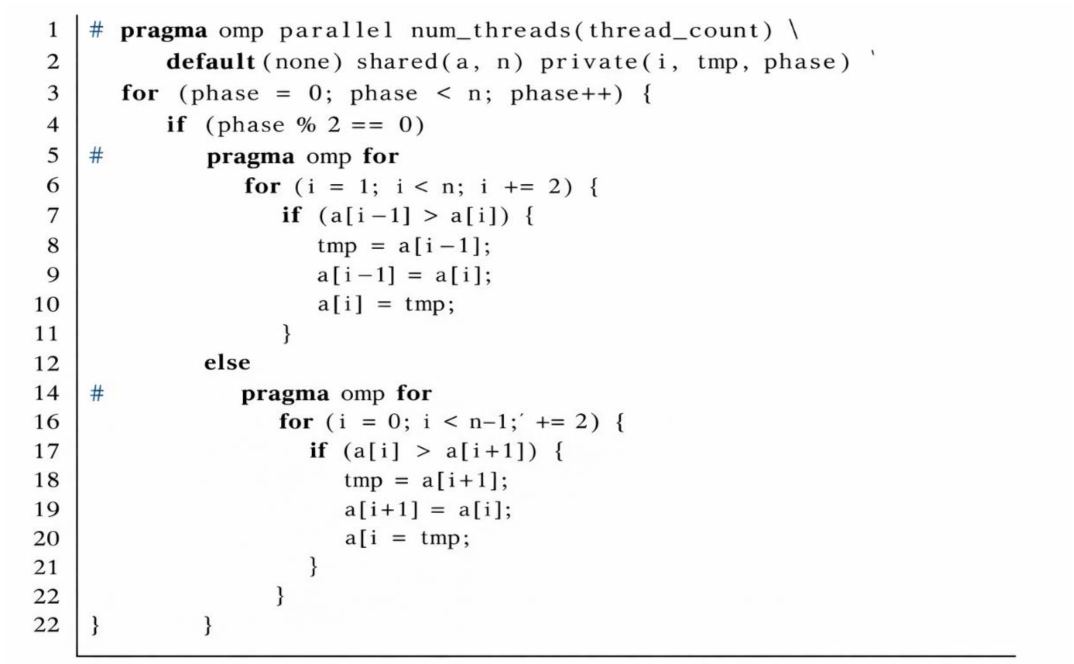
For example, if the initial list is a = {9, 7, 8, 6}, the algorithm performs a sequence of phases until the list becomes {6, 7, 8, 9}.

It can be observed that the **outer loop** has a **loop-carried dependence**, since each phase depends on the outcome of the previous phase. Therefore, the outer loop cannot be parallelized.
However, the **inner loops** within each phase **do not have any loop-carried dependence**, since they operate on **disjoint pairs**. Hence, they can be safely parallelized.

In OpenMP, this can be achieved using the **parallel** and **for directives**.
The team of threads is **forked once** before the outer loop and reused for each phase,

thus avoiding the repeated cost of creating and joining threads.
Each phase still executes synchronously because OpenMP provides an **implicit barrier** at the end of every for directive, ensuring that no thread begins the next phase until all have completed the current one.

This improved OpenMP implementation is shown in **Program 5.5**, where the threads are created once using a parallel directive and the for directive is used within the block to parallelize the inner loops.

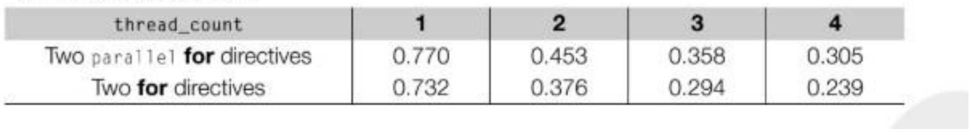


```
# pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp for
            for (i = 1; i < n; i += 2) {
               if (a[i-1] > a[i]) {
                  tmp = a[i-1];
                  a[i-1] = a[i];
                  a[i] = tmp;
               }
         else
#           pragma omp for
               for (i = 0; i < n-1;´ += 2) {
                  if (a[i] > a[i+1]) {
                     tmp = a[i+1];
                     a[i+1] = a[i];
                     a[i = tmp;
                  }
               }
}        }
```

Program 5.5: Second OpenMP implementation of odd-even sort.

Performance results (Table 5.3) show that this version achieves better speedup, being about **17% faster** than the version that uses two parallel for directives, due to reduced thread management overhead.

Table 5.3 Odd-even sort with two `parallel` **for** directives and two **for** directives. Times are in seconds.



`thread_count`	1	2	3	4
Two `parallel` **for** directives	0.770	0.453	0.358	0.305
Two **for** directives	0.732	0.376	0.294	0.239

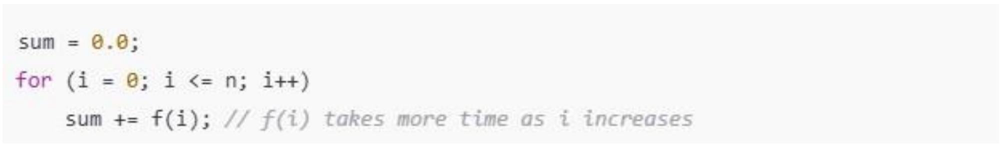
4.7 Scheduling loops

When using parallel for, the assignment of loop iterations to threads is **system dependent**. Most OpenMP implementations use **block partitioning**, where:

- If there are n iterations and t threads,
    - Thread 0 gets 0 → n/t − 1

- Thread 1 gets n/t → 2n/t – 1
- ...and so on.

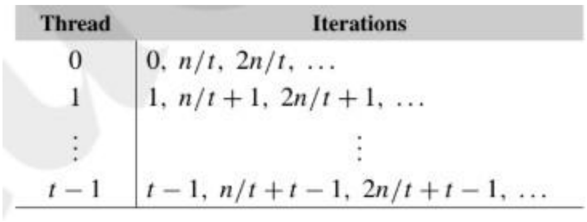
Problem: If the work per iteration is **uneven**, some threads may do much more work than others.



```c
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i); // f(i) takes more time as i increases
```

Using **block partitioning**, thread t-1 gets heavier iterations → poor load balancing.

Using **cyclic partitioning**, iterations are assigned in a **round-robin** manner:



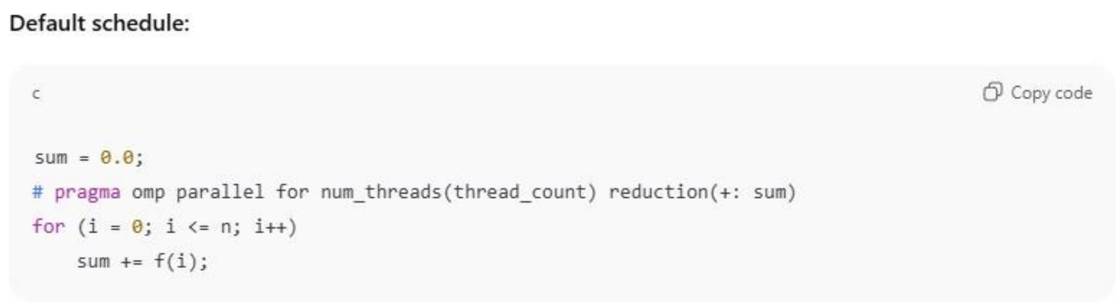
Thread	Iterations
0	$0,\ n/t,\ 2n/t,\ \ldots$
1	$1,\ n/t+1,\ 2n/t+1,\ \ldots$
$\vdots$	$\vdots$
$t-1$	$t-1,\ n/t+t-1,\ 2n/t+t-1,\ \ldots$

- Example program with f(i) increasing in time:
    - Block assignment (2 threads) → speedup = 1.33
    - Cyclic assignment (2 threads) → speedup = 1.99

OpenMP solution: Use the **schedule clause** in parallel for or for directives to control iteration assignment and improve performance.

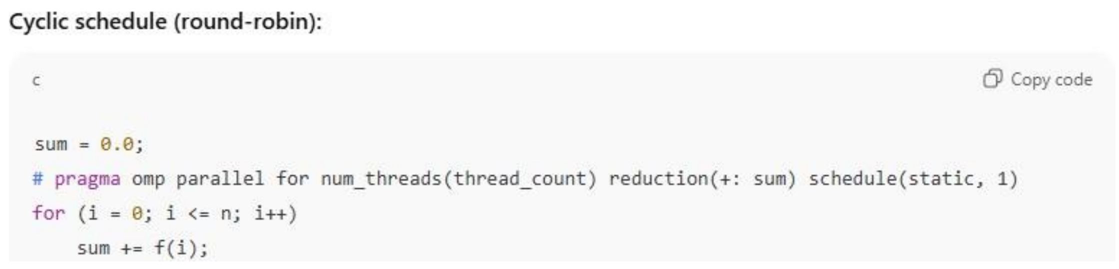
4.7.1 The *Schedule* Clause

The **schedule clause** controls how loop iterations are assigned to threads.



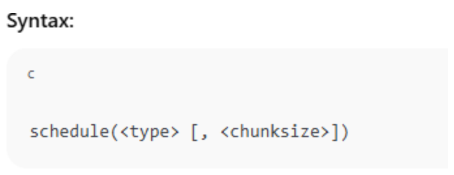
Default schedule:

```c
sum = 0.0;
# pragma omp parallel for num_threads(thread_count) reduction(+: sum)
for (i = 0; i <= n; i++)
    sum += f(i);
```



Cyclic schedule (round-robin):

```c
sum = 0.0;
# pragma omp parallel for num_threads(thread_count) reduction(+: sum) schedule(static, 1)
for (i = 0; i <= n; i++)
    sum += f(i);
```



Syntax:

```c
schedule(<type> [, <chunksize>])
```

Schedule types:

- **static:** Iterations assigned to threads before loop execution (can specify chunk size).
- **dynamic / guided:** Iterations assigned while the loop executes; threads request more work after finishing their current chunk.
- **auto:** Compiler/runtime decides.
- **runtime:** Determined at runtime via environment variable.

Chunk size:

- A block of consecutive iterations.
- Only applies to **static, dynamic, and guided** schedules.
- Determines how iterations are grouped before assignment to threads.

Visualization

- Shows static, dynamic, and guided scheduling and how work is divided among threads.

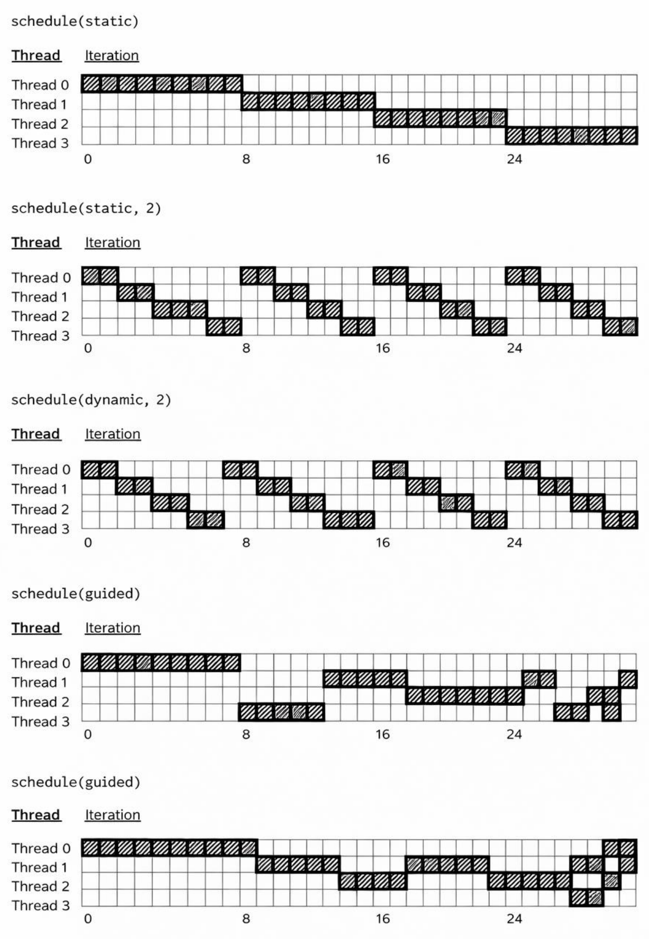


FIGURE 5.5

Scheduling visualization for the static, dynamic, and guided schedule types with 4 threads and 32 iterations. The first static schedule uses the default chunksize, whereas the second uses a chunksize of 2. The exact distribution of work across threads will vary between different executions of the program for the dynamic and guided schedule types, so this visualization shows one of many possible scheduling outcomes.

4.7. 2 The *Static* Schedule type

In a **static schedule**, iterations are divided into **chunks** and assigned to threads **before the loop starts**, often in a **round-robin** fashion.

Examples: Suppose 12 iterations (0–11) and 3 threads:

- `schedule(static, 1)` – each thread gets 1 iteration at a time in round-robin:

  ```yaml
  Thread 0: 0, 3, 6, 9
  Thread 1: 1, 4, 7, 10
  Thread 2: 2, 5, 8, 11
  ```

- `schedule(static, 2)` – 2 iterations per chunk:

  ```yaml
  Thread 0: 0, 1, 6, 7
  Thread 1: 2, 3, 8, 9
  Thread 2: 4, 5, 10, 11
  ```

- `schedule(static, 4)` – 4 iterations per chunk:

  ```yaml
  Thread 0: 0, 1, 2, 3
  Thread 1: 4, 5, 6, 7
  Thread 2: 8, 9, 10, 11
  ```

- Most OpenMP implementations: schedule(static, total_iterations / thread_count)
- If chunksize is omitted, it's roughly total_iterations / thread_count.

- Iterations take roughly **equal time**.
- Can improve **memory access speed** on NUMA systems by keeping threads assigned to the same ranges across loops.

4.7.3 The *dynamic* and guided *Schedule* types

Dynamic Schedule

- Iterations are divided into **chunks** of consecutive iterations.
- **Threads execute a chunk** and request another from the run-time system when done.
- If chunksize is omitted, the default is **1**.
- **Advantages:** Useful when **iteration times vary**, since threads get work on a first-come, first-served basis.
- **Disadvantage:** Slight overhead due to dynamic allocation of chunks.
- **Tip:** Increasing chunksize reduces overhead but may reduce load balancing.

Guided Schedule

- Similar to **dynamic**, but **chunk sizes decrease** over time.
- The first chunk is large; subsequent chunks shrink roughly as:

new_chunk_size ≈ remaining_iterations / thread_count

- If chunksize is not specified, chunks shrink to 1; if specified, they shrink to the given minimum.
- **Advantages:** Better load balancing for loops where later iterations are more compute-intensive.
- **Example:** Trapezoidal rule with n = 10,000 and 2 threads: first chunk ≈ 5000, next ≈ 2500, and so on.
- Table 5.4. We see that the size of the chunk is approximately the number of iterations remaining divided by the number of threads.

4.7.4 The *runtime* Schedule type

- Allows the **loop scheduling type** to be determined **at run-time** instead of at compile-time.
- Achieved using the clause:

```
#pragma omp parallel for schedule(runtime)
```

- The **environment variable OMP_SCHEDULE** controls the schedule type and chunk size.

- Values for OMP_SCHEDULE can be any valid OpenMP schedule: static, dynamic, guided, optionally with a chunksize.

  Example:

```
$ export OMP_SCHEDULE="static,1"
```

  This sets a **cyclic assignment** (iterations assigned one at a time to threads).

Advantages:

- Allows testing different schedules **without recompiling** the program.
- Useful for tuning performance, especially when loop iteration times are unpredictable.

Example usage in a bash script:

```
#!/usr/bin/env bash
declare -a schedules=("static" "dynamic" "guided")
declare -a chunk_sizes=(" " 1000 100 10 1)

for schedule in "${schedules[@]}"; do
    echo "Schedule: ${schedule}"
    for chunk_size in "${chunk_sizes[@]}"; do
        echo "Chunk Size: ${chunk_size}"
        sched_param="${schedule}"
        if [[ "${chunk_size}" != " " ]]; then
            sched_param="${schedule},${chunk_size}"
        fi
        OMP_SCHEDULE="${sched_param}" ./omp_mat_vect 4 2500 2500
    done
    echo
done
```

4.7.5 Which Schedule?

Choosing a Schedule in OpenMP

- The **schedule clause** affects how loop iterations are divided among threads.
- Each schedule type introduces **different overheads**:
    - **Static:** Least overhead
    - **Dynamic:** More overhead (assigned at run-time)
    - **Guided:** Highest overhead (adaptive chunking)

Condition	**Recommended Schedule**	**Reason**
Each iteration takes **similar time**	**Default or static**	Minimal overhead, best performance
Iteration cost **increases/decreases gradually**	**static, small chunksize** (e.g., schedule(static,1))	Balances workload smoothly
Iteration cost **varies unpredictably**	**dynamic** or **guided**	Allows flexible load balancing at run-time
Want to **test** different scheduling strategies	**runtime**	Can change behavior via OMP_SCHEDULE variable

Performance Tip

- Start with the **default schedule**.
- If performance is poor, **experiment** with static, dynamic, or guided schedules and different chunk sizes.
- Use schedule(runtime) for **easy tuning without recompiling**.

Example Observation

```
schedule(static,1)
```

Changing from the default to in a 2-thread, 10,000-iteration loop improved speedup from **1.33 → 1.99** (near optimal).

4.8 Producers and consumers

4.8.1 Queues

A **queue** is a linear data structure where:

- **Insertion** happens at the **rear**, and
- **Deletion** happens at the **front**.

It follows the **FIFO (First In, First Out)** principle — like customers waiting in line at a supermarket: new customers join at the end, and service starts from the front.

Key operations:

- **Enqueue:** Add an element to the rear.
- **Dequeue:** Remove an element from the front.

Applications in Computer Science:

- **Process scheduling:** Ensures fair access to resources (e.g., disk access).
- **Multithreaded systems:** Used for synchronization between
    - **Producer threads** (which generate tasks or requests) and
    - **Consumer threads** (which process those tasks).

Example:
Producers enqueue data requests (like stock prices), and consumers dequeue and process them — maintaining order and coordination between multiple threads.

4.8.2 Message Passing using Queues

In a **shared-memory system**, message passing between threads can be efficiently managed using **message queues**.

Each thread maintains its own **message queue**:

- To **send a message**, a thread **enqueues** a message into another thread's queue.
- To **receive a message**, a thread **dequeues** from its own queue.

This allows threads to **communicate and coordinate** safely without direct data sharing.

Program Behavior:

- Each thread repeatedly performs two actions:
    1. **Send_msg()** → generates a random integer message and a random destination thread, then enqueues the message.
    2. **Try_receive()** → checks its own queue; if a message is available, dequeues and prints it.
- Each thread sends a fixed number of messages (send_max).
- After sending all messages, it continues to receive until all threads finish.

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
    Send_msg();       // Enqueue a random message to another thread
    Try_receive();    // Check and dequeue any received message
}
while (!Done())
    Try_receive();    // Keep receiving until all threads complete
```

4.8.3 Sending Messages

- When implementing message passing using queues, **sending (enqueueing)** a message becomes a **critical section**.
- This is because:
- The message queue usually maintains a **rear pointer** (tail) to efficiently add new messages.
- If **multiple threads** attempt to enqueue at the same time, the rear pointer may be **corrupted or overwritten**, leading to **message loss or inconsistency**.
- Hence, enqueue operations must be **protected** to ensure mutual exclusion.
- **Critical Section:**
  A code block where **shared data** (like the queue's rear pointer) is accessed or modified by multiple threads and therefore must be executed **by only one thread at a time**.

```
mesg = random();                          // Generate random message
dest = random() % thread_count;          // Select random destination thread

#pragma omp critical                     // Protect enqueue operation
Enqueue(queue, dest, my_rank, mesg);     // Add message to destination queue
```

- The #pragma omp critical directive ensures **safe concurrent access**.

- A thread can even send a **message to itself**, since all threads maintain their own message queue.
- Proper synchronization prevents **data corruption** and ensures **message integrity** in parallel environments.

4.8.4 Receiving Messages

Receiving (dequeueing) messages has **different synchronization requirements** from sending.

Each thread **owns its own queue**, meaning:

- **Only the owner thread** will dequeue messages.
- **Multiple threads** may enqueue messages to that queue.

Since dequeue is done only by one thread, **conflicts are minimal**.

If we track the number of **enqueued** and **dequeued** messages separately, we can compute the **queue size** as:

```
queue_size = enqueued - dequeued
```

- Only the owner thread updates dequeued.
- Multiple threads may update enqueued.

If two threads access enqueued simultaneously (one reading, one writing), the queue size might be off by **±1**, but this only causes a **minor delay**, not data corruption.

Synchronization Strategy:

- No synchronization needed when queue has **2 or more messages**.
- Use a **critical section** only when there's **exactly 1 message** — to avoid possible race conditions between enqueue and dequeue.

```
queue_size = enqueued - dequeued;

if (queue_size == 0)
    return;                         // No messages to receive
else if (queue_size == 1)
    #pragma omp critical
    Dequeue(queue, &src, &mesg);    // Safely dequeue the only message
else
    Dequeue(queue, &src, &mesg);    // Safe: multiple messages exist

Print_message(src, mesg);           // Display received message
```

4.8.5 Termination Detection

In a message-passing program, threads must know **when to stop** receiving messages. This is handled by the **Done()** function.

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

is **incorrect**, because:

- Another thread may **send a message** immediately **after** this check.
- The receiving thread might **terminate too early**, leaving some messages undelivered.

Improved Approach:
After all threads finish sending messages, no new messages will be enqueued.
So, introduce a global counter **done_sending**, incremented by each thread when it finishes sending.

```
queue_size = enqueued - dequeued;

if (queue_size == 0 && done_sending == thread_count)
    return TRUE;        // No pending messages, and all threads finished sending
else
    return FALSE;
```

4.8.6 Startup

When the program starts, a **single master thread** begins execution. It reads the command-line arguments and **allocates an array of message queues**, one for each thread. This array must be **shared among all threads** so that any thread can send messages to any other thread.

Each **message queue** (implemented as a struct) contains:

- a list of messages
- front and rear pointers (or indices)
- count of messages enqueued
- count of messages dequeued

To reduce copying, the array should hold **pointers to these structs.** Once the master thread allocates the array, it starts all threads using a **parallel directive**, and each thread allocates memory for its own queue.

However, a problem can occur if some threads finish allocation earlier and try to enqueue messages into a queue that hasn't been allocated yet—this can cause a

program crash. To prevent this, all threads must wait until every queue is allocated before sending messages.

OpenMP provides a way to synchronize threads using the **barrier directive**:

```
#pragma omp barrier
```

This ensures that **no thread proceeds** beyond the barrier until **all threads** have reached it, guaranteeing safe and synchronized startup.

4.8.7 The *atomic* directive

After completing its message sends, each thread increments the variable **done_sending** before entering its final loop of receives. Since multiple threads update this variable, it forms a **critical section** that must be protected from concurrent access.

Although we could use the **critical** directive, OpenMP provides a more efficient option — the **atomic** directive:

```
#pragma omp atomic
```

The **atomic directive** ensures that a single **assignment operation** on a shared variable is executed **atomically** — that is, **without interruption** by other threads. However, it can only protect **simple update statements** such as:

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

where <op> can be one of +, -, *, /, &, ^, |, <<, or >>.

Note that <expression> must **not reference x**, and only the **load and store of x** are guaranteed to be atomic. For example:

```c
#pragma omp atomic
x += y++;
```

Here, the update to x is atomic, but the increment of y is **not protected**, possibly leading to unpredictable results.

The **purpose of atomic** is performance: many processors support a **hardware-level load–modify–store instruction**, which allows such simple updates to be done safely and much faster than a general critical section.

`4.8.8 Critical sections and locks

In the message-passing program, OpenMP's **critical directive** is used to ensure that only one thread executes a block of code at a time. However, this can affect performance if used carelessly.

In earlier examples, we had just one critical section, so mutual exclusion was simple. But in this program, there are three such operations:

- done_sending++;
- Enqueue(q_p, my_rank, mesg);
- Dequeue(q_p, &src, &mesg);

We don't need complete mutual exclusion across all these—different threads can safely enqueue messages in different queues at the same time. But OpenMP treats **all unnamed critical sections as one composite section**, which **serializes** all such operations and **reduces parallelism**.

To improve this, OpenMP allows **named critical sections**:

```
#pragma omp critical(name)
```

Blocks with **different names** can execute simultaneously. However, names are fixed at **compile time**, so we can't create a unique critical section for each thread dynamically. Hence, we use **locks** instead.

A **lock** is a shared data structure that allows programmers to manually control access to critical sections. Its usage follows this pattern:

```
// Executed by one thread
Initialize the lock;

// Executed by multiple threads
Set or acquire the lock;
Critical section;
Unset or release the lock;

// Executed by one thread
Destroy the lock;
```

When a thread calls the **lock function**, it either acquires the lock (if available) or waits until it is released. Once inside the critical section, no other thread can enter until the lock is **unset**.

OpenMP provides two kinds of locks:

- **Simple locks** – can be set once before being unset.
- **Nested locks** – can be set multiple times by the same thread.

For our program, **simple locks** are sufficient. The functions are declared in omp.h:

```
void omp_init_lock(omp_lock_t *lock_p);    // Initialize the lock
void omp_set_lock(omp_lock_t *lock_p);     // Acquire the lock (blocks if in use)
void omp_unset_lock(omp_lock_t *lock_p);   // Release the lock
void omp_destroy_lock(omp_lock_t *lock_p); // Destroy the lock
```

Thus, by using locks instead of generic critical sections, we can ensure **safe and efficient synchronization** while preserving **maximum parallel performance**.

4.8.9 Using locks in the message-passing program

In the message-passing program, we need **mutual exclusion** for each **individual message queue**, not for a single block of code shared by all threads. Using **locks** allows this finer control.

By adding a data member of type **omp_lock_t** to the **queue struct**, each queue maintains its own lock. Whenever a thread accesses a queue, it first sets the lock to ensure exclusive access, performs the operation, and then unsets the lock.

For example, the earlier code using a *critical* section:

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

is replaced with:

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

Similarly, for receiving messages:

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

Now, **only threads accessing the same queue** block each other, allowing true parallel execution. In contrast, the earlier critical directive forced **all threads** to wait, even if they were accessing different queues.

It's also possible to place the lock operations inside the Enqueue and Dequeue functions themselves. However, this could reduce performance, especially if Dequeue is frequently called by Try_receive. To maintain the current program structure and efficiency, it's better to keep the omp_set_lock and omp_unset_lock calls in the **Send** and **Try_receive** functions.

Since each queue now contains its own lock, it should be:

initialized when the queue is created (in the queue initialization function), using

```
omp_init_lock(&q_p->lock);
```

and **destroyed** by the owning thread before freeing the queue, using

```
omp_destroy_lock(&q_p->lock);
```

This ensures safe, fine-grained synchronization and better parallel performance in the message-passing system.

4.8.10 *Critical* directives, *atomic* directives, or locks?

When deciding between **critical directives, atomic directives, and locks** in OpenMP, the choice depends on the **type of mutual exclusion required** and **performance considerations**.

1. **Atomic directive**
   - Typically the **fastest option** for simple updates.
   - Works only for single assignment or increment/decrement statements of the forms allowed by OpenMP.
   - Guarantees atomicity for that statement, but may enforce mutual exclusion **across all atomic directives** in some implementations.
   - Limitation: If multiple atomic directives protect **different variables**, one might still block the other in some implementations, though standard does not require it.
2. **Critical directive**
   - Simple and easy to use for any block of code.
   - **Unnamed critical sections** are treated as a single composite section; only one thread executes any unnamed critical section at a time.
   - **Named critical sections** allow different sections to execute concurrently.
   - Performance is generally comparable to locks in most OpenMP implementations.
3. **Locks**
   - Provide **fine-grained control** over mutual exclusion, especially useful for **data structures**, such as individual queues, rather than general code blocks.
   - Can be initialized, set, unset, and destroyed explicitly, allowing concurrent access to different resources safely.
   - Slightly more complex to use than critical directives but essential when **per-resource synchronization** is required.

4.8.11 Some caveats

When using **mutual exclusion** in OpenMP, there are several important **caveats** to keep in mind:

1. **Don't mix atomic and critical on the same variable**

   Example:

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

   Here, the atomic directive does not protect the second operation, so the two blocks **can interfere**, producing incorrect results.

Solution: either rewrite the code so atomic can be used for both, or protect both with a **critical directive**.

2. **No guarantee of fairness**

   Threads waiting to enter a critical section may be **blocked indefinitely** if other threads repeatedly acquire it.

   Example:

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

   Thread 1 could potentially never execute the assignment if other threads keep entering the critical section first.

3. **Avoid nesting critical sections**

   Example of **deadlock**:

```
#pragma omp critical
y = f(x);

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

   If a thread inside the first critical section tries to enter the second, it **blocks forever**, causing a deadlock.

   Using **named critical sections** can sometimes solve this:

```
#pragma omp critical(one)
y = f(x);

double f(double x) {
  #pragma omp critical(two)
  z = g(x);
}
```

However, deadlocks can still occur if threads enter critical sections in **different orders**. For example:

Time	Thread u	Thread v
0	Enter one	Enter two
1	Attempt two	Attempt one

2	Block	Block

Both threads remain **blocked forever**.

4.9 Caches, Cache Coherence, and False Sharing

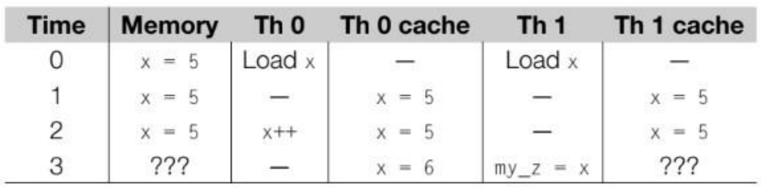
1. Introduction to Cache Memory

- Over the years, the **speed of processors** has increased much faster than the speed of **main memory**.
- When a processor accesses data directly from main memory for each operation, most of its time is wasted waiting for the data to arrive.
- To overcome this, modern processors use **cache memory**, which is a small but much faster memory located close to the processor.
- Cache design is based on the principles of **temporal locality** (recently used data is likely to be reused soon) and **spatial locality** (data near recently accessed locations is likely to be used soon).
- Instead of transferring just one memory location, a **block of memory** (called a *cache line* or *cache block*) is transferred between the processor's cache and main memory.

2. Cache and Shared Memory

- In shared-memory systems, caches can significantly affect performance.
- Suppose a shared variable x = 5 is accessed by both thread 0 and thread 1.
  - Each thread loads x into its own cache and executes statements like my_y = x;.
  - If thread 0 then executes x++;, and thread 1 later executes my_z = x;, what value will my_z hold—5 or 6?
- The issue arises because there are **multiple copies** of x:
  - one in **main memory**,
  - one in **thread 0's cache**, and
  - one in **thread 1's cache**.
- When thread 0 updates x, the other copies may become outdated.
  - This is known as the **cache coherence problem**.
- To solve this, most systems enforce **cache coherence** by:
  - marking cache lines invalid when data is modified elsewhere, and
  - ensuring updated data is fetched before further access.
- Thus, caches must communicate to stay synchronized.
  - For more details, refer to **Table 5.5** (sequence of accesses)

Table 5.5 Memory and cache accesses.



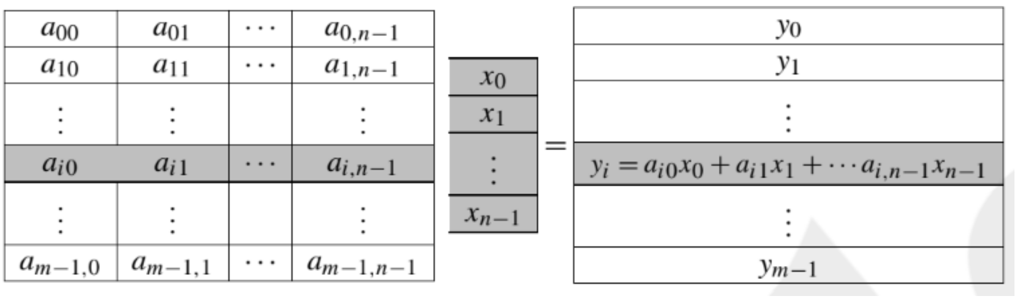
Time	Memory	Th 0	Th 0 cache	Th 1	Th 1 cache
0	x = 5	Load x	—	Load x	—
1	x = 5	—	x = 5	—	x = 5
2	x = 5	x++	x = 5	—	x = 5
3	???	—	x = 6	my_z = x	???

3. Illustration: Matrix–Vector Multiplication

- Consider a matrix $A = (a_{ij})$of size $m \times n$and a vector $x$with $n$components.
  - Their product $y = Ax$is a vector with $m$components.
  - Each element is computed as:

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

  - Refer to **Figure 5.6** for a schematic view.



a_{00} a_{01} $\cdots$ $a_{0,n-1}$
a_{10} a_{11} $\cdots$ $a_{1,n-1}$
a_{i0} a_{i1} $\cdots$ $a_{i,n-1}$
$a_{m-1,0}$ $a_{m-1,1}$ $\cdots$ $a_{m-1,n-1}$
x_0 x_1 x_{n-1}
$=$
y_0
y_1
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
y_{m-1}

- The **serial implementation** is as follows:

```
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j] * x[j];
}
```

- There are **no loop-carried dependences** in the outer loop because:
  - A and x are read-only, and
  - each iteration updates a separate element of y.
- Hence, we can **parallelize** the outer loop using OpenMP:
- #pragma omp parallel for num_threads(thread_count) \
- default(none) private(i,j) shared(A,x,y,m,n)

```
#pragma omp parallel for num_threads(thread_count) \
default(none) private(i,j) shared(A,x,y,m,n)
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j] * x[j];
}
```

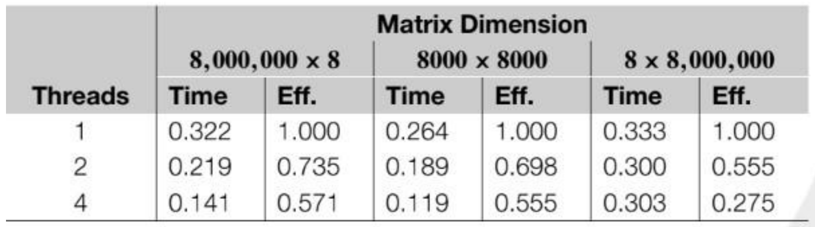
4. Performance Metrics

- If T_{serial} is the run time of the serial version and $T_{parallel}$ is the parallel version run time,
  then **Efficiency (E)** is given by:

$$E = \frac{S}{t} = \frac{T_{serial}}{t \times T_{parallel}}$$

where S is the speedup and t is the number of threads.

Table 5.6 Run-times and efficiencies of matrix-vector multiplication (times are in seconds).



	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

- **Table 5.6** lists the run times and efficiencies for various input sizes.
  - All cases involve the same total number of floating-point operations (64 million).
  - However, execution times differ due to cache performance.

5. Cache Misses and Performance Variation

- **Write-miss**: occurs when a processor writes to a variable that is not in cache, requiring access to main memory.
- **Read-miss**: occurs when a processor reads a variable that is not in cache.
- Using a cache profiler such as **Valgrind**, the following was observed:
  - For the **8,000,000 × 8** input, there were more *write-misses*—mostly due to initializing the larger y array.
  - For the **8 × 8,000,000** input, there were more *read-misses*—mostly due to frequent access of the larger x array.
- Hence, differences in cache performance significantly affect the execution time even when arithmetic operations are constant.

6. Parallel Efficiency and Cache Behavior

- As the number of threads increases, efficiency decreases—especially for the **8 × 8,000,000** case.
- For two threads, efficiency drops by over **20%**, and for four threads, by **50%**, compared to the other two cases.
- The reason again lies in **cache behavior** and particularly **cache coherence traffic**.

7. Understanding False Sharing

- **Cache coherence is maintained at the cache-line level**, typically 64 bytes (i.e., 8 doubles).
- When one core modifies *any variable* in a cache line, all other cores' copies of that line are invalidated—even if they use different variables in the same line.
- This leads to a phenomenon called **false sharing**.

Example:

- In the **8 × 8,000,000** problem with four threads:
  - If all y elements lie within one cache line, each thread's update to its assigned y element invalidates others' cache lines.
  - Each thread must repeatedly reload data from main memory, drastically slowing execution.
  - This occurs even though threads access **different variables**, not shared ones.
- Therefore, false sharing occurs when:
  - Threads update **different variables located in the same cache line**, and
  - Cache controllers treat the entire line as modified and invalidate others' copies.

8. Why False Sharing Does Not Always Occur

- For the **8000 × 8000** case:
  - Each thread processes a distinct range of y values, e.g.
    - Thread 2 → y[4000] to y[5999]
    - Thread 3 → y[6000] to y[7999]
  - Only at the interface (between y[5999] and y[6000]) might a single cache line be shared.
  - However, accesses occur at different times, minimizing interference.
- Hence, false sharing has **negligible impact** on this case and the **8,000,000 × 8** case.
- Arrays A and x are read-only, so **no cache updates** occur there.

9. Techniques to Avoid False Sharing

- Two common approaches to prevent false sharing:
  1. **Padding –** Insert dummy elements in y to ensure each thread's data falls into different cache lines.
  2. **Private Buffers –** Let each thread use its private copy of y during computation and then update the shared array afterward.

4.10 Tasking in OpenMP

Traditional OpenMP constructs such as *parallel* and *parallel for* work efficiently when the number of iterations or parallel blocks is fixed and known. However, several real-world problems—like **web servers**, **recursive algorithms**, and **producer–consumer programs**—have dynamic or unpredictable workloads that cannot be parallelized using these constructs, since *while* and *do–while* loops or unbounded *for* loops are not supported. To address this, **OpenMP 3.0 introduced Tasking functionality**, enabling the creation of independent computation units called **tasks**, which the runtime schedules dynamically.

The **task directive** (#pragma omp task) defines a new unit of work. When a thread encounters it, the OpenMP runtime creates a task that may execute later depending on system load. Tasks must be generated **within a parallel region**, usually under a **single directive**, ensuring that only one thread launches them:

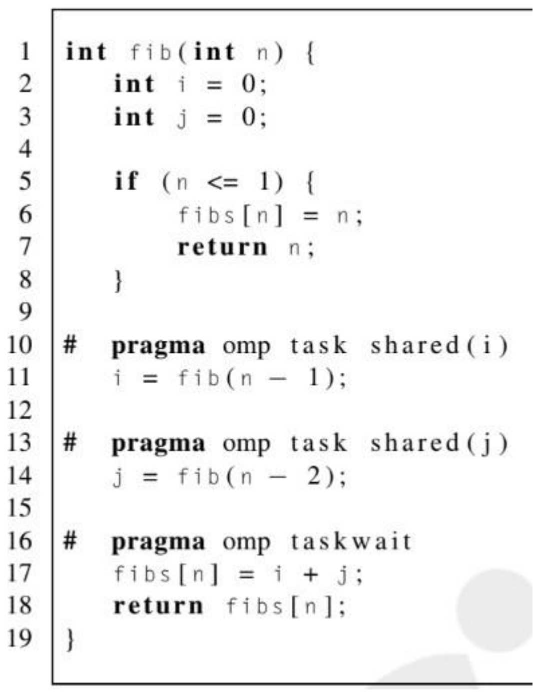
```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task
   ...
}
```

This structure prevents multiple threads from redundantly creating identical tasks.

A classic example is the **recursive Fibonacci computation**. In the serial version, recursive calls fib(n-1) and fib(n-2) are executed sequentially. By adding task directives before these calls, each can be executed in parallel. However, results initially appear incorrect because variables like *i* and *j* become **private** to each task. Declaring them as **shared** ensures all tasks reference the same memory locations:

```
#pragma omp task shared(i)
i = fib(n - 1);
#pragma omp task shared(j)
j = fib(n - 2);
```

Even then, execution order is undefined, and a thread may proceed before subtasks complete. To synchronize them, we use the **taskwait directive**, which acts as a barrier ensuring all child tasks finish before the parent continues. The corrected program (Program 5.6) now produces accurate results.



```
int fib(int n) {
    int i = 0;
    int j = 0;

    if (n <= 1) {
        fibs[n] = n;
        return n;
    }

#   pragma omp task shared(i)
    i = fib(n - 1);

#   pragma omp task shared(j)
    j = fib(n - 2);

#   pragma omp taskwait
    fibs[n] = i + j;
    return fibs[n];
}
```

Despite correctness, **performance may degrade** for large *n* due to **task creation overhead**. Each task requires its own data environment, increasing runtime costs. To mitigate this, OpenMP allows **conditional task creation** using the *if* clause—for example, #pragma omp task if(n > 20)—so tasks are created only for larger values of *n*. Another optimization eliminates redundant tasks by letting the parent execute one recursive call directly. On a 64-core system, these optimizations roughly halved the execution time for *n = 35*.

4.11 Thread-Safety

In shared-memory programming, **thread-safety** refers to the ability of a block of code or a function to be executed simultaneously by multiple threads **without causing errors or data inconsistencies**. A program is thread-safe if it ensures correct results even when multiple threads access shared resources concurrently.

For example, consider a program that uses multiple threads to **tokenize lines of a text file**. The function strtok() from <string.h> is commonly used for tokenization. Its first call takes a string to be tokenized, and subsequent calls with a NULL argument continue tokenizing the same cached string. Internally, strtok() maintains a **static variable** to remember the position in the string. When used by multiple threads, this cached pointer becomes **shared**, not private, leading to **incorrect results—**threads overwrite each other's tokenization data. Hence, strtok() is **not thread-safe**.

This problem is not unique to strtok; other C library functions such as rand() in <stdlib.h> and localtime() in <time.h> are also not guaranteed to be thread-safe. To overcome this issue, the C standard provides **re-entrant or thread-safe versions** of such functions. The thread-safe version of strtok() is strtok_r(), whose prototype is:

```
char *strtok_r(char *string, const char *separators, char **saveptr);
```

The third argument, saveptr, is a user-managed pointer that keeps track of the position in the string, replacing the internal static variable. Each thread can maintain its own saveptr, making the function **re-entrant** and **thread-safe**.

Thread-safety issues are often difficult to detect because incorrect programs may sometimes produce correct output due to the **non-deterministic scheduling of threads**. Such errors, known as **race conditions**, occur unpredictably and may appear only in certain runs. Therefore, ensuring thread-safety through **proper data scoping** and **use of re-entrant functions** is essential for reliable parallel programs.