



Search Creators

**Visvesvaraya Technological
University(VTU)**

Prepared By

Hanumanthu

Dedicated To.

All Engineering Students

For More Updates

Website: <https://searchcreators.org/>
YouTube: <https://www.youtube.com/@searchcreators7348>
Instagram : <https://www.instagram.com/searchcreators/>
Telegram: <https://t.me/SearchCreators>
WhatsApp: +917348878215
Contact Number: +91 73488 78215

Scan Us | One Stop for All



COPYRIGHT & OWNERSHIP DECLARATION

Full Ownership:

All the content in this study material is the **exclusive intellectual property of Search Creators, Tumakuru.**

It is fully **created, compiled, and owned** by Search Creators.

Source:

The material is **prepared strictly based on VTU-prescribed textbooks** under the **2022 Scheme**, ensuring syllabus accuracy.

No AI / Website Use:

-  No ChatGPT, AI tools, or online generators were used.
-  No third-party websites or digital platforms contributed content.
-  100% manually prepared by the **Search Creators academic team.**

Usage Terms:

- ❖  Minimum cost for **personal student use only** (learning, revision, internal exam prep).
- ❖  **Not allowed** for resale, upload, sharing on public platforms, or reproduction in any form without written permission.
- ❖  Strictly prohibited from being published or claimed by any other party.

Prepared & Owned By:

Search Creators, Tumakuru

Final Year Project & Academic Support Center

 **Contact: 73488 78215**

 **Email: aryahanumanthu@gmail.com**

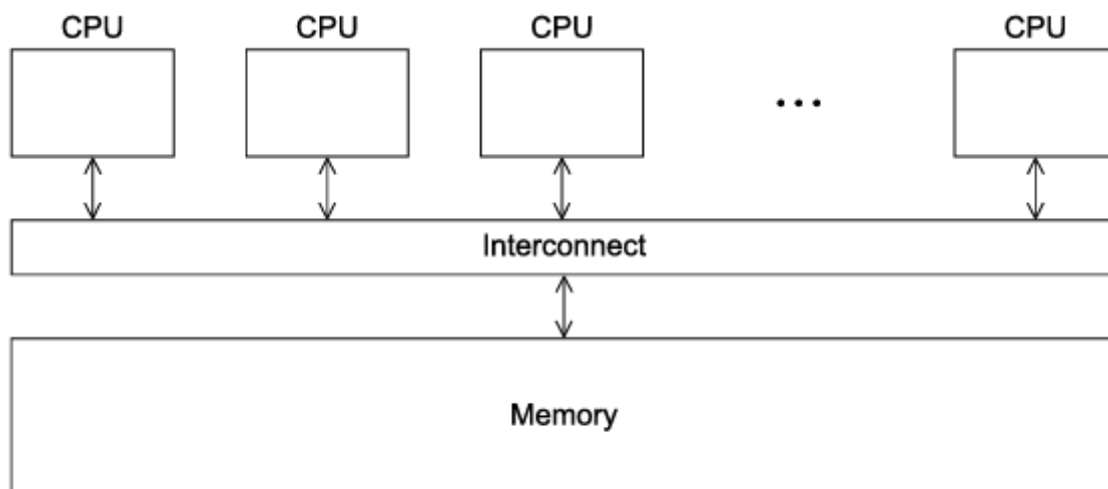
Module -04Shared-memory programming with OpenMP**Openmp pragmas and directives**

OpenMP (Open Multi-Processing) is an **API (Application Programming Interface)** for **shared-memory MIMD (Multiple Instruction, Multiple Data)** programming.

The "MP" in OpenMP signifies "multiprocessing," which is synonymous with shared-memory MIMD computing.

Key Characteristics of OpenMP**Shared-Memory Model:**

OpenMP is designed for systems where all autonomous cores or CPUs have access to a common main memory, as depicted in a shared-memory system (Fig. 5.1).

**FIGURE 5.1**

A shared-memory system.

This means threads can directly read from and write to the same memory locations.

Higher-Level Abstraction than Pthreads:**Pthreads:**

Requires explicit programmer control over each thread's behavior, including thread creation, management, and synchronization. It's a lower-level, function-based library.

OpenMP:

Allows programmers to express parallelism using **directives (pragmas)**. The compiler and run-time system often handle the precise determination of tasks and which thread executes them.

This makes it simpler for certain parallel behaviors, especially parallelizing loops, but can make low-level thread interactions more challenging to program.

Compiler Support Required:

Unlike Pthreads (which is a library that can be linked to any C compiler), OpenMP requires specific compiler support for its pragmas.

A C compiler that doesn't support OpenMP will ignore the directives, resulting in sequential execution.

Incremental Parallelization:

A significant design goal of OpenMP was to enable programmers to **incrementally parallelize existing serial programs**.

This is extremely difficult with MPI (which is for distributed memory) and often challenging with Pthreads due to its low-level nature.

Getting Started with OpenMP

OpenMP uses a "**directives-based**" API. In C and C++, these directives are implemented as special preprocessor instructions called **pragmas**.

- **Pragma Syntax:** Pragmas start with #pragma. For OpenMP, they specifically start with #pragma omp.

```
#pragma omp parallel num_threads(thread_count)
// ... structured block of code to be executed in parallel ...
```

- **Compiler Handling:** Compilers that don't support OpenMP will simply ignore these pragmas. This allows a single OpenMP program source file to be compiled and run either sequentially (if OpenMP is not supported) or in parallel (if it is).

Example: "Hello, World" with OpenMP (Program 5.1)

```
#include <stdio.h>

#include <stdlib.h>#include <omp.h> // OpenMP header file
```

```
void Hello(void); // Thread function prototype

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

    #pragma omp parallel num_threads(thread_count) // OpenMP parallel directive
    Hello(); // Structured block: call to Hello function

1 return 0;
} // main

void Hello(void) {
    int my_rank = omp_get_thread_num(); // Get thread ID
    int thread_count = omp_get_num_threads(); // Get total number of threads

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

Compiling and Running OpenMP Programs

Compilation with GCC: Use the -fopenmp option:

Bash

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

Execution: Specify the number of threads as a command-line argument:

Bash

```
$ ./omp_hello 4
```

Output Behavior: The output order is **non-deterministic** because threads contend for access to stdout. You might see "Hello from thread 0", then "Hello from thread 2", etc., in any order.

The Program

1. **omp.h Inclusion (Line 3):** This header file provides prototypes for OpenMP functions and macro definitions.
2. **strtol (Line 9):** Used to parse the number of threads from the command-line argument.
3. **#pragma omp parallel num_threads(thread_count) (Line 11):** This is the core OpenMP directive.
 - #pragma omp parallel: Indicates that the immediately following structured block of code should be executed in parallel by multiple threads.
 - num_threads(thread_count): This is a **clause** that modifies the parallel directive, explicitly requesting thread_count threads. While not guaranteed (system limitations may apply), in most cases, the requested number of threads will be created.
4. **Execution Flow at parallel directive:**
 - Initially, the program runs as a single **master thread** (often referred to as thread 0).
 - When the master thread encounters the parallel directive, it forms a **team** of thread_count threads. The original master thread continues as part of this team, and thread_count - 1 **child threads** are created.
 - Each thread in the team then executes the **structured block** that follows the directive (in this case, the Hello() function call).
5. **Hello() Function (Lines 17-24):**
 - omp_get_thread_num(): Returns the unique integer ID (rank) of the calling thread within its team (0 to thread_count - 1).
 - omp_get_num_threads(): Returns the total number of threads in the current team.
 - Each thread calls printf independently, leading to the non-deterministic output order.
6. **Implicit Barrier and Termination:**
 - When all threads in the team complete the structured block (e.g., Hello() returns), there is an **implicit barrier**. This means any thread that finishes early will wait for all other threads in its team to complete.
 - After all threads have completed the block, the child threads are **terminated**, and only the **parent thread** (the original master thread) continues execution with the code following the parallel region (Line 14 in main).

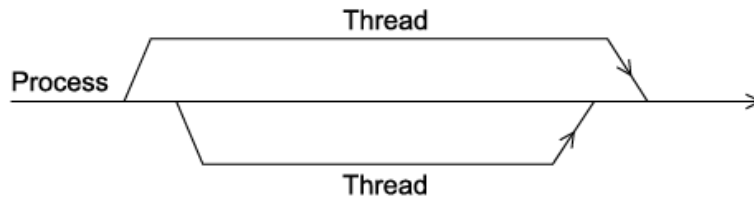


FIGURE 5.2

A process forking and joining two threads.

Error Checking in OpenMP

- **Command-line Arguments:** It's good practice to check if `argv[1]` exists and if `thread_count` is a positive value.
- **Compiler Support (`_OPENMP` macro):** To make code more portable to compilers that might not support OpenMP, check for the preprocessor macro `_OPENMP`. This macro is defined by OpenMP-compliant compilers.

```
#ifdef _OPENMP
#include <omp.h>
#else
// ... in Hello function or wherever OpenMP calls are made ...
#endif

#ifdef _OPENMP
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
#else
    int my_rank = 0;    // Assume single-threaded
    int thread_count = 1; // Assume single-threaded
#endif
```

This allows the program to gracefully compile and run sequentially if OpenMP support is absent. (The textbook examples will often omit this for clarity).

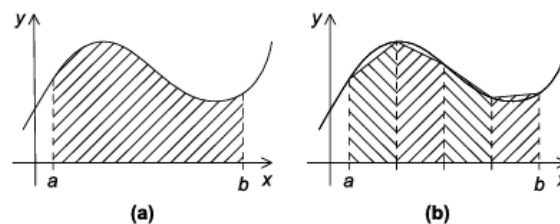


FIGURE 5.3

The trapezoidal rule.

The trapezoidal rule

The trapezoidal rule is a numerical integration method used to estimate the area under a curve $y = f(x)$ between two points $x = a$ and $x = b$.

Serial Algorithm Review:

The interval $[a, b]$ is divided into n equal subintervals, each of length $h = (b - a)/n$. The approximation is the sum of the areas of trapezoids formed over each subinterval. The formula is:

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

where $x_i = a + i \cdot h$.

The serial pseudocode 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;
```

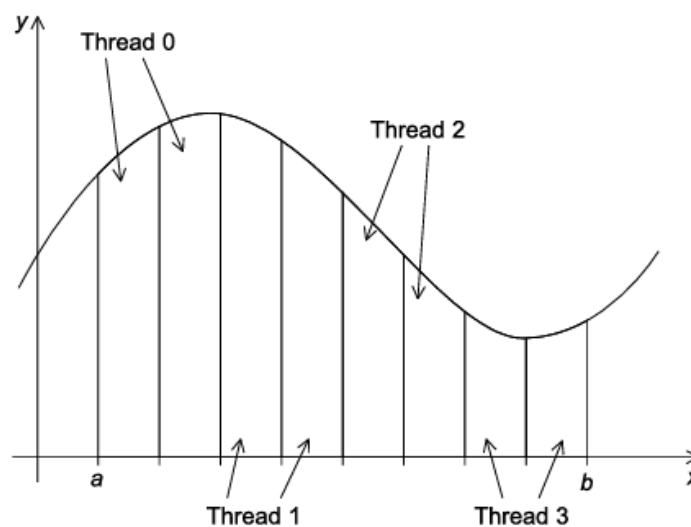


FIGURE 5.4

Assignment of trapezoids to threads.

First OpenMP Version

To parallelize the trapezoidal rule using OpenMP, we apply Foster's parallel program design methodology:

1. Task Partitioning:

- **Type A tasks:** Computing the area of individual trapezoids.
- **Type B task:** Summing all the individual trapezoid areas.

2. **Communication:** No communication is needed among the Type A tasks. However, each Type A task must communicate its computed area to the single Type B task (the global sum).

3. Aggregation and Mapping:

- Assuming many more trapezoids (n) than available cores (`thread_count`), we aggregate tasks by assigning a **contiguous block of trapezoids to each thread**.
- Effectively, this partitions the overall interval $[a,b]$ into smaller, contiguous subintervals, and each thread applies the serial trapezoidal rule to its assigned subinterval to compute a `my_result` (local integral contribution). (See Figure 5.4 for a visual representation of this assignment).

The Challenge: Global Sum and Race Conditions

After each thread computes its `my_result`, these individual contributions need to be summed into a `global_result`. A naive approach like `global_result += my_result;` leads to a **race condition**.

Race Condition:

Occurs when multiple threads attempt to access and modify a shared resource (here, `global_result`), and at least one of the accesses is an update.

The outcome becomes unpredictable because the final value depends on the non-deterministic interleaving of thread operations.

Example Scenario:

- `global_result` starts at 0.
- Thread 0 computes `my_result = 1`.
- Thread 1 computes `my_result = 2`.
- **Problematic Timetable:**
 - **Time 0:** Thread 0 loads `global_result` (0) into its register.
 - **Time 1:** Thread 1 loads `global_result` (0) into its register.
 - **Time 2:** Thread 0 adds its `my_result` (1) to its register (now 1).

- **Time 3:** Thread 1 adds its my_result (2) to its register (now 2).
- **Time 4:** Thread 0 stores its register value (1) back to global_result.
- **Time 5:** Thread 1 stores its register value (2) back to global_result.
- **Result:** global_result ends up as 2, instead of the correct sum ($1 + 2 = 3$). Thread 0's contribution was overwritten.

Critical Section:

The code segment `global_result += my_result;` is a **critical section**. It's a block of code that updates a shared resource and must only be executed by one thread at a time to ensure correctness.

Solution: The critical Directive

OpenMP provides the `#pragma omp critical` directive to ensure **mutually exclusive access** to a structured block of code. Only one thread can execute the code block immediately following this directive at any given time.

```
#pragma omp critical
*global_result_p += my_result;
```

This ensures that the update to `global_result` is atomic and correct.

OpenMP Trapezoidal Rule Program (Program 5.2)

Let's examine the structure of the OpenMP program for the trapezoidal rule:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h> // For OpenMP functions and directives

// Function prototype for the parallel Trap function
void Trap(double a, double b, int n, double* global_result_p);

// Placeholder for the function f(x) to be integrated
double f(double x) {
    return x * x; // Example: f(x) = x^2
}
```

```
int main(int argc, char* argv[]) {
    double global_result = 0.0; // Shared variable for the total integral
    double a, b;                // Left and right endpoints of the integral
    int n;                      // Total number of trapezoids
    int thread_count;           // Number of threads to use

    // Get number of threads from command line argument
    thread_count = strtol(argv[1], NULL, 10);

    // Get input for a, b, and n from the user (executed by master thread)
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);

    // Start parallel region: thread_count threads will execute Trap function
#pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result); // Pass global_result by pointer

    // After parallel region, master thread continues and prints the 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

void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result; // h: trapezoid base length; my_result: thread's local integral
    double local_a, local_b; // Thread's local subinterval endpoints
    int i, local_n;          // Loop counter; local number of trapezoids
    int my_rank;             // Thread's ID (rank)
    int thread_count;        // Total number of threads in the team

    // Get thread-specific information
    my_rank = omp_get_thread_num();
    thread_count = omp_get_num_threads();
```

```
// Calculate common step size and local trapezoid count
h = (b - a) / n;
local_n = n / thread_count; // Assumes n is divisible by thread_count
```

```
// Determine the thread's local subinterval
local_a = a + my_rank * local_n * h;
local_b = local_a + local_n * h;
```

```
// Calculate the thread's local integral contribution
my_result = (f(local_a) + f(local_b)) / 2.0;
for (i = 1; i <= local_n - 1; i++) {
    x = local_a + i * h;
    my_result += f(x);
}
my_result = my_result * h;
```

```
// Add thread's local result to the shared global_result using a critical section
```

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

Code Walkthrough:

main function:

- Runs as a single thread initially.
- Reads `thread_count` from command line and `a`, `b`, `n` from user input.
- `#pragma omp parallel num_threads(thread_count)`: This directive creates a team of `thread_count` threads. Each thread in this team will execute the `Trap` function. The `global_result` variable is passed by pointer, making it a shared variable accessible to all threads.
- After all threads finish `Trap` (implicit barrier), the master thread continues to print the final `global_result`.

Trap function:

- Each thread within the parallel team executes this function.

- `my_rank = omp_get_thread_num();` and `thread_count = omp_get_num_threads();`: Each thread gets its unique ID and the total number of threads in the team.
- **Work Distribution:**
 - $h = (b-a)/n$; h is the same for all threads.
 - `local_n = n/thread_count;`: Each thread is assigned an equal number of trapezoids.
 - `local_a = a + my_rank * local_n * h;`: Calculates the starting point of the subinterval for the current thread based on its rank.
 - `local_b = local_a + local_n * h;`: Calculates the ending point of the subinterval.
- **Local Calculation:** The thread then performs the standard serial trapezoidal rule calculation over its `[local_a, local_b]` interval to get `my_result`.
- **Global Sum (Critical Section):**

```
#pragma omp critical
*global_result_p += my_result;
```

This ensures that only one thread at a time can update the shared `global_result_p`, preventing the race condition.

Important Details

Variable Naming:

The `local_` prefix (e.g., `local_a`, `local_b`, `local_n`) is used to emphasize that these variables hold values specific to each thread's portion of the work, distinct from the global `a`, `b`, `n`.

Divisibility Check:

The code assumes `n` is evenly divisible by `thread_count`. If not, `local_n` would be `n / thread_count` (integer division), leading to fewer trapezoids being computed in total $((n / \text{thread_count}) * \text{thread_count})$ than the requested `n`.

Proper error checking (e.g., `if (n % thread_count != 0) { ... exit(0); }`) is recommended in practice to handle this.

Interval Calculation:

The calculation of `local_a` and `local_b` correctly partitions the original interval `[a, b]` among the threads. For example, thread 0 gets `[a, a + local_n*h]`, thread 1 gets `[a + local_n*h, a + 2*local_n*h]`, and so on.

Scope of variables

In serial programming, the **scope** of a variable refers to the regions of the program text where that variable can be accessed or "seen."

- **Function-wide scope:** A variable declared inside a C function is only accessible within that function's body. It's typically allocated on the function's call stack.
- **File-wide scope (Global/Static):** A variable declared at the top of a .c file, outside any function, can be accessed by any function within that same file. It's typically allocated in static memory.

OpenMP Variable Scope:

In OpenMP, the **scope** of a variable in a parallel block refers to the **set of threads that can access that variable** within the parallel region.

Shared Scope:

A variable with **shared scope** can be accessed by **all threads** in the OpenMP team.

Default Behavior: Variables that are declared **before** a `#pragma omp parallel` directive (i.e., in the enclosing scope, like `main` or globally) typically have **shared scope** by default within the parallel region.

Example (Trapezoidal Rule):

In the main function of the trapezoidal rule program, `a`, `b`, `n`, `global_result`, and `thread_count` are declared before the `#pragma omp parallel` directive.

When `Trap` is called within the parallel region, these variables are implicitly shared.

- Each thread needs to access the global `a`, `b`, and `n` to determine its `local_a`, `local_b`, and `local_n`. This access happens through the function call's arguments, which effectively receive the shared values.
- `global_result` is a crucial example. It's declared in `main` and needs to be accessible to *all* threads to accumulate their `my_result` contributions.
- If `global_result` were private, each thread would have its own independent copy, and the final sum in `main` would be incorrect (likely just the initial value or the last thread's value). This is why the critical directive is needed for updates to `global_result` – because it's shared.

Private Scope:

A variable with **private scope** can only be accessed by a **single thread**. Each thread gets its own unique, independent copy of the variable.

Default Behavior: Variables declared **inside** a structured block that is executed in parallel (e.g., local variables within a function called from a parallel region, or variables declared directly within the parallel region's block) typically have **private scope**. These variables are allocated from each thread's private stack.

Example ("Hello, World"):

In the Hello function, `my_rank` and `thread_count` are declared locally. Each thread calling Hello gets its own private copies of these variables, which it initializes independently using `omp_get_thread_num()` and `omp_get_num_threads()`.

Example (Trapezoidal Rule's Trap function):

In the Trap function, variables like `h`, `x`, `my_result`, `local_a`, `local_b`, `i`, `local_n`, `my_rank` are all declared locally within the function. Therefore, each thread has its own private version of these variables, residing on its own stack. This is essential for independent computation (`my_result`).

Pointer to Shared Variable:

While `global_result_p` itself is a private pointer variable within the Trap function (each thread has its own `global_result_p` on its stack), what it *points to* (`*global_result_p`) is the `global_result` variable which has shared scope. This indirection allows threads to access and update the single shared instance.

The reduction clause

The **reduction clause** in OpenMP provides a powerful and convenient way to handle computations where multiple threads contribute to a single, combined result using an associative binary operator.

This is a common pattern in parallel programming, often referred to as a "reduction."

The Problem with Simple Critical Sections

Consider the attempt to use a simpler function prototype `double Local_trap(double a, double b, int n);` where each thread calculates and returns its `my_result`. A naive approach might be:

```
global_result = 0.0;
#pragma omp parallel num_threads(thread_count)
{
    // Problematic: forces serialization of Local_trap calls
    #pragma omp critical
    global_result += Local_trap(a, b, n); // Assumes a,b,n are correctly set for each thread
}
```

The problem here is that the entire `global_result += Local_trap(...)` expression, including the call to `Local_trap`, is inside the critical section.

This means **only one thread can execute Local_trap at a time**, effectively serializing the most computationally intensive part of the program.

This defeats the purpose of parallelization and can even make the parallel version slower than the serial one due to the overhead of mutex locking/unlocking.

A better approach using a private variable and a critical section would be:

```
global_result = 0.0;
#pragma omp parallel num_threads(thread_count)
{
    double my_result = 0.0; // Private to each thread
    // Each thread calculates its part independently (in parallel)
    my_result = Local_trap(a, b, n); // Assuming a,b,n are properly passed/calculated for the local range
    // (e.g., Local_trap needs to determine local_a, local_b, local_n based on thread ID)

    #pragma omp critical // Only this small part is serialized
    global_result += my_result;
}
```

This version works correctly and allows `Local_trap` to execute in parallel. Each thread computes its `my_result` privately and then, in a very small critical section, adds its `my_result` to the shared `global_result`.

Introducing the reduction Clause

OpenMP offers a cleaner, more concise, and often more optimized way to achieve the same result using the reduction clause.

What is a Reduction?

A reduction is a computation that applies an **associative binary operator** repeatedly to a sequence of operands to produce a single result.

Examples include summing numbers, finding the maximum/minimum, or multiplying values.

Syntax of the reduction Clause:

reduction(<operator>: <variable list>)

- <operator>: A supported reduction operator (e.g., +, *, -, &, |, ^, &&, ||, max, min).
- <variable list>: One or more variables (separated by commas) that are involved in the reduction.

How it works (for the trapezoidal rule example):

```
global_result = 0.0; // Initial value for the shared variable
#pragma omp parallel num_threads(thread_count) \
    reduction(+: global_result) // Note the backslash for line continuation
global_result += Local_trap(a, b, n); // This line is executed by each thread
```

Here's what the OpenMP runtime system conceptually does with the reduction clause:

1. **Private Copies:** For each variable specified in the reduction clause (global_result in this case), OpenMP automatically creates a **private copy** for each thread in the team.
2. **Initialization:** Each private copy is initialized to the **identity value** of the specified operator (e.g., 0 for +, 1 for *, etc.). (See Table 5.1 below).
3. **Local Computation:** Inside the parallel region, when a thread operates on global_result (e.g., global_result += Local_trap(...)), it is actually operating on its **private copy** of global_result. This means the calls to Local_trap happen in parallel, without contention.
4. **Combination (Implicit Synchronization):** At the end of the parallel region (at the implicit barrier), OpenMP automatically performs a **final combination** of all the private copies into the original shared global_result variable, using the specified reduction operator. This combination is typically done efficiently, often using optimized tree-based reductions or other methods that minimize serialization, making it generally more performant than a manual critical section for summation.

Equivalence to Manual Critical Section:

The code with the reduction clause:

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

is effectively equivalent to the more verbose manual approach:

```
global_result = 0.0;
#pragma omp parallel num_threads(thread_count)
{
    double my_result = 0.0; /* private */ // This is what OpenMP implicitly creates
    my_result += Local_trap(double a, double b, int n);
    #pragma omp critical // This is what OpenMP implicitly manages
    global_result += my_result;
}
```

Specifics of Operators and Floating Point Arithmetic:

- **Supported Operators:** +, *, -, & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), && (logical AND), || (logical OR), max (not explicitly listed in the text but commonly supported), min (not explicitly listed).
- **Subtraction (-):** While subtraction is not associative, OpenMP handles it by effectively summing the negated private results.

If result -= i is reduced, thread 0 might compute $-(1+2)=-3$ and thread 1 $-(3+4)=-7$. The final result will be $\text{global_result} + (-3) + (-7) = \text{global_result} - 10$, which is the correct behavior for repeated subtraction.

- **Floating Point Precision:** For float or double variables, the exact numerical result of a reduction might differ slightly from a serial calculation, or even between different numbers of threads.

This is because floating-point addition (and multiplication) is not strictly associative due to limited precision. For example, $(a+b)+c$ might not be exactly equal to $a+(b+c)$ for floating-point numbers.

This is a common characteristic of parallel floating-point computations, not specific to OpenMP.

Identity Values for Reduction Operators (Table 5.1):

When OpenMP creates private copies of reduction variables, they are initialized to an "identity value" (or "neutral element") for the specified operation.

This ensures that when the partial results are combined, the initial state of the private variable does not alter the final outcome.

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

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

Loop carried dependency

When discussing parallel programming, especially in the context of loops, a **loop-carried dependency** (also known as a **loop dependence** or **inter-iteration dependency**) refers to a situation where an iteration of a loop depends on the result of a previous iteration of the same loop.

In simpler terms:

- **Iteration i** needs something that was computed or changed by **Iteration j**, where j comes *before* i in the loop's execution sequence.

This dependency is crucial because it **prevents simple parallelization** of the loop. If iterations are executed concurrently without respecting the dependency, the results can be incorrect, leading to race conditions, data inconsistencies, or wrong final values.

Why is it a problem for parallelization?

Imagine you have a loop:

```
for (i = 0; i < N; i++) {
```

```
// some computation involving array[i] and array[i-1]
array[i] = array[i-1] + some_value;
}
```

If we try to run iterations $i=1$, $i=2$, $i=3$, etc., simultaneously:

- Iteration $i=1$ needs `array[0]`.
- Iteration $i=2$ needs `array[1]`.
- Iteration $i=3$ needs `array[2]`.

If Iteration $i=2$ starts before Iteration $i=1$ has finished computing and storing `array[1]`, then Iteration $i=2$ will use an old or incorrect value for `array[1]`, leading to a wrong result.

This is a **read-after-write (RAW)** dependency across loop iterations.

Types of Loop-Carried Dependencies:

Read-After-Write (RAW) / Flow Dependence:

- Iteration i reads a value that was written by Iteration j (where $j < i$).
- **Example:** `A[i] = B[i] + C[i-1]`; (Here, `C[i-1]` is written in the previous iteration and read in the current one, assuming `C` is modified by the loop). Or the example above: `array[i] = array[i-1] + some_value`;

Write-After-Read (WAR) / Anti-dependence:

- Iteration i writes to a location that was read by Iteration j (where $j < i$). This is less common as a direct *problem* for parallelization if variables are privatized, but it can still occur.
- **Example:** `A[i+1] = ...`; `B[i] = A[i]`; then later `A[i+1] = ...`; in the next iteration.

Write-After-Write (WAW) / Output Dependence:

- Iteration i writes to the same location that was written by Iteration j (where $j < i$).

Example:

```
for (...) {

    temp_result = ...;

    shared_variable = temp_result; // or some_array[k] = temp_result;
```

```
}
```

If `shared_variable` or `some_array[k]` is the same for multiple iterations, the final value will be determined by whichever iteration finishes last, leading to non-determinism.

How to Handle Loop-Carried Dependencies

Identify and Analyze:

The first step is to carefully analyze the loop and identify if any variables or memory locations are accessed in a way that creates dependencies between iterations.

Serialization:

If a strong, unavoidable loop-carried dependency exists (like the `array[i] = array[i-1] + ...` example), the loop **cannot be parallelized directly** without changing the algorithm or introducing explicit synchronization that effectively serializes parts of it. In such cases, trying to parallelize can lead to incorrect results.

Restructuring/Transformation:

- **Reduction:** If the dependency is a cumulative operation (like summation, product, min, max), it can often be handled using **reduction clauses** in OpenMP or similar constructs. As seen in the trapezoidal rule example, `global_result += my_result;` is a reduction. Each thread works on a private copy, and these copies are combined at the end.
- **Privatization:** If variables appear to cause a dependency but are actually used independently within each iteration, they can be declared as private to each thread.
- **Loop Transformations:** Techniques like loop fission (splitting a loop into multiple loops), loop fusion, or data restructuring might break or manage dependencies.
- **Algorithmic Changes:** Sometimes, a completely different algorithm is needed that is inherently parallel. For example, some scan/prefix sum algorithms have loop-carried dependencies but can be re-expressed in a parallel fashion.

Synchronization (if absolutely necessary):

For complex dependencies that cannot be easily transformed into reductions or private variables, explicit synchronization mechanisms like mutexes, atomic operations, or barriers might be used.

However, this often introduces significant overhead and can largely negate the benefits of parallelism, especially for fine-grained dependencies.

The critical section used in the initial trapezoidal rule program before the reduction clause was introduced is an example of such explicit synchronization.

Scheduling

Scheduling Loops in OpenMP

When parallelizing for loops in OpenMP using the `#pragma omp parallel for` directive, the way in which loop iterations are assigned to threads is called **scheduling**.

The default scheduling behavior (often a block partition) might not always be optimal, especially when the work associated with each iteration varies.

OpenMP provides the `schedule` clause to give programmers control over this distribution.

The Problem with Default (Block) Scheduling

By default, most OpenMP implementations use a **block partitioning**. If there are n iterations and `thread_count` threads, thread 0 gets iterations 0 to $n/\text{thread_count} - 1$, thread 1 gets the next block, and so on.

This can be inefficient if the computational cost of iterations is not uniform.

Example: `sum += f(i);` where $f(i)$ takes time proportional to i .

- Thread 0 would get iterations 0 to $n/\text{thread_count} - 1$, performing relatively cheap computations.
- Thread $\text{thread_count} - 1$ would get iterations $(\text{thread_count} - 1) * n/\text{thread_count}$ to n , performing the most expensive computations. This leads to **load imbalance**, where some threads finish early and others are stuck with a heavy workload, meaning the total execution time is dictated by the slowest thread.

The provided example with $f(i)$ calling `sin i` times demonstrates this:

- Serial ($n=10,000$, 1 thread): 3.67 seconds.
- 2 threads, **default (block) schedule**: 2.76 seconds (Speedup = 1.33, poor for 2 threads).
- 2 threads, **cyclic schedule**: 1.84 seconds (Speedup = 1.99, much better!). This clearly shows how a good schedule can significantly impact performance.

The schedule Clause

The schedule clause is used with `#pragma omp for` or `#pragma omp parallel for` directives to specify how iterations should be distributed. Its general form is:

`schedule([type][, chunksize])`

- **type:** Determines the scheduling algorithm.
- **chunksize (optional):** A positive integer that defines the size of blocks of iterations. Its interpretation depends on the type.

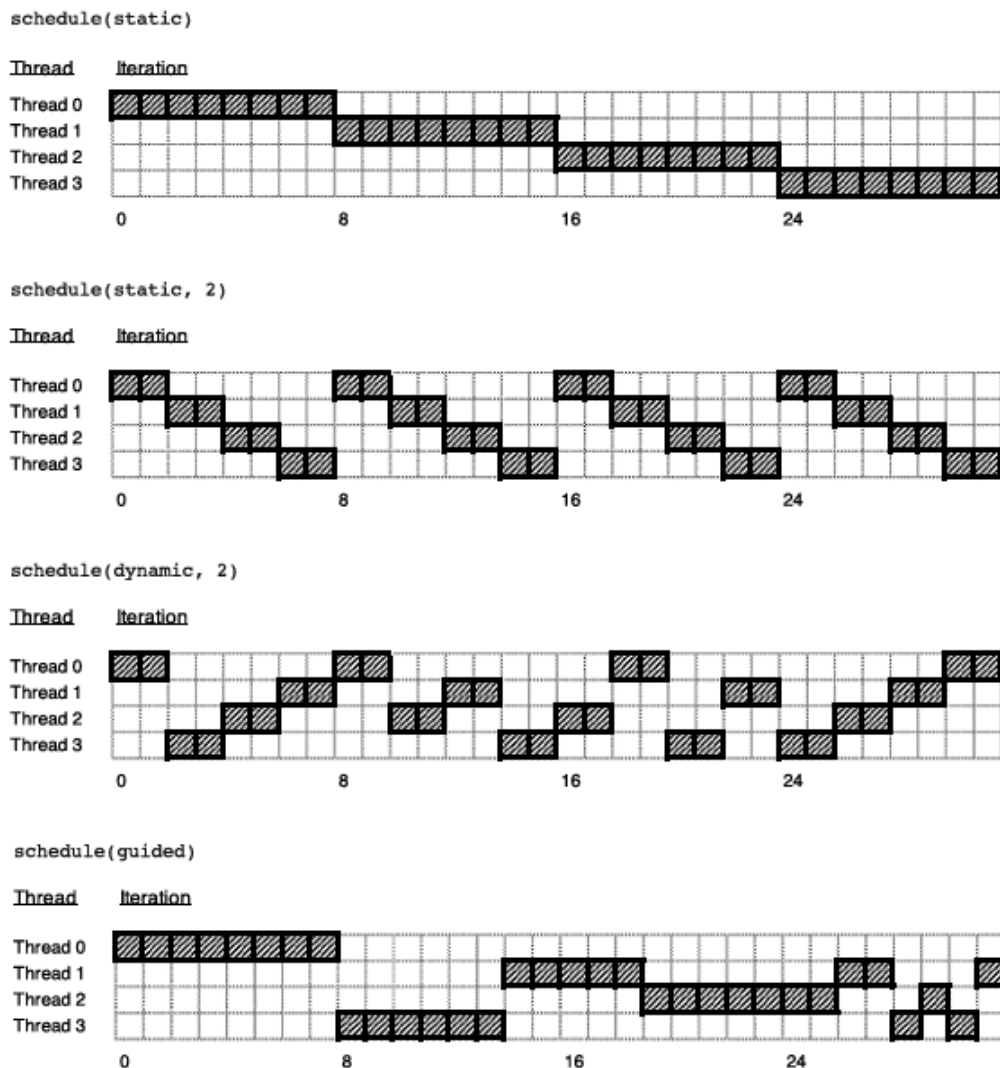


FIGURE 5.5

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.

The static Schedule Type

Behavior: Iterations are assigned to threads *before* the loop begins. Chunks of chunksize iterations are distributed in a **round-robin** fashion.

Default chunksize: If chunksize is omitted or set to the default, it's approximately $\text{total_iterations} / \text{thread_count}$, which results in the block partitioning.

Example with static:

- `schedule(static, 1)`: Cyclic assignment (as in the improved example).
 - Thread 0: Iterations 0, 3, 6, 9
 - Thread 1: Iterations 1, 4, 7, 10
 - Thread 2: Iterations 2, 5, 8, 11
- `schedule(static, 2)`: Chunks of 2, round-robin.
 - Thread 0: 0,1, 6,7
 - Thread 1: 2,3, 8,9
 - Thread 2: 4,5, 10,11

When to Use:

- Ideal when each loop iteration takes **roughly the same amount of time**.
- Offers **minimal overhead** because assignments are fixed at compile time (or early runtime).
- Can improve **memory locality** on NUMA systems if subsequent loops use the same schedule and data is reused by the same threads.

The dynamic and guided Schedule Types

These are **run-time scheduling** types, meaning iterations are assigned to threads *while the loop is executing*. Threads request new chunks of work as they complete their current ones.

This helps with load balancing when iteration costs are unpredictable.

dynamic:

- **Behavior:** Iterations are broken into chunks of chunksize consecutive iterations. Threads acquire chunks on a **first-come, first-served** basis.
- **Default chunksize:** If omitted, chunksize defaults to 1.

- **When to Use:** When loop iterations have **highly variable or unpredictable computational costs**.
- **Trade-off:** Introduces **overhead** due to dynamic assignment requests from the runtime system. A larger chunksize reduces this overhead but might worsen load imbalance for highly variable workloads.

guided:

- **Behavior:** Similar to dynamic, but the **size of the chunks decreases** over time. Initially, chunks are large, and they get progressively smaller until they reach a minimum chunksize (defaulting to 1, or the specified chunksize). The idea is to assign large chunks initially to reduce overhead, then smaller chunks later to fine-tune load balance.
- **When to Use:** When the computational cost of iterations **decreases (or increases) non-linearly** as the loop progresses, or when there's an unknown but varying workload per iteration. It tries to quickly dispatch the bulk of the work, then use smaller chunks to balance the tail end.
- **Example (Table 5.4):** If $n=10,000$ and $\text{thread_count}=2$, the first chunk might be ~ 5000 iterations, the next ~ 2500 , then ~ 1250 , and so on.

Table 5.4 Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

The runtime Schedule Type

- **Behavior:** The schedule type and chunksize are determined at **runtime** by inspecting the `OMP_SCHEDULE` environment variable.
- **How to Use:**
 1. In your C/C++ code, use `schedule(runtime)`:

```
#pragma omp parallel for reduction(+:sum) schedule(runtime)
for (i = 0; i <= n; i++)
    sum += f(i);
```

2. Before running your executable, set the OMP_SCHEDULE environment variable.

- **Bash/Unix:** export OMP_SCHEDULE="static,1" or export OMP_SCHEDULE="dynamic,10"
- The provided bash script shows how to automate testing different schedules and chunk sizes using this method.

When to Use: Extremely useful for **experimentation and tuning**. It allows you to try different scheduling strategies without recompiling the code.

Which Schedule to Choose?

Choosing the optimal schedule is often an **empirical process** involving experimentation.

Default is Best (No schedule clause):

If loop iterations have **roughly the same computational cost**, the default static schedule (with a large chunk size, typically `total_iterations / thread_count`) is usually the best choice due to its minimal overhead. If performance is satisfactory, don't change it.

Linearly Varying Costs (static,1 or small chunksize):

If the cost of iterations **decreases or increases linearly** as the loop progresses (like the `f(i)` example), a static schedule with a **small chunksize (e.g., static,1 for cyclic)** often yields the best performance by distributing the varied workload more evenly.

Unpredictable/Highly Variable Costs (dynamic or guided):

If the cost of each iteration **cannot be determined in advance** or varies significantly, dynamic or guided schedules are good candidates.

- Start with dynamic or guided without a specified chunksize (defaults to 1 for dynamic, adaptive for guided).
- Experiment with larger chunksize values to find a balance between load balancing and scheduling overhead.
- The runtime schedule is invaluable here for rapid testing.

No Universal Best Schedule:

The optimal schedule can depend on the specific algorithm, the number of threads, the total number of iterations, and the underlying hardware (e.g., cache sizes, NUMA architecture). Extensive experimentation might be necessary for fine-tuning.

Producers and consumers

The "producers and consumers" problem is a classic example in parallel programming that illustrates the need for explicit synchronization mechanisms beyond simple loop parallelization.

It's not amenable to a straightforward `#pragma omp parallel` for because the interactions between threads are more complex than just independent loop iterations or simple reductions.

Queues

A **queue** is a First-In, First-Out (FIFO) abstract data type.

- **Enqueue:** Adding an element to the "rear" or "tail" of the queue.
- **Dequeue:** Removing an element from the "front" or "head" of the queue.

Queues are crucial in multithreaded applications for managing shared resources and inter-thread communication.

Producers and Consumers Model

In this model:

- **Producer threads** generate data or tasks.
- **Consumer threads** process that data or tasks.
- A **queue** acts as a buffer between producers and consumers. Producers enqueue items, and consumers dequeue them.

Example:

- **Producers:** Threads requesting stock prices from a server.
- **Consumers:** Threads fetching/generating the actual stock price data.

Message-Passing Implementation (Example)

The text describes implementing a simple message-passing system where threads alternate between sending and receiving messages using queues. Each thread has its own message queue.

Pseudocode for each thread:

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {  
    Send_msg();  
    Try_receive();  
}  
while (!Done()) // Loop until all messages are processed and all threads are done  
    Try_receive();
```

Sending Messages (Critical Section)

Accessing a message queue to **enqueue** a message is a **critical section**.

Reason: If multiple threads try to enqueue simultaneously into the *same* queue, operations like updating the "rear" pointer or managing the list structure can conflict, leading to lost messages or a corrupted queue.

Pseudocode for Send_msg():

```
mesg = random();  
dest = random() % thread_count; // Destination thread's queue  
#pragma omp critical // Protects the Enqueue operation  
Enqueue(queue, dest, my_rank, mesg);
```

The critical directive ensures that only one thread can execute Enqueue at a time. This, however, poses a performance issue: if any thread tries to send a message, it will block if *any other thread* (even to a different destination queue) is currently in a critical section. This leads to global serialization.

Receiving Messages (Lesser Synchronization Needs)

Dequeuing is different:

- Only the **owner** (destination thread) of a queue dequeues from it.
- If the queue has at least two messages, a Dequeue operation by the owner typically doesn't conflict with Enqueue operations by other threads (since Enqueue modifies the tail, and Dequeue modifies the head).

Pseudocode for Try_receive():

```
queue_size = enqueued - dequeued; // Assuming enqueued and dequeued counters exist
if (queue_size == 0) return; // No messages, return immediately
else if (queue_size == 1)
    #pragma omp critical // Protects if only one message is left (dequeue might conflict with enqueue
into an empty queue)
    Dequeue(queue, &src, &mesg);
else
    Dequeue(queue, &src, &mesg); // No critical section if queue_size > 1
Print_message(src, mesg);
```

Note on queue_size calculation:

The calculation `queue_size = enqueued - dequeued` might read `enqueued` while another thread is updating it. This might lead to `queue_size` being momentarily incorrect (e.g., 0 instead of 1, or 1 instead of 2). However, this is deemed acceptable here as it only causes a slight delay or an unnecessary critical section entry, not correctness issues.

Termination Detection

A crucial aspect is determining when all threads are truly done, including processing all messages.

A simple `Done()` based only on `queue_size == 0` is problematic: a thread might check, find the queue empty, terminate, and then another thread sends it a message that will never be received.

Solution: Introduce a shared counter `done_sending`. Each thread increments this counter *after* completing its sending loop.

```
// In each thread after its for loop
done_sending++; // This needs protection, as seen below

// Done function
if (queue_size == 0 && done_sending == thread_count)
    return TRUE;
else
    return FALSE;
```

Startup (Implicit vs. Explicit Barriers)

- The master thread allocates an array of queue pointers (shared).
- Each thread then allocates its individual queue struct.
- **Problem:** Threads might try to send messages to queues that haven't been allocated yet by their respective owner threads.
- **Solution:** Use an **explicit barrier** to ensure all queues are allocated before any messages are sent.

```
#pragma omp barrier
```

When a thread encounters this, it pauses until *all* other threads in the team also reach this point. Only then do all threads proceed.

The atomic Directive

The atomic directive provides a potentially higher-performance way to protect simple critical sections.

Syntax: `#pragma omp atomic` followed by a single assignment statement of a specific form.

Forms:

- `x = expr_binop x; (e.g., x += y;)`
- `x = x binop expr; (e.g., x = x + y;)`
- `x++; ++x; x--; --x;`

Limitation: It can only protect single, specific assignment statements. The `expr` part must not reference `x`.

Mechanism: It leverages special "load-modify-store" hardware instructions, which are often much faster than general-purpose mutexes used by critical.

Example for done_sending:

```
#pragma omp atomic
done_sending++;
```

Caveat: Only the operation on `x` is guaranteed atomic. For `x += y++;`, `y++` might not be atomic.

Critical Sections and Locks (Addressing Global Serialization)

The default critical directive (without a name) enforces **global mutual exclusion** across *all* unnamed critical sections in the program. This is detrimental for the message-passing example because:

- `done_sending++` is a critical section.
- `Enqueue` is a critical section.
- `Dequeue` (sometimes) is a critical section.
- If one thread is incrementing `done_sending`, *no other thread* can be enqueueing or dequeuing, even if it's for a different queue. This is unnecessarily restrictive.

Named Critical Sections: `#pragma omp critical (name)` allows different critical sections to execute concurrently if they have different names.

- **Example:** `#pragma omp critical (queue_access)`
- **Limitation:** Names are compile-time constants. We need a different critical section *per queue*, and the queue to access is determined at runtime (`dest`). Thus, named critical sections aren't sufficient here.

Locks (`omp_lock_t`): Locks provide the necessary flexibility to enforce mutual exclusion **on a per-data-structure basis**, not just per code block.

- **Concept:** A lock is a data structure (`omp_lock_t`) shared among threads.
 - `omp_init_lock()`: Initializes the lock (unlocked state).
 - `omp_set_lock()`: A thread tries to acquire the lock. If unavailable, it blocks.
 - `omp_unset_lock()`: A thread releases the lock.
 - `omp_destroy_lock()`: Uninitializes the lock.
- **Advantage:** Allows for **fine-grained locking**. Threads wanting to access different queues can proceed concurrently, as long as they acquire the lock for their *specific* queue.

Using Locks in the Message-Passing Program

To avoid global serialization, each queue struct should contain its own `omp_lock_t` member.

- **Replacing critical for Enqueue:**

```
// q_p = msg_queues[dest]
omp_set_lock(&q_p->lock); // Acquire lock for the specific destination queue
Enqueue(q_p, my_rank, msg); // Now safe to modify q_p
```

```
omp_unset_lock(&q_p->lock); // Release lock
```

- **Replacing critical for Dequeue:**

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

This ensures that only threads accessing the *same* queue are serialized, maximizing concurrency.

Critical Directives, Atomic Directives, or Locks? (Choosing the Right Tool)

- **atomic:**
 - **Pros:** Potentially fastest due to hardware support.
 - **Cons:** Limited to specific single assignment statements. Unnamed atomic directives *might* enforce global exclusion on some implementations (though not mandated by standard if variables are different), so use with caution if multiple distinct atomic operations are present.
- **critical (unnamed):**
 - **Pros:** Very easy to use.
 - **Cons:** Enforces global mutual exclusion across *all* unnamed critical directives, which can severely limit concurrency if not all critical sections truly conflict.
- **critical (name):**
 - **Pros:** Allows distinct critical sections to execute concurrently.
 - **Cons:** Names are compile-time. Not suitable when the "resource" to be protected depends on a runtime value (like a queue index).
- **locks (omp_lock_t):**
 - **Pros:** Most flexible. Allows fine-grained, data-structure-specific mutual exclusion. Essential when you need to protect *instances* of a data structure rather than just code blocks.
 - **Cons:** More verbose (requires explicit init, set, unset, destroy calls).

General Guidelines:

1. **atomic first:** If your critical section fits the exact form of atomic, use it.
2. **critical next:** If atomic isn't suitable, but you need simple mutual exclusion for a *code block* (and don't have many distinct, unrelated critical sections that would suffer from global serialization), critical is simple. Use named critical sections if different code blocks can indeed run concurrently.

3. **locks for data structures:** If mutual exclusion is needed for specific *instances* of a shared data structure (like per-queue locking), locks are the correct choice.

Some Caveats (Dangers of Mutual Exclusion)

Using synchronization primitives incorrectly can lead to serious bugs:

1. **Mixing Mutual Exclusion Types:** Don't mix different types of protection for the *same* critical section. For example, if part of a variable's modification is atomic and another part is critical, there's no guarantee the critical section will exclude the atomic one, leading to race conditions. Choose one consistent mechanism.
2. **No Guarantee of Fairness (Starvation):** Mutual exclusion constructs (especially general ones like mutexes/locks) don't guarantee fairness. A thread can potentially be blocked forever waiting for a lock if other threads repeatedly acquire and release it before the blocked thread gets a chance. This is called **starvation**.
3. **Deadlock:** This is a common and severe problem when using locks. It occurs when two or more threads are permanently blocked, each waiting for a resource that another blocked thread holds.
 - **Nested Locks:** Trying to acquire a lock *within* another lock can lead to deadlock if the inner lock is already held by the same thread or another thread that needs the outer lock.
 - **Circular Waiting:** If threads attempt to acquire multiple locks in different orders (e.g., Thread U acquires Lock A then tries Lock B; Thread V acquires Lock B then tries Lock A), they can get into a state where U holds A, V holds B, and both are waiting for the other's lock.
 - **Prevention:** The primary rule to prevent deadlock with multiple locks is to ensure that **all threads acquire locks in the same, consistent order**.

Caches, cache coherence and false sharing in openmp

Caches and Locality

Processor-Memory Speed Gap: Processors operate significantly faster than main memory can supply data.

Caches: To bridge this gap, fast, small memory blocks called caches are integrated into processors.

Locality Principles: Caches work by exploiting:

- **Temporal Locality:** If a memory location is accessed, it's likely to be accessed again soon.
- **Spatial Locality:** If a memory location is accessed, nearby locations are likely to be accessed soon.

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	???

Cache Lines (Cache Blocks): When a processor needs data from main memory, it doesn't just retrieve that single data item. Instead, an entire block of memory containing the requested item is transferred to the cache. This block is called a cache line (typically 64 bytes or more).

Cache Coherence Problem

In shared-memory systems with multiple cores, each having its own cache, the cache coherence problem arises when multiple copies of the same data exist in different caches and main memory.

Scenario:

1. x = 5 (in main memory).
2. Thread 0 reads x into its cache (cache 0).
3. Thread 1 reads x into its cache (cache 1).
4. Thread 0 executes x++ (changes x in cache 0 to 6).
5. Thread 1 then reads x to set my_z = x.

The Problem: Without cache coherence, thread 1's cache copy of x would still be 5, leading to my_z being 5, which is incorrect.

Solution (Cache Coherence Protocols): Most modern systems implement hardware-level cache coherence protocols (e.g., MESI protocol).

- When Thread 0 modifies x in its cache, the cache coherence protocol detects this write to a shared variable.
- It then invalidates (marks as stale) all other cached copies of the cache line containing x (e.g., in Thread 1's cache).
- Before Thread 1 reads x, its cache controller sees that its copy is invalid. It then fetches the most up-to-date value of x (either from main memory or directly from Thread 0's cache if it's the "owner" of the modified line).

Impact: Cache coherence ensures correctness (all threads see a consistent view of shared memory) but can impact performance due to cache line invalidations and subsequent reloads from main memory.

Matrix-Vector Multiplication Example

The text uses matrix-vector multiplication ($y=Ax$) to illustrate cache performance and false sharing.

Serial Code:

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

Parallel OpenMP Code:

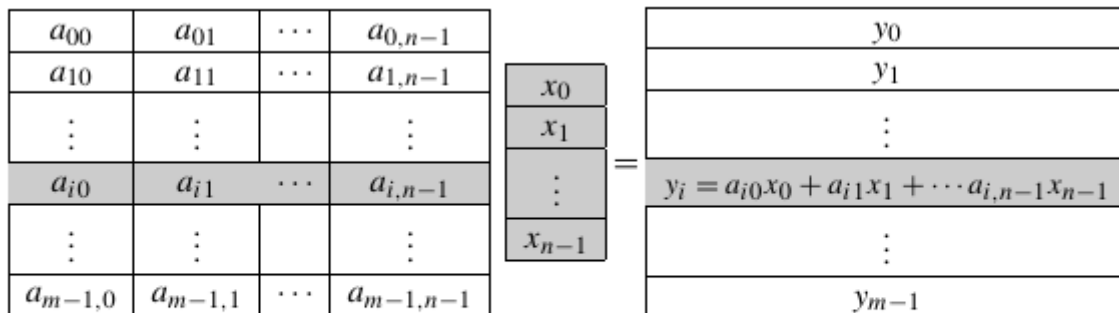
```
#pragma omp parallel for num_threads(thread_count) \  
    default(none) private(i, j) shared(A, x, y, m, n)  
for (i = 0; i < m; i++) {  
    y[i] = 0.0;  
    for (j = 0; j < n; j++)  
        y[i] += A[i][j] * x[j];  
}
```

The outer loop is parallelized because each iteration i updates a unique element $y[i]$, eliminating loop-carried dependencies on y . A and x are read-only.

Performance Observations (Table 5.6): The total number of floating-point operations is constant (64,000,000) across different matrix dimensions, but run-times vary significantly for the serial program:

- 8000x8000: Baseline performance.
- 8,000,000x8: Slower (22% more time). Many **write-misses** when initializing $y[i] = 0.0$ (Line 4), as y is a very long vector, leading to frequent main memory writes.
- 8x8,000,000: Slower (26% more time). Many **read-misses** when accessing $x[j]$ (Line 6), as x is a very long vector, leading to frequent main memory reads.

This highlights that **memory access patterns (cache performance)** are as critical as arithmetic operation count for overall performance.

**FIGURE 5.6**

Matrix-vector multiplication.

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

Threads	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
	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

False Sharing

Now, let's focus on the parallel performance, especially the 8x8,000,000 input where parallel efficiency is significantly lower.

Scenario (8x8,000,000 input, 4 threads, 64-byte cache line = 8 doubles):

- y has 8 components (y[0] to y[7]).
- Each thread is assigned a small number of y components (e.g., Thread 0 computes y[0], y[1]).
- **Crucially, if all elements of y reside within a single cache line**, then every write to *any* element of y by *any* thread will invalidate the *entire cache line* in other processors' caches.
- For example, Thread 0 updates y[0]. If Thread 2's cache also holds y[0] (as part of the same cache line), that cache line becomes invalid for Thread 2.
- When Thread 2 subsequently accesses y[2] (which is also in the same invalidated cache line), it must reload the entire cache line from main memory, even though Thread 2 never touched y[0].

Definition of False Sharing:

- Occurs when two or more threads access **different variables** that happen to reside in the **same cache line**.
- At least one of the threads **updates its variable**.
- Even though there's no true data sharing (threads aren't accessing the same variable), the cache coherence protocol invalidates the entire shared cache line.
- This forces other threads to reload the entire line from main memory, incurring significant performance penalties, *as if* they were truly sharing the same variable. Hence, "false sharing."

Why it's a problem for 8x8,000,000 but not others:

- In the 8x8,000,000 case, y is a very short vector (8 doubles), easily fitting into one 64-byte cache line. When threads update different $y[i]$ elements, they are all contending for the same cache line. Every write causes an invalidation for other cores that might have a copy.
- In 8,000,000x8 and 8000x8000 cases, y is a much longer vector. Each thread is assigned a large, contiguous block of y elements (e.g., 2000 or 2,000,000 elements).
- While there might be false sharing at the *boundaries* between blocks assigned to different threads (where their assigned elements fall into the same cache line), these occurrences are rare relative to the total number of accesses.
- Most of the time, threads are accessing different cache lines entirely, or their access to the shared boundary lines is sufficiently separated in time that invalidations don't cause significant contention.

Avoiding False Sharing:

1. **Padding:** Insert dummy variables (padding) between array elements or data structures to ensure that variables accessed by different threads fall into different cache lines. This increases memory usage but can dramatically improve performance.
2. **Thread-Private Storage with Final Update:** Each thread can compute its partial results into a *private* array (allocated on its own stack or heap). After all threads complete their computations, they then atomically update the *shared* global array. This completely eliminates false sharing during the main computation phase. (Similar to how reduction works, but applied to an array).

False sharing is a subtle but critical performance trap in shared-memory programming. It makes correct parallelization (threads working on different parts of the data) perform poorly due to the underlying hardware's cache coherence mechanisms operating at the granularity of cache lines. Understanding cache line size and data layout is essential for optimizing parallel code.

Tasking

Tasking in OpenMP

The OpenMP constructs discussed so far (like parallel for) are excellent for parallelizing loops with a fixed and predetermined number of iterations.

However, many parallel problems don't fit this model. This includes:

- **Dynamic problems:** Where the workload size or arrival rate is unpredictable (e.g., a web server handling incoming requests).
- **Recursive algorithms:** Where the number and structure of sub-problems are defined at runtime (e.g., tree traversals, quicksort, Fibonacci).
- **Producer-Consumer patterns:** Where tasks are generated and consumed asynchronously.
- **Loops with unbounded iterations:** while or do-while loops, or for loops where the upper bound is not fixed.

To address these challenges, OpenMP 3.0 introduced **Tasking functionality**.

The task Directive

The core of OpenMP tasking is the `#pragma omp task` directive:

```
#pragma omp task
{
    // Block of code that defines a task
}
```

Task Generation:

When a thread encounters a task directive, a new independent unit of computation (a "task") is generated by the OpenMP runtime.

Scheduling:

This task is then placed into a pool of tasks to be scheduled for execution by any available thread in the team. It's crucial to understand that the task is **not necessarily executed immediately** by the thread that created it. It might be executed by the same thread, or by a different idle thread.

Parallel Region Requirement:

Tasks *must* be launched from within an OpenMP parallel region.

single Directive for Task Launchers:

Often, you only want one thread to be responsible for *generating* tasks, especially if the task generation logic is inherently serial. The `#pragma omp single` directive can be used for this:

```
#pragma omp parallel // Creates a team of threads
#pragma omp single // Only one thread executes the following block
{
    // This is where tasks are typically generated
    #pragma omp task
    { /* ... task code ... */ }

    #pragma omp task
    { /* ... another task code ... */ }
}
```

If `single` is omitted, every thread in the team will execute the task-generating code, potentially creating `thread_count` identical tasks for each logical task you intended.

Example: Parallel Fibonacci Calculation with Tasks

The Fibonacci sequence ($F_n = F_{n-1} + F_{n-2}$) is a classic recursive problem demonstrating tasking. A naive recursive implementation has a natural tree structure of computations.

Serial Recursive fib function:

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

    if (n <= 1) {
        fibs[n] = n; // fibs is a global array
        return n;
    }
```

```
i = fib(n - 1); // Recursive call 1
j = fib(n - 2); // Recursive call 2
fibs[n] = i + j;
return fibs[n];
}
```

Attempting to parallelize fib with tasks:

1. Initial Attempt (Incorrect due to Data Scope): Adding #pragma omp task before each recursive call:

```
// ... inside fib(n)
#pragma omp task
i = fib(n - 1); // Task 1 created
#pragma omp task
j = fib(n - 2); // Task 2 created
// ...
fibs[n] = i + j; // Problem: i and j are local to the *parent* task's stack
```

- **Problem:** The default data scope for variables in tasks is **private**. This means *i* and *j* inside the *fib* function that *creates* the tasks are distinct from the *i* and *j* used by the *tasks themselves* when they execute *fib(n-1)* and *fib(n-2)*.
- The *fib(n-1)* and *fib(n-2)* calls return their results into the private *i* and *j* variables *within the context of their respective tasks*. However, the parent *fib* call (the one that *created* these tasks) still has its own *i* and *j* variables, which remain at their initial values (e.g., 0). Thus, *fibs[n]* ends up being 0 + 0.

2. Addressing Data Scope (shared clause):

To make the *i* and *j* variables of the *parent* task accessible and writable by the *child tasks* that compute *fib(n-1)* and *fib(n-2)*, we must declare them as shared within the task directive:

```
// Program 5.6
int fib(int n) {
    int i = 0; // Local variables for current fib call
    int j = 0;

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

```
    return n;
}

#pragma omp task shared(i) // i is shared with this task's scope
i = fib(n - 1);

#pragma omp task shared(j) // j is shared with this task's scope
j = fib(n - 2);

// Problem: The parent task might continue and use stale i/j values
// fibs[n] = i + j;
// return fibs[n];
}
```

New Problem:

Even with `shared(i)` and `shared(j)`, the order of execution is not guaranteed. The thread executing the `fib(n)` call might continue and try to use `i` and `j` *before* the tasks computing `fib(n-1)` and `fib(n-2)` have completed and updated them. This results in unpredictable and often incorrect results.

Ensuring Completion (taskwait directive):

To guarantee that the parent task waits for its child tasks to complete before proceeding, OpenMP provides the `#pragma omp taskwait` directive.

This acts as a barrier for tasks: the thread encountering `taskwait` will pause until all tasks *generated by the current task* (or by tasks generated in the same task region) have completed.

```
// Program 5.6 (Corrected)
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); // This call generates a new task

#pragma omp task shared(j)
j = fib(n - 2); // This call generates another new task

#pragma omp taskwait // Wait for the two child tasks to complete
fibs[n] = i + j; // Now i and j will have the correct results
return fibs[n];
}
```

With taskwait, the parallel Fibonacci program now produces correct results.

Performance Considerations and Optimizations

While tasking solves correctness issues for recursive algorithms, it introduces overhead:

Task Creation Overhead:

Each task directive involves overhead for creating the task's data environment, scheduling it, etc. For fine-grained tasks (like fib(2), fib(3)), this overhead can easily outweigh the benefits of parallelism, leading to significant slowdowns compared to the serial version.

Optimizations:

if Clause for Task Creation Threshold: The if clause on the task directive allows conditional task creation. Tasks are only generated if the if condition evaluates to true. Below a certain problem size, it's often more efficient to execute the work serially.

```
#pragma omp task shared(i) if(n > 20) // Create task only if n is large enough
i = fib(n - 1);
```

This ensures that very small, cheap fib calls are executed serially by the thread that encounters them, avoiding tasking overhead. The threshold (e.g., 20) is typically determined through experimentation.

Eliminating Redundant Tasks (Parent Task Doing Work):

In the Fibonacci example, the parent task creates two child tasks and then just waits. This means the parent task is mostly idle.

A common optimization is to have the parent task perform one of the recursive calls directly, without creating a new task for it:

```
// Optimized fib function (parent does one recursive call)
int fib(int n) {
    int i, j;

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

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

    j = fib(n - 2); // Current thread directly executes fib(n-2) without creating a task

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

This reduces the number of tasks created, hence reducing overhead. On a 64-core system with $n=35$, these two optimizations (if clause + parent doing work) can halve execution time.

Thread safety

What is Thread-Safety?

A block of code or a function is **thread-safe** if it can be simultaneously executed by multiple threads without causing race conditions, data corruption, or incorrect results.

In other words, calling the function from multiple threads at the same time will always produce the correct and expected behavior, as if the calls were executed sequentially.

The strtok Example: A Case Study in Non-Thread-Safety

The text provides an excellent example using the strtok function from string.h.

How strtok works (and why it's problematic):

First Call: strtok(string, separators)

- It takes the input string and the separators.
- It finds the first token, null-terminates it, and returns a pointer to it.
- **Crucially**, it stores an internal, **static** pointer to where it left off in the string. This static pointer persists across calls.

Subsequent Calls: strtok(NULL, separators)

- It uses the previously cached static pointer to continue tokenizing from the last position in the *original* string.

The Problem in a Multithreaded Context (Program 5.7):

Shared Internal State:

The core issue is that strtok uses an **internal static variable** to store its state (the pointer to the current position in the string).

This static variable is global (or file-scope static), meaning it's **shared** among all threads.

Race Condition:

When Thread 0 calls strtok(lines[0], ...) it sets this internal static pointer to lines[0]. If Thread 1 then calls strtok(lines[1], ...) *before* Thread 0 completes its tokenization of lines[0], Thread 1 **overwrites** this shared static pointer with lines[1].

Incorrect Results:

- Thread 0 then makes a subsequent call strtok(NULL, ...) expecting to continue from lines[0], but now strtok is looking at lines[1] (due to Thread 1's overwrite).

- This leads to corrupted output, as seen in the example where Thread 0 suddenly finds a token ("days") that belongs to Thread 1's line.

Conclusion: strtok is **not thread-safe** because it relies on shared, mutable internal state without any synchronization mechanisms.

Other Non-Thread-Safe C Library Functions

The text notes that it's "not uncommon" for C library functions to be non-thread-safe. Other common examples include:

- **rand() from stdlib.h:** Often uses an internal seed that gets updated. Multiple threads calling rand() simultaneously can lead to incorrect or non-random sequences.
- **localtime() from time.h:** May use a static buffer to store the struct tm return value, which can be overwritten by concurrent calls.

Solving Thread-Safety: Re-entrant Functions (_r suffix)

For common non-thread-safe functions, the C standard or specific library implementations often provide **re-entrant** (and thus thread-safe) versions.

These versions typically take an extra argument that allows the caller to provide the "state" explicitly, rather than relying on internal static storage.

strtok_r: The thread-safe alternative to strtok.

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

- **saveptr_p:** This is the crucial third argument. It's a pointer to a char* variable that *your thread* owns. strtok_r will use this char* variable (pointed to by saveptr_p) to store its internal state for *this specific tokenization operation*.
- Since each thread will declare its own saveptr variable (which is a private variable on the thread's stack), there's no shared state, and thus no race condition.

Corrected Tokenization in Program 5.7 (Conceptual change):

```
// In Tokenize function
```

```
char *my_token;
```

```
char *saveptr; // This variable is private to each thread due to OpenMP private clause
```

```
#pragma omp parallel for schedule(static, 1) // my_token and saveptr are private
for (i = 0; i < line_count; i++) {
    // ...
    my_token = strtok_r(lines[i], "\t\n", &saveptr); // First call with string
    while (my_token != NULL) {
        // ...
        my_token = strtok_r(NULL, "\t\n", &saveptr); // Subsequent calls with NULL
        // ...
    }
}
```

Key Takeaways on Thread-Safety:

Shared Mutable State is the Enemy of Thread-Safety:

Any function or code block that modifies shared data without proper synchronization is inherently not thread-safe. This includes global variables, static variables, or parameters passed by reference that are shared.

Identify and Replace:

When parallelizing code, be vigilant about library functions. Consult documentation or assume functions using internal state might not be thread-safe. Look for `_r` versions or other thread-safe alternatives.

Design for Thread-Safety: When writing your own functions that might be called concurrently:

- **Minimize Shared State:** Design functions to operate primarily on local, private data whenever possible.
- **Encapsulate State:** If shared state is necessary, encapsulate it within a data structure and protect access to that data structure with explicit synchronization primitives (like mutexes/locks, atomic operations, or OpenMP critical sections/atomics).
- **Immutable Data:** Functions that only read immutable (unchanging) shared data are generally thread-safe.

Understanding and addressing thread-safety is paramount in concurrent programming to ensure correctness and avoid subtle, hard-to-debug errors.