Parallel Concepts

Concurrency

- When multiple operations are making progress within the same time period.
- Usually on the same core/thread.

Parallelism

- When multiple operations are making progress at the same time.
- Usually requires multiple threads/cores.

Process

An instance of the computer that is being executed. These are its components:

- Executable machine language program.
- Block of memory.
- Descriptor of the OS resources allocated to it.
- Security info.
- Information about the state of the process.

Threading

- Threads are contained within processes.
- They allow programmers to divide their programs into independent tasks.
- A stream of instructions that can be scheduled to run independently from its main program.
- The hope is that when one thread blocks because it is waiting for resources, the other can run.

Processes vs Threads

- Threads exist within a process; they're like the children of the process.
- A process has at least one thread.
- If a process has more than one thread, it is multithreaded.
- Starting a thread within a process is known as **forking**.
- Terminating a thread is known as **joining**.
- Both threads and processes are units of execution or **tasks**.
- **Processes do not share memory** (each gets its own block of memory from the system).
- **Threads within a process share memory** (since they are children of the process, they have access to its resources).
- Data stored in a process's memory can be **shared or private**:
    - If **private**, only the thread that owns it can use it.

How is Data Shared?

Shared Memory

- Allows processors to have access to a global address space.

- Multiple processes can operate independently but share the same memory resources.
- Changes in a memory location affected by one task are visible to others.

Uniform Memory Access (UMA)

- The time to access all memory locations is the same for all cores.

Non-Uniform Memory Access (NUMA)

- A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

Task Scheduling

- A **scheduler** is a program that uses a **scheduling policy** to decide which process should run next.
- It uses a **selection function** to make the decision.
- The selection function considers:
  - Resources the process requires.
  - Time the process has been waiting.
  - The process's priority.
- Scheduling policy should try to optimize:
  - **Responsiveness** of interactive processes.
  - **Turnaround time** (the time the user waits for the process to finish).
  - **Resource utilization**.
  - **Fairness** (ensuring each process gets a chance to run).

Some Scheduling Policies

Non-Preemptive Policies

- Each task runs to completion before the next one can run.
- **First In First Out (FIFO)**.
- **Shortest-Job-First (SJF)**.

Preemptive Policies

- **Round-Robin**: Each task is assigned a fixed time before it is required to give way to the next task and move back to the queue.
- **Earliest-Deadline-First**: The process with the closest deadline is picked next.
- **Shortest Remaining Time First**: The process with the shortest remaining time is picked first.

Key Terms

- **Shared resource**: A resource available to all processes in the concurrent program.
- **Critical section**: Sections of code within a process that require access to shared resources. Cannot be executed while another process is in a corresponding section of code.
- **Mutual exclusion**: Requirement that when one process is in a critical section accessing a shared resource, no other process may be in a critical section accessing any of those shared resources.

- **Condition synchronization**: A mechanism ensuring that a process does not proceed until a certain condition is satisfied.
- **Deadlock**: A situation where two or more processes are unable to proceed because each is waiting for another process to act.
- **Livelock**: A situation where two or more processes continuously change their state in response to changes in other processes without making any progress.
- **Race Condition**: A situation where multiple tasks read/write a shared data item, and the result depends on the relative timing of their execution.
- **Starvation**: A situation where a runnable process is overlooked indefinitely by the scheduler.

Dead or Alive(lock)

Concurrent programs must satisfy two properties:

1. **Safety**: The program doesn't enter a bad state.
2. **Liveness**: The program must progress.

Two problems that can occur:

- **Deadlock**: A process is waiting for a shared resource that will never be available (e.g., another process is waiting for this process to act).
- **Livelock**: Multiple processes continuously change state in response to each other without making progress.

Conditions for Deadlock

For deadlock to occur, **four conditions must hold**:

1. **Mutual Exclusion**: The program involves a shared resource protected by mutual exclusion.
2. **Hold While Waiting**: A process can hold a resource while waiting for others.
3. **No Preemption**: The OS cannot force a process to deallocate a resource it holds.
4. **Circular Wait**: P1 is waiting for a resource held by P2, and P2 is waiting for a resource held by P1.

Preventing Deadlock

To prevent deadlock, **prevent at least one of the four conditions** from occurring.

POSIX Threads

A **POSIX thread** is a thread associated with a process's shared resources. Each thread has its own:

- **Stack**
- **Program counter**
- **Registers**
- **Thread ID**

Races

A **race condition** occurs when the **parent process exits before its child threads complete**. This does not allow enough time for child threads to finish execution.

Fixes for Race Conditions

- Best fix for race conditions, use mutual exclusions and join the threads

```
pthread_mutex_t lock;

void* say_something(void *ptr) {
    pthread_mutex_lock(&lock); //this now becomes critical section! it
uses mutual exclusion
    printf("%s ", (char*)ptr);
    pthread_mutex_unlock(&lock); //end the critical condition
    pthread_exit(0);
}

int main() {
    pthread_t thread_1, thread_2;
    char *msg1 = "Hello ";
    char *msg2 = "World!";
    //  create the lock -> error checking?
    pthread_mutex_init(&lock, NULL);
    pthread_create( &thread_1, NULL, say_something, msg1);
    pthread_create( &thread_2, NULL, say_something, msg2);

    // the main thread has to wait for the other threads to terminate
before it can terminate
    pthread_join(thread_1, NULL);
    pthread_join(thread_2, NULL);

    printf("Done!");
    fflush(stdout);
    pthread_mutex_destroy(&lock);
    exit(0);
}
```

This is conditional synchronization

```
void* say_something(void *ptr) {
    pthread_mutex_lock(&lock);//this now becomes critical section!
    //check on some condition - if it is hello, wait for world....

    if (strcmp(”World!",(char*)ptr) == 0) {
        printf("Waiting on condition variable cond1\n");
        if (done == 0) //only wait in the event that you need to…
            pthread_cond_wait(&cond1, &lock);
    } else {
        printf("Signaling condition variable cond1\n");
```

```
            done == 1;
            pthread_cond_signal(&cond1);
        }

        printf("%s ", (char*)ptr);
        pthread_mutex_unlock(&lock);
        pthread_exit(0);
    }
```

5 - Intro to OpenMP

OpenMP = open Multi-Processing

An api for multithreaded shared parallel programming

OpenMP is:

- higher level than Pthreads
- programmer only states that a block of code is to be executed in parallel
- requires compiler support

Task Parallelisms

- Share the tasks among each core ie on core does the tasks on all data

Data parallelism

- Share the data among each core

OpenMP API

OpenMP is based on directives

OpenMP API components

- Compiler directive
- Runtime library routines
- Environment variables

Fork Join

OpenMP uses the fork join model. The enforces synchronization so every thread must wait till everyone is finished
before proceeding to the next region A group of threads executing the parallel block is known as a team, the original thread is called the master,
the children are called slaves

Task parallelism

```
#pragma omp parallel num_threads(4)
{
  int id = omp_get_thread_num();
  printf("T%d:A\n", id);
  printf("T%d:B\n", id);

  if (id == 0)
    printf("T0:special task\n");

  if (id == 1)
    printf("T1:special task\n");

  if(id == 2)
    printf("T2:special task\n");
}

printf("End");
```

Data Parallelism

```
#pragma omp parallel num_threads(2)
{
  int id = get_thread_num()
  int my_a = id * 3;  \\ where you want the thread to start doing work
  int my_b = id * 3 + 3; \\ where it should stop doing work

  printf("T%d will process indexes %d to ");

  for (int index = my_a; index < my_b; index++)
    printf("do work\n");
}

printf("done\n");

return 0;
```

6 - OpenMP Mutexes, Exclusions, and Synchronization

Race Conditions

A **race condition** occurs when multiple threads **simultaneously access and modify shared data**, leading to **unpredictable behavior**.

Example:

```
#pragma omp parallel
{
    global_sum += my_sum; // Potential race condition
}
```

To prevent this, we use **mutual exclusion** techniques.

Barriers

Barriers ensure that all threads reach a synchronization point before continuing execution.

Types of Barriers:

1. **Implicit Barriers** - Automatically added at the end of parallel regions.
2. **Explicit Barriers** - Defined using `#pragma omp barrier`.

Example:

```
#pragma omp parallel
{
    compute_part();
    #pragma omp barrier // Ensures all threads finish before proceeding
    finalize_part();
}
```

Barrier Limitations:

- All threads must encounter the barrier.
- Conditional execution may lead to **illegal barriers**.

`nowait` Clause

Using `nowait` allows threads **to skip synchronization** when it is unnecessary, improving performance.

Example:

```
#pragma omp single nowait
{
    expensive_task();
}
// Other threads continue execution without waiting.
```

Mutual Exclusion

Mutual exclusion ensures that only **one thread at a time** accesses a critical section.

OpenMP Mutual Exclusion Mechanisms:

1. **Critical Directive** - Ensures exclusive execution.
2. **Atomic Directive** - Ensures atomic updates to a shared variable.
3. **Locks** - Explicit locking mechanisms.

1. Critical Directive

```
#pragma omp critical
{
    shared_var += local_val;
}
```

Named Critical Sections:

```
#pragma omp critical(name1)
x = compute_x();
#pragma omp critical(name2)
y = compute_y();
```

- Allows **simultaneous execution** of **different** critical sections.

---

2. Atomic Directive

`#pragma omp atomic` is **faster** than `critical` for **simple updates**.

```
#pragma omp atomic
sum += value;
```

Supported Operations:

- `x++`, `x--`, `x += expr`, `x = x + expr`

---

3. Locks

Locks **manually enforce** mutual exclusion.

```
#include <omp.h>
static omp_lock_t mylock;

int main() {
    omp_init_lock(&mylock);
```

```
    #pragma omp parallel
    {
        omp_set_lock(&mylock);
        critical_section();
        omp_unset_lock(&mylock);
    }

    omp_destroy_lock(&mylock);
    return 0;
}
```

Key Lock Functions:

- `omp_init_lock(&lock);`
- `omp_set_lock(&lock);`
- `omp_unset_lock(&lock);`
- `omp_destroy_lock(&lock);`

When to Use Which?

Mechanism	Use Case
Atomic	Single-variable updates (fastest)
Critical	Protects complex code sections
Locks	Fine-grained control over execution

Caveats & Best Practices

1. **Avoid Mixing** different mutual exclusion methods.
2. **Fairness is NOT guaranteed** - Some threads may starve.
3. **Avoid Nesting** critical sections (deadlocks possible).

---

7 - OpenMP Variable Scope and Reductions

Variable Scope

In OpenMP, variable scope determines which **threads** can access a variable inside a parallel block.

Shared Variables

- Exist in **one memory location**, accessible by all threads.
- Default behavior for variables declared **before** the parallel block.

```c
int x = 5;
#pragma omp parallel
{
    // All threads access the same x
}
```

Private Variables

- Each thread gets **its own copy** of the variable.
- Uninitialized unless explicitly set.

```c
int y = 5;
#pragma omp parallel private(y)
{
    // Each thread gets its own y (uninitialized)
}
```

Firstprivate Variables

- Like `private`, but **initialized** with the original value.

```c
int z = 5;
#pragma omp parallel firstprivate(z)
{
    // Each thread gets its own z, initialized to 5
}
```

Default Clause

Sets the default scope for all variables.

```c
int x = 0, y = 0;
#pragma omp parallel num_threads(4) default(none) private(x) shared(y)
{
    x = omp_get_thread_num();
    #pragma omp atomic
    y += x;
}
```

Reductions

Reduction operations allow threads to **aggregate results** safely without manual synchronization.

Syntax

```
#pragma omp parallel reduction(<operator> : <variable list>)
```

Example 1: Summing Across Threads

```
int sum = 0;
#pragma omp parallel reduction(+:sum)
{
    sum += omp_get_thread_num();
}
printf("Total sum = %d", sum);
```

Example 2: Multiple Variables

```
int x = 10, y = 10;
#pragma omp parallel reduction(+:x, y)
{
    x = omp_get_thread_num();
    y = 5;
}
printf("Shared: x=%d, y=%d\n", x, y);
```

Reduction Operations

Operator	Description
`+`	Summation
`*`	Multiplication
`&`	Bitwise AND
`|`	Bitwise OR
`^`	Bitwise XOR
`&&`	Logical AND
`||`	Logical OR

Parallel Summation with Reduction

Instead of using a **critical section**, reductions optimize aggregation.

```c
double global_sum = 0;
#pragma omp parallel num_threads(4) reduction(+:global_sum)
{
    global_sum += compute_value(omp_get_thread_num());
}
```

Area Under a Curve (Trapezoidal Rule)

Using **reduction** to integrate a function:

```c
double global_result = 0.0;
#pragma omp parallel num_threads(4) reduction(+:global_result)
{
    global_result += Local_trap(a, b, n);
}
printf("Approximate area: %f\n", global_result);
```

---

8 - Work Sharing (Parallel For, Single)

1. Work-Sharing Constructs

- Used to distribute work among threads inside a parallel region.
- **Types:**
  - `for` – Divides loop iterations across threads.
  - `single` – Assigns work to a single thread.
  - `sections` – Splits tasks into sections executed by different threads.
- There is an **implied barrier** at the exit unless `nowait` is specified.

2. Parallel For

- Loop iterations are divided across threads dynamically.
- The loop variable is **private** by default.
- The execution order is **non-deterministic**.

Syntax Options:

1. **Inside an existing parallel region:**

   ```c
   #pragma omp for
   for(i = start; i < end; i += step) {
       // Loop body
   }
   ```

2. **Creating a parallel region just for the loop:**

   ```
   #pragma omp parallel for
   for(i = start; i < end; i += step) {
       // Loop body
   }
   ```

Example Without OpenMP Parallelization

```
#pragma omp parallel num_threads(4)
{
    int i, n = omp_get_thread_num();
    for(i=0; i<4; i++)
        printf("T%d: i=%d\n", n , i);
}
```

Each thread executes the whole loop, leading to redundant iterations.

Example With OpenMP Parallel For

```
#pragma omp parallel
{
    int i, n;
    #pragma omp for
    for (i = 0; i < 4; i++) {
        n = omp_get_thread_num();
        printf("T%d: i=%d\n", n, i);
    }
}
```

Iterations are divided among the threads, reducing redundancy.

---

3. Data Dependency & Loop-Carried Dependencies

- Parallel loops should avoid **loop-carried dependencies** (when one iteration depends on results from another).
- **Example of incorrect parallelization:**

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

This will produce incorrect results because `fibo[i-1]` and `fibo[i-2]` might not be computed yet.

---

4. Reduction in Parallel Loops

- Reduction avoids data races when accumulating results.
- **Example: Summing values in an array**

```
double sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < n; i++)
    sum += array[i];
```

---

5. Assigning Work to a Single Thread

- Use `#pragma omp single` for operations that should only be done once.
- **Example:**

```
#pragma omp parallel
{
    printf("Hi from T%d\n", omp_get_thread_num());
    #pragma omp single
    printf("One Hi from T%d\n", omp_get_thread_num());
}
```

Only one thread will execute the `single` block.

---

9 - Work Sharing (Sections, Scheduling, Ordered Iterations)

1. Parallel Sections

- `#pragma omp sections` allows different sections of code to be executed by different threads.
- **Example:**

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        printf("Section 1 executed by thread %d\n",
omp_get_thread_num());
    }
    #pragma omp section
```

```
    {
        printf("Section 2 executed by thread %d\n",
omp_get_thread_num());
    }
}
```

- There is an **implicit barrier** at the end of the sections unless `nowait` is used.

---

2. Loop Scheduling

- The `schedule` clause determines how loop iterations are assigned to threads.

Scheduling Type	Description
`static`	Equal chunks assigned at compile time.
`dynamic`	Threads take chunks dynamically.
`guided`	Starts with large chunks, then reduces.
`auto`	Compiler decides the best method.

- **Example using dynamic scheduling:**

```
#pragma omp parallel for schedule(dynamic,2)
for(int i = 0; i<8; i++)
    printf("T%d: %d\n", omp_get_thread_num(), i);
```

---

3. Ordered Iterations

- Ensures that iterations follow a strict order when needed.
- **Example:**

```
#pragma omp for ordered schedule(dynamic)
for(int i=0; i<100; i++) {
    f(a[i]); // Can run in parallel
    #pragma omp ordered
    g(a[i]); // Runs in order
}
```

---

10 - OpenMP Examples, Functions, SIMD

1. Parallel Matrix Multiplication

```c
#pragma omp parallel for collapse(2)
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
        C[i][j] = 0;
        for (k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
```

2. Finding the Maximum Value

```c
int max_parallel(int *arr){
    int i, m = arr[0];
    #pragma omp parallel for reduction(max:m)
    for (i = 0; i < N; i++)
        if (m < arr[i])
            m = arr[i];
    return m;
}
```

3. Producer-Consumer Model

```c
void produce() {
    while (i < NUM_ITEMS) {
        #pragma omp critical(one)
        if (!full) {
            put(item);
            i++;
        }
    }
}

void consume() {
    while (j < NUM_ITEMS) {
        #pragma omp critical(two)
        if (!empty) {
            get();
            j++;
        }
    }
}
```

Ensures only one thread modifies shared data at a time.

```
# Parallel Computing Practice Midterm - Long Answer Solutions

## **Question 1: Parallelizing Nested Loops**
```

````
### **Given Code:**
```c
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        A[i][j] = max(A[i][j], B[i][j]);
````

(a) Parallelizing the Code

Using OpenMP, we can parallelize the outer loop to allow multiple threads to work on different rows concurrently.

```c
#pragma omp parallel for private(j)
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        A[i][j] = max(A[i][j], B[i][j]);
```

- The `#pragma omp parallel for` ensures each thread handles a different value of `i`.
- `private(j)` ensures each thread has its own copy of `j`.

(b) Choosing the Best Schedule

- `static` scheduling: Assigns equal chunks of rows to threads. Good if workload is uniform.
- `dynamic` scheduling: Threads request new rows when they finish processing assigned rows. Best for non-uniform workloads.
- `guided` scheduling: Similar to `dynamic`, but chunk sizes decrease over time.

For this case, **static scheduling** is the most efficient since each iteration has equal workload.

```c
#pragma omp parallel for schedule(static) private(j)
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        A[i][j] = max(A[i][j], B[i][j]);
```

Question 2: Difference Between Parallel Structures

Code Snippets & Explanation

(a) `#pragma omp master`

```c
#pragma omp parallel {
    int n = omp_get_thread_num();
    printf("T%d:A\n", n);
    #pragma omp master
    printf("T%d:X\n", n);
    printf("T%d:B\n", n);
```
```

```
}
printf("Finished");
```

- `#pragma omp master`: Only **one thread (master)** executes `printf("T%d:X\n", n);`.
- All threads execute `printf("T%d:A\n", n);` and `printf("T%d:B\n", n);`.

(b) `#pragma omp single`

```
#pragma omp parallel {
    int n = omp_get_thread_num();
    printf("T%d:A\n", n);
    #pragma omp single
    printf("T%d:X\n", n);
    printf("T%d:B\n", n);
}
printf("Finished");
```

- `#pragma omp single`: **Only one thread executes `printf("T%d:X\n", n);`**, but it can be any thread, not necessarily the master thread.

(c) Explicit Check for Thread 0

```
#pragma omp parallel {
    int n = omp_get_thread_num();
    printf("T%d:A\n", n);
    if(omp_get_thread_num() == 0)
        printf("T%d:X\n", n);
    printf("T%d:B\n", n);
}
printf("Finished");
```

- This explicitly checks if the thread number is `0`, similar to `master`, but allows more flexibility.

Summary of Differences:

- `master`: Only the master thread executes the block.
- `single`: A single (but arbitrary) thread executes the block.
- Explicit check: A thread with a specific ID executes the block.

Question 3: Parallelizing Loops with Dependencies

(a) Serial Code:

```
C[0] = 1;
for (i = 1; i < N; i++) {
```

```
    C[i] = C[i - 1];
    for (j = 0; j < N; j++) {
        C[i] *= A[i][j] + B[i][j];
    }
}
```

Parallelized Version:

- The loop **depends on `C[i-1]`**, so it **cannot** be fully parallelized.
- However, the inner loop can be parallelized:

```
C[0] = 1;
for (i = 1; i < N; i++) {
    C[i] = C[i - 1];
    #pragma omp parallel for
    for (j = 0; j < N; j++) {
        C[i] *= A[i][j] + B[i][j];
    }
}
```

Question 5: Parallelizing Floyd-Warshall Algorithm

Given Code:

```
for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if ( (d[i][k] + d[k][j]) < d[i][j] )
                d[i][j] = d[i][k] + d[k][j];
```

Parallelizing It:

Since `d[i][j]` depends on previous iterations of `k`, only the **inner two loops** can be parallelized:

```
for (k = 0; k < n; k++) {
    #pragma omp parallel for collapse(2)
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if ((d[i][k] + d[k][j]) < d[i][j])
                d[i][j] = d[i][k] + d[k][j];
}
```

- `collapse(2)`: Merges the two loops so that OpenMP distributes **both** `i` and `j` iterations among threads.

Question 6: Explicit OpenMP Parallelization

Given OpenMP Code:

```c
void vector_add(double *a, double *b, double *sum, int n) {
    int i;
    #pragma omp parallel for
    for (i = 0; i < n; i++)
        sum[i] = a[i] + b[i];
}
```

Manually Managing Threads

Instead of `#pragma omp`, we create threads explicitly:

```c
void vector_add(double *a, double *b, double *sum, int n) {
    int TID, TOT;
    #pragma omp parallel private(TID)
    {
        TID = omp_get_thread_num();
        TOT = omp_get_num_threads();
        int range = n / TOT;
        int start = TID * range;
        int end = start + range;

        for (int i = start; i < end; i++) {
            sum[i] = a[i] + b[i];
        }
    }
}
```

- `omp_get_thread_num()`: Each thread gets its unique ID.
- `omp_get_num_threads()`: Gets the total number of threads.
- `range = n / TOT`: Each thread processes an equal chunk.

Code Snippets

```c
pthread_mutex_t lock;
void* say_something(void *ptr) {
    pthread_mutex_lock(&lock);
    printf("%s ", (char*)ptr);
    pthread_mutex_unlock(&lock);
    pthread_exit(0);
}
```

```
int main() {
    pthread_t t1, t2;
    char *msg1 = "Hello ", *msg2 = "World!";
    pthread_mutex_init(&lock, NULL);
    pthread_create(&t1, NULL, say_something, msg1);
    pthread_create(&t2, NULL, say_something, msg2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Done!");
    pthread_mutex_destroy(&lock);
    exit(0);
}
```

Mutex synchronization for thread safety.

```
#pragma omp parallel num_threads(4)
{
  int id = omp_get_thread_num();
  printf("T%d:A\nT%d:B\n", id, id);
  if (id == 0) printf("T0:special task\n");
  if (id == 1) printf("T1:special task\n");
  if (id == 2) printf("T2:special task\n");
}
printf("End");
```

Task parallelism in OpenMP.

```
#pragma omp parallel num_threads(2)
{
  int id = omp_get_thread_num();
  int my_a = id * 3, my_b = id * 3 + 3;
  printf("T%d will process indexes %d to %d\n", id, my_a, my_b);
  for (int index = my_a; index < my_b; index++) printf("do work\n");
}
printf("done\n");
```

Data parallelism using OpenMP.

```
#pragma omp parallel
{
    global_sum += my_sum;
}
```

Race condition due to unsynchronized access.

```
#pragma omp atomic
sum += value;
```

Atomic directive ensures safe updates.

```
#pragma omp critical
{
    shared_var += local_val;
}
```

Critical section to prevent concurrent access.

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++)
    sum += array[i];
```

Reduction safely aggregates results.

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++) {
        C[i][j] = 0;
        for (int k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
```

Parallel matrix multiplication using OpenMP.

```
#pragma omp parallel for schedule(dynamic,2)
for (int i = 0; i<8; i++)
    printf("T%d: %d\n", omp_get_thread_num(), i);
```

Dynamic scheduling distributes workload efficiently.