Parallel and Distributed Programming

Imad Kissami[1]

[1]Mohammed VI Polytechnic University, Benguerir, Morocco

October 28, 2025

What's OpenMP?

- OpenMP is a parallel programming model initially designed for shared memory architectures.
- It now also targets accelerators, integrated systems, and real-time systems.
- Computation tasks can access a common memory space, which limits data redundancy and simplifies data exchange.
- In practice, parallelization is done through lightweight processes (threads), making it a `multithreaded` model..

OpenMP Specifications

- `OpenMP 2 (2000)`: Fortran 95 parallel extensions.
- `OpenMP 3 (2008)`: Introduced the concept of tasks.
- `OpenMP 4 / 4.5 (2013 / 2015)`: Accelerator support, task dependencies, SIMD, and thread placement.
- `OpenMP 5 / 5.1 (2018 / 2020)`: Improved accelerator support, non-uniform memory support, C11/C++17/Fortran 2008 support.
- `OpenMP 6 (2024)`: Enhanced support for modern C++ (20/23), unified memory features, deep copy capabilities, improved accelerator support and Fortran enhancements.

Key Concepts in OpenMP

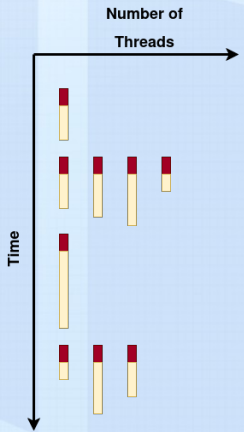
- `Thread`: An execution entity with its own local memory (`stack`).
- `Team`: A group of one or more threads executing a parallel region.
- `Task`: A block of executable code with associated data, generated by `parallel` or `task` constructs.
- `Shared variable`: A variable accessible to all threads within a parallel region.
- `Private variable`: A variable for which each thread has its own distinct copy.
- `Host device`: The CPU system where OpenMP begins execution.
- `Target device`: A hardware accelerator (GPU, Xeon Phi) to which code and data can be offloaded.

CPU Core vs. Thread

- CPU Core (Physical):
  - A real hardware unit capable of executing instructions independently.
  - More cores mean better native parallel processing.
  - Example: A quad-core CPU has 4 physical cores.
- Thread (Logical):
  - A virtual execution unit via Hyper-Threading.
  - Shares core resources; increases throughput but with limited performance.
  - Example: A quad-core CPU with HT provides 8 threads.

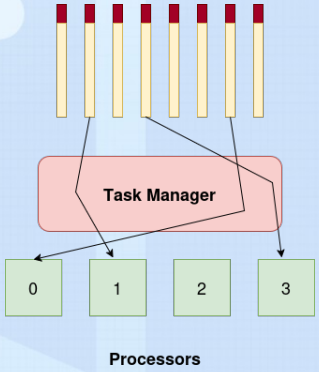
General concepts: Execution model

- At start, an OpenMP program runs sequentially with a single master thread (rank 0).
- OpenMP allows defining parallel regions which are code portions destined to be executed in parallel.
- On entering a parallel region, OpenMP creates threads and associated implicit tasks.
- Each thread executes its own task in parallel with others to share the workload.
- The program alternates between sequential and parallel regions.



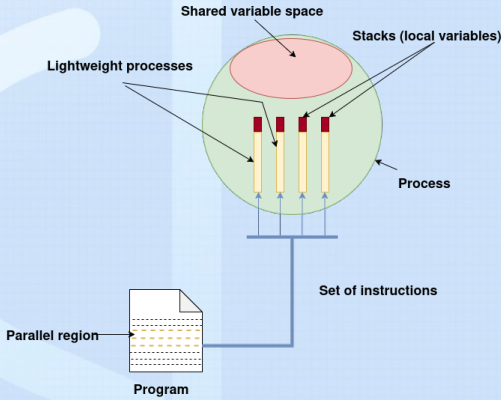
General concepts: Threads (light-weight processes)

- Each thread runs its own sequence of instructions (its own task).
- The operating system decides when and on which CPU core each thread runs.
- There is no guaranteed order in how threads run — it can change each time.



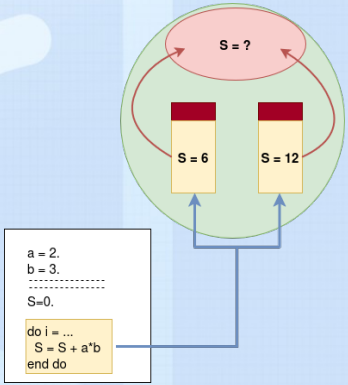
General concepts: Shared and Private Memory

- All threads share the same memory space from the main program (shared memory).
- Each thread also has its own local memory called the `stack`.
- You can define:
  - `Shared` variables — accessible by all threads.
  - `Private` variables — each thread has its own copy.



General concepts: Synchronization

- In shared memory, multiple threads can access and modify the same variable.
- To avoid incorrect results, `synchronization` is sometimes needed.
- `Synchronization` ensures that two threads do not change a shared variable at the same time.
- A common case is with `reduction` operations, where multiple values are combined into one.

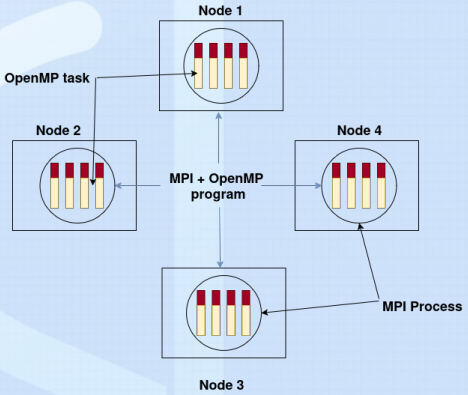


Functionalities of OpenMP

- Makes it easier to write parallel programs using shared memory.
- Provides tools to:
  - Split the work between threads (e.g., loops).
  - Define shared or private variables.
  - Synchronize threads to avoid race conditions.
  - Offload computations to accelerators (e.g., GPUs).
  - Express parallelism using tasks.

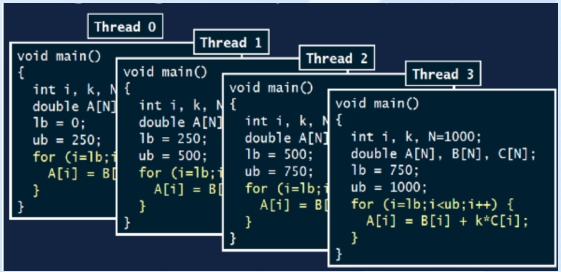
OpenMP versus MPI

- MPI: Programming model for distributed memory systems.
  - Each process has its own memory space.
  - Data is exchanged through explicit messages.
- OpenMP: Programming model for shared memory systems.
  - All threads share the same memory space.
  - Easier communication, but limited to a single node.
- Both can be combined in hybrid programs to run on clusters of shared-memory nodes.



OpenMP & SPMD

- Single Program: All threads execute the same code.
- Multiple Data: Each thread processes a different portion of the data.



Programming Interface: Directive Format and Examples

- OpenMP directives follow the general form:

```
sentinelle directive [clause [ clause] ...]
```

- The directive looks like a comment to compilers that don't support OpenMP.
- The sentinel depends on the programming language.

In C/C++:

```
#ifdef _OPENMP
#include <omp.h>
#endif
...
#pragma omp parallel private(a,b) ↩
      firstprivate(c,d,e)
{
  ...
}
```

In Fortran:

```
!$ use OMP_LIB
...

!$OMP PARALLEL PRIVATE(a,b) &
!$OMP FIRSTPRIVATE(c,d,e)
...
!$OMP END PARALLEL
```

Compilation

Compilation options for activating the interpretation of OpenMP directives by some compilers:

- The GNU compiler: `-fopenmp`

```
gcc -fopenmp prog.c    # C compiler
```

- The Intel compiler: `-fopenmp` or `-qopenmp`

```
icc -fopenmp prog.c    # C compiler
```

- The NVIDIA compiler : `-Xcompiler -fopenmp`

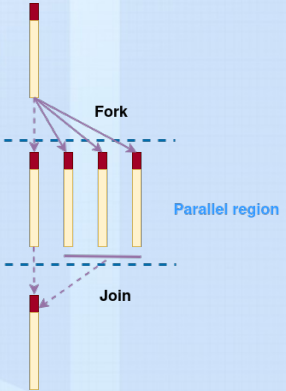
```
nvcc -Xcompiler -fopenmp prog.c      # C compiler
```

- Execution example:

```
export OMP_NUM_THREADS=3  # Number of desired threads
./a.out           # Execution
```

Parallel construct: Fork-Join Model

- An OpenMP program alternates between sequential and parallel regions.
- When a parallel region begins:
  - The master thread (rank 0) creates additional threads.
  - Each thread runs its own implicit task.
- When the parallel region ends:
  - All threads synchronize and the program continues sequentially.
- This is known as the fork-join model.



Parallel Construct: Code Example

- Inside a parallel region, each thread runs the same code.
- The data-sharing attribute (DSA) of the variables are shared, by default. So `a` is **`shared`** by default across all threads
- There is an implicit synchronization barrier at the end of the parallel region.

```
A = 92290.000000
A = 92290.000000
A = 92290.000000
Parallel ?: T
```

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

int main(void)
{
  float a = 92290.;
  bool p = false;

#pragma omp parallel
  {
    p = omp_in_parallel();
    printf("A = %f\n", a);
  }

  printf("Parallel ?: %s\n", p ? "T" : "F");
  return 0;
}
```

```
gcc -fopenmp example1.c -o example1
export OMP_NUM_THREADS=3
./example1
```

Data-sharing attribute of variables: Private variables

- The private clause allows changing the DSA of a variable to private.
- If a variable has a private DSA, it is allocated in the stack of each thread.
- The private variables are not initialized when the parallel region starts.
- The original value of a outside the region remains unchanged.

```c
#include <stdio.h>
#include <omp.h>
int main(void){
  float a = 92000.0;
  int rank;

#pragma omp parallel private(rank, a)
  {
    rank = omp_get_thread_num();
    a  = a + 290.0;
    printf("Rank: %d ; a = %f\n", rank, a);
  }
  printf("Out of region, a = %f\n", a);
  return 0;
}
```

```
Rank: 1 ; a = 290.000000
Rank: 0 ; a = 290.000000
Rank: 2 ; a = 290.000000
Out of region, a = 92000.000000
```

Data-sharing attribute of variables: Private variables

- firstprivate copies the initial value of a variable to each thread.
- Threads use their private copy, but it's initialized from the original.
- The original variable remains unchanged after the parallel region.

```c
#include <stdio.h>
#include <omp.h>
int main(void){
  float a = 92000.;

#pragma omp parallel firstprivate(a)
  {
    a = a + 290.0;
    printf("a = %f\n", a);
  }
  printf("Out of region, a = %f\n", a);
  return 0;
}
```

```
A = 92290.000000
A = 92290.000000
A = 92290.000000
Out of region, A = 92000.000000
```

Data-sharing attribute of variables: The DEFAULT clause

- With default(none), you must explicitly declare all variable sharing types.
- This helps prevent mistakes and improves code clarity.
- In this example, `p` is declared as `shared`.

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

int main(void)
{
  bool p = false;

#pragma omp parallel default(none) shared(p)
  {
    p = omp_in_parallel();
  }

  printf("Parallel ?: %s\n", p ? "True" : "False");
  return 0;
}
```

```
Parallel ?: True
```

Dynamic Memory Allocation in Parallel Regions

- Memory can be dynamically allocated inside a parallel region.
- If the pointer is declared inside the region, each thread has its own private allocation.
- Each thread must also free its own memory.
- This avoids race conditions and improves data locality.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
  int n = 1000;

#pragma omp parallel
  {
    double* array = (double*) malloc(n * sizeof(double));

    int id = omp_get_thread_num();
    for (int i = 0; i < n; i++) {
      array[i] = id * 1.0;
    }

    printf("Thread %d example: %f\n", id, array[10]);
    free(array);
  }
  return 0;
}
```

Dynamic Allocation of Shared Variables

- If the dynamically allocated variable is shared by threads, only one thread (e.g., the master thread) should allocate and initialize it.
- This avoids concurrent memory conflicts and ensures consistency.
- Due to data locality, it is better to initialize the array inside the parallel region ("first touch" policy).
- Use `#pragma omp master` or conditionals to restrict memory allocation to one thread.

```
#pragma omp parallel
{
#pragma omp master
  {
    shared_array = malloc(...);
  }
#pragma omp barrier
  // Threads can now safely use the array
}
```

Data-sharing attribute of variables

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

int main(void)
{
  int n = 1024;
  float *a = (float *) malloc(n * sizeof(float));
  int start, end, rank, nb_tasks, i;

  /*
  The 'if(n > 512)' directive: It tells OpenMP to use parallelization only if n > 512.
  The 'default(none)' enforces explicit sharing of variables.
  'private(start, end, rank, nb_tasks, i)' means these variables will have separate copies in each thread.
  'shared(a, n)' means 'a' and 'n' are shared among all threads.
  */
#pragma omp parallel if(n > 512) default(none) shared(a, n) private(start, end, rank, nb_tasks, i)
  {
```

Data-sharing attribute of variables

```
    nb_tasks = omp_get_num_threads();
    rank  = omp_get_thread_num();

    start = (rank * n) / nb_tasks;
    end =  (((rank + 1) * n) / nb_tasks) - 1;

    for (i = start; i <= end; i++) a[i] = 92290. + (float) i;

    printf("Rank: %d; a[%d],..., a[%d]: %f,...,%f\n", rank, start, end, a[start], a[end]);
    }

  free(a);
}
```

```
Rank: 2; a[682],..., a[1023]: 92972.000000,...,93313.000000
Rank: 1; a[341],..., a[681]: 92631.000000,...,92971.000000
Rank: 0; a[0],..., a[340]: 92290.000000,...,92630.000000
```

Extent of a parallel region

- The `extent` of a parallel region includes:
  - The code directly inside the region (static extent).
  - Any functions or routines called from inside (dynamic extent).
- In this example, the function `sub()` is part of the parallel region's extent.

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

// function prototype
void sub();

int main(void){
#pragma omp parallel
  {
    sub();
  }
  return 0;
}

void sub()
{
  int p = omp_in_parallel();
  printf("Parallel ?: %s\n", p ? "T" : "F");
}
```

```
Parallel ?: T
Parallel ?: T
Parallel ?: T
```

Extent of a parallel region

- Variables declared inside a function are automatically private when called in a parallel region.
- This is because each thread gets its own stack.
- In C/C++, the variables declared inside a parallel region are private.

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

void sub();

int main(void){
#pragma omp parallel default(shared)
  {
    sub();
  }
  return 0;
}
void sub(){
  // local variable is private by default
  float a = 92290.0;

  a += omp_get_thread_num();
  printf("a = %f\n", a);
}
```

```
a = 92290.000000
a = 92292.000000
a = 92291.000000
```

Transmission by arguments

- Variables passed as arguments inherit the DSA from the calling context.
- In this example:
  - `a` is `shared` → same value for all threads.
  - `b` is `private` → each thread has its own copy.
- The function `sub()` updates `b` using the thread ID.

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

void sub(int x, int *y);

int main(void){
  int a = 92000, b = 0;

#pragma omp parallel shared(a) private(b)
  {
    sub(a, &b);
    printf("a = %d, b = %d\n", b);
  }
  return 0;
}

void sub(int x, int *y){
  *y = x + omp_get_thread_num();
}
```

```
a = 92000, b = 92002
a = 92000, b = 92000
a = 92000, b = 92001
```

Static variables

- A `static` variable keeps its value across multiple calls.
- In a parallel region, a static variable is `shared` by default.
- Each thread sees the same instance of the variable.
- This can lead to race conditions if not handled properly.

```
Values of a, b et c : 1.000000, 1.000000, 1.000000
Values of a, b et c : 1.000000, 1.000000, 0.000000
Values of a, b et c : 1.000000, 1.000000, 2.000000
```

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

float static b;
void sub() {
  int rank;
  float static a;
  float c;
  rank = omp_get_thread_num();

  a = b = c = rank;

  // Barrier to synchronize threads
#pragma omp barrier
  printf("Values of a, b et c : %f, %f, %f\n", a, b, c);
}

int main() {
#pragma omp parallel
  {
    sub();
  }
  return 0;
}
```

Static variables: Threadprivate and Copyin

- `threadprivate` makes a static variable private and persistent for each thread.
- `copyin` copies the value from the master thread to others at entry.
- Useful to give each thread an initial value and preserve state across regions.

```
b vaut : 92291.000000
b vaut : 92290.000000
b vaut : 92292.000000
Hors region, a vaut: 92000.000000
```

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

static float a = 0.;
#pragma omp threadprivate(a)
void sub() {
  float b = a + 290;
  printf("b vaut : %f\n", b);
}
int main() {
  a = 92000.;
#pragma omp parallel copyin(a)
  {
    a += omp_get_thread_num();
    sub();
  }
  printf("Hors region, a vaut: %f\n", a);
  return 0;
}
```

Clauses: `reduction` and `num_threads`

- `num_threads(n)` allows setting a specific number of threads for a region. (similar to calling `omp_set_num_threads`).
- `reduction(+:var)` combines values across threads using an operation (like +).
- Reduction clauses ensure thread-safe accumulation with implicit synchronization.

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

int main(void)
{
#pragma omp parallel num_threads(2)
  {
    printf("Good Morning !\n");
  }

#pragma omp parallel num_threads(3)
  {
    printf("Hello !\n");
  }
  return 0;
}
```

```
Good Morning !
Good Morning !
Hello !
Hello !
Hello !
```

Nested Parallel Regions

- By default, nested parallel regions are disabled.
- To enable them, set the environment variable `OMP_NESTED=true`.
- Each parallel region can have its own number of threads.
- Nested regions allow hierarchical parallelism.

```
gcc -fopenmp set_nested.c -o set_nested
export OMP_NESTED=true
./a.set_nested
```

```c
#include <stdio.h>
#include <omp.h>
int main(void){
  int rank;
#pragma omp parallel num_threads(3) private(rank)
  {  rank = omp_get_thread_num();
    printf("My rank in region 1 : %d\n", rank);
#pragma omp parallel num_threads(2) private(rank)
    {
      rank = omp_get_thread_num();
      printf(" My rank in region 2 : %d\n", rank);
    }
  }
  return 0;}
```

```
My rank in region 1 : 1
My rank in region 1 : 2
My rank in region 1 : 0
 My rank in region 2 : 1
 My rank in region 2 : 1
 My rank in region 2 : 0
 My rank in region 2 : 0
 My rank in region 2 : 1
 My rank in region 2 : 0
```

Introduction to Worksharing

- Creating a parallel region is not enough — the programmer must still manage how the work is split among threads.
- OpenMP provides specific directives to help with this, such as:
  - `for`, `sections`, and `workshare`
- Some code inside a parallel region can be executed by only one thread using:
  - `single` and `master` directives
- Thread synchronization will be covered in the next chapter.

Parallelizing Loops with `for` Directive

- A loop can be parallelized if all iterations are independent.
- OpenMP distributes loop iterations among threads using `#pragma omp for`.
- Only `for` loops (not `while` or infinite loops) can be parallelized this way.
- Use explicit tasks for complex loop structures.
- The `schedule` clause allows control over iteration distribution.
- Loop indices are private by default — no need to declare them as such.
- By default, there is a synchronization barrier at the end of the loop (can be removed with `nowait`).

The `schedule` Clause

- The `schedule` clause controls how loop iterations are divided among threads.
- General syntax:

```
#pragma omp for schedule(mode, chunk_size)
```

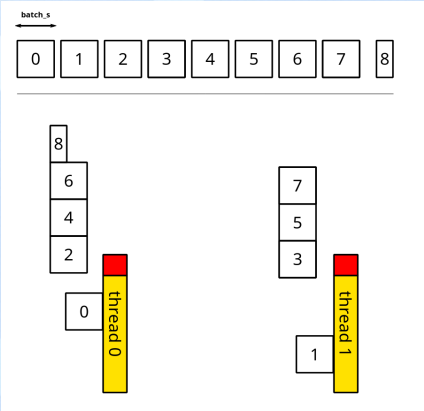
- Common scheduling modes:
  - `static[, chunk_size]` – fixed-size chunks assigned in round-robin.
  - `dynamic[, chunk_size]` – threads request new chunks when free.
  - `guided[, min_chunk_size]` – chunk size decreases over time.
- Choosing the right mode can improve load balancing and performance.

Static Scheduling & Chunk Mapping

- The simplest distribution mode is `STATIC`.

```
#pragma omp for schedule(static, batch_size)
```

- The chunk size is fixed.
- If `batch_size` is omitted, it defaults to the largest possible chunk.
- Chunks are distributed cyclically among threads.
- Useful when: iteration workload is uniform.
- Comments:
  - Compiler optimizations are possible.
  - Prefer larger `batch_size` values when applicable.



Static Scheduling Code (Part 1)

```
#include <stdio.h>
#include <omp.h>
#define N 4096

int main() {
  float a[N];
  int i, rank, nb_tasks, i_min, i_max;

#pragma omp parallel private(rank, nb_tasks, i_min, i_max)
  {
  rank = omp_get_thread_num();
  nb_tasks = omp_get_num_threads();
  i_min = N;
  i_max = 0;
```

Static Scheduling Code (Part 2) & Output

```
#pragma omp for schedule(static, N/nb_tasks) nowait
  for (i = 0; i < N; i++) {
  a[i] = 92290.0 + (float)(i + 1);
  if (i < i_min) i_min = i;
  if (i > i_max) i_max = i;
  }
  printf("Rank: %d; i_min: %d; i_max: %d\n", rank, i_min, i_max);
}
return 0;
}
```

```
gcc -fopenmp static_range.c -o static_range
export OMP_NUM_THREADS=3
./static_range

Rank: 2; i_min: 2730; i_max: 4094
Rank: 0; i_min: 0; i_max: 4095
Rank: 1; i_min: 1365; i_max: 2729
```

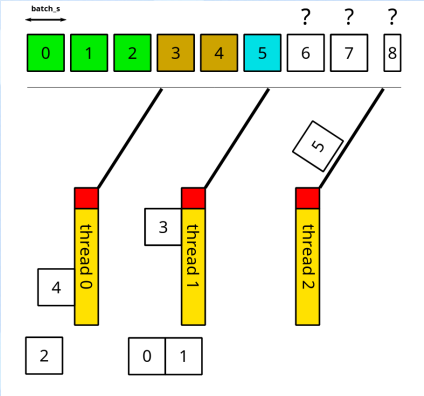
- Why i_max for Rank 0 is 4095?

Dynamic Scheduling & Chunk Mapping

- We can relax the constraint of fixed distribution with `DYNAMIC`.

```
#pragma omp for schedule(DYNAMIC, batch_size)
```

- The chunk size is fixed.
- If `batch_size` is omitted, it defaults to 1.
- The next chunk is assigned to a thread as soon as it becomes free.
- Useful when:
  - The machine's resources are shared.
  - The workload varies from one iteration to another.



Dynamic Scheduling & Chunk Mapping

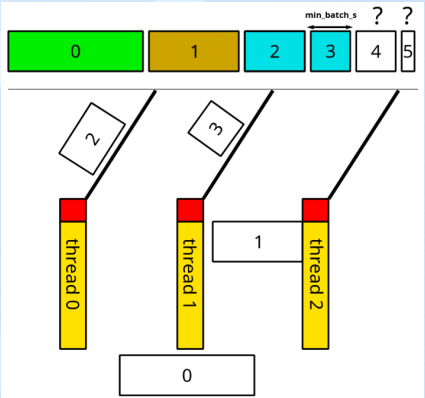
```
include <stdio.h>
#include <omp.h>
int main() {
  int N = 13;
  int i;
#pragma omp parallel
  {
#pragma omp for schedule(dynamic, 3)
  for (i = 0; i < N; i++) {
    printf("Thread %d: Iteration %d\n", omp_get_thread_num(), i);
  }
  }
  return 0;
}
```

Guided Scheduling & Chunk Mapping

- With `GUIDED`, chunk sizes decrease over time while remaining above a minimum threshold.

```
#pragma omp for schedule(GUIDED, batch_size)
```

- The first chunks are large and are assigned quickly.
- Later chunks are smaller, improving load balancing.
- `min_batch_size` sets the minimum chunk size.
- If `batch_size` is omitted, it defaults to 1.
- Useful when:
  - The workload is heavier at the beginning of the loop.
  - Finer granularity is required near the end for better balance.



Guided Scheduling & Chunk Mapping

```
#include <stdio.h>
#include <omp.h>
int main() {
  int N = 20;
  int i;
#pragma omp parallel
  {
#pragma omp for schedule(guided, 2)
  for (i = 0; i < N; i++) {
    printf("Thread %d: Iteration %d\n", omp_get_thread_num(), i);
  }
  }
  return 0;
}
```

Additional Scheduling Options

- Two other scheduling options are available in OpenMP:
- `schedule(runtime)`:
  - The scheduling policy and chunk size are set at runtime.
  - They are specified using the environment variable:

```
export OMP_SCHEDULE='DYNAMIC,200'
export OMP_NUM_THREADS =4
./a.out
```

  - Useful for experimenting with different schedules without recompiling.
- `schedule(auto)`:
  - The compiler and/or runtime system choose the best scheduling strategy.
  - May take advantage of performance models or heuristics.
  - Useful when performance is difficult to predict.

An Ordered Execution

- It is sometimes useful to execute a loop in an ordered way (e.g., for debugging).
- The iteration order will then be identical to that of a sequential execution.

```
export OMP_SCHEDULE="STATIC,2"
export OMP_NUM_THREADS=4
./ordered
rank: 0; iteration: 0
rank: 0; iteration: 1
rank: 1; iteration: 2
rank: 1; iteration: 3
rank: 2; iteration: 4
rank: 2; iteration: 5
rank: 3; iteration: 6
rank: 3; iteration: 7
rank: 0; iteration: 8
```

```
int main(void)
{
  const int n = 9;
  int rang;

#pragma omp parallel private(rang)
  {
    rang = omp_get_thread_num();
#pragma omp for schedule(runtime) ordered
    for (int i = 0; i < n; i++)
    {
#pragma omp ordered
      {
        printf("rank: %d; iteration: %d\n", rang, i);
      }
    }
  }
  return 0;
}
```

Reductions with OpenMP

- A reduction is an associative operation on a shared variable.
- Supported operations:
  - Arithmetic: `+`, `-`, `*`
  - Logical: `&&`, `||`, `==`, `!=`
  - Intrinsic: `max`, `min`
- Each thread computes a local result.
- A final step merges all results into the shared variable.

```
s = 5; p = 32; r = 243
```

```c
#include <stdio.h>
#include <omp.h>
int main(void)
{
  const int n = 5;
  int i;
  int s = 0, p = 1, r = 1;

#pragma omp parallel
  {
#pragma omp for reduction(+:s) reduction(*:p,r)
  for(i = 0; i < n; i++) {
    s = s + 1;
    p = p * 2;
    r = r * 3;
  }
  }
  printf("s = %d; p = %d; r = %d\n", s, p, r);
  return 0;
}
```

Fusion of loop nests

Using the `collapse(n)` Clause

- The `collapse(n)` clause transforms `n` nested loops into a single iteration space.
- This increases available parallelism when outer loops have small bounds.
- Improves load balancing and performance.
- Requires perfectly nested loops with canonical form.

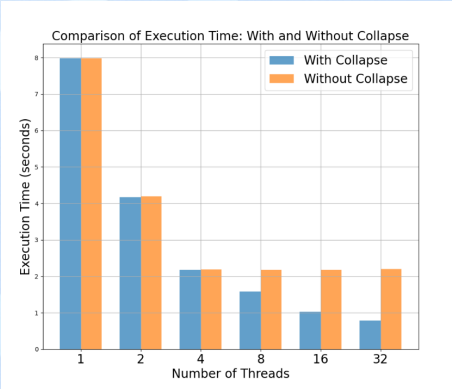
```
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; i++) {
  for (int j = 0; j < M; j++) {
    A[i][j] = B[i][j] + C[i][j];
  }
}
```

Fusion of loops: Using the `collapse(n)` Clause

```
int main() {
  int N1 = 4; N2 = 8, N3 = 10000000;
  float *A = (float *)malloc(N1 * N2 * N3 * sizeof(float));
  int i, j, k;
  double itime, ftime;
  itime = omp_get_wtime();
#pragma omp parallel
  {
#pragma omp for schedule(static) collapse(3)
  for (i = 0; i < N1; i++) {
    for (j = 0; j < N2; j++) {
      for (k = 1; k < N3; k++)
        A[i * N2 * N3 + j * N3 + k] = expf(sinf(A[i * N2 * N3 + j * N3 + k - 1]) + ↩
              cosf(A[i * N2 * N3 + j * N3 + k])) / 2.0;
      }
    }
  }
  ftime = omp_get_wtime();
  printf("Time = %.3f seconds.\n", ftime - itime);
  free(A);}
```

Fusion of loops: Performance Impact of `collapse(n)`

- Execution of the same nested loop program with and without the `collapse` clause.
- The elapsed time (in seconds) is measured for different thread counts.
- The use of `collapse` allows better load balancing across threads.
- Especially useful when outer loops have small bounds.



Additional Clauses for `for` Directive

- Several additional clauses are accepted in the `for` directive:
  - `private`: declares a variable as thread-local.
  - `firstprivate`: creates a private copy initialized with the pre-loop value.
  - `lastprivate`: keeps the value of the last iteration (thread that executes it).
- These clauses provide precise control over data scoping:
  - Avoid data races.
  - Maintain initialization or final values.
  - Useful in nested loops and conditionally-executed iterations.

Example of `private` Clause

- Variable `x` is shared before entering the parallel region.
- The `private(x)` clause creates a local copy of `x` for each thread.
- Each thread updates its own copy independently.
- Outside the parallel region, the original value of `x` remains unchanged.

```
Thread 0: x = 0
Thread 1: x = 10
Thread 2: x = 20
Thread 3: x = 30
After loop: x = 10
```

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

int main() {
  int x = 10;
#pragma omp parallel
  {
#pragma omp for private(x)
    for (int i = 0; i < 4; i++) {
      // print x => Garbage value
      x = i * 10;
      printf("Thread %d: x = %d\n", omp_get_thread_num(), x);
    }
  }
  printf("After loop: x = %d\n", x);
  return 0;
}
```

Example of `firstprivate` Clause

- The variable `x` is initialized to 10 before entering the parallel region.
- The `firstprivate(x)` clause creates a private copy of `x` for each thread, initialized with the value of `x` before the parallel region.
- Each thread modifies its own private copy.
- The original value of `x` remains unchanged after the parallel region.

```
Thread 0: x = 10
Thread 1: x = 11
Thread 2: x = 12
Thread 3: x = 13
After loop: x = 10
```

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

int main() {
  int x = 10;
#pragma omp parallel
  {
#pragma omp for firstprivate(x)
    for (int i = 0; i < 4; i++) {
      x += i; // Each thread starts with x=10
      printf("Thread %d: x = %d\n", omp_get_thread_num(), x);
    }
  }
  printf("After loop: x = %d\n", x);
  return 0;
}
```

Example of `lastprivate` Clause

- The variable `x` is declared outside the parallel region.
- With `lastprivate(x)`, each thread has its private copy of `x`.
- At the end of the loop, the value of the last iteration (as if in sequential order) is copied to the original `x`.
- The final value printed reflects the computation from the last logical iteration.

```
Thread 0: x = 0
Thread 1: x = 10
Thread 2: x = 20
Thread 3: x = 30
After loop: x = 30
```

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

int main() {
  int x = 10;
#pragma omp parallel
  {
#pragma omp for lastprivate(x)
    for (int i = 0; i < 4; i++) {
      x = i * 10;
      printf("Thread %d: x = %d\n", omp_get_thread_num(), x);
    }
  }
  printf("After loop: x = %d\n", x);
  return 0;
}
```

Example of `parallel for` Directive

- The directive `parallel for` merges both `parallel` and `for`.
- It accepts clauses from both directives.
- It implies a global synchronization at the end of the loop.
- `nowait` is not allowed after a `parallel for`.
- In this example, the variable `temp` is updated in each iteration.

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

int main(void)
{
  const int n = 9;
  int i;
  float temp;

  #pragma omp parallel for lastprivate(temp)
  for (i = 0; i < n; i++) {
    temp = (float)i;
  }

  printf("Final value of temp = %.1f\n", temp);
  return 0;
}
```

Example of `nowait` Clause

- By default, threads synchronize at the end of a worksharing construct.
- The `nowait` clause removes the implicit barrier.
- Threads do not wait for others to finish before continuing.
- Useful for improving performance when synchronization is not needed.

```
Thread 0: in 1st loop
Thread 1: in 1st loop
Thread 1: in 2nd loop
Thread 0: in 2nd loop
```

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

int main() {
#pragma omp parallel
  {
#pragma omp for nowait
   for (int i = 0; i < 2; i++)
     printf("Thread %d: in 1st loop\n", omp_get_thread_num());

#pragma omp for
   for (int i = 0; i < 2; i++)
     printf("Thread %d: in 2nd loop\n", omp_get_thread_num());
  }
  return 0;
}
```

Parallel loop: `#pragma omp parallel` vs. `#pragma omp parallel for`

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

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

Feature Comparison

Feature	`#pragma omp parallel + for`	`#pragma omp parallel for`
Thread Creation	Once for the whole block	Created/destroyed for each loop
Multiple Loops	Efficient for multiple loops	New threads for each loop
Synchronization	Barrier after each `for`	Barrier after loop
Best Use Case	Multiple parallel loops	Single parallel loop

Parallel Sections

- A `section` is a block executed by one and only one thread.
- Several independent code blocks can be run in parallel using `sections`.
- OpenMP assigns one thread per section.
- The `nowait` clause is optional and removes the implicit barrier.
- Good for distributing independent tasks like I/O, pre/post-processing, etc.

```
Thread 0: Task A
Thread 1: Task B
Thread 2: Task C
```

```c
int main() {
#pragma omp parallel sections
  {
#pragma omp section
    {
      printf("Thread %d: Task A\n", omp_get_thread_num());
    }
#pragma omp section
    {
      printf("Thread %d: Task B\n", omp_get_thread_num());
    }
#pragma omp section
    {
      printf("Thread %d: Task C\n", omp_get_thread_num());
    }
  }
  return 0;
}
```

Parallel Sections — Key Features & Example Execution

- Task Parallelism: Different tasks execute concurrently.
- Thread Assignment: OpenMP assigns each section to a distinct thread.
- Automatic Load Balancing: Sections are distributed based on thread availability.
- Ideal for heterogeneous computations or pre/post-processing.
- There is no guarantee that Task A will be executed before Task B or C.

Example Execution

Thread	Executed Task
0	Task B
1	Task C
2	Task A

Complementary Information on `parallel sections`

- All `section` directives must appear in the lexical scope of the enclosing `sections` block.
- The following clauses are accepted in the `sections` construct:
  - `private`
  - `firstprivate`
  - `lastprivate`
  - `reduction`
- The directive `#pragma omp parallel sections` is a shorthand combining `parallel` and `sections`, inheriting all their clauses.
- The terminating barrier is implicit and cannot be removed (the `nowait` clause is not allowed).

Exclusive Execution

- In some cases, we may want only one thread to execute specific portions of code within a parallel region.
- OpenMP provides two directives for this purpose: `single` and `master`.
- Although their objective is the same — restricting execution to a single thread — their behaviors are fundamentally different.
  - `single`: any thread may execute the block, with an implicit barrier at the end (unless `nowait` is specified).
  - `master`: only the master thread (thread 0) executes the block, and no barrier is implied.

Exclusive Execution with `single` Directive

- Only one thread executes the code block under **`single`**.
- Other threads skip it and wait at the implicit barrier.
- Output:

```
Rank: 1 ; a is: 92290.000000
Rank: 2 ; a is: -92290.000000
Rank: 0 ; a is: 92290.000000
```

- Output shows different values of `a` depending on whether the thread entered the **`single`** block.

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

int main(void)
{
int rank;
float a;
#pragma omp parallel default(private)
{
    a = 92290.0;
#pragma omp single
    {
        a = -92290.0;
    }
    rank = omp_get_thread_num();
    printf("Rank: %d ; a is: %f\n", rank, a);
}
return 0;
}
```

Exclusive Execution with `single` + `copyprivate`

- It allows the thread which is charged with executing the **`single`** region, to broadcast the value of a list of private variables to other threads before exiting this region.
- A supplementary clause accepted only by the termination directive is the **`copyprivate`** clause.
- Output:

```
Rank: 1 ; a is: -92290.000000
Rank: 2 ; a is: -92290.000000
Rank: 0 ; a is: -92290.000000
```

- The other clauses accepted by the **`single`** directive are **`private`** and **`firstprivate`**.

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

int main(void)
{
  int rank;
  float a;
#pragma omp parallel default(private)
  {
    a = 92290.0;
#pragma omp single copyprivate(a)
    {
      a = -92290.0;
    }
    rank = omp_get_thread_num();
    printf("Rank: %d ; a is: %f\n", rank, a);
  }
  return 0;
}
```

Exclusive Execution with `master` Directive

- Only the master thread (thread 0) executes the code block under `master`.
- Other threads skip it without synchronization (no implicit barrier).
- Output:

```
Rank: 1 ; a is: 92290.000000
Rank: 2 ; a is: 92290.000000
Rank: 0 ; a is: -92290.000000
```

- Output shows that only thread 0 changes the value of `a`.

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

int main(void)
{
  int rank;
  float a;
#pragma omp parallel default(private)
  {
    a = 92290.0;
#pragma omp master
    {
      a = -92290.0;
    }
    rank = omp_get_thread_num();
    printf("Rank: %d ; a is: %f\n", rank, a);
  }
  return 0;
}
```

Summary of Supported Clauses by Construct

- The table below shows which clauses are supported by each OpenMP construct:

Construct	default	shared	private	firstprivate	lastprivate	copyprivate	if	reduction	schedule	ordered	copyin	nowait
parallel	X	X	X	X			X	X			X	
for/do			X	X	X			X	X	X		X
sections			X	X	X			X				X
single			X	X		X						X
master												

Synchronization — When is it needed?

- To ensure that all threads reach the same instruction point (global barrier).
- The absence of the `nowait` clause implies a synchronization barrier at the end of the construct.
- The barrier can also be explicitly applied using the `barrier` directive.

- Useful to coordinate parallel execution safely.
- Guarantees consistency and correct ordering between threads.

Synchronization — Mutual Exclusion and Coherence

- When threads execute the same code accessing shared variables.
- Memory coherence (read/write) must be guaranteed.
- Mechanisms: `reduction`, `ordered`, `atomic`, and `critical`.

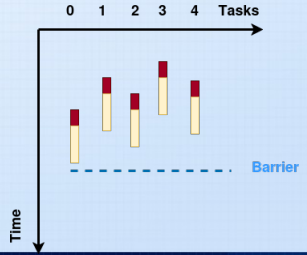
- Required for deterministic and correct results.
- Prevents race conditions and undefined behaviors.

Synchronization — The `flush` Directive

- Ensures memory visibility between threads.
- Refreshes the value of a shared variable in main memory.
- Particularly important in systems with hierarchical caches.
- Can be used to enforce synchronization without a full barrier.
- Often used in spin-wait loops for thread coordination.
- Requires careful use to avoid inconsistencies.

Global Synchronization: `barrier`

- All threads print a message before the barrier.
- Each thread waits at the `barrier` until all arrive.
- Then, all threads print another message after the barrier.
- Guarantees that the second part starts only after all threads complete the first.




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

int main(void)
{
#pragma omp parallel
  {
    int rank = omp_get_thread_num();
    printf("Before barrier: Thread %d\n", rank);

#pragma omp barrier

    printf("After barrier: Thread %d\n", rank);
  }

  return 0;
}
```

Protecting Updates: `atomic`

- The `atomic` directive ensures a shared variable is updated atomically.
- Prevents race conditions on operations like incrementing a counter.
- Only affects the instruction immediately following it.

```
export OMP_NUM_THREADS=4; ./atomic
Final counter = 4
```

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

int main(void)
{
  int counter = 0;

#pragma omp parallel
  {
#pragma omp atomic
    counter += 1;
  }

  printf("Final counter = %d\n", counter);
  return 0;
}
```

Supported Forms of `atomic` Updates

- The `atomic` directive supports specific types of expressions:
  - `x = x op expr`
  - `x = expr op x`
  - `x = f(x, expr)`
  - `x = f(expr, x)`
- Operators: `+`, `-`, `*`, `/`, `&&`, `||`, etc.
- Intrinsic functions: `min`, `max`, ...
- The variable `x` must be shared and scalar.

```
// x = x + 1
#pragma omp atomic
x = x + 1;

// x = 2 * x
#pragma omp atomic
x = 2 * x;

// x = max(x, new_value)
#pragma omp atomic
x = fmax(x, new_value);

// x = min(100, x)
#pragma omp atomic
x = fmin(100, x);
```

Mutual Exclusion: `critical`

- The `critical` directive protects a code section so that only one thread executes it at a time.
- Used when multiple operations need to be performed atomically.
- Multiple `critical` sections can share the same name to synchronize across them.
- Avoid using `critical` for simple updates — use `atomic` instead.

```
export OMP_NUM_THREADS=4; ./critical
Final sum = 4
```

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

int main(void)
{
  int sum = 0;

#pragma omp parallel
  {
#pragma omp critical
    {
      sum += 1;
    }
  }
  printf("Final sum = %d\n", sum);
  return 0;
}
```

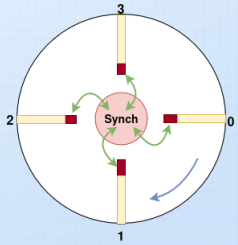
Named Critical Sections with `critical(name)`

- `critical(name)` allows the definition of distinct mutual exclusion zones.
- Only threads entering the same named section are mutually exclusive.
- Improves performance by avoiding unnecessary blocking between unrelated sections.
- All unnamed critical sections share the same implicit name.

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

int main(void)
{
  int sum = 0, prod = 1;

#pragma omp parallel
  {
#pragma omp critical
    sum += 1;

#pragma omp critical(RC1)
    prod *= 2;
  }

  printf("sum = %d ; prod = %d\n", sum, prod);
  return 0;
}
```

Memory Visibility: `flush`

- `flush` ensures that updates to a shared variable are made visible to all threads.
- Useful in architectures with hierarchical memory (e.g., caches).
- Without `flush`, a thread may see stale values from its private cache.



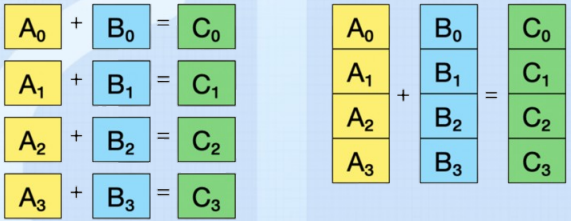

```
static int ready = 0;
int main(void)
{
#pragma omp parallel num_threads(2)
  {
    int tid = omp_get_thread_num();

    if (tid == 0) {
      ready = 1;
#pragma omp flush(ready)
    }
    else {
      while (1) {
#pragma omp flush(ready)
        if (ready == 1)
        break;
      }
      printf("Thread %d detected ready.\n", tid);
    }
  }
  return 0;}
```

SIMD VECTORIZATION

What is SIMD?

- SIMD = Single Instruction Multiple Data.
- A single instruction operates on multiple data elements simultaneously.

```
sum += A[i] * B[i]; // executed on 4-8 elements at ↩
      once
```



SISD vs. SIMD

SIMD VECTORIZATION

Loop Vectorization with `#pragma omp simd`

- `omp simd` splits the loop into chunks fitting vector registers.
- It enables vectorization without introducing thread-level parallelism.
- Can be used inside or outside a parallel region.
- Suitable for regular loops with no loop-carried dependencies.

```c
#include <stdio.h>
#include <omp.h>
#define N 500000

int main(void) {
  double A[N], B[N], sum = 0.0;
  for (int i = 0; i < N; i++) {
    A[i] = 1.0;
    B[i] = 2.0;
  }

#pragma omp simd reduction(+:sum)
  for (int i = 0; i < N; i++) {
    sum += A[i] * B[i];
  }

  printf("%f\n", sum);
  return 0;
}
```

SIMD VECTORIZATION

Combining Thread and Data Parallelism with `parallel for simd`

- The directive `parallel for simd` combines thread-level and vector-level parallelism.
- The loop is divided among threads, then vectorized within each thread.
- Improves performance by exploiting multicore and SIMD units together.
- The directive also creates the parallel region.

```c
#include <stdio.h>
#include <omp.h>
#define N 500000

int main(void) {
  double A[N], B[N], sum = 0.0;
  for (int i = 0; i < N; i++) {
    A[i] = 1.0;
    B[i] = 2.0;
  }

#pragma omp parallel for simd reduction(+:sum)
  for (int i = 0; i < N; i++) {
    sum += A[i] * B[i];
  }

  printf("%f\n", sum);
  return 0;
}
```

SIMD VECTORIZATION

Vectorizing Scalar Functions with `declare simd`

- `declare simd` generates a vector version of a scalar function.
- Enables the function to be used inside vectorized loops.
- Prevents vectorization from being blocked by scalar calls.
- Function is executed in vector mode on blocks of data.

```c
#define N 1000

#pragma omp declare simd
static inline double dist_f(double x, double y) {
  return sqrt(x*x + y*y);
}

int main(void) {
  double A[N], B[N], dist_max = 0.0;

  for (int i = 0; i < N; i++) {
    A[i] = (double)i;
    B[i] = (double)(2*i);
  }
#pragma omp parallel for simd reduction(max:dist_max)
  for (int i = 0; i < N; i++)
    dist_max = fmax(dist_max, dist_f(A[i], B[i]));

  printf("%f\n", dist_max);
  return 0;
}
```

Aligned vs Unaligned Memory Access

- The CPU cache is structured in fixed-size lines, typically 64 bytes each.
- SIMD instructions (e.g., AVX) load fixed-size memory blocks (e.g., 32 or 64 bytes). AVX-256 => 256 bits = 32 bytes
- For best performance, a SIMD load should start at an address aligned with the SIMD register size (e.g., 32 for AVX, 64 for AVX-512).
- If a SIMD load starts at an unaligned address (e.g., 5), it may cross a cache line boundary => In that case, the processor must fetch two cache lines, which can be slower than a single aligned access.

Improving Performance with `aligned` Clause

- Memory alignment improves SIMD efficiency by matching vector register size.
- The **`aligned`** clause informs the compiler that data is properly aligned.
- Avoids costly unaligned memory accesses.
- Especially effective on architectures with wide SIMD units (e.g., AVX-256).

```c
#define N 1024

int main() {
  // Align array A to 32 bytes
  double A[N] __attribute__((aligned(32)));
  double sum = 0.0;

#pragma omp simd aligned(A:32)
  for (int i = 0; i < N; i++)
    A[i] = i * 2.0;

#pragma omp simd aligned(A:32) reduction(+:sum)
  for (int i = 0; i < N; i++)
    sum += A[i];

  printf("Sum = %f\n", sum);
}
```

Controlling Vector Length with `safelen`

- The `safelen(N)` clause limits the number of iterations that can be executed simultaneously using SIMD.
- It helps prevent vectorization when there is a risk of data dependency across iterations beyond a safe distance.
- Useful when your loop contains computations that are correct only if iterations are no more than `N` steps apart.
- Enforcing a safe vector length allows better control over correctness and partial vectorization.

```
#define N 1024

int main() {
  double A[N], B[N];

#pragma omp simd safelen(4)
  for (int i = 0; i < N; i++) {
    A[i] = i * 2.0;
    B[i] = A[i] + 1.0;
  }

  printf("B[0] = %f, B[1023] = %f\n",  B[0], B[1023]);

  return 0;
}
```

SIMD VECTORIZATION

Branch-Free Conditional Execution

- Avoiding `if` statements inside loops improves SIMD efficiency.
- Conditional branches can interrupt the flow of vectorized execution.
- Using ternary operators (e.g., `? :`) helps maintain uniform control flow.
- This technique ensures that all SIMD lanes stay active without divergence.

```
#define N 1024

int main() {
  double A[N];

#pragma omp simd
  for (int i = 0; i < N; i++) {
    A[i] = (i % 2 == 0) ? i * 2.0 : i * 1.5;
  }

  printf("A[0] = %f, A[1] = %f\n",
  A[0], A[1]);

  return 0;
}
```

Why Tasks?

- Traditional OpenMP parallelism is based on the fork-join model.
- Efficient for data parallelism, but not well-suited for irregular or recursive computations.
- Introduced in OpenMP 3.0, the `task` construct enables flexible, dynamic parallelism.
- Tasks decouple work specification from execution, allowing better load balancing.

The Concept Bases — Task Types & Behavior

- A task = code + data, executed by one thread.
- Two kinds of tasks:
  - `Implicit`: created by `parallel`.
  - `Explicit`: created by `task`.
- Tasks offer more flexibility than the thread-based model.
- Well-suited for recursive or pointer-based workloads.

- Task Synchronization:
  - `taskwait`: waits for direct child tasks.
  - `taskgroup`: waits for all descendant tasks.
  - Barriers: wait for all tasks created before.
- Data sharing:
  - DSA is per task.
  - Exception: `threadprivate` is per thread.

The Task Execution Model

- Execution begins with only the master thread.
- Upon entering a `parallel` region:
  - A team of threads is created.
  - One implicit task is created for each thread.
- Upon encountering a `workshare` construct:
  - The work is distributed among threads (or implicit tasks).
- Upon encountering a `task` construct:
  - Explicit tasks are created.
  - Execution of these explicit tasks can be deferred.

Executing Explicit Tasks

- Explicit tasks are executed at specific task scheduling points:
  - `task`
  - `taskwait`
  - `barrier`
- At these points, any available thread may begin executing a pending task.
- A thread can switch from executing one task to another.
- At the end of a `parallel` region:
  - All tasks must complete before exiting the region.
  - Only one thread resumes execution of the sequential part.

Parallel vs Sequential Execution

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

int main() {
#pragma omp parallel
{
  printf("Started studying\n");
  printf("Attended a lecture\n");
  printf("Completed an assignment\n");
}
return 0;
}
```

```
export OMP_NUM_THREADS=2; ./task
Started studying
Attended a lecture
Started studying
Completed an assignment
Attended a lecture
Completed an assignment
```

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

int main() {
  printf("Started studying\n");
  printf("Attended a lecture\n");
  printf("Completed an assignment\n");
  return 0;
}
```

```
export OMP_NUM_THREADS=2; ./task
Started studying
Attended a lecture
Completed an assignment
```

First Task-Based Execution

- Tasks are created inside a `single` block to avoid duplication.
- Each `task` is executed asynchronously by any available thread.
- Execution order of tasks is not guaranteed.

```
export OMP_NUM_THREADS=2; ./task_print; ./task_print

Started studying
Attended a lecture
Completed an assignment

Started studying
Completed an assignment
Attended a lecture
```

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

int main() {
#pragma omp parallel
{
#pragma omp single
  {
    printf("Started studying.\n");
#pragma omp task
    printf("Attended a lecture.\n");
#pragma omp task
    printf("Completed an assignment.\n");
  }
}
return 0;
}
```

- How to always terminate the phrase with "Took the exam"?

Tasks May Not Finish Before the Next Statement

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

int main() {
#pragma omp parallel
  {
#pragma omp single
    {
      printf("Started studying\n");
#pragma omp task
      printf("Attended a lecture\n");
#pragma omp task
      {
        printf("Completed an assignment\n");
      }
      printf("Took the exam\n");
    }
  }
}
```

- No synchronization is used before `"Took the exam"`.
- Tasks are not guaranteed to have completed before this statement.
- Tasks are only executed at specific task scheduling points: `task`, `taskwait`, `barrier`.

Tasks May Not Finish Before the Next Statement

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

int main() {
#pragma omp parallel
  {
#pragma omp single
    {
      printf("Started studying\n");
#pragma omp task
      printf("Attended a lecture\n");
#pragma omp task
      {
        printf("Completed an assignment\n");
      }
      printf("Took the exam\n");
    }
  }
}
```

- No synchronization is used before `"Took the exam"`.
- Tasks are not guaranteed to have completed before this statement.
- Tasks are only executed at specific task scheduling points: `task`, `taskwait`, `barrier`.

```
export OMP_NUM_THREADS =2; ./ task_print2; ./ task_print2

Started studying
Took the exam
Attended a lecture
Completed an assignment

Started studying
Took the exam
Completed an assignment
Attended a lecture
```

Proper Synchronization with `taskwait`

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

int main() {
#pragma omp parallel
  {
#pragma omp single
    {
      printf("Started studying\n");
#pragma omp task
      printf("Attended a lecture\n");
#pragma omp task
      printf("Completed an assignment\n");
#pragma omp taskwait
      printf("Took the exam\n");
    }
  }
  return 0;
}
```

- `taskwait` ensures both tasks finish before executing `"Took the exam"`.
- Now the output is logically consistent.
- Useful for controlling task completion within a region.

Proper Synchronization with `taskwait`

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

int main() {
#pragma omp parallel
  {
#pragma omp single
    {
      printf("Started studying\n");
#pragma omp task
      printf("Attended a lecture\n");
#pragma omp task
      printf("Completed an assignment\n");
#pragma omp taskwait
      printf("Took the exam\n");
    }
  }
  return 0;
}
```

- `taskwait` ensures both tasks finish before executing `"Took the exam"`.
- Now the output is logically consistent.
- Useful for controlling task completion within a region.

```
Started studying
Completed an assignment
Attended a lecture
Took the exam

Started studying
Attended a lecture
Completed an assignment
Took the exam
```

Managing Dependencies Between Tasks

- The `depend` clause allows expressing dependencies between tasks.
- The second task will not execute before the first one completes.
- This ensures a specific execution order without requiring `taskwait`.

```
Started studying
Attended a lecture
Completed an assignment
All tasks done

Started studying
Attended a lecture
Completed an assignment
All tasks done
```

```c
int main() {
  int T1, T2;

#pragma omp parallel
  {
#pragma omp single
    {
      printf("Started studying\n");
#pragma omp task depend(out:T1)
      {
        printf("Attended a lecture\n");
      }
#pragma omp task depend(in:T1)
      {
        printf("Completed an assignment\n");
      }
#pragma omp taskwait
      printf("All tasks done\n");
    }
  }
  return 0;}
```

Default Data-Sharing Attributes (DSA)

- Implicit tasks (e.g., threads in `parallel`): variables are `shared` by default.
- Explicit tasks (created by `#pragma omp task`):
  - If the variable is shared in the parent task, it stays `shared`.
  - Otherwise, the default DSA is `firstprivate` (i.e., a copy is made).
- You can override default DSA with:
  - `shared(list)`, `private(list)`, `firstprivate(list)`.
  - Or specify global behavior using: `default(shared)`, `default(private)`, etc.

Default Behavior

- Variable `val` is declared outside the `parallel` region.
- It is therefore shared by all threads, including tasks.
- The task modifies `val`; this change is visible globally.
- The implicit barrier at the end of the parallel region waits for all explicit tasks created by the thread team.

```
Initial value: 10
Task sees: 10
Final value: 42
```

```c
int main() {
  int val = 10;
  printf("Initial value: %d\n", val);

#pragma omp parallel
  {
#pragma omp single
    {
#pragma omp task
      {
        printf("Task sees: %d\n", val);
        val = 42;
      }
    }
  }

  printf("Final value: %d\n", val);
  return 0;
}
```

Using `firstprivate(val)` in a Task

- The variable `val` is copied into the task.
- The task operates on its own private copy.
- The change to `val` does not affect the original.
- Output order may vary if no `taskwait` is used.

```
Initial value: 10
Task sees: 10
Final value: 10
```

```c
int main() {
  int val = 10;
  printf("Initial value: %d\n", val);

#pragma omp parallel
  {
#pragma omp single
    {
#pragma omp task firstprivate(val)
      {
        printf("Task sees: %d\n", val);
        val = 42;
      }
    }
  }
  printf("Final value: %d\n", val);
  return 0;
}
```

Fibonacci Numbers (Recursive Function)

- Recursive parallelism builds a binary task tree.
- Each recursive call spawns two new tasks.
- Variables `i` and `j` must be **`shared`** to accumulate result.
- **`taskwait`** ensures that both sub-results are computed before summing.
- This model is typical for divide-and-conquer strategies.

```
res_fib = 55
```

```
int fib(int n) {
  int res, i, j;
  if (n < 2) return n;
  else {
#pragma omp task shared(i)
    i = fib(n - 1);
#pragma omp task shared(j)
    j = fib(n - 2);
#pragma omp taskwait
    res = i + j;
  }
  return res;}
```

```
int main() {
  int nn = 10, res_fib;
#pragma omp parallel
  {
#pragma omp single
    res_fib = fib(nn);
  }
  printf("res_fib = %d\n", res_fib);
  return 0;}
```

Task Overhead in Recursive Divide and Conquer

- In recursive algorithms like Fibonacci, a large number of tasks are generated dynamically.
- Each recursive call creates two subtasks — forming a binary tree of tasks.
- As recursion goes deeper, the size of the subproblems decreases.
- Near the leaves of the tree, tasks become too small to justify their creation.
- These small tasks increase:
  - runtime scheduling overhead,
  - context switching cost,
  - and memory pressure.
- => This overhead can dominate computation time and reduce efficiency.

Optimizing Fibonacci with `final` and `mergeable`

- `final(n <= 5)` disables task creation for small subproblems.
- `mergeable` allows merging the task context with its parent.
- This reduces task management overhead near the leaves of recursion.
- Useful to avoid performance loss from too many tiny tasks.
- The result remains identical; only the performance improves.

```
res_fib = 55
```

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

int fib(int n) {
  int res, i, j;
  if (n < 2) return n;
  else {
#pragma omp task shared(i) final(n <= 5) mergeable
    i = fib(n - 1);
#pragma omp task shared(j) final(n <= 5) mergeable
    j = fib(n - 2);
#pragma omp taskwait
    res = i + j;
  }
  return res;
}
```

Manual Cut-off Strategy in Recursive Tasks

- Instead of relying on `final`, use an explicit `if` condition to avoid task creation for small values of `n`.
- When `n <= 5`, compute sequentially.
- This gives more control and ensures portability across compilers.
- Avoids creating too many fine-grained tasks near the base case.
- Slightly more verbose, but often more efficient and predictable.

```
res_fib = 55
```

```
int fib(int n) {
int res, i, j;
if (n < 2) return n;

if (n <= 5) {
  i = fib(n - 1);
  j = fib(n - 2);
} else {
#pragma omp task shared(i)
  i = fib(n - 1);
#pragma omp task shared(j)
  j = fib(n - 2);
#pragma omp taskwait
}

res = i + j;
return res;
}
```

Synchronization with `taskgroup`

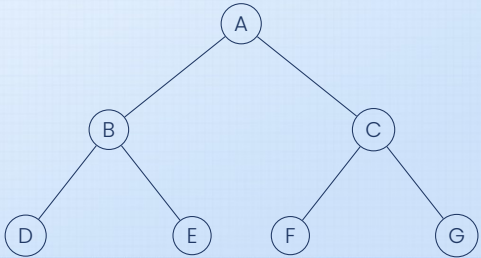
- `taskgroup` waits for all tasks inside the block, including nested tasks.
- Unlike `taskwait`, it captures the entire task tree of descendants.
- Ensures correct synchronization in complex hierarchies of tasks.
- Useful for divide-and-conquer patterns with recursive subtask creation.

```
Start
Task 1
Task 2
Task 2.1
End of group
```

```
int main() {
#pragma omp parallel
  {
#pragma omp single
    {
      printf("Start\n");
#pragma omp taskgroup
      {
#pragma omp task
        printf("Task 1\n");

#pragma omp task
        {
          printf("Task 2\n");
#pragma omp task
          printf("Task 2.1\n");
        }
      }
      printf("End of group\n");
    }
  }
  return 0;}
```

Binary Tree Traversal with `taskgroup`

- Each node of the tree spawns recursive traversal tasks for its left and right subtrees.
- A task is also created to process the current node.
- A `taskgroup` ensures that all three tasks (left, right, process) complete before returning.
- This guarantees depth-first synchronization.



- For node A:
  - Task 1: traverse left subtree (B)
  - Task 2: traverse right subtree (C)
  - Task 3: process node A
- The `taskgroup` waits for all 3 tasks to finish before returning from A.

Parallel Binary Tree Traversal with `taskgroup`

- Each node creates:
  - A task for the left child (if exists),
  - A task for the right child (if exists),
  - A task to process the current node.
- All are synchronized using a `taskgroup`.
- This structure ensures that no node returns before its children and itself are processed.

```
typedef struct TreeNode {
  struct TreeNode* left;
  struct TreeNode* right;
} TreeNode;
void process_leaf(TreeNode* node) {
  printf("Processing node at %p\n", (void*)node);
}
void traverse_tree(TreeNode* node) {
  if (node == NULL) return;

#pragma omp taskgroup
  {
    if (node->left) {
#pragma omp task
      traverse_tree(node->left);
    }
    if (node->right) {
#pragma omp task
      traverse_tree(node->right);
    }
#pragma omp task
    process_leaf(node);
  }
}
```

Main Function to Launch Parallel Tree Traversal

- A complete binary tree is initialized to a given depth.
- Only one thread (via `single`) starts the traversal.
- Tasks are created recursively for each node.
- The tree is freed after traversal.

```
Start traversal...
Processing node at 0x55b3a0d2a260
...
Done.
```

```
TreeNode* init_tree(int depth) {
  /* initiate the tree */
}

void free_tree(TreeNode* node) {
  /* free the tree */
}

int main() {
  TreeNode* tree = init_tree(3);

  printf("Start traversal...\n");

#pragma omp parallel
  {
#pragma omp single
    traverse_tree(tree);
  }

  printf("Done.\n");
  free_tree(tree);
  return 0;
}
```

Task Cancellation

- OpenMP 4.0 introduced the `#pragma omp cancel` directive.
- It allows cancelling tasks, taskgroups, parallel regions, or loops.
- When `cancel taskgroup` is executed, all tasks in that group are cancelled.
- Remaining tasks in the group may never execute.
- Useful for search algorithms or early exit strategies.

```
Task 1 started
Task 2 running
```

```
int main() {
#pragma omp parallel
  {
#pragma omp single
    {
#pragma omp taskgroup
      {
#pragma omp task
        {
          printf("Task 1 started\n");
#pragma omp cancel taskgroup
          printf("Task 1 canceled\n"); // likely ↩
               not printed
        }
#pragma omp task
        {
          printf("Task 2 running\n");
        }
      }
    }
  }
  return 0;
}
```

Loop Parallelism with `taskloop` (OpenMP 4.5)

- `taskloop` distributes loop iterations as separate tasks.
- Tasks are created by a single thread and then executed by any thread.
- Combines the benefits of loop parallelism and dynamic task scheduling.
- Especially useful for irregular iteration workloads or nested parallelism.

```
Task iteration 0 executed by thread 1
Task iteration 1 executed by thread 0
Task iteration 2 executed by thread 1
...
```

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

int main() {
  int N = 10;

#pragma omp parallel
  {
#pragma omp single
    {
#pragma omp taskloop
      for (int i = 0; i < N; i++) {
        printf("Task iteration %d executed by ↩
            thread %d\n",
        i, omp_get_thread_num());
      }
    }
  }
  return 0;
}
```

Summary of Task Constructs

Construct	Description
`task`	Creates an explicit task to be executed by any available thread.
`taskwait`	Waits for all child tasks to complete before continuing.
`taskgroup`	Synchronizes all tasks and their descendants within the group.
`depend(in, out, inout)`	Specifies task dependencies to enforce execution order.
`cancel`	Cancels running tasks or taskgroups when a certain condition is met.
`final(expression)`	Prevents child task creation when the condition is true, reducing task overhead.
`mergeable`	Allows merging a task with its parent, reducing overhead for small tasks.