Better Multi-Threading Programming? OpenMP

Jinzhen Wang
jwang96@charlotte.edu

Department of Computer Science
UNC Charlotte

Outline

OpenMP Concept

- Abbreviation for Open Multi-Processing.
- A shared-memory application programming interface (API).
- Used to direct multi-threaded shared memory parallelism.

OpenMP vs. Pthreads

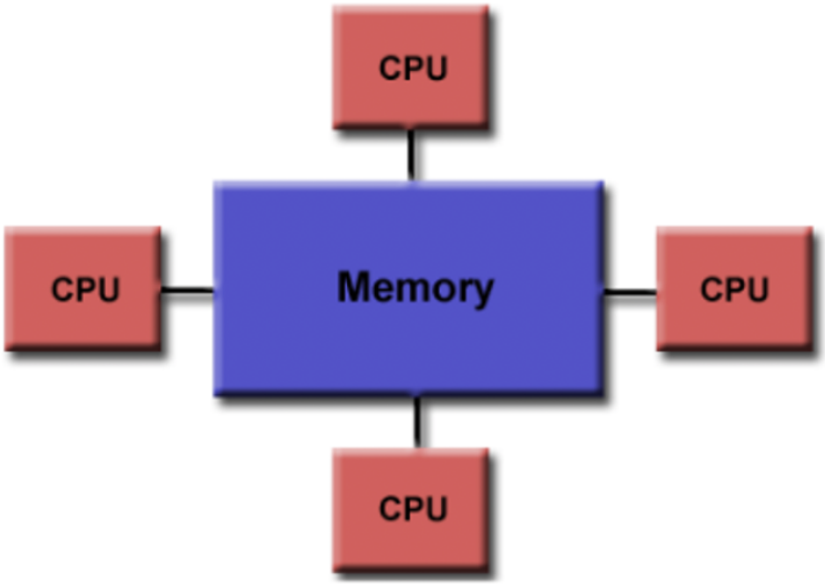
- Both are APIs for shared-memory programming.
- Pthreads require the programmer to explicitly specify the behavior of each thread.
- OpenMP allows the programmer to state that a block of code should be executed in parallel, and the compiler and runtime systems make decisions on actual execution.

What is NOT OpenMP?

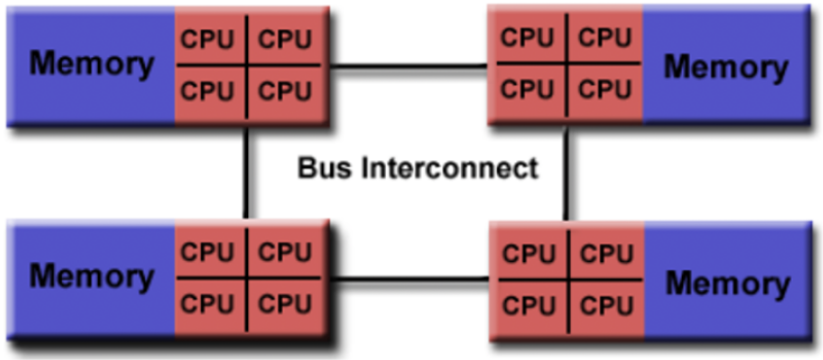
- Not designed for distributed memory parallel systems.
- Not necessarily implemented identically by all vendors.
- Not guaranteed to make the most efficient use of shared memory or CPU cores.
- Not required to check for data dependencies, data conflicts, race conditions, deadlocks, etc.
- Not designed to handle all parallel sections in your program, such as parallel I/O.
  - The programmer need to handle I/O and its synchronization.

Shared Memory Model

- OpenMP is designed for multi-processor/core, shared memory machines.
- The underlying architecture can be shared memory UMA or NUMA.



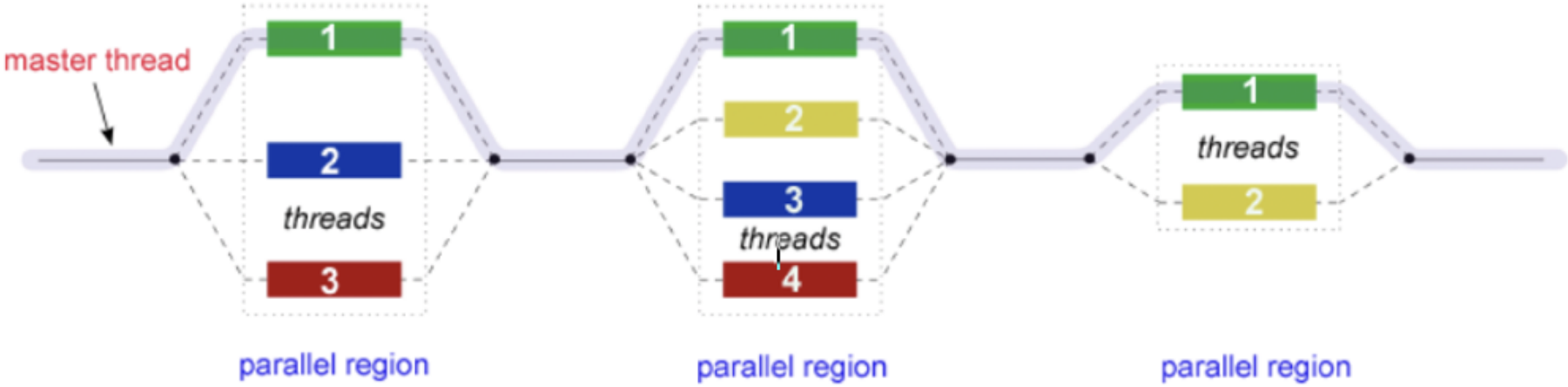
Uniform Memory Access



Non-Uniform Memory Access

OpenMP Programming Model

- OpenMP works based on threads.
  - It is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
  - Parallelization can be as simple as taking a serial program and inserting compiler directives.
- Fork-Join Model
  - Similar to Pthreads, OpenMP uses the fork-join model of parallel execution:



OpenMP 3 Components

- **Compiler Extension**:
  - Compiler directives such as `#pragma omp`.
  - They declares blocks of code to be parallel and how they should be scheduled. And the compiler inserts callbacks to the runtime.
- **Runtime Library**:
  - Manages thread pools, synchronization, task scheduling.
  - Include omp.h and compile with -fopenmp (in GCC).
- **Environment Variables**:
  - Controls execution of parallel code at run-time.

OpenMP Code Example: Hello World

Printing Helloword

- Hello World from multiple threads
- Include omp.h to use OpenMP library functions: omp_get_thread_num()

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main() {
5      #pragma omp parallel num_thread(5)
6      {
7          int thread_id = omp_get_thread_num();
8          printf("Hello World from thread %d\n", thread_id);
9      }
10     return 0;
11 }
```

OpenMP Code Example: Hello World

Printing Helloword

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main() {
5      #pragma omp parallel
6      {
7          int thread_id = omp_get_thread_num();
8          printf("Hello World from thread %d\n", thread_id);
9      }
10     return 0;
11 }
```

Example output

```
1 Hello World from thread 0
2 Hello World from thread 2
3 Hello World from thread 1
4 Hello World from thread 3
5 Hello World from thread 4
```

Compiling OpenMP Programs

- Different compilers might implement different versions of OpenMP.
- Compile with `-fopenmp` to enable OpenMP.
- For more information: `https://www.openmp.org/resources/openmp-compilers-tools/`

OpenMP Directives

`#pragma omp parallel default(shared) private(pi)`

- Format
  - `#pragma omp | directive-name | [clause, ...] | newline`
- #pragma omp
  - Required for all OpenMP C/C++ directives
- Directive name
  - A valid OpenMP directive
  - Must be after the pragma and before any clauses
- Clauses
  - Optional, any order
- Newline (required)

OpenMP Directives

- `#pragma omp parallel`:
  - Defines a parallel region.
- `#pragma omp for`:
  - Specifies a for-loop that is executed in parallel.
- `#pragma omp critical`:
  - Ensures a section of code is executed by only one thread at a time.
- `#pragma omp barrier`:
  - Synchronizes threads at this point in the code.
- `#pragma omp atomic`:
  - Ensures a specific memory location is updated atomically.

OpenMP Directives

- General Rules
  - Case sensitive
  - Directives follow conventions of the C/C++ standard for compiler directives
  - Only one directive-name may be specified per directive
  - Each directive applies to at most one succeeding statement, which must be a structured block
  - Long directive lings can be "continued" on succeeding lines by escaping the newline character with a backslash \ at the end of the directive line

Parallel Construct

- The parallel construct plays a crucial role in OpenMP/
- It specifies the computation that should be executed in parallel, *i.e.*, *where the fork starts*.
- All threads will execute the same parallel region (it is possible to use thread id to execute a different path though).
- There is an implied barrier at the end of a parallel region. Only the main thread continues execution past this point.
- If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is **undefined**.
- A program without a parallel construct will be executed sequentially.

Parallel Construct: Example 1

```
1  #include <omp.h>
2  #include <iostream>
3
4  int main () {
5    std::cout<<"Before"<<std::endl;
6
7  #pragma omp parallel num_thread(2)
8    {
9      std::cout<<"During"<<std::endl;
10   }
11   std::cout<<"After"<<std::endl;
12
13   return 0
14 }
```

Parallel Construct: Example 1

```
1  #include <omp.h>
2  #include <iostream>
3
4  int main () {
5    std::cout<<"Before"<<std::endl;
6
7  #pragma omp parallel num_thread(2)
8    {
9      std::cout<<"During"<<std::endl;
10   }
11   std::cout<<"After"<<std::endl;
12
13   return 0
14 }
```

Possible outcome

```
1 Before
2 During
3 During
4 After
```

Parallel Construct: Example 1

```
1  #include <omp.h>
2  #include <iostream>
3
4  int main () {
5    std::cout<<"Before"<<std::endl;
6
7  #pragma omp parallel num_thread(2)
8    {
9      std::cout<<"During"<<std::endl;
10   }
11   std::cout<<"After"<<std::endl;
12
13   return 0
14 }
```

Possible outcome

```
1 Before
2 During
3 During
4 After
```

Possible outcome

```
1 Before
2 DuringDuring
3
4 After
```

Parallel Construct: Example 2

```
1  #include <omp.h>
2  #include <iostream>
3  int main()
4  {
5    #pragma omp parallel
6    {
7      std::cout<<"The parallel region is executed by thread "
8          <<omp_get_thread_num()<<std::endl;
9      if (omp_get_thread_num() == 2)
10     {
11       std::cout<<omp_get_thread_num()
12         <<" does different things"<<std::endl;
13     }
14   }
15   return 0;
16 }
```

Parallel Construct: Example 2

```
1  #include <omp.h>
2  #include <iostream>
3  int main()
4  {
5    #pragma omp parallel
6    {
7      std::cout<<"The parallel region is executed by thread "
8          <<omp_get_thread_num()<<std::endl;
9      if (omp_get_thread_num() == 2)
10     {
11       std::cout<<omp_get_thread_num()
12         <<" does different things"<<std::endl;
13     }
14   }
15   return 0;
16 }
```

Thoughts

Possible outcomes? Assuming `OMP_NUM_THREADS=4`

Number of Threads

- How many threads to execute the parallel section?
- Following the order of precedence:
  - Setting of the NUM_THREAD clause
  - Use of the `omp_set_num_threads()` library function
  - Setting of the `OMP_NUM_THREADS` environment variable
  - Implementation default – usually the number of CPUs on a node, though it could be dynamic
- Threads are numbered from 0 (main thread) to N-1

Directive Scoping

- Variables declared outside a parallel section are shared among threads.
- Variables declared inside a parallel section are local to each thread.
- Manual control over scoping is possible with `#pragma parallel shared(var)`.

```
1 void f() {
2     int a;
3
4 #pragma omp parallel
5   {
6     // all threads here
          see the same 'a'
7   }
8 }
```

```
1 void f() {
2
3 #pragma omp parallel
4   {
5     int a;
6     // each thread sees
          its own 'a'
7   }
8 }
```

Manual Scoping

- `shared(list)`: Specifies variables that will be shared.
- `private(list)`: Each thread has its own copy of the variable.
- `lastprivate(list)`: The last value of a variable is accessible after the parallel region.
- `firstprivate(list)`: Variables are pre-initialized before the parallel region.
- `default(shared/private/none)`: Default data-sharing attribute.
  - In general, we recommend that the programmer not rely on the OpenMP default rules for data-sharing attributes.

Scoping Example - Private

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main (void){
4    int i = 10, a = 12;
5    #pragma omp parallel private (i, a) num_threads (4)
6    {
7      for (i=0; i < 2; i++){
8        a = i + 1;
9        printf("thread Id: %d, a: %d\n", omp_get_thread_num(), a);
10     }
11   }
12   printf("after parallel, a: %d, i: %d\n", a, i);
13 }
```

Scoping Example - Private

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main (void){
4    int i = 10, a = 12;
5    #pragma omp parallel private (i, a) num_threads (4)
6    {
7      for (i=0; i < 2; i++){
8        a = i + 1;
9        printf("thread Id: %d, a: %d\n", omp_get_thread_num(), a);
10     }
11   }
12   printf("after parallel, a: %d, i: %d\n", a, i);
13 }
```

Outcome?

Scoping Example - Private

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main (void){
4    int i = 10, a = 12;
5    #pragma omp parallel private (i, a) num_threads (4)
6    {
7      for (i=0; i < 2; i++){
8        a = i + 1;
9        printf("thread Id: %d, a: %d\n", omp_get_thread_num(), a);
10     }
11   }
12   printf("after parallel, a: %d, i: %d\n", a, i);
13 }
```

Outcome?

```
1 thread Id: 0, a: 1
2 thread Id: 0, a: 2
3 thread Id: 1, a: 1
4 ...
5 after parallel, a: 12, i: 10
```

Scoping Example - FirstPrivate

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main (void){
4    int i = 1, a = 12;
5    #pragma omp parallel firstprivate (i, a) num_threads (4)
6    {
7      for (i; i < 2; i++){
8        a = i + 1;
9        printf("thread Id: %d, a: %d\n", omp_get_thread_num(), a);
10     }
11   }
12   printf("after parallel, a: %d, i: %d\n", a, i);
13 }
```

Scoping Example - FirstPrivate

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main (void){
4    int i = 1, a = 12;
5    #pragma omp parallel firstprivate (i, a) num_threads (4)
6    {
7      for (i; i < 2; i++){
8        a = i + 1;
9        printf("thread Id: %d, a: %d\n", omp_get_thread_num(), a);
10     }
11   }
12   printf("after parallel, a: %d, i: %d\n", a, i);
13 }
```

Outcome?

Scoping Example - FirstPrivate

```c
#include <omp.h>
#include <stdio.h>
int main (void){
  int i = 1, a = 12;
  #pragma omp parallel firstprivate (i, a) num_threads (4)
  {
    for (i; i < 2; i++){
      a = i + 1;
      printf("thread Id: %d, a: %d\n", omp_get_thread_num(), a);
    }
  }
  printf("after parallel, a: %d, i: %d\n", a, i);
}
```

Outcome?

```
thread Id: 0, a: 1
thread Id: 1, a: 2
thread Id: 2, a: 1
...
after parallel, a: 12, i: 1
```

Scoping Example - LastPrivate

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main (void){
4    int i = 1, a = 12;
5    #pragma omp parallel lastprivate (a) num_threads (4)
6    {
7      for (i=0; i < 8; i++){
8        a = i + 100*omp_get_thread_num();
9        printf("thread Id: %d, a: %d\n", omp_get_thread_num(), a);
10     }
11   }
12   printf("after parallel, a: %d, i: %d\n", a, i);
13 }
```

Scoping Example - LastPrivate

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main (void){
4    int i = 1, a = 12;
5    #pragma omp parallel lastprivate (a) num_threads (4)
6    {
7      for (i=0; i < 8; i++){
8        a = i + 100*omp_get_thread_num();
9        printf("thread Id: %d, a: %d\n", omp_get_thread_num(), a);
10     }
11   }
12   printf("after parallel, a: %d, i: %d\n", a, i);
13 }
```

Outcome?

Scoping Example - LastPrivate

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main (void){
4    int i = 1, a = 12;
5    #pragma omp parallel lastprivate (a) num_threads (4)
6    {
7      for (i=0; i < 8; i++){
8        a = i + 100*omp_get_thread_num();
9        printf("thread Id: %d, a: %d\n", omp_get_thread_num(), a);
10     }
11   }
12   printf("after parallel, a: %d, i: %d\n", a, i);
13 }
```

Outcome?

```
1 For thread Id: 0, a will be: 0, 1, 2, 3, 4, 5, 6, 7.
2 For thread Id: 1, a will be: 100, 101, 102, ... 107.
3 ...
4 After parallel, a: 307, i: 1
```

Scoping Example - LastPrivate

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main (void){
4    int i = 1, a = 12;
5    #pragma omp parallel lastprivate (a) num_threads (4)
6    {
7      for (i=0; i < 8; i++){
8        a = i + 100*omp_get_thread_num();
9        printf("thread Id: %d, a: %d\n", omp_get_thread_num(), a);
10     }
11   }
12   printf("after parallel, a: %d, i: %d\n", a, i);
13 }
```

Does the value of a depends on the launch sequence?

Scoping Example - LastPrivate

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main (void){
4    int i = 1, a = 12;
5    #pragma omp parallel lastprivate (a) num_threads (4)
6    {
7      for (i=0; i < 8; i++){
8        a = i + 100*omp_get_thread_num();
9        printf("thread Id: %d, a: %d\n", omp_get_thread_num(), a);
10     }
11   }
12   printf("after parallel, a: %d, i: %d\n", a, i);
13 }
```

Does the value of a depends on the launch sequence?

The lastprivate clause in OpenMP does not depend on the order in which threads are launched but rather on the last logical iteration of the loop executed within each thread.

Synchronization Constructs

- Whenever multiple threads trying to update the same variable, synchronization is necessary.
- OpenMP provides a variety of **Synchronization Constructs** that control how the execution of each thread proceeds relative to other team threads.
  - **Main**: Executed only by the main thread.
  - **Single**: Executed by only one thread, not necessarily the main thread.
  - **Barrier**: Synchronizes threads, all must reach this point to proceed.
  - **Ordered**: Ensures the execution order of a specific block of code inside a parallel for loop.
  - **Critical**: Ensures a section of code is executed by only one thread at a time.
  - **Atomic**: Guarantees atomic updates to a memory location.
  - **Lock**: Provides mutual exclusion with locks.

Barrier Construct

```
#pragma omp barrier
```

- A point in the execution of a program where threads wait for each other
  - no one will proceed unless all threads have reached that point.
- Many OpenMP constructs imply a barrier.

Main and Single Construct

- These two define section of code that will be executed only ONCE
- For a main section, it is executed only by the main thread (Thd 0)

```
1  #include <omp.h>
2  #include <stdio.h>
3  int main(void) {
4    int num_threads, thread_id;
5    #pragma omp parallel private(thread_id)
6    {
7      #pragma omp main
8      {
9        num_threads = omp_get_num_threads();
10       printf("main thread %d: Number of threads = %d\n",
             omp_get_thread_num(), num_threads);
11     }
12     printf("Hello from thread %d\n", omp_get_thread_num());
13   }
14   return 0;
15 }
```

Main and Single Construct

- These two define section of code that will be executed only ONCE
- For a Single section, it is executed by any of the parallel threads (Often the first one comes)

```
1  #include <omp.h>
2  #include <iostream>
3  int main() {
4    int data = 0;
5    #pragma omp parallel num_threads(4)
6    {
7      #pragma omp single
8      {
9        std::cout << "This message is printed only once by thread
             : " << omp_get_thread_num() << std::endl;
10       data = 42; // Initialize shared data
11     }
12     #pragma omp barrier // Optional, implied barrier
13     std::cout << "Thread " << omp_get_thread_num() << " sees
           data = " << data << std::endl;
14   }
15   return 0;
16 }
```

Ordered Construct

`#pragma omp ordered`

- Execute a structured block within a parallel loop in sequential order.
- Only valid on the loop construct (parallel for).

```
1  int a[6]={0};
2  #pragma omp parallel for ordered schedule(option, [
       chunk size])
3  for(int i=0;i<6;i++)
4  {
5    a[i] += i;
6    //#pragma omp ordered
7    {
8      printf("a[%d]=%d\n", i, a[i]);
9    }
10 }
```

Critical Construct

`#pragma omp critical [(name)]`

- Specifies a region of code that must be executed by only one thread at a time.
- Ensure that multiple threads do not attempt to update the same shared data.
- When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name.

```
1  sum = 0;
2  #pragma omp parallel
3  {
4      sumLocal=0;
5      #pragma omp for
6      for (i=0; i<n; i++)
7          sumLocal += a[i];
8      #pragma omp critical (update_sum)
9      {
10         sum += sumLocal;
11     }
12 }
```

Atomic Construct

```
#pragma omp atomic
```

- Enables multiple threads to update shared data w.o. interference
- Only applies to the **single assignment statement** that immediately follows it.
- It actually is protecting an individual memory location denoted by the left-hand side of the assignment.
- Applies to all threads, not just the same team.
- Only work with expression statement in C/C++, such as $+, -, *, /, \ldots$

Atomic Construct

Example 1

```
1 int ic = 0, 1, n;
2 #pragma omp parallel for
3 {
4     for (i=0; i<n; i++)
5         #pragma omp atomic
6         ic += 1;
7 }
8 printf("counter = %d\n", ic);
```

Atomic Construct

Example 2

```
1  #include <omp.h>
2  #include <iostream>
3  int main() {
4    int shared_counter = 0;
5    // First parallel region
6    #pragma omp parallel num_threads(4){
7      for(int i=0; i<100; i++) {
8        #pragma omp atomic
9        shared_counter++;}}
10   std::cout << "Value of shared_counter after first parallel
         region: " << shared_counter << std::endl;
11   // Second parallel region
12   #pragma omp parallel num_threads(4){
13     for (int i = 0; i < 100; i++) {
14       #pragma omp atomic
15       shared_counter++;}}
16   std::cout << "Final value of shared_counter: " <<
         shared_counter << std::endl;
17   return 0;
18 }
```

Locks

- Supports simple locks and nested locks.
- void omp_func_lock(omp_lock_t *lck)
- Pretty much similar to mutex lock. Check the document.

Example 2

```
1  #include <omp.h>
2  #include <iostream>
3  int main() {
4    int shared_counter = 0;
5    omp_lock_t lock;
6    omp_init_lock(&lock);
7    #pragma omp parallel num_threads(4)
8    {
9      for (int i = 0; i < 100; i++) {
10       omp_set_lock(&lock);
11       shared_counter++;
12       omp_unset_lock(&lock);}}
13   omp_destroy_lock(&lock);
14   std::cout << "Final value of shared_counter: " <<
         shared_counter << std::endl;
15   return 0;
16 }
```

Work-Sharing Constructs

- **For**: Distributes loop iterations among threads.
- **Sections**: Divides execution among threads, each thread executes a section.

Parallel for construct

```
1 #pragma omp parallel for [clauses]
2 for (int i = 0; i < N; i++) {
3     // Loop body
4 }
```

- Iterations of a loop to be executed in parallel by multiple threads.
- The OpenMP runtime handles dividing the work among threads.
- Customization the behavior using clauses like schedule, private, shared, reduction, etc.

Parallel for construct: Private/Shared Clause

```
#pragma omp parallel for private(i)
for (int i = 0; i < N; i++) {
    // Each thread gets its own 'i'.
}
```

- Declares that each thread will have its own copy of the specified variables. Changes made by one thread do not affect the variables in other threads.

```
#pragma omp parallel for shared(a)
for (int i = 0; i < N; i++) {
    // 'a' is shared among threads.
}
```

- Specifies that the variables are shared among all threads. All threads can read and write to the shared variables.

Parallel for construct: Reduction Clause

```
1 int sum = 0;
2 #pragma omp parallel for reduction(+:sum)
3 for (int i = 0; i < N; i++) {
4     sum += i;
5 }
6 // The final value of 'sum' will be the total sum of all iterations
```

`reduction(operator: variable_list)`

- This clause is used for parallel reduction operations. It creates private copies of the variables for each thread and combines them at the end of the parallel region using the specified operator.

Identifier	Initializer	Combiner
+	omp_priv = 0	omp_out += omp_in
-	omp_priv = 0	omp_out += omp_in
*	omp_priv = 1	omp_out *= omp_in
&	omp_priv = ~ 0	omp_out &= omp_in
|	omp_priv = 0	omp_out |= omp_in
^	omp_priv = 0	omp_out ^= omp_in
&&	omp_priv = 1	omp_out = omp_in && omp_out
||	omp_priv = 0	omp_out = omp_in || omp_out
max	omp_priv = *Least representable number in the reduction list item type*	omp_out = omp_in > omp_out ? omp_in : omp_out
min	omp_priv = *Largest representable number in the reduction list item type*	omp_out = omp_in < omp_out ? omp_in : omp_out

Parallel for construct: Schedule Clause

```
#pragma omp parallel for schedule(clause, [chunk size])
```

- **static**: Iterations divided into chunks, assigned statically.
- **dynamic**: Iterations divided into chunks, assigned dynamically.
- **guided**: Like dynamic but chunk size decreases over time.
- **runtime**: Scheduling is determined at runtime.

Parallel `for` construct: Scheduling options

Option	Description	Workload distribution	Control over chunksize	Overhead	Use case
static	Divide the iterations into equal-sized chunks assigned to threads before the loop begins.	Precomputed, fixed chunks for each thread	Yes	Very low overhead because chunk assignment happens once before execution.	Best for workloads with uniform iteration times, where all iterations take roughly the same time.
dynamic	Iterations are distributed into chunks and assigned to threads as they become available.	Threads take the next available chunk after completing their current one.	Yes	Higher overhead due to dynamic assignment at runtime.	Best for workloads with uneven or unpredictable iteration times (e.g., when iterations vary in complexity).
guided	Similar to dynamic, but the chunk size decreases exponentially as threads complete their work.	Starts with larger chunks and decreases hunk size over time.	Min chunk size	Moderate overhead, as it combines dynamic distribution with decreasing chunk sizes.	Good for workloads that become lighter over time, where fewer iterations need to be handled as the loop progresses.
auto	The decision of how to distribute iterations is delegated to the OpenMP runtime system.	OpenMP runtime decides the best distribution.	No	Depends on the runtime's decision. Could be low or high.	When you're unsure of the best schedule to use and want the runtime to optimize based on system and workloads.
runtime	Scheduling is determined at runtime based on the value of the environment variable.	Defined by OMP_SCHEDULE environment variable at runtime.	Yes	Depends on the schedule selected by the environment variable.	When you want the flexibility to change the schedule without recompiling the program, allowing for experimentation

Other Clauses in Parallel For

- **nowait**: Threads do not synchronize at the end of the loop.
  - The nowait clause removes the implicit barrier that occurs at the end of a parallel loop.

```
#pragma omp parallel for nowait
for (int i = 0; i < N; i++) {
    // Loop body
}
```

- **collapse**: Collapses nested loops into a single loop.
  - Used to combine multiple (n) nested loops into a single loop for the purposes of iteration distribution among threads.

```
#pragma omp parallel for collapse(n)
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        // Loop body
    }
}
```

Task Directive

- Task parallelism in OpenMP allows you to define independent units of work, or `tasks`, which can be executed **concurrently by different threads**. Tasks are useful for dealing with irregular parallelism, where work is divided dynamically rather than being evenly distributed across threads, as in loop-based parallelism.
- OpenMP provides a tasking model using the `#pragma omp task` directive. Tasks can be nested, and dependencies between tasks can be explicitly defined, allowing for flexible and dynamic execution.

```
1  #pragma omp parallel
2  {
3      #pragma omp single
4      {
5          #pragma omp task
6          { /* Task 1 */ }
7          #pragma omp task
8          { /* Task 2 */ }
9          #pragma omp taskwait
10     }
11 }
```

Task Directive: Key Clauses

- shared(list): Variables in the list are shared among all tasks.
- private(list): Variables in the list are private to each task.
- firstprivate(list): Variables in the list are initialized with their value at the point of task creation.
- depend(in | out | inout : list): Specifies task dependencies, ensuring certain tasks are completed before others begin.
- if(condition): If the condition evaluates to false, the task is executed immediately by the current thread.

Task Directive: depend Clause

```
1  #pragma omp parallel {
2    #pragma omp single {
3      #pragma omp task { // Task 1
4        int thread_id = omp_get_thread_num();
5        printf("Task 1 executed by thread %d\n", thread_id);}
6      #pragma omp task { // Task 2
7        int thread_id = omp_get_thread_num();
8        printf("Task 2 executed by thread %d\n", thread_id);}
9      #pragma omp task { // Task 3
10       int thread_id = omp_get_thread_num();
11       printf("Task 3 executed by thread %d\n", thread_id);}
12     #pragma omp taskwait
13     printf("All tasks are done!\n");}
14   }
```

- Without task dependencies.

Task Directive: depend Clause

```
1  int a = 0, b = 0, c = 0;
2  #pragma omp parallel {
3    #pragma omp single{
4      #pragma omp task depend(out: a) { // Task 1: Initialize a
5        a = 10;
6        std::cout << "Task 1 (initialize a) executed by thread " <<
             omp_get_thread_num() << std::endl;}
7      #pragma omp task depend(out: b) { // Task 2: Initialize b
8        b = 20;
9        std::cout << "Task 2 (initialize b) executed by thread " <<
             omp_get_thread_num() << std::endl;}
10     #pragma omp task depend(in: a, b) depend(out: c) { // Task 3:
         Compute c = a + b (depends on Task 1 and Task 2)
11       c = a + b;
12       std::cout << "Task 3 (compute c = a + b) executed by thread "
              << omp_get_thread_num() << std::endl;}
13     #pragma omp taskwait // Wait for all tasks to finish
14     std::cout << "Final value of c: " << c << std::endl;
15   }
16 }
```

- With task dependencies

Task Directive: `if` Clause

Allows the specification of a condition under which a task should be deferred for parallel execution or executed immediately by the current thread. If true: the task is deferred for parallel execution; Else: task is executed immediately by the thread that encounters the task creation

```
1 #pragma omp task if(condition)
2 {
3     // Task body
4 }
```

Task Directive: if Clause

```
1  // Recursive function to compute Fib. numbers with task parallelism
2  int fib(int n) {
3      int x, y;
4      if (n < 2) { return n;} else {
5          // Create task for fib(n - 1), but only if n > 10
6          #pragma omp task shared(x) if (n > 10)
7          x = fib(n - 1);
8          // Create task for fib(n - 2), but only if n > 10
9          #pragma omp task shared(y) if (n > 10)
10         y = fib(n - 2);
11         // Ensure both tasks are completed before proceeding
12         #pragma omp taskwait
13         return x + y;}}
14 int main() {
15     int n = 20, result=0;  // Fibonacci number to calculate
16     // Parallel region for task creation
17     // Only one thread creates the initial task
18     #pragma omp parallel { #pragma omp single {result = fib(n);}}
19     std::cout << "Fibonacci("<<n<<") = " << result << std::endl;
20     return 0;}
```

Sections Construct

Specifies multiple, independent code sections that can be executed concurrently by different threads. Each section represents a block of code that can be executed by one thread in the team.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            // Code block for section 1
        }

        #pragma omp section
        {
            // Code block for section 2
        }

        // Add more sections if needed
    }
}
```

Task Construct vs Sections Construct

Key Differences Between `sections` and `task` Constructs

Feature	`sections` Construct	`task` Construct
Purpose	Divide independent tasks among threads at compile-time.	Divide work dynamically, enabling runtime scheduling.
Type of Workload	Best for parallelizing a fixed, predefined number of tasks.	Best for dynamic, recursive, or irregular workloads.
Thread Assignment	Threads are statically assigned one section each.	Tasks are dynamically assigned to available threads.
Implicit Barrier	Implicit barrier at the end of the `sections` block (can be avoided with `nowait`).	No implicit barrier unless you use `taskwait`.
Execution Control	All sections are distributed among the available threads and executed concurrently.	Tasks are placed in a task pool and executed asynchronously by available threads.
Use of Dependencies	No built-in support for dependencies between sections.	Supports task dependencies using `depend` clauses.

Summary

- OpenMP provides a simple and flexible interface for parallel programming in shared memory systems.
- Understanding the different constructs and directives is key to efficient parallel programming.
- Experimenting with different scheduling policies and optimizations can significantly impact performance.