Introduction of OpenMP

Contents

In this section, we will first introduce what OpenMP is and some advantages of using OpenMP, and then we will introduce the execution model and memory model of OpenMP.

1.1.1. What is OpenMP?

OpenMP is a standard parallel programming API for shared memory environments, written in C, C++, or FORTRAN. It consists of a set of compiler directives with a “lightweight” syntax, library routines, and environment variables that influence run-time behavior. OpenMP is governed by OpenMP Architecture Review Board (or OpenMP ARB), and is defined by several hardware and software vendors.

OpenMP behavior is directly dependent on the OpenMP implementation. Capabilities of this implementation can enable the programmer to separate the program into serial and parallel regions rather than just concurrently running threads, hides stack management, and provides synchronization of constructs. That being said OpenMP will not guarantee speedup, parallelize dependencies, or prevent data racing. Data racing, keeping track of dependencies, and working towards a speedup are all up to the programmer.

1.1.2. Why do we use OpenMP?

OpenMP has received considerable attention in the past decade and is considered by many to be an ideal solution for parallel programming because it has unique advantages as a mainstream directive-based programming model.

First of all, OpenMP provides a cross-platform, cross-compiler solution. It supports lots of platforms such as Linux, macOS, and Windows. Mainstream compilers including GCC, LLVM/Clang, Intel Fortran, and C/C++ compilers provide OpenMP good support. Also, with the rapid development of OpenMP, many researchers and computer vendors are constantly exploring how to optimize the execution efficiency of OpenMP programs and continue to propose improvements for existing compilers or develop new compilers. What’s more. OpenMP is a standard specification, and all compilers that support it implement the same set of standards, and there are no portability issues.

Secondly, using OpenMP can be very convenient and flexible to modify the number of threads. To solve the scalability problem of the number of CPU cores. In the multi-core era, the number of threads needs to change according to the number of CPU cores. OpenMP has irreplaceable advantages in this regard.

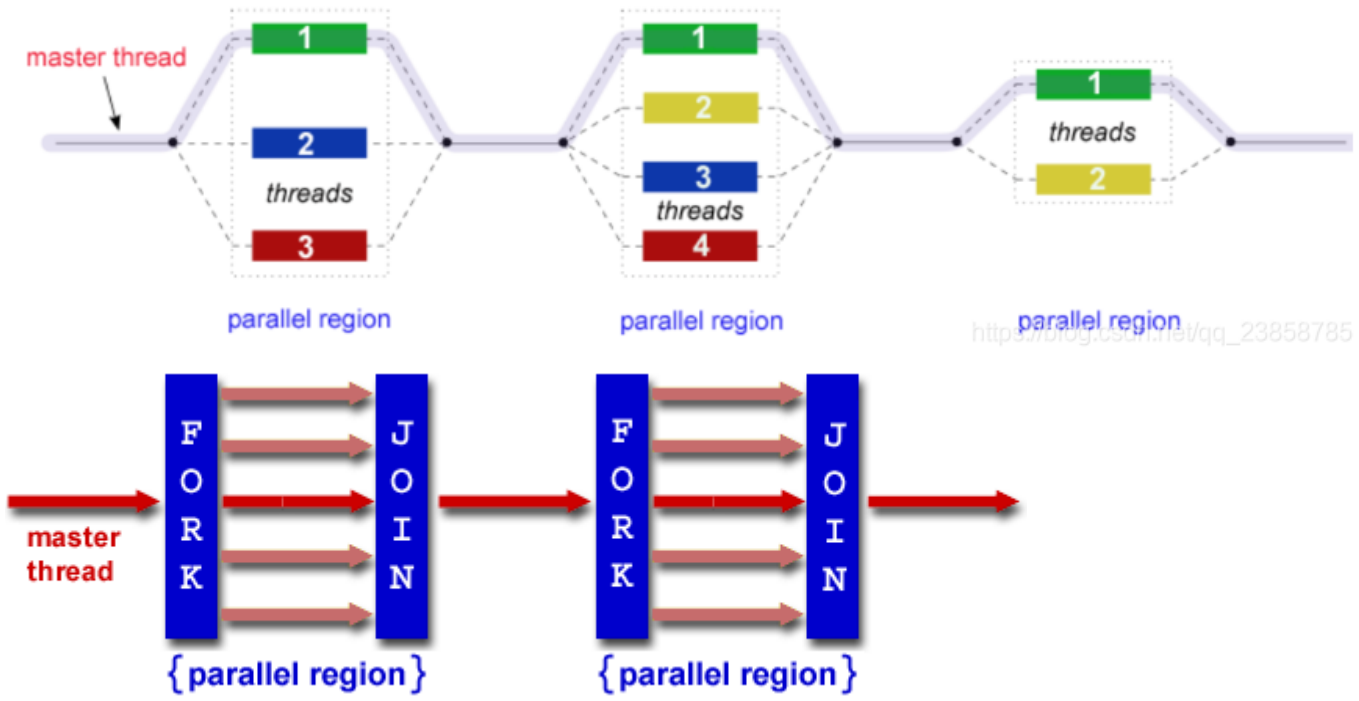
Thirdly, using OpenMP to create threads is considered to be convenient and relatively easy because it does not require an entry function, the code within the same function can be decomposed into multiple threads for execution, and a for loop can be decomposed into

multiple threads for execution. If OpenMP is not used, when the operating system API creates a thread, the code in a function needs to be manually disassembled into multiple thread entry functions.

To sum up, OpenMP has irreplaceable advantages in parallel programming. More and more new directives are being added to achieve more functions, and they are playing an important role on many different platforms.

1.1.3. OpenMP Fork-Join Excution Model

The parallel model used by OpenMP is called the fork-join model, as shown in the following figure:



Drwa a new figure, merge these 2 figures and shown the barrier in the new figure
All OpenMP programs start with a single thread which is the master thread. The master thread executes serially until it encounters the first parallel region. Parallel region is a block of code executed by all threads in a team simultaneously.

When the master thread encounters a parallel region, i.e. when it encounters an OpenMP parallel instruction, it creates a thread group consisting of itself and some additional (possibly zero) worker threads. This process called fork. These threads can execute tasks in parallel and are uniformly dispatched by the master thread. There is an implicit barrier at the end of the parallel region. When the thread has finished executing its task, it will wait at the barrier. When all threads have completed their tasks, the threads can leave the barrier. The master thread is left and continues to execute the serial code after the parallel region. This process called join.

Thread management is done by the runtime library, which maintains a pool of worker threads that can be used to work on parallel regions. It will allocate or reclaim the threads based on the OpenMP instructions and sometimes adjust the number of threads to allocate based on the availability of threads.

1.1.4. SPMD Program Models

The program model in the parallel region is Single Program, Multiple Data (SPMD). Processor Elements (PE) execute the same program in parallel, but has its own data. Each PE uses a unique ID to access its portion of data and different PEs can follow different paths through the same code. Each PE knows its own ID, and while programming, we can use conditional statements to control one or more threads. The commonly used format to follow is

```
if(my_id ==x) { }
else{}
```

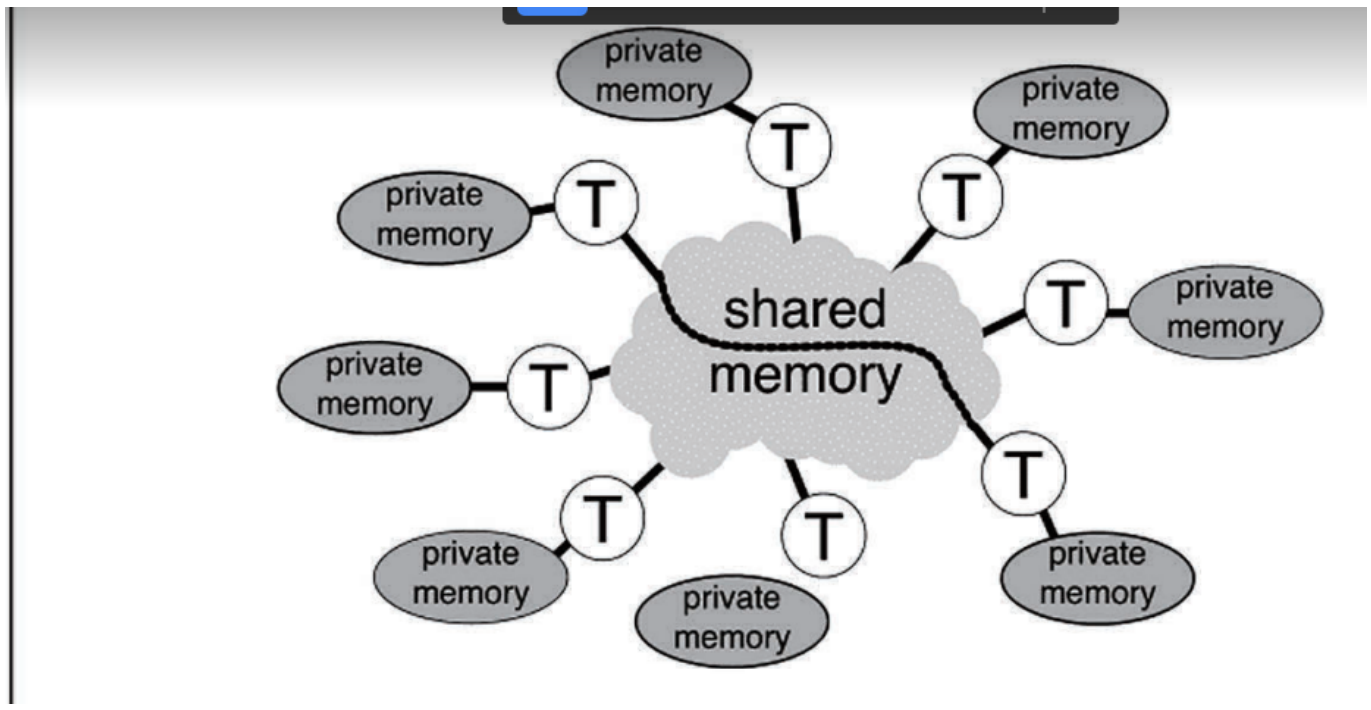
SPMD is by far the most commonly used pattern for structuring parallel programs. In addition to openmp, other common parallel programming languages, such as MPI that can be applied to distributed memory systems, and CUDA that can be applied to CPU+GPU heterogeneous systems, all adopt this model.

1.1.5. OpenMP Memory Model

At the beginning of this part, we briefly introduce two common memory models, namely shared memory and distributed memory. Shared memory means that multiple cores share a single memory, while distributed memory means that each computing node (possibly one or more cores) has its own memory. General large computers combine distributed memory and shared memory models, that is, shared memory within each computing node and distributed memory between nodes.

OpenMP supports the shared memory model that reduces execution time by creating multithreading to distribute parallelizable loads to multiple physical computing cores. Before the support for heterogeneous computing in OpenMP 4.0, there was only one address space.

The following figure shows the OpenMP memory model.



Let’s introduce some features of this model. First, all threads have access to the same, globally shared memory.

Second,OpenMP programs have two different basic types of memory: private and shared. Variables can also be divided into private variables and public variables. The type of the variable can be defined using an explicit private clause, or it can be defined according to default rules. Although the default rules are clear, we still recommend programmers explicitly define the types of variables, because using default rules does not always meet our expectations, and many errors may occur because of the subtlety of default rules.

Private variables are stored in private memory that is specially allocated to each thread, and other threads cannot access or modify private variables. Access conflicts will never happen when accessing the private variables. Even if there are other variables with the same name in the program or variables with the same name for multiple threads, these variables will be stored in different locations and will not be confused. The specific mechanism will be explained in detail when introducing the private clause.

There are two differences between shared variables and private variables. The first is that shared variables are not allowed to share one name. Each variable is unique. Secondly, shared variables are stored in memory that all threads can access, and any thread can access shared variables at any time. The control of access conflicts is the responsibility of the user.

The other two features of this memory model are that data transfer is transparent to the programmer and synchronization takes olace, but it is mostly implicit.

By Michael Womack, Anjia Wang, Patrick Flynn, Xinyao Yi, Yonghong Yan

Creating a Parallel Program with OpenMP

Contents

In this section, we will introduce the syntax of OpenMP, how to compile OpenMP programs, and give the readers an overall picture of OpenMP programs through two simple examples.

1.2.1. How to compile OpenMP programs?

When compiling OpenMP programs, you need to use comiler flags to enable OpenMP, such as –openmp, -xopenmp, -fopenmp, -mp

In this book, all our examples are compiled using LLVM/Clang on Ubuntu 20.04. LLVM/Clang has the advantages of fast compilation speed, less memory usage, and modular design. To execute OpenMP code, you need to add -fopenmp when compiling. The full command to compile is

```
clang -fopenmp filename.c -o filename.o
```

It is also worth mentioning that when writing OpenMP programs, you need to include the <omp.h> header file.

1.2.2. OpenMP Directives and Syntax

A series of directives and clauses in OpenMP identify code blocks as parallel regions. Programmers only need to insert these directives into the code, so OpenMP is defined as a kind of directive_based language. In C/C++, the directive is based on the #pragma omp construct. In Fortran, instructions begin with ! $omp. This is a regular language pragma (in C/C++) or a regular comment (in Fortran) for Compilers. So special option is need when the compiler is required to generate OpenMP code. Otherwise the compiler won't recognize it and simply ignore it.

The basic format of OpenMP directive in C/C++ is as follows:

```
#pragma omp directive-name [clause[ [,] clause]...]
```

In fortran, the directives take one of the forms:

Fixed forms:

```
*$OMP directive-name [clause[ [,] clause]...]
C$OMP directive-name [clause[ [,] clause]...]
```

Free form (but works for fixed form too):

```
!$omp directive-name [clause[ [,] clause]...]
```

Where '[]' means optional. A directive acts on the statement immediately following it or a block of statements enclosed by '{}'. Common diretcives are parallel, for, sections, single, atomic, barrier, simd, target, etc. Clause is equivalent to the modification of directive, which can be used to specify additional information with the directive. The specific clause(s) that can be used, depends on the directive.

In Fortran, OpenMP directives specify a paired end directive, where the directive-name of the paired end directives is:

• If directive-name starts with begin, the end-directive-name replaces begin with end

• otherwise it is end directive-name unless otherwise specified.

1.2.3. OpenMP Parallel Regions

As we have mentioned in the previous section, parallel region is a block of code executed by all threads in a team simultaneously. A block is a logically connected group of program statements considered as a unit. Next, let's take a look at how code blocks are defined in C/C++ and fortran. In C/C++, a block is a single statement or a group of statement between opening and closing curly braces. for example:

```
#pragma omp parallel
{
      id = omp_get_thread_num();
      res[id] = lots_of_work(id);
}
```

The two statements between the curly braces are a logical unit where one statement cannot be executed without the other being executed. They form a code block.

Another example:

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

Same with the above example, the two statements between the curly braces are a code block.

In Fortran, a block is a single statement or a group of statements between directive/end-directive pairs. For example:

```
!$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(.not.conv(res(id)) goto 10
!$OMP END PARALLEL
```

Another example:

```
!$OMP PARALLEL DO
    do i=1,N
        res(i)=bigComp(i)
    end do
!$OMP END PARALLEL DO
```

In fortran, OpenMP directives specify a paired end directive. Therefore, in the above two examples, the statements between the OpenMP directive and the corresponding end directive form a code block, which is also a parallel region.

1.2.4. Creating a simple OpenMP Program

This simple OpenMP program execute in parallel using multiple threads, and each thread outputs a ‘Hello World’.

```
//%compiler: clang
//%cflags: -fopenmp

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

int main(int argc, char *argv[]){
    #pragma omp parallel
    printf("%s\n", "Hello World");

    return 0;
}
```

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

“#pragma omp parallel” indicates that the subsequent statement will be executed by multiple threads in parallel, and the number of threads is preset by the system (generally equal to the number of logical processors, for example, i5 4-core 8-thread CPU has 8 logical processors),

If we want to specify the number of threads, we can add optional clauses to this directive, such as “#pragma omp parallel num_threads(4)” still means that the subsequent statement will be executed by multiple threads in parallel, but the number of threads is 4 .

```
//%compiler: clang
//%cflags: -fopenmp

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

int main(int argc, char *argv[]){
    #pragma omp parallel num_threads(4)
    printf("%s\n", "Hello World");

    return 0;
}
```

```
Hello World
Hello World
Hello World
Hello World
```

Through these two simple examples, I believe the readers already understood the basic structure of an OpenMP program. Next, we will introduce the factors that affect the performance of the OpenMP program.

By Michael Womack, Anjia Wang, Patrick Flynn, Xinyao Yi, Yonghong Yan

Performance Analysis

Contents

For parallel programs, performance is the primary concern. The relative ease of using OpenMP is a mixed blessing. We can quickly write a correct OpenMP program, but without the desired level of performance. Even though there are certain “best practices” to avoid common performance problems, extra work is needed to program with a large thread count. In this section, we briefly introduce some factors that affect performance when programming, and some “best practices” to improve performance.

1.3.1. Factors Impacting Performance

The OpenMP program includes serial regions and parallel regions, firstly, and the performance of the serial regions will affect the performance of the whole program. There are many optimization methods for serial programs, such as eliminating data dependencies, constant folding, copy propagation, removing dead code, etc. Although the compiler has made some optimization efforts for serial programs, in fact, for current compilers the effect of this optimization is minimal. A simpler and more effective method is to parallelize and vectorize the parts that can be parallelized or vectorized. For the purposes of this book, we won’t go into too much detail about the optimization of the serial parts.

The second factor is the proportion of the serial parts and the parallel parts. It is well understood that the more parts that can be parallelized in a program, the greater the room for program performance improvement. Amdahl’s law(or Amdahl’s argument) gives the theoretical speedup in latency of the execution of a task at a fixed workload that can be expected of a system whose resources are improved. This formula is often applied to parallel processing systems. For the speedup S describing the effect of parallel processing under a fixed load, Amdahl gave the following formula:

$$S = 1/((1-a) + a/n)$$

where **a** is the proportion of the parallel computing part, and **n** is the number of parallel processing nodes. In this way, when **1-a=0**, (ie, no serial, only parallel), the maximum speedup ratio **s=n**; when **a=0** (ie, only serial, no parallelism), the minimum speedup ratio **s=1**; when **n→∞**, the limit acceleration ratio **s→1/(1-a)**, which is also the upper limit of the acceleration ratio. For example, 90% of the code of a program is parallel, but there is still 10% of the serial code, the maximum speedup that can be achieved even by an infinite number of processors in the system is still 9. This formula has been accepted by academia , and it can indicate that the maximum speedup that a program can achieve is limited by the serial portion of the program.

Next, let’s analyze what factors can affect performance in the parallel regions. To analyze this problem, we can consider where the overhead of parallelism is, and then reduce the overhead or balance the overhead and benefits to obtain ideal performance. Common parallel overheads are mainly concentrated in two aspects. The first is the overhead of thread management. Thread management usually includes creating, resuming, managing, suspending, destroying and synchronizing. The management overhead of threads is often large compared to computation,

especially synchronization. Multiple threads waiting for synchronization cause a lot of computing resources to be wasted. The idea to solve this problem is to distribute tasks as evenly as possible, making the thread load as evenly as possible.

The second aspect is memory access, usually, memory is the main limiting factor in the performance of shared memory programs. Problems such as memory access conflicts will seriously affect the performance of the program. The access location of data is the key to the latency and bandwidth of memory access. The most commonly used method is to optimize data locality. We can improve this problem by explicitly managing the data which will be introduced in detail in the following chapters.

1.3.2. Ideas for Improving the performance

Based on the factors which affect the performance, there are several performance-enhancing efforts that can be considered:

- Thread load balancing
- Minimizing synchronization.
- Using more efficient directives. For example, using PARALLEL DO/FOR instead of worksharing DO/FOR directives in parallel regions.
- Expansion or merging of parallel regions
- Improving cache hit rate
- Optimization of round-robin scheduling
- Variable property optimization
- Optimization to reduce false sharing problem

In the following chapters of this book, we will introduce how to create parallel programs using OpenMP based on three different architectures, and explain how to optimize performance based on some of the ideas mentioned above.

By Michael Womack, Anjia Wang, Patrick Flynn, Xinyao Yi, Yonghong Yan

Parallel Programming for Multicore and Multi-CPU Machines

Moores's Law states CPUs double in frequency every two years. This was possible by increasing the transistor density. As time has gone on this has proven to be more difficult because with more transistors more heat is produced. As this problem became more apparent Computer Architects turned to another solution, a cheaper yet effective solution. Which was the implementation of multiple cores within a CPU. IBM was the first to release a Multi-core CPU back in 2001. Intel's attempt to release processors of this caliber came in 2002 with the Intel Pentium 4. This processor didn't implement parallel computing more or less just hyperthreading. This allowed it to switch between programs really fast so it gave the appearance of doing multiple processes at once. Since then more and more computers have been implementing multicore CPUs, some even have multiple CPUs. Because of this computer scientists needed to find a way to utilize these extra cores, thus was the birth of parallel computing.

In this book, we will be discussing multicore and multi CPU machines and how they work, how to use them, and how to optimize your code to use them effectively.

By Michael Womack, Anjia Wang, Patrick Flynn, Xinyao Yi, Yonghong Yan

Multicore and Multi-CPU shared memory systems

Contents

In this section, we will introduce the MIMD architecture, multi-CPU systems, multi-core systems, and multithreading. Through this section, the readers will understand the structure of computers for parallel computing. To lay the foundation for the introduction to OpenMP programming.

2.1.1. MIMD Architecture

Multiple Instruction Multiple Data(MIMD) architecture consists of processors that operate independently and asynchronously from one another. MIMD architecture systems consist of many processors that can have much common memory or memory that is exclusive to one. These terms mean many processes can execute instructions at any time with any piece or segment of data. MIMD architecture consists of two different types of architecture: Shared Memory MIMD architecture and Distributed Memory MIMD architecture.

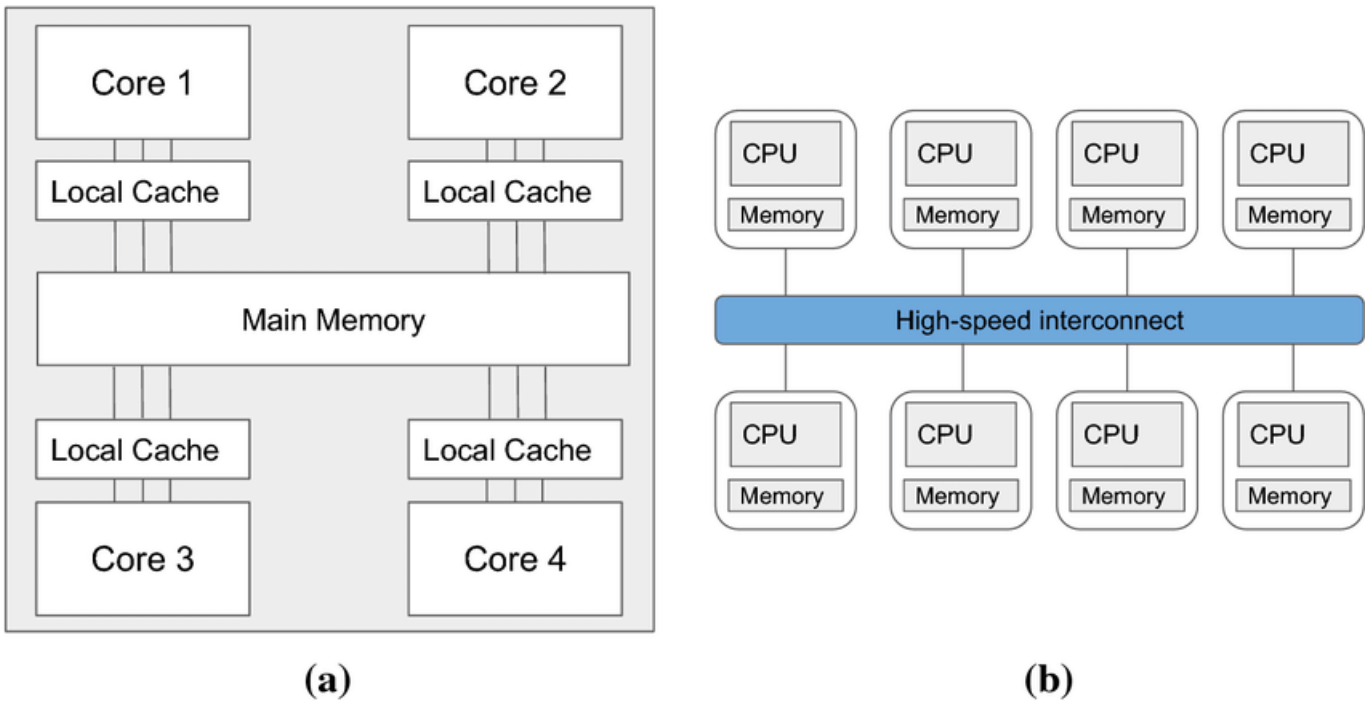
Shared Memory MIMD systems have the following characteristics. Shared Memory MIMD groups processors and memory. Any processor can access that memory through a communication channel designated by the programmer. Finally, the memory addresses are shared between processors. On the other hand, Distributed Memory MIMD systems have their own set of characteristics. First, memory and processors are grouped together in pairs known as processing elements. Which is linked to other PEs through the use of communication channels. Second, each PE communicates with each other by sending messages, communication isn't as open like that of a Shared Memory MIMD system.

The multi-CPU systems and multicore systems we introduce next both belong to the MIMD Architectures.

2.1.2. Multi-CPU Systems

A multiple CPU machine is made up of two or more CPUs, sharing system bus, memory and I/O. Multiple CPU machines were the original attempt at increasing computational power but since the birth of Multicore machines, most consumer computers have adopted that approach. That being said there is still a purpose for Multiple CPU machines, you will find them in supercomputers, servers, and other nonconsumer equipment that need raw computing power.

Multi-CPU systems can be divided into two types according to different memory systems, one uses a shared memory system, and the other uses a distributed memory system. As shown below:

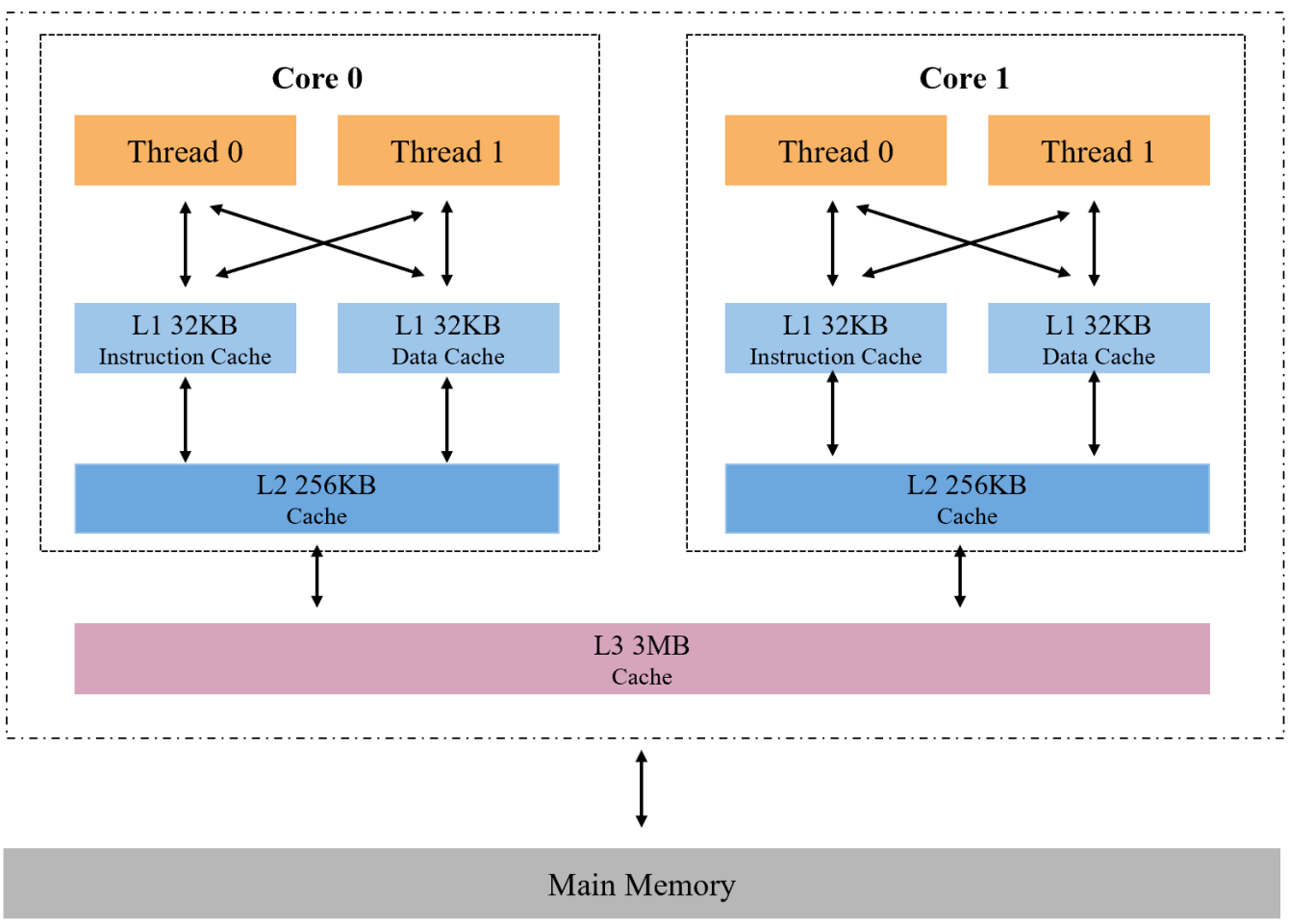


On the left is a diagram of a multi-core CPU, where each core has a small local cache, but all cores have direct access to large shared memory. The diagram on the right shows a distributed memory system where multiple CPUs, each with their own local memory, are connected via a high-speed network. Distributed memory systems are common in distributed clusters and can communicate using MPI. Since OpenMP only supports shared memory systems, we won't discuss distributed memory systems here.

2.1.3. Multicore Systems

Multicore machines just mean that the computer has "n" number of central processing units. Although these processors have multiple cores they still fit in one CPU slot on a motherboard all using the same power, cooling, and other hardware.

A multi-core example is shown below:



Only two cores are shown in this diagram, but real multi-core processors tend to use more cores. Multiple threads can be loaded onto a core for execution. Each core has L1 and L2 caches that only it can access, but they share the same memory system. For multi-core systems, the difference from multi-CPU systems is that all processors are on the same chip. The operating system treats each core as a separate processor, and each core can work independently.

2.1.4. Comparation of Multi-CPU Systems and

Multicore Systems

Both systems have their own advantages and disadvantages.

For multi-CPU systems:

Advantages:

- Multiple processors can work at the same time, and the throughput will be greatly increased.
- When a processor stops working, other processors can help to complete the work, which greatly improves the reliability of the entire system.

Disadvantages:

- Multiple processors work at the same time, and the coordination between them becomes very complicated.
- Buses, memory and I/O devices are shared. So if some processor is using some I/O, the other processor has to wait for its turn, which will result in lower throughput.
- Increased requirements for memory devices. In order to make all processors work efficiently at the same time, we need to have a large main memory, which increases the cost.

For multi-core systems:

Advantages:

- Multiple cores are on the same die, which results in higher cache coherence.
- The core is very energy efficient, so it can get more performance with less power consumption.

Disadvantages:

- We don't get as good performance as a multi-CPU system. For example, if we have two cores on our CPU, we can theoretically get twice the speedup compared to a single core, but in practice we can only get 70-80% of the ideal speedup.
- Some operating systems do not provide good support for multi-core systems.

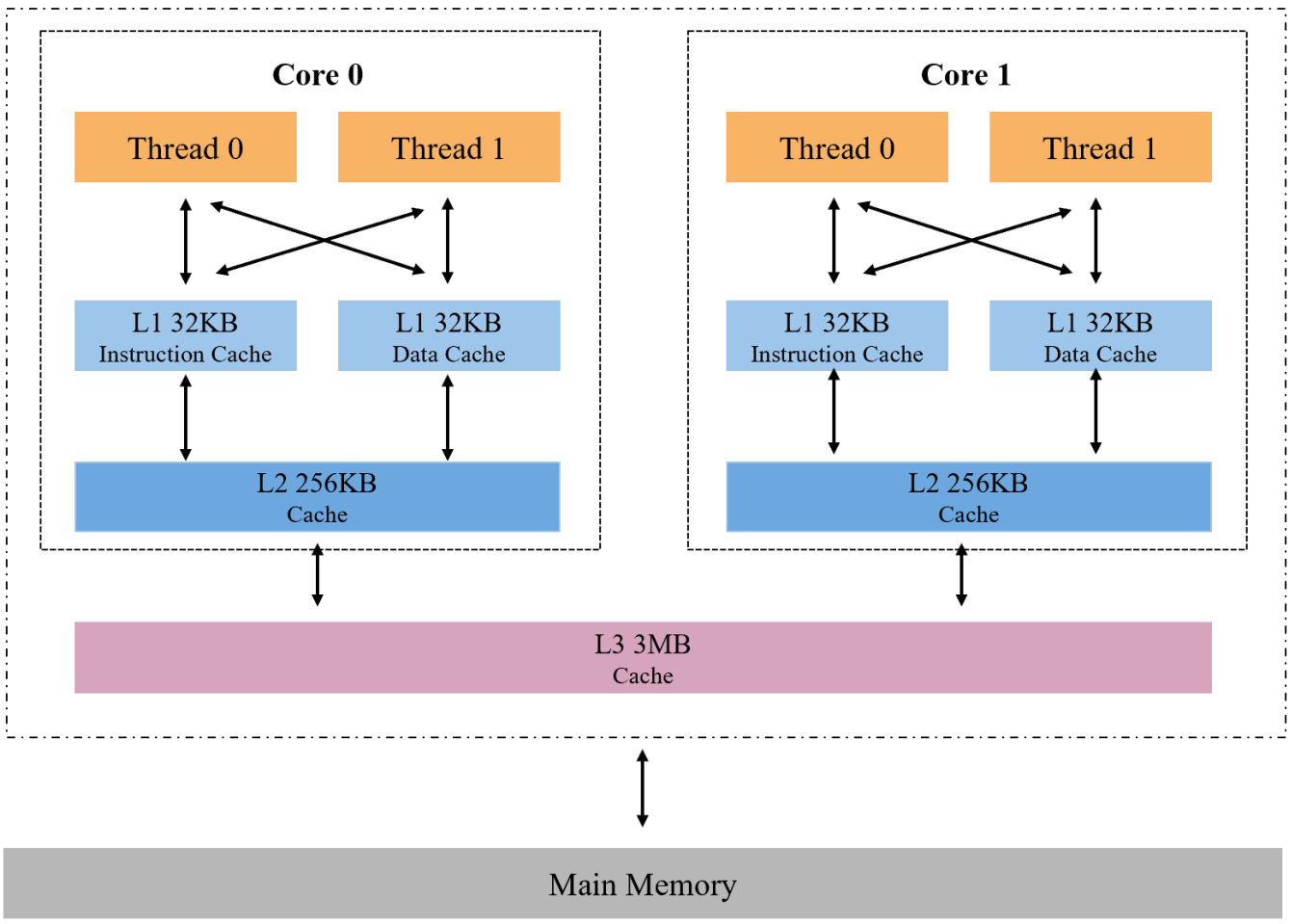
But the readers may have noticed that although the two approaches are slightly different in how fast the processors work together and how they access memory, their design is very similar. And the important thing is that the working logic of OpenMP is the same, which is to create multiple threads and allocate them on different cores. Therefore, we treat them as the same.

2.1.5. Multithreading

Next we will introduce a concept that is very important to OpenMP programming - multithreading. Multithreading is a technology that implements concurrent execution of multiple threads on software or hardware. Thread is usually a concept at the software level and is the actual operating unit in the operating system process. We can divide one huge task into multiple small tasks at the software level and load these tasks on different threads for execution.

The multi-threaded concurrency is a pseudo-concurrency in a single-core CPU, which is just a quick switch between multiple tasks through the scheduling strategy. Implementing multithreading on a single-core CPU is essentially an efficient use of CPU core, and try the best to make the CPU core always running at full capacity. The interaction with the memory is dragging down the program's execution speed. Moreover, multithreading can hide part of the interaction time through thread scheduling. The hyper-threading technology essentially simulates multiple logical units in one CPU core. It is an optimization of threads based on the premise that each thread in multithreading is not efficiently used to improve efficiency further. It is used on Intel's top CPUs to pursue the ultimate efficiency.

The multi-threading in the multi-core CPU is the actual parallel execution of threads. Even if a core is fully occupied, other cores are not affected and can handle thread scheduling. When the number of threads is less than the number of CPU cores, only some of the cores will work concurrently. When the number of threads is more than the number of cores, they will be loaded on the cores for execution according to a particular scheduling strategy.



We still use this figure as an example to show the architecture of a multi-core CPU that supports hyper-threading technology， Suppose there are 12 cores in the CPU, and we create 24 threads. Since the CPU shown in the figure supports hyper-threading technology, two threads are loaded into one core for execution, so 24 threads can be executed simultaneously. If we have 48 threads, 24 will be executed first and the others will be queued. Whenever a core completes its current task, the first two threads in the queue are loaded onto that core for execution. The cache in the CPU is an essential part of ensuring that threads can access data at high speed. There are two types of L1 caches which are L1i, which stores instructions, and L1d, which stores data. Different threads in the same core share the L1 and L2 cache, and different cores communicate through the L3 cache. The same thread can only run on one CPU core at one time.

By Michael Womack, Anjia Wang, Patrick Flynn, Xinyao Yi, Yonghong Yan

Creating SPMD parallelism using OpenMP parallel directive

Contents

From this part, we begin to introduce how to use OpenMP directives to write programs. We first introduce the most basic and most commonly used **parallel** directive.

2.2.1. Get Started with **Parallel** Directive to Create Parallelism

The **parallel** directive is used to mark a parallel region. When a thread encounters a parallel region, a group of threads is created to execute the parallel region. The original thread that executed the serial part will be the primary thread of the new team. All threads in the team execute parallel regions together. After a team is created, the number of threads in the team remains constant for the duration of that parallel region.

> Primary thread is also known as the master thread

When a thread team is created, the primary thread will implicitly create as many tasks as the number of threads, each task is assigned and bounded to one thread. When threads are all occupied, implicit tasks that have not been allocated will be suspended waiting for idle threads.

The following example from Chapter 1 shows how to use the parallel directive in C.

```c
//%compiler: clang
//%cflags: -fopenmp

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

int main(int argc, char *argv[]){
    #pragma omp parallel
    printf("%s\n", "Hello World");

    return 0;
}
```

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

This example prints *Hello World* 8 times, which means 8 threads are created by default. The default number of threads is determined by the computer hardware, 8 threads are created on the author's computer. The following example shows how to use the **num_threads** clause in the parallel directive to specify the number of threads to create.

```
//%compiler: clang
//%cflags: -fopenmp

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

int main(int argc, char *argv[]){
    #pragma omp parallel num_threads(4)
    printf("%s\n", "Hello World");

    return 0;
}
```

```
Hello World
Hello World
Hello World
Hello World
```

In this example, we use the **num_threads** clause to specify the use of 4 threads to execute the parallel region. When the master thread encounters OpenMP constructs, three threads are created, and together with these three threads, a thread group of 4 is formed. *Hello World* is printed four times, once per thread.

The next two examples show how to use the **parallel** directive in Fortran, and they have the exactly same meaning as the two examples in C above.

```
!!%compiler: gfortran
!!%cflags: -fopenmp

PROGRAM Parallel_Hello_World
USE OMP_LIB

!$OMP PARALLEL

  PRINT *, "Hello World"

!$OMP END PARALLEL

END
```

```
 Hello World
 Hello World
 Hello World
 Hello World
 Hello World
 Hello World
 Hello World
 Hello World
```

```
!!%compiler: gfortran
!!%cflags: -fopenmp

PROGRAM Parallel_Hello_World
USE OMP_LIB

!$OMP PARALLEL num_threads(4)

  PRINT *, "Hello World"

!$OMP END PARALLEL

END
```

```
 Hello World
 Hello World
 Hello World
 Hello World
```

2.2.2. Syntax and Semantics of **Parallel** Directive

and Its Clauses

Through the examples shown in the last section, it is not difficult to conclude that the syntax of the **parallel** directive in C is:

```
#pragma omp parallel [clause[ [,] clause] ... ] new-line
    structured-block
```

The syntax of the **parallel** directive in Fortran is:

```
!$omp parallel do [clause[ [,] clause] ... ]
    loop-nest
[!$omp end parallel do]
```

As we have already introduced in chapter 1, clauses are used to specify additional information with the directive. Ten clauses can be used with the **parallel** directive, listing as follows:

```
if([ parallel :] scalar-expression)
num_threads(integer-expression)
default(data-sharing-attribute)
private(list)
firstprivate(list)
shared(list)
copyin(list)
reduction([reduction-modifier ,] reduction-identifier : list)
proc_bind(affinity-policy)
allocate([allocator :] list)
```

2.2.2.1. **if** Clause

The **if** clause is used to achieve conditional parallelism. It can be used with many directives, such as **parallel** directive, **task** directive, **simd** directive, etc. Its effect depends on the construct to which it is applied. The syntax is as follows:

```
if([ directive-name-modifier :] scalar-expression)
```

or

```
if([ directive-name-modifier :] scalar-logical-expression)
```

The *directive-name-modifier* is optional and is very useful in combined constructs, which we will cover in the later chapters. When the **if** claue is used with the **parallel** directive, only *parallel* can be used as the *directive-name-modifier*. Its semantics are that the parallel region is active when the *scaler-expression* or *scalar-logical-expression* is true, otherwise the parallel region that follows will be inactive. At most one **if** clause can be used in each parallel construct.

2.2.2.2. **num_threads** Clause

The syntax for **num_threads** is as follows:

```
num_threads(integer-expression)
```

We used the **num_threads** clause in the previous example to indicate the number of threads used to execute parallel regions. The number of threads in the parallel region is determined by *integer-expression*. At most one **num_threads** clause can be used in each parallel construct. Because the number of threads used must be uniquely determined when entering the parallel region， whether explicitly specified by the programmer using **num_threads** clause or implicitly specified by the compiler. Of course, the *scaler-expression* must be an integer greater than zero.

2.2.2.3. Data-Sharing Attribute Clauses

Data-sharing attribute clauses are used to control the data-sharing attributes of variables.

There are four data-sharing attribute clauses that can be used with the **parallel** directive, namely **default** clause, **private** clause, **firstprivate** clause and **shared** clause. The **lastprivate** clause and **linear** clause are two other data-sharing attribute clauses. They cannot be used with **parallel** directives, so we will skip them for now and describe them in detail in later chapters.

We first introduce the **private** clause, **firstprivate** clause and **shared** clause.

2.2.2.3.1. **private** Clause

The syntax of the **private** clause is as follows:

```
private(list)
```

As mentioned before, a set of implicit tasks, equal in number to the number of threads in the team, is generated by the master thread when it encountering the parallel region. The **private** caluse is used to declare private variable(s) in the *list* for a task or a SIMD lane. A private variable has a different address in the execution context of every thread. These variables are private to threads, and threads cannot access private variables owned by other threads. Programmers can use the **private** clause as many times as needed.

2.2.2.3.2. **firstprivate** Clause

The **firstprivate** clause is very similar to the **private** clause. They both indicate that the variables in the *list* are private to the thread. Its syntax is as follows:

```
firstprivate(list)
```

But unlike the **private** clause, the variables in the *list* are initialized to the initial value that the original item had in the serial area. Like **private** clause, the **firstprivate** clause can be used multiple times in the parallel construct.

2.2.2.3.3. **shared** Clause

The **shared** clause is used to declare that one or more items in the *list* can be shared by tasks or SIMD lane. Its syntax is as follows:

```
shared(list)
```

Shared variables have the same address in the execution context of every thread. In a parallel region, all threads or SIMD lanes can access these variables. The **shared** clause can also be used multiple times within a parallel struct.

2.2.2.3.4. **default** Clause

The syntax of **default** clause：

```
default(data-sharing-attribute)
```

The **default** clause is used to define the default data-sharing attribute of variables in a parallel construct(it also can be used in a teams, or task-generating construct). The *data-sharing-attribute* is one of the following:

> private
> firstprivate
> shared
> none

When the data-sharing attribute of a variable is not specified, the data-sharing attribute of this variable will be set to the attribute specified in the **default** clause. If we have a variable *a* and the *data-sharing-attribute* in the **default** clause is *shared*, then we can understand it like this:

```
if(variable a is not assigned data-sharing attribute && we have clause
default(shared)) {
    equals to : we have clause shared(a)
}
```

A special note for the **default(none)** clause:

> The **default(none)** clause means that we do not define any data-sharing attribute for variables. The compiler does not implicitly define this for us, therefore, the variables need to be listed explicitly in other data-sharing attribute clauses.

Unlike the above three clauses, **default** clause can only appear once in a directive.

2.2.2.3.5. Implicit Scoping Rules

You may ask, if the **default** clause is not used, what data-sharing attribute is the variable that is not explicitly listed in other clauses? In fact, OpenMP defines a set of implicit scope rules. It determines the data-sharing attribute of a variable based on the characteristics of the variable being accessed. For example, for threads in a group, a variable is scoped to *shared* if using this variable in the parallel region does not result in a data race condition. A variable is scoped to *private* if, in each thread executing the parallel region, the variable is always written before being read by the same thread. The detailed rules can be found in the OpenMP specification.

Although OpenMP proposes as detailed and explicit rules as possible for implicit scoping, there is still a high possibility of unpredictable errors. Therefore, it is undoubtedly the best choice for programmers to use clauses to explicitly specify the data-sharing attributes of variables. This is an important aspect of OpenMP program optimization.

2.2.2.4. **copyin** Clause

The **copyin** clause is the one of two data copying clauses that can be used on the **parallel** construct or combined parallel worksharing constructs. The other one is the **copyprivate** clause which is only allowed on the **single** construct.

Before introducing these two clauses, we need to introduce a new data-sharing attribute, named *threadprivate*. The difference between private variables and thread private variables can be briefly described as follows:

- Private variables are local to a region and are placed on the stack most of the time. The lifetime of a private variable is the duration defined by the data scope clause. Every thread, including the main thread, makes a private copy of the original variable, and the new variable is no longer storage-associated to the original.
- Threadprivate variables are persist across regions, most likely to be placed in the heap or in thread-local storage, which can be seen as being stored in a local memory local to the thread. The main thread uses the original variable and other threads make private copies of the original variable. The host variable is still store-associated with the original variable.

The syntax of the **copyin** Clause is as follows:

```
copyin(list)
```

It can copy the value of the main thread's threadprivate variable into the threadprivate variable of every other member in the team that is executing the parallel region. And it can be used multiple times within a parallel struct.

2.2.2.5. **reduction** Clause

The **reduction** clause belongs to the *reduction scoping clauses* and the *reduction participating clauses*. The *reduction scoping clauses* define the region of a reduction computed by a task or a SIMD lane. The *reduction participating clauses* are used to define a task or SIMD channel as a

reduction participant. The **reduction** clause is specifically designed for reduction operations, and it allows the user to specify one or more thread-private variables that accept reduction operation at the end of the parallel region.

The syntax of the **reduction** clause is as follows:

```
reduction([ reduction-modifier,]reduction-identifier : list)
```

The *reduction-modifier* is optional and is used to describe the characteristics of the reduction operation. It can be one of the following:

> inscan
> task
> default

When the *reduction-modifier* is *inscan*, the list items on each iteration of a loop nest will be updated with scan computation. When *inscan* is used, one list item must be as a list item in an **inclusive** or **exclusive** clause on a **scan** directive enclosed by the construct. It separates the items in the list from the reduction operations, and decides whether the storage statement includes or excludes the scan input of the present iteration.

When the *reduction-modifier* is *task*, an indeterminate number of additional private copies will be generated to support task reduction, and these reduction-related copies will be initialized before they are accessed by the tasks.

When *reduction-modifier* is *default* or when no *reduction-modifier* is specified, the behavior will be relative to the construct in which the reduction is located. For **parallel**, **scope** and **simd** construcrs, one or more private copies of each list item are created for each implicit task (for **parallel** and **scope**) or SIMD lane (for **simd**), as if the **private** clause had been used. Some other rules for other constructs can be found in the OpenMP specification and we will not go into details here.

The *reduction-identifier* is used to specify the reduction operator. According to the OpenMP specification, the *reduction-identifier* has the following syntax:

- For C language， a *reduction-identifier* is either an identifier or one of the following operators: +, - (deprecated), *, &, |, ^, && and ||.
- For C++, a *reduction-identifier* is either an id-expression or one of the following operators: +, - (deprecated), *, &, |, ^, && and ||.
- For Fortran， a *reduction-identifier* is either a base language identifier, or a user-defined operator, or one of the following operators: +, - (deprecated), *, .and., .or., .eqv., .neqv., or one of the following intrinsic procedure names: max, min, iand, ior, ieor.

The following two tables, also from the OpenMP specification, show implicitly declared reduction-identifiers for numeric and logical types, including the initial value settings, and semantics for the reduction-identifiers.

For C/C++:

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
max	omp_priv = Minimal representable number in the reduction list item type	omp_out = omp_in > omp_out ? omp_in : omp_out
min	omp_priv = Maximal representable number in the reduction list item type	omp_out = omp_in < omp_out ? omp_in : omp_out

For Fortran:

Identifier	Initializer	Combiner
+	omp_priv = 0	omp_out = omp_in + omp_out
-	omp_priv = 0	omp_out = omp_in + omp_out
*	omp_priv = 1	omp_out = omp_in * omp_out
.and.	omp_priv = .true.	omp_out = omp_in .and. omp_out
.or.	omp_priv = .false.	omp_out = omp_in .or. omp_out
.eqv.	omp_priv = .true.	omp_out = omp_in .eqv. omp_out
.neqv.	omp_priv = .false.	omp_out = omp_in .neqv. omp_out
max	omp_priv = Minimal representable number in the reduction list item type	omp_out = max(omp_in, omp_out)
min	omp_priv = Maximal representable number in the reduction list item type	omp_out = min(omp_in, omp_out)
iand	omp_priv = All bits on	omp_out = iand(omp_in, omp_out)
ior	omp_priv = 0	omp_out = ior(omp_in, omp_out)
ieor	omp_priv = 0	omp_out = ieor(omp_in, omp_out)

The **reduction** clause can be used multiple times as needed within a parallel struct.

2.2.2.6. **proc_bind** Clause

The **proc_bind** clause is used to specify a mapping of OpenMP threads to places within the current place partition for implicit tasks of the encountering threads. At most one proc_bind clause can appear on the directive. Its syntax is as follows:

```
proc_bind(affinity-policy)
```

and the *affinity-policy* is one of the following:

> primary
> master [deprecated]
> close
> spread

When *affinity-policy* is specified as *primary*, it means that the execution environment assigns each thread in the group to the same location as the primary thread. The *master* affinity policy has been deprecated, it has identical semantics as *prime*.

The *close* thread affinity policy instructs the execution environment to assign threads in the group closer to the parent thread's location. When the number of threads **P** in the group is less than the number of locations **P** in the partition where the parent thread is located, each thread will be assigned one place, otherwise, **T/P** threads will be assigned to one place. When **P** does not divide **T** equally, the exact number of threads at a particular place is implementation-defined. The principle of allocation is that the thread number with the thread with the smallest thread number is executed at the position of the parent thread, and then the threads are allocated backward in the order of increasing thread number, and wrap-around with respect to the place partition of the primary thread.

The *spread* thread affinity policy is to create **P** sub-partitions in the parent partition and distribute the threads in these sub-partitions, thus achieving a sparse distribution. When the number of threads is equal to or less than **P**, the number of sub-partitions is equal to the number of threads **T**, and the threads are distributed in the first position of each partition in order of thread number from small to large. Conversely, when the number of threads is greater than **P**, **T/P** threads with consecutive thread numbers are allocated to each sub-partition. Sorted by thread number, the first **T/P** threads are allocated in the subpartition that contains the place of the parent thread. The remaining thread groups are allocated backward in turn, with a wrap-around with respect to the original place partition of the primary thread.

2.2.2.7. **allocate** Clause

The **allocate** clause is used to specify the memory allocator used to obtain storage of the private variables. When it is used with **parallel** construct, the syntax is shown as following:

```
allocate([allocator:] list)
```

In C/C++, the *allocator* is an expression of the omp_allocator_handle_t type. In Fortran, the *allocator* is an integer expression of the omp_allocator_handle_kindkind. We will not list these allocators here, readers can find them in the OpenMP specification, but we will explain some commonly used allocators in the examples in the following section.

The **allocate** clause can be used multiple times as needed within a parallel struct.

2.2.3. More Advanced Examples of Using Other Clauses and the Analysis of Performance and Improvement

In the above section, we have introduced the syntax and semantics of all the clauses that can be used with the parallel directive. In this section, we introduce some examples of how to use these clauses.

Meanwhile, in some of the examples, we will show some ways to improve performance, and discuss some potential possibilities for improving performance.

2.2.3.1. **if** clause and **num_threads** clause

The **if** clause can achieve conditional parallelism and the **num_threads** clause will identify the number of threads used. In the following example, we use three cases to display the effect of the if clause and the **num_trheads** clause. No **if** clause is used in case 1, and the **num_threads** clause indicated we should use 4 threads, so we can get the output from four threads in parallel. The scalar expression in the **if** clause in case 2 is evaluated to be false, so the statements

following the openMP directive (maybe code block in other cases) will be executed serially by only one thread. And in case 3, the scalar expression is evaluated to be true, so the statements following the openMP directive will be executed parallel by four threads.

```
//%compiler: clang
//%cflags: -fopenmp

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

int main(int argc, char *argv[]){
    int M =10;
    //case 1
    #pragma omp parallel num_threads(4)
    printf("Hello World in case 1 from thread %d\n", omp_get_thread_num());

    printf("-------------------------------------------------\n");

    //case 2
    #pragma omp parallel num_threads(4) if(M > 10)
    printf("Hello World in case 2 from thread %d\n", omp_get_thread_num());

    printf("-------------------------------------------------\n");

    //case 3
    #pragma omp parallel num_threads(4) if(M <= 10)
    printf("Hello World in case 3 from thread %d\n", omp_get_thread_num());

    return 0;
}
```

```
Hello World in case 1 from thread 2
Hello World in case 1 from thread 3
Hello World in case 1 from thread 1
Hello World in case 1 from thread 0
-------------------------------------------------
Hello World in case 2 from thread 0
-------------------------------------------------
Hello World in case 3 from thread 3
Hello World in case 3 from thread 1
Hello World in case 3 from thread 0
Hello World in case 3 from thread 2
```

Within a parallel region, the thread number uniquely identifies each thread. A thread can obtain its own thread number by calling the **omp_get_thread_num** library routine.

2.2.3.2. Data-Sharing Attribute Clauses

The following example is a little more complicated. It shows how to use the **omp_get_thread_num** library routine and shows how to use two other clauses, the **default** clause and the **private** clause. It assigns tasks to each thread explicitly.

```
//%compiler: clang
//%cflags: -fopenmp
//This example is from https://www.openmp.org/wp-content/uploads/openmp-examples-5.1.pdf. The size of array and output are changed

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

void subdomain(float *x, int istart, int ipoints) {
    int i;
    for (i = 0; i < ipoints; i++)
         x[istart+i] = istart+i;
}

void sub(float *x, int npoints) {
    int iam, nt, ipoints, istart;
    #pragma omp parallel default(shared) private(iam,nt,ipoints,istart)
    {
        iam = omp_get_thread_num();
        nt = omp_get_num_threads();
        ipoints = npoints / nt; /* size of partition */
        istart = iam * ipoints; /* starting array index */
        if (iam == nt-1) /* last thread may do more */
            ipoints = npoints - istart;
        subdomain(x, istart, ipoints);
    }
}

void print(float *x, int npoints) {
    for (int i = 0; i < npoints; i++) {
        if(i % 10 == 0)
            printf("\n");
        printf("%.5f ", x[i]);
    }
}

int main() {
    float array[100];
    sub(array, 100);
    print(array, 100);
    return 0;
}
```

```
0.00000 1.00000 2.00000 3.00000 4.00000 5.00000 6.00000 7.00000 8.00000 9.00000
10.00000 11.00000 12.00000 13.00000 14.00000 15.00000 16.00000 17.00000 18.00000
19.00000
20.00000 21.00000 22.00000 23.00000 24.00000 25.00000 26.00000 27.00000 28.00000
29.00000
30.00000 31.00000 32.00000 33.00000 34.00000 35.00000 36.00000 37.00000 38.00000
39.00000
40.00000 41.00000 42.00000 43.00000 44.00000 45.00000 46.00000 47.00000 48.00000
49.00000
50.00000 51.00000 52.00000 53.00000 54.00000 55.00000 56.00000 57.00000 58.00000
59.00000
60.00000 61.00000 62.00000 63.00000 64.00000 65.00000 66.00000 67.00000 68.00000
69.00000
70.00000 71.00000 72.00000 73.00000 74.00000 75.00000 76.00000 77.00000 78.00000
79.00000
80.00000 81.00000 82.00000 83.00000 84.00000 85.00000 86.00000 87.00000 88.00000
89.00000
90.00000 91.00000 92.00000 93.00000 94.00000 95.00000 96.00000 97.00000 98.00000
99.00000
```

In the above example, we use the default number of threads to perform assignment operations on 100 elements in the array. Tasks are evenly distributed to each thread, and when the number of tasks is not divisible by the number of threads, the remaining tasks will be completed by the last thread.

When programming in parallel, the most important and hardest part is how to assign tasks and manage threads. We already introduced that a thread can get its own id through the **omp_get_thread_num** routine. Another important routine is **omp_get_num_threads**, which returns the number of threads in the current team.

In the above example, variable *npoints* presents the total number of elements in the array, and it is divided into *nt* parts, each of size *ipoints*. The starting address of each part is *istart*. Each part is completed by one thread, and a total of 8 threads execute tasks in parallel.

The **default** clause is used to define the default data-sharing attributes of variables that are referenced in a parallel, teams, or task-generating construct. In the above example, *default(shared)* indicates that by default, the variables in the parallel region are shared variables. The **private** clause is used to explicitly specify variables that are private in each task or SIMD lane (SIMD will be introduced in the next chapter). In the above example, the variables *iam, nt, ipoints* and *istart* are private variables for each thread, which means a thread cannot access these variables of another thread.

The corresponding Fortran program is shown below.

```
!!%compiler: gfortran
!!%cflags: -fopenmp

    SUBROUTINE SUBDOMAIN(X, ISTART, IPOINTS)
        INTEGER ISTART, IPOINTS
        REAL X(0:99)

        INTEGER I

        DO 100 I=0,IPOINTS-1
            X(ISTART+I) = ISTART+I
100     CONTINUE

    END SUBROUTINE SUBDOMAIN

    SUBROUTINE SUB(X, NPOINTS)
        INCLUDE "omp_lib.h" ! or USE OMP_LIB

        REAL X(0:99)
        INTEGER NPOINTS
        INTEGER IAM, NT, IPOINTS, ISTART

!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)

        IAM = OMP_GET_THREAD_NUM()
        NT = OMP_GET_NUM_THREADS()
        IPOINTS = NPOINTS/NT
        ISTART = IAM * IPOINTS
        IF (IAM .EQ. NT-1) THEN
            IPOINTS = NPOINTS - ISTART
        ENDIF
        CALL SUBDOMAIN(X,ISTART,IPOINTS)

!$OMP END PARALLEL
    END SUBROUTINE SUB

    SUBROUTINE print(X, NPOINTS)
        INTEGER I
        REAL X(0:99)
        INTEGER NPOINTS
        DO I = 0,NPOINTS-1
            IF (mod(I,10) .EQ. 0) THEN
                print*,' '
            END IF
            WRITE(*,'(1x,f9.5,$)') X(I)
        END DO
    END SUBROUTINE PRINT

    PROGRAM PAREXAMPLE
        REAL ARRAY(100)
        CALL SUB(ARRAY, 100)
        CALL PRINT(ARRAY, 100)
    END PROGRAM PAREXAMPLE
```

```
   0.00000   1.00000   2.00000   3.00000   4.00000   5.00000   6.00000   7.00000
8.00000   9.00000
  10.00000  11.00000  12.00000  13.00000  14.00000  15.00000  16.00000  17.00000
18.00000  19.00000
  20.00000  21.00000  22.00000  23.00000  24.00000  25.00000  26.00000  27.00000
28.00000  29.00000
  30.00000  31.00000  32.00000  33.00000  34.00000  35.00000  36.00000  37.00000
38.00000  39.00000
  40.00000  41.00000  42.00000  43.00000  44.00000  45.00000  46.00000  47.00000
48.00000  49.00000
  50.00000  51.00000  52.00000  53.00000  54.00000  55.00000  56.00000  57.00000
58.00000  59.00000
  60.00000  61.00000  62.00000  63.00000  64.00000  65.00000  66.00000  67.00000
68.00000  69.00000
  70.00000  71.00000  72.00000  73.00000  74.00000  75.00000  76.00000  77.00000
78.00000  79.00000
  80.00000  81.00000  82.00000  83.00000  84.00000  85.00000  86.00000  87.00000
88.00000  89.00000
  90.00000  91.00000  92.00000  93.00000  94.00000  95.00000  96.00000  97.00000
98.00000  99.00000
```

```
//%compiler: clang
//%cflags: -fopenmp
//This example is from https://www.openmp.org/wp-content/uploads/openmp-examples-
5.1.pdf.

#include <assert.h>

int A[2][2] = {1, 2, 3, 4};

void f(int n, int B[n][n], int C[]) {
    int D[2][2] = {1, 2, 3, 4};
    int E[n][n];

    assert(n >= 2);
    E[1][1] = 4;

    #pragma omp parallel firstprivate(B, C, D, E)
    {
        assert(sizeof(B) == sizeof(int (*)[n]));
        assert(sizeof(C) == sizeof(int*));
        assert(sizeof(D) == 4 * sizeof(int));
        assert(sizeof(E) == n * n * sizeof(int));

        /* Private B and C have values of original B and C. */
        assert(&B[1][1] == &A[1][1]);
        assert(&C[3] == &A[1][1]);
        assert(D[1][1] == 4);
        assert(E[1][1] == 4);
    }
}

int main() {
    f(2, A, A[0]);
    return 0;
}
```

In the above example, we explored how to use the **firstprivate** clause. The size and values of the array or pointer appearing in the list of the **firstprivate** clause depend on the original arrays or pointers in the serial part. The type of A is array of two arrays of two ints. B and C are function variables, which will be adjusted according to the type defined in the function. The type of B is pointer to array of n ints, and the type of C is pointer to int. The type of D is exactly the same as A, and the type of E is array of n arrays of n ints.

In the example, we use multiple assert statements to check whether B, C, D, and E have the expected types and values. Thus to verify the function of the **firstprivate** clause.

By Michael Womack, Anjia Wang, Patrick Flynn, Xinyao Yi, Yonghong Yan

Creating SPMD parallelism using OpenMP teams directive

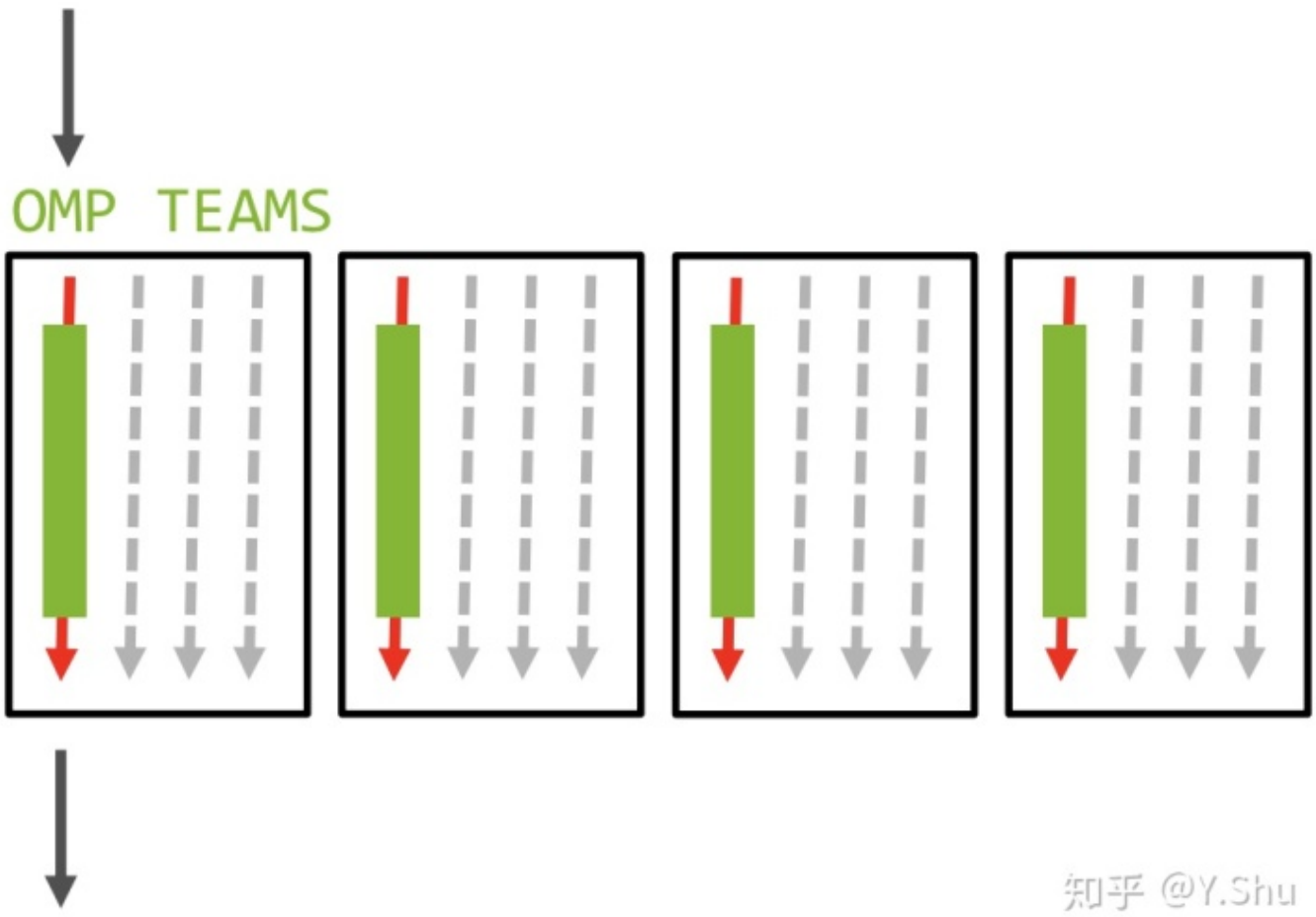
Contents

In this part, we will introduce how to use OpenMP **teams** directives to create SPMD parallelism.

2.3.1. **teams** Directive

The **teams** directive indicates that the loop that follows is split among multiple thread teams, one thread team computing one part of the task. Developers can use the **teams** directive to use a large number of thread teams.

The following figure shows the execution model of the **teams** directive:



A league of teams is created when a thread encounters a **teams** construct. Each team is an initial team, and the initial thread in each team executes the team area. After a team is created, the number of initial teams remains the same for the duration of the **teams** region. Within a **teams** region, the initial team number uniquely identifies each initial team. A thread can obtain its own initial team number by calling the *omp_get_team_num* library routine. The teams directive has the following characteristics:

- the **teams** directive can spawn one or more thread teams with the same number of threads
- code is portable for one thread team or multiple thread teams
- only the primary thread of each team continues to execute
- no synchronization between thread teams
- programmers don’t need to think about how to decompose loops

OpenMP was originally designed for multithreading on shared-memory parallel computers, so the parallel directive only creates a single layer of parallelism. The team instruction is used to express the second level of scalable parallelization. Before OpenMP 5.0, it can be only used on the GPU (with an associated target construct). In OpenMP 5.0 the **teams** construct was extended to enable the host to execute a teams region.

```
//%compiler: clang
//%cflags: -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda  -std=c++11 -Wall -Wno-
unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -fopenmp -fopenmp-
targets=nvptx64
#include <stdlib.h>
#include <omp.h>
float dotprod(float B[], float C[], int N) {
    float sum0 = 0.0;
    float sum1 = 0.0;
    #pragma omp target map(to: B[:N], C[:N]) map(tofrom: sum0, sum1)
    #pragma omp teams num_teams(2)
    {
        int i;
        if (omp_get_num_teams() != 2)
            abort();
        if (omp_get_team_num() == 0) {
            #pragma omp parallel for reduction(+:sum0)
            for (i=0; i<N/2; i++)
                sum0 += B[i] * C[i];
        } else if (omp_get_team_num() == 1) {
            #pragma omp parallel for reduction(+:sum1)
            for (i=N/2; i<N; i++)
                sum1 += B[i] * C[i];
        }
    }
    return sum0 + sum1;
}
/* Note: The variables sum0,sum1 are now mapped with tofrom, for correct
 execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
 */
```

```
clang: fatal error: cannot find libdevice for sm_35. Provide path to different CUDA
installation via --cuda-path, or pass -nocudalib to build without linking with
libdevice.
[Native kernel] clang exited with code 1, the executable will not be executed
```

```
//%compiler: clang
//%cflags: -fopenmp

// Need to update the native kernel, or specified that our kernel doesn't support
OpenMP 5.0


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#define N 1000

int main(){
    int nteams_required=2, max_thrds, tm_id;
    float sp_x[N], sp_y[N], sp_a=0.0001e0;
    double dp_x[N], dp_y[N], dp_a=0.0001e0;

    // Create 2 teams, each team works in a different precision
    #pragma omp teams num_teams(nteams_required) thread_limit(max_thrds)
private(tm_id)
    {
        tm_id = omp_get_team_num();
        if( omp_get_num_teams() != 2 ) //if only getting 1, quit
        {
            printf("error: Insufficient teams on host, 2 required\n");
            exit(0);
        }
        if(tm_id == 0) // Do Single Precision Work (SAXPY) with this team
        {
            #pragma omp parallel
            {
                #pragma omp for //init
                for(int i=0; i<N; i++){sp_x[i] = i*0.0001; sp_y[i]=i; }
                #pragma omp for simd simdlen(8)
                for(int i=0; i<N; i++){sp_x[i] = sp_a*sp_x[i] + sp_y[i];}
            }
        }
        if(tm_id == 1) // Do Double Precision Work (DAXPY) with this team
        {
            #pragma omp parallel
            {
                #pragma omp for //init
                for(int i=0; i<N; i++){dp_x[i] = i*0.0001; dp_y[i]=i; }
                #pragma omp for simd simdlen(4)
                for(int i=0; i<N; i++){dp_x[i] = dp_a*dp_x[i] + dp_y[i];}
            }
        }
    }
    printf("i=%d sp|dp %f %f \n",N-1, sp_x[N-1], dp_x[N-1]);
    printf("i=%d sp|dp %f %f \n",N/2, sp_x[N/2], dp_x[N/2]);
    //OUTPUT1:i=999 sp|dp 999.000000 999.000010
    //OUTPUT2:i=500 sp|dp 500.000000 500.000005
    return 0;
}
```

```
OMP: Warning #96: Cannot form a team with 12 threads, using 6 instead.
OMP: Hint Consider unsetting KMP_DEVICE_THREAD_LIMIT (KMP_ALL_THREADS),
KMP_TEAMS_THREAD_LIMIT, and OMP_THREAD_LIMIT (if any are set).
```

```
i=999 sp|dp 999.000000 999.000010
i=500 sp|dp 500.000000 500.000005
```

Its syntax is:

```
#pragma omp teams [clause[ [,] clause] ... ] new-line
    structured-block
```

The syntax in Fortran is:

```
!$omp teams [clause[ [,] clause] ... ]
    loosely-structured-block
!$omp end teams
```

2.3.2. Clauses

2.3.3. Examples

By Michael Womack, Anjia Wang, Patrick Flynn, Xinyao Yi, Yonghong Yan

Synchronization of threads using barrier and order directive

OpenMP synchronization for barrier and order directive.

By Michael Womack, Anjia Wang, Patrick Flynn, Xinyao Yi, Yonghong Yan