Module 04: Introduction to OpenMP

This module is largely focussed on parallelization of algorithms using the OpenMP API. The building an understanding of multithreading execution model to speed up the algorithms (e.g., PDE solvers) you will develop throughout this course.

As CPU speeds no longer improve as significantly as they did before, multicore systems are becoming more popular. To harness that power, it is becoming important for programmers to be knowledgeable in parallel programming — making a program execute multiple things simultaneously.

This document attempts to give a quick introduction to OpenMP, a simple C/C++/Fortran compiler extension that allows to add parallelism into existing source code without significantly having to entirely rewrite it.

What is OpenMP = Open Multi-Parallelism

OpenMP is an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism in C/C++ programs. It is not intrusive on the original serial code in that the OpenMP instructions are made in pragmas interpreted by the compiler.

OpenMP is managed by the nonprofit technology consortium OpenMP Architecture Review Board (or OpenMP ARB), jointly defined by a group of major computer hardware and software vendors, including AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments, Oracle Corporation, and more.

Goals of OpenMP

- **Standardization:**
  - Provide a standard among a variety of shared memory architectures/platforms
  - Jointly defined and endorsed by a group of major computer hardware and software vendors
- **Lean and Mean:**
  - Establish a simple and limited set of directives for programming shared memory machines.
  - Significant parallelism can be implemented by using just 3 or 4 directives.
- **Ease of Use:**
  - Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
  - Provide the capability to implement both coarse-grain and fine-grain parallelism
- **Portability:**
  - The API is specified for C/C++ and Fortran
  - Public forum for API and membership
  - Most major platforms have been implemented including Unix/Linux platforms and Windows

OpenMP Programming Model

Shared Memory Model

OpenMP is deisgned for multi-processor/core, shared memory machines.

From a strictly hardware point of view, we see a typical shared-memory architecture where all processors can access to a common physical memory. If you have multiple processors, then those would be able to access exactly the same memory locations (see Figure 5). This architecture is very similar to what you have in your personal computers. Hence, developing high-performance parallel algorithms for shared-memory systems is straightforward and very similar to developing serial applications that you would normally run on your laptop. However, there are two big problems with this approach: 1) building CPUs with hundred thousands of cores is not possible for now; 2) all CPU-cores compete for access to memory over a shared bus. That's this kind of systems are not manufactured today.

However, the idea of communicating directly through memory remains a useful programming abstraction. So in many systems, programmers use common programming techniques (e.g., OpenMP) to perform parallel tasks can directly access the same **"virtual"** memory regardless of where the physical memory actually exists. So, this programming model can be thought as an illusion that all memory is actually shared.

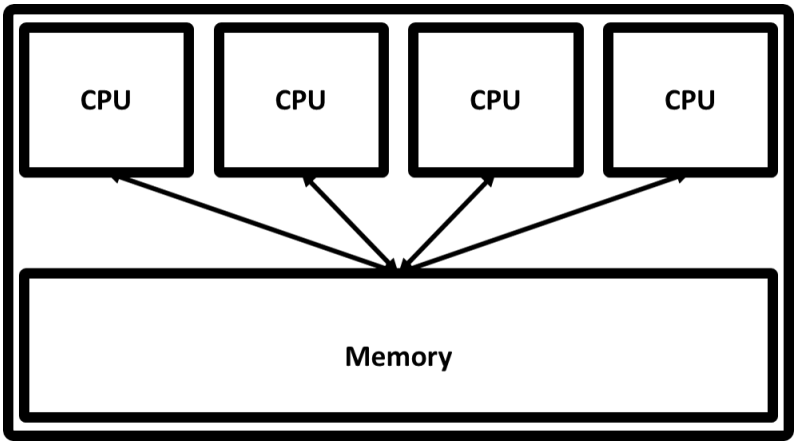

Figure 1. Shared memory architecture.

Because OpenMP is designed for shared memory parallel programming, it largely limited to single node parallelism. Typically, the number of processing elements (cores) on a node determine how much parallelism can be implemented.

Program vs Process vs Thread

Program

A program is a set of instructions and associated data that resides on the disk and is loaded by the operating system to perform some task. An executable file or a python script file are examples of programs. In order to run a program, the operating system's kernel is first asked to create a new process, which is an environment in which a program executes.

Process

A process is a program in execution. A process is an execution environment that consists of instructions, user-data, and system-data segments, as well as lots of other resources such as CPU, memory, address-space, disk and network I/O acquired at runtime. A program can have several copies of it running at the same time but a process necessarily belongs to only one program.

Thread

Thread is the smallest unit of execution in a process. A thread simply executes instructions serially. A process can have multiple threads running as part of it. Usually, there would be some state associated with the process that is shared among all the threads and in turn each thread would have some state private to itself. The globally shared state amongst the threads of a process is visible and accessible to all the threads, and special attention needs to be paid when any thread tries to read or write to this global shared state. There are several constructs offered by various programming languages to guard and discipline the access to this global state, which we will go into further detail in upcoming lessons.

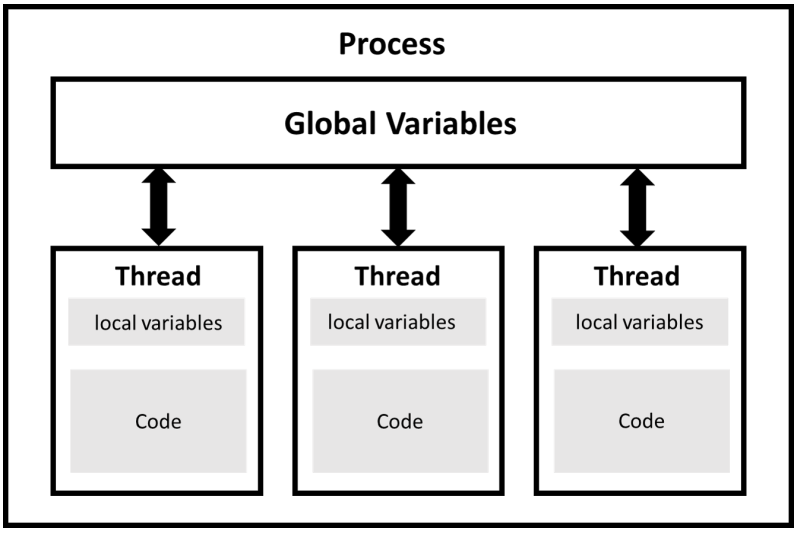

Figure 2. A cartoon showing a process with three threads.

Processes do not share any resources amongst themselves whereas threads of a process can share the resources allocated to that particular process, including memory address space.

Thread-based Parallelism

OpenMP programs accomplish parallelism exclusively through the use of threads by utilizing the fork-join model of parallel execution. All OpenMP programs begin with a single master thread which executes sequentially until a parallel region is encountered, when it creates a team of parallel threads (FORK). When the team threads complete the parallel region, they synchronize and terminate, leaving only the master thread that executes sequentially (JOIN).

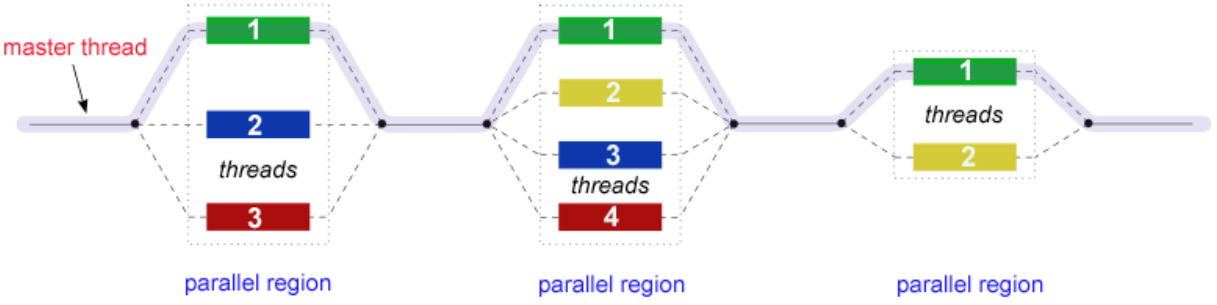

Figure 3. The fork-join model.

Data scoping

Because OpenMP is a shared memory programming model, most data within a parallel region is shared by default. All threads in a parallel region can access this shared data simultaneously. OpenMP provides a way for the programmer to explicitly specify how data is "scoped" if the default shared scoping is not desired.

I/O

OpenMP spicifies nothing about parallel I/O. This is particularly important if multiple threads attempt to read/write from the same file. If every threads conducts I/O to a different file. this issue is not as significant. It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

Scalability and Measuring Parallel Performance

The real power of high-performance computer clusters comes from many processors in parallel to solve big problems. In parallel computing, a large number of processors work simultaneously to produce exceptional computational power and to significantly reduce the total computational time. In such scenarios, scalability or scaling is widely used to indicate the ability of hardware and software to deliver greater computational power when the amount of resources is increased. Scalability is an

important aspect of HPC clusters demonstrating how the capacity of the whole system can be proportionally increased by adding more hardware. From a software point of view, scalability is sometimes referred to as parallelization efficiency—the ratio between the actual speedup and the ideal speedup obtained when using a certain number of processors.

The speedup in parallel computing can be straightforwardly defined as

$$\text{Speedup } (S_P) = \frac{T_1}{T_P}, \tag{1}$$

where T_1 is the computational time for running the software using one processor, and T_P is the computational time running the same software with P processors. Ideally, we would like software to have a linear speedup that is equal to the number of processors (speedup = P), as that would mean that every processor would be contributing 100% of its computational power. Unfortunately, this is a very challenging goal for real applications to attain.

The parallel efficiency is defined as the ratio of speedup to the number of processors. Efficiency measures the fraction of time for which a processor is usefully utilized and can be calculated as follows

$$E = \frac{S_P}{P}, \tag{2}$$

There are two basic ways to measure the parallel performance of a given application, depending on whether or not one is CPU-bound or memory-bound. These are referred to as strong and weak scaling, respectively.

Amdahl's law and strong scaling

Amdahl's law states that the speedup is limited by the fraction of the serial part of the software that is not amenable to parallelization. Amdahl's law can be formulated as follows

$$S_p \leq \frac{1}{f + \frac{1-f}{P}}, \tag{3}$$

where f is the fraction of the execution time spent on the serial part, $1 - f$ is the fraction of the execution time spent the part that can be parallelized, and P is the number of processors.

Amdahl's law states that the upper limit of speedup is determined by the serial fraction of the code. This law implicitly assumes that the problem size is fixed and the communication cost between processors are negligible. Problems for which these assumptions are reasonable can use this model for performance analysis. Such analysis is called **strong scaling**.

Q: What is the maximum speedup of a code with 5% of its part sequential, executed on a 1000 processor cores? (What if it ran on 100 cores? What if the 5% is reduced to 1%?)

Gustafson's law and weak scaling

Amdahl's law provides an upper limit of speedup for fixed-sized problems. However, sizes of problems scale with the amount of resources, and hence, using small amounts of resources for small problems and larger quantities of resources for big problems is a more reasonable approach. In real-life situations, we often want to keep the execution time constant and increase the problem size. We often want to keep the execution time constant and increase the problem size.

Gustafson's law is based on the approximations that the parallel part scales linearly with the amount of resources, and that the serial part does not increase with respect to the size of the problem. The Gustafson's law can be formulated as follows

$$\text{Scaled Speedup } S_p = f + (1 - f) * P, \tag{4}$$

where f is the fraction of the execution time spent on the serial part, $1 - f$ is the fraction of the execution time spent the part that can be parallelized, and P is the number of processors. According to the Gustafson's law, the scaled speedup increases linearly with respect to the number of processors and there is no upper limit for the scaled speedup. This is called **weak scaling**, where the scaled speedup is calculated based on the amount of work done for a scaled problem size. In other words, Gustafson's Law scales relative to the parallel fraction of a code and the serial fraction does not affect the scaling.

Summary

There are two ways of determining the scaling of a software: strong scaling and weak scaling. Some of the key points are summarized below:

- Scalability is important for parallel computing efficiency

- Strong scaling concerns the speedup for a fixed problem size with respect to he number of processors, and is governed by Amdahl's law.
- Weak scaling concerns the speedup for a scaled problem size with respect to the number of processors, and is governed by Gustafson's law.
- The results of strong and weak scaling tests provide good indications for the best match between job size and the mount of resources that should be requested for a particular job.

OpenMP API Overview

OpenMP API has three distinct components:

- Compiler directives
- Runtime library routines
- Environment variables

As the new API versions are introduced, new directives, runtime library routines, and environment variables are added to the API. The programmer decides how to employ these components. In the simplest case, only a few of them are needed.

Compiler Directives

OpenMP compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise. This is usually done by specifying the appropriate compiler flag when compiling your source code.

OpenMP directives are used for various purposes:

- Spawning a parallel region
- Dividing blocks of code among threads
- Distributing loop iterations between threads
- Serializing sections of code
- Synchronization of work among threads

The standard syntax for compiler directives are as given below:

sentinel directive-name [clause, ...]

For example:

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

Run-time Library Routines

The OpenMP API includes several run-time library routines that are used for a variety of purposes:

- Setting and querying the number of threads
- Querying a thread's unique identifier (ID)
- Querying if in a parallel regions, and at what level
- Setting and querying nested parallelism

For C/C++, all of the run-time library routines are actual functions. For example:

```
#include <omp.h>
int omp_get_num_threads(void)
```

Note that for C/C++, you need ot include the **<omp.h>** header file.

Environment Variables

OpenMP provides several environment variables for controlling the execution of parallel code at run-time.

These environment variables can be used to control such things as:

- Setting the number of threads
- Specifying how loop interations are divided
- Binding threads to processors
- Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism

- Enabling/disabling dynamic threads
- Setting thread wait policy

You can set the OpenMP environment variables the same way you set any other environment variables. For example:

```
export OMP_NUM_THREADS=8
```

Example OpenMP Code Structure

```
#include <omp.h>

int main ()  {

int var1, var2, var3;

Serial code
      .
      .
      .

Beginning of parallel region. Fork a team of threads.
Specify variable scoping

#pragma omp parallel private(var1, var2) shared(var3)
   {

   Parallel region executed by all threads
              .
   Other OpenMP directives
              .
   Run-time Library calls
              .
   All threads join master thread and disband

   }

Resume serial code
      .
      .
      .

    return 0;
}
```

Compiling OpenMP Programs

OpenMP programs require you to use the appropriate compiler flag to "turn on" OpenMP compilations. For the gcc compiler you generally use in this class, the OpenMP compiler flag is **"-fopenmp"**.

OpenMP Directives

OpenMP directives is one of the core components of the OpenMP API. Parallel code with OpenMP marks, through a special directive, sections to be executed in parallel. The part of the code that is marked to run in parallel will cause threads to form. The main tread is the master thread. The slave threads all run in parallel and run the same code. Each thread executes the parallelized section of the code independently. When a thread finishes, it joins the master. When all threads finished, the master continues with code following the parallel section.

Parallel Region Construct

In OpenMP framework, a parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct. When a thread reaches a parallel directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number **0** within the that team. The compiler copies

duplicates the source code starting from the beginning of the **parallel region and all threads will execute that code. Once all threads finish their job and reach the end of the **parallel** region, they will either be destroyed or hit a barrier. After this point, only the master thread continues the execution.

The syntax for the **"parallel"** construct is as follows

```
#pragma omp parallel [clause ...]  newline
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)
                     num_threads (integer-expression)


   structured_block
```

The number of threads in a parallel region can be determined by various ways. One can set the **"num_threads"** clause when defining a parallel region. Alternatively, the number of threads in a parallel region can be set by the OpenMP library function **"omp_set_num_threads()"** or setting the environment variable **"OMP_NUM_THREADS"** on your terminal shown as below:

```
export OMP_NUM_THREADS=8
```

When defining a parallel region, one should be aware that a parallel region must be a structure block that does not span multiple routines or code files. It is illegal to branch (goto) into ot out of a parallel region (goto should be literally illegal!). Additionally, only a single **num_threads** clause is permitted in a parallel regions definition.

Hello, World

Now, we know how to create a parallel region in OpenMP, we can go ahead and write our first parallel program.

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

int main (int argc, char *argv[]) {
  int tid, nthreads;
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread %d\n", tid);
    #pragma omp barrier
    if ( tid == 0 ) {
      nthreads = omp_get_num_threads();
      printf("There are %d threads\n",nthreads);
    }
  }
  return EXIT_SUCCESS;
}
```

OpenMP assumes data is shared among threads. However, we want each thread ot have their unique ID, so we can distinguish them. In the code above, we set the **"tid"** variable as **private** to each thread. When a variable is flagged as **private**, each threads generates a copy of that variable. Therefore, values of private variables can be different for each thread.

Work-Sharing Constructs

A work-sharing construct divides the execution of an enclosed code region among the members of the team that encounter it. The work-sharing constructs do not launch new threads and use the threads that are created by the **parallel** construct.

There are three types of work-sharing constructs: **"for"**, **"sections"**, and **"single"**. These constructs must be enclosed dynamically within a parallel region in for the directive to execute in parallel. All threads in a team must be able to encounter work-sharing constructs. When one uses successive work-sharing constructs, all members of a team must encounter these

constructs in the same order.

For directive

The **for** directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor. This approach represents a type of **"data parallelism"**.

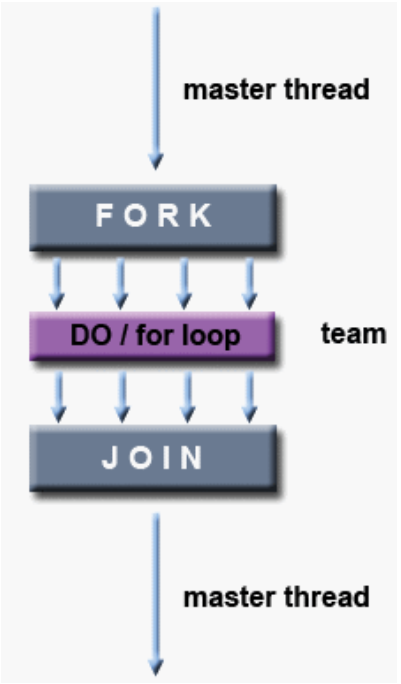

Figure 4. The for directive in OpenMP.

The syntax for the **for** directive is given below:

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

   for_loop
```

The explanations for each clause is given below.

Schedule

Here, the **"schedule"** clause describes how iterations of the loop are divided among the threads in the team.

In **"static"** scheduling, loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

Figure 5. The static scheduling in OpenMP.

In **"dynamic"** scheduling, loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

Figure 6. The synamic scheduling in OpenMP.

nowait

In OpenMP, threads usually synchronize at the end of a parallel region, If one defines the **"nowait"** clause in **for** directive, the threads do not synchronize at the end of the parallel loop.

ordered

The **"ordered"** clause specifies that the iterations of the loop must be executed as they would be in a serial program.

collapse

The **"collapse"** clause specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The order of the iterations in the collapsed iteration space is determined as though they were executed sequentially.

An example of collapsing a loop is given below:

```
#pragma omp parallel for private(j) collapse(2)
for (i = 0; i < 4; i++)
    for (j = 0; j < 100; j++)
```

The for loop above can be manually collapsed as shown below:

```
#pragma omp parallel for
for(int n=0; n<4*100; n++) {
    int i = n/100; int j=n%100;
```

It is important that one is aware of the restrictions of the **for** directive, which can be summarized as below:

- The loop iteration variable must be an integer and the loop control parameters must be the same for all threads.
- Program correctness must not depend upon which thread executes a particular iteration.
- It is illegal to branch out of a loop associated with a for directive.
- The chunk size must be specified as a loop invariant integer expression, as there is no synchronization during its evaluation by different threads.
-

An example for the for directive

The following code performs a parallel vector addition:

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

#define N 1000
#define CHUNKSIZE 100

int main(int argc, char *argv[]) {

    int i, chunk;
    float a[N], b[N], c[N];

    /* initialize arrays */
    for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;

    /* set the chunk size */
    chunk = CHUNKSIZE;

/* the beginning of the parallel region */
#pragma omp parallel shared(a,b,c,chunk) private(i)
{
    /* parallelize the following for loop */
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++){
        c[i] = a[i] + b[i];
    }

    /* who are you ? */
    printf("I am : %d\n",omp_get_thread_num());
}   /* the end of the parallel region */
return 0;
}
```

In the code above, arrays "A", "B", and "C" and the variable "N" will be shared by all threads. The variable "i" will be private to each thread; each thread will have its own unique copy. The iterations of the loop will be distributed dynamically in **chunk**-sized pieces. Finally, threads will not synchronize upon completing their individual pieces of work (**nowait**).

Sections directive

The **sections** is a non-iterative work-sharing construct. It specifies that the enclosed sections(s) of code are to be divided among the threads in the team.

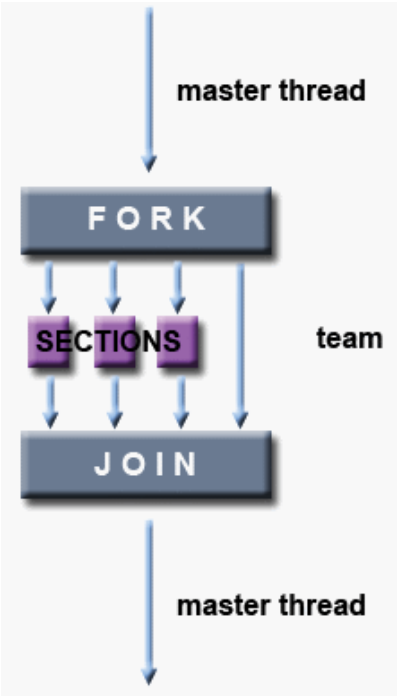

Figure 7. The sections directive in OpenMP.

Independent **section** directives are nested within a **sections** directive. Each **section** is executed once by a thread in the team. Different sections may be executed by different threads. It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such.

The standard syntax for the **sections** directive is given below:

```
#pragma omp sections [clause ...]  newline
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
  {

  #pragma omp section   newline

     structured_block

  #pragma omp section   newline

     structured_block

  }
```

There is an implied barrier at the end of a **sections** directive, unless the **nowait** clause is used.

An example for the sections directive

The code below shows an example of the usage of the **sctions** directive where a vector addition and multiplication are performed in parallel.

```
#include <omp.h>
#define N 1000

int main(int argc, char *argv[]) {

int i;
float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }

#pragma omp parallel shared(a,b,c,d) private(i)
  {

  #pragma omp sections nowait
    {

    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

    }  /* end of sections */

  }  /* end of parallel region */

    return 0;
}
```

Single directive

The **single** directive specifies that the enclosed code is to be executed by only one thread in the team. Although the **single** directive may seem pointless in the first look, it may be useful when dealing with sections of code that are not thread safe (such as I/O).

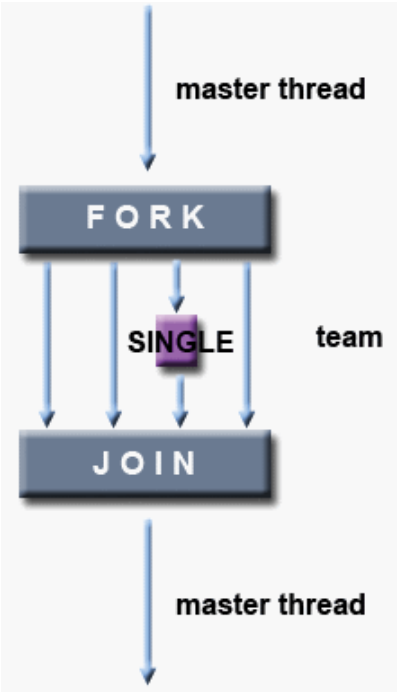

Figure 8. The single directive in OpenMP.

The syntax for the **single** directive is as given below:

```
#pragma omp single [clause ...]  newline
                   private (list)
                   firstprivate (list)
                   nowait

        structured_block
```

The threads in the team that do not execute the **single** directive wait at the end of the enclosed code block, unless a **nowait** clause is specified.

Combined Parallel Work-Sharing Constructs

OpenMP allows programmers to combine the parallel region directive with work-sharing constructs for convenience. Instead of creating a parallel region and work-sharing constructs within that region, you can just type:

- parallel for
- parallel sections

The expressions above will launch threads, parallel region, and parallelize the code within the specified code block.

For the most part, these directives behave identically to an individual PARALLEL directive being immediately followed by a separate work-sharing directive. Most of the rules, clauses and restrictions that apply to both directives are in effect. Therefor, you will mostly use these combined directives to parallelize your code.

Below, you can find an example for combined parallel for directives:

```
#include <omp.h>
 #define N       1000
 #define CHUNKSIZE   100

 int main(int argc, char *argv[]) {

 int i, chunk;
 float a[N], b[N], c[N];

 /* Some initializations */
 for (i=0; i < N; i++)
   a[i] = b[i] = i * 1.0;
 chunk = CHUNKSIZE;

 #pragma omp parallel for \
   shared(a,b,c,chunk) private(i) \
   schedule(static,chunk)
   for (i=0; i < N; i++)
     c[i] = a[i] + b[i];

   return 0;
 }
```

Data Scope (Data-Sharing) Attribute Clauses

An important consideration for OpenMP programming is the understanding the data scoping. Because OpenMP is based upon the shared memory programming model, most variables are shared by default. However, you will extensively use data scope attribute clauses in conjunction with several directives (parallel, for, and sections) to control the scoping of enclosed variables.

The **parallel**, **for**, and **sections** constructs allow programmer to define how and which data variables in the serial section of the program are transferred to the parallel regions of the program (and back). Additionally, as we saw in previous sections, you can define which variables will be visible to all threads in the parallel regions and which variables will be privately allocated to all threads.

Private clause

The **private** clause declares variables in its list to be private to each thread.

The syntax for the **private** clause is as given below:

```
private (list of variables separated by comma)
```

In your codes, you will almost always define loop index variables as private.

Private variables behave as follows:

- A new object of the same type is declared once for each thread in the team
- All references to the original object are replaced with references to the new object
- Should be assumed to be uninitialized for each thread

Shared clause

The **shared** clause declares variables in its list to be shared among all threads in the team.

The syntax for the **shared** clause is the same as private clause:

```
shared(list of variables separated by comma)
```

A shared variable exists in only one memory location and all threads can read or write to that address. It is the programmer's responsibility to ensure that multiple threads properly access **shared** variables.

Default clause

The **default** clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region. You can use the **default** clause as follows:

```
    default(shared | private | none)
```

Using "none" as a default requires that the programmer explicitly scope all variables.

In [ ]: