Aim: To learn basics of OpenMP API

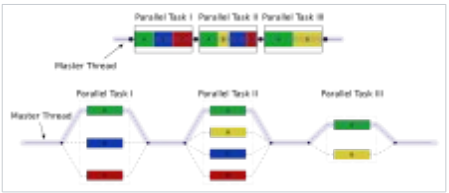
OpenMP (**Open Multi-Processing**) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran,[3] on many platforms, instruction set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.[2][4][5]

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.[1]

OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.

An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI), such that OpenMP is used for parallelism *within* a (multi-core) node while MPI is used for parallelism *between* nodes. There have also been efforts to run OpenMP on software distributed shared memory systems, to translate OpenMP into MPI and to extend OpenMP for non-shared memory systems.

Design



An illustration of multithreading where the master thread forks off a number of threads which execute blocks of code in parallel.

See also: Fork–join model

OpenMP is an implementation of multithreading, a method of parallelizing whereby a *master* thread (a series of instructions executed consecutively) *forks* a specified number of *slave* threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a compiler directive that will cause the threads to form before the section is executed.[3] Each thread has an *id* attached to it which can be obtained using a function (called `omp_get_thread_num()`). The thread id is an integer, and the master thread has an id of *0*. After the execution of the parallelized code,

the threads *join* back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. *Work-sharing constructs* can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions. The OpenMP functions are included in a header file labelled `omp.h` in C/C++.

History[edit]

The OpenMP Architecture Review Board (ARB) published its first API specifications, OpenMP for Fortran 1.0, in October 1997. In October the following year they released the C/C++ standard. 2000 saw version 2.0 of the Fortran specifications with version 2.0 of the C/C++ specifications being released in 2002. Version 2.5 is a combined C/C++/Fortran specification that was released in 2005.

Up to version 2.0, OpenMP primarily specified ways to parallelize highly regular loops, as they occur in matrix-oriented numerical programming, where the number of iterations of the loop is known at entry time. This was recognized as a limitation, and various task parallel extensions were added to implementations. In 2005, an effort to standardize task parallelism was formed, which published a proposal in 2007, taking inspiration from task parallelism features in Cilk, X10 and Chapel.[10]

Version 3.0 was released in May 2008. Included in the new features in 3.0 is the concept of *tasks* and the *task* construct,[11] significantly broadening the scope of OpenMP beyond the parallel loop constructs that made up most of OpenMP 2.0.[12]

Version 4.0 of the specification was released in July 2013.[13] It adds or improves the following features: support for accelerators; atomics; error handling; thread affinity; tasking extensions; user defined reduction; SIMD support; Fortran 2003 support.[14][*full citation needed*]

The current version is 5.0, released in November 2018.

Note that not all compilers (and OSes) support the full set of features for the latest version/s.

Core elements

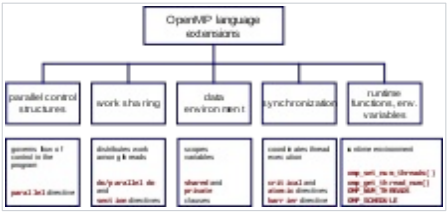


Chart of OpenMP constructs

The core elements of OpenMP are the constructs for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables.

In C/C++, OpenMP uses #pragmas. The OpenMP specific pragmas are listed below.

Thread creation

The pragma *omp parallel* is used to fork additional threads to carry out the work enclosed in the construct in parallel. The original thread will be denoted as *master thread* with thread ID 0.

Example (C program): Display "Hello, world." using multiple threads.

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

int main(void)
{
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

Use flag -fopenmp to compile using GCC:

```
$ gcc -fopenmp hello.c -o hello
```

Output on a computer with two cores, and thus two threads:

```
Hello, world.
Hello, world.
```

However, the output may also be garbled because of the race condition caused from the two threads sharing the standard output.

```
Hello, wHello, woorld.
rld.
```

(Whether `printf` is thread-safe depends on the implementation. C++ `std::cout`, on the other hand, is always thread-safe.)

Work-sharing constructs

Used to specify how to assign independent work to one or all of the threads.

•*omp for* or *omp do*: used to split up loop iterations among the threads, also called loop constructs.
•*sections*: assigning consecutive but independent code blocks to different threads
•*single*: specifying a code block that is executed by only one thread, a barrier is implied in the end
•*master*: similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.
Example: initialize the value of a large array in parallel, using each thread to do part of the work

```
int main(int argc, char **argv)
{
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

This example is embarrassingly parallel, and depends only on the value of `i`. The OpenMP `parallel for` flag tells the OpenMP system to split this task among its working threads. The threads will each receive a unique and private version of the variable. For instance, with two worker threads, one thread might be handed a version of `i` that runs from 0 to 49999 while the second gets a version running from 50000 to 99999.

Variant directives

Variant directives is one of the major features introduced in OpenMP 5.0 specification to facilitate programmers to improve performance portability. They enable adaptation of OpenMP pragmas and user code at compile time. The specification defines traits to describe active OpenMP constructs, execution devices, and functionality provided by an implementation, context selectors based on the traits and user-defined conditions, and *metadirective* and *declare directive* directives for users to program the same code region with variant directives.

•The *metadirective* is an executable directive that conditionally resolves to another directive at compile time by selecting from multiple directive variants based on traits that define an OpenMP condition or context.
•The *declare variant* directive has similar functionality as *metadirective* but selects a function variant at the call-site based on context or user-defined conditions.
The mechanism provided by the two variant directives for selecting variants is more convenient to use than the C/C++ preprocessing since it directly supports variant selection in OpenMP and allows an OpenMP compiler to analyze and determine the final directive from variants and context.

```
// code adaptation using preprocessing directives

int v1[N], v2[N], v3[N];
#if defined(nvptx)
 #pragma omp target teams distribute parallel loop map(to:v1,v2)
map(from:v3)
  for (int i= 0; i< N; i++)
     v3[i] = v1[i] * v2[i];
#else
 #pragma omp target parallel loop map(to:v1,v2) map(from:v3)
  for (int i= 0; i< N; i++)
     v3[i] = v1[i] * v2[i];
#endif


// code adaptation using metadirective in OpenMP 5.0

int v1[N], v2[N], v3[N];
#pragma omp target map(to:v1,v2) map(from:v3)
  #pragma omp metadirective \
     when(device={arch(nvptx)}: target teams distribute parallel
loop)\
     default(target parallel loop)
  for (int i= 0; i< N; i++)
     v3[i] = v1[i] * v2[i];
```

Clauses

Since OpenMP is a shared memory programming model, most variables in OpenMP code are visible to all threads by default. But sometimes private variables are necessary to avoid race conditions and there is a need to pass

values between the sequential part and the parallel region (the code block executed in parallel), so data environment management is introduced as *data sharing attribute clauses* by appending them to the OpenMP directive. The different types of clauses are:

Data sharing attribute clauses

•*shared*: the data declared outside a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are shared except the loop iteration counter.
•*private*: the data declared within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are private.
•*default*: allows the programmer to state that the default data scoping within a parallel region will be either *shared*, or *none* for C/C++,
or *shared*, *firstprivate*, *private*, or *none* for Fortran. The *none* option forces the programmer to declare each variable in the parallel region using the data sharing attribute clauses.
•*firstprivate*: like *private* except initialized to original value.
•*lastprivate*: like *private* except original value is updated after construct.
•*reduction*: a safe way of joining work from all threads after construct.

Synchronization clauses

•*critical*: the enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data from race conditions.
•*atomic*: the memory update (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic; only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using *critical*.
•*ordered*: the structured block is executed in the order in which iterations would be executed in a sequential loop
•*barrier*: each thread waits until all of the other threads of a team have reached this point. A work-sharing construct has an implicit barrier synchronization at the end.
•*nowait*: specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish. In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.

Scheduling clauses

•*schedule(type, chunk)*: This is useful if the work sharing construct is a do-loop or for-loop. The iteration(s) in the work sharing construct are assigned to

threads according to the scheduling method defined by this clause. The three types of scheduling are:
1.*static*: Here, all the threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default. However, specifying an integer for the parameter *chunk* will allocate chunk number of contiguous iterations to a particular thread.
2.*dynamic*: Here, some of the iterations are allocated to a smaller number of threads. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are left. The parameter *chunk* defines the number of contiguous iterations that are allocated to a thread at a time.
3.*guided*: A large chunk of contiguous iterations are allocated to each thread dynamically (as above). The chunk size decreases exponentially with each successive allocation to a minimum size specified in the parameter *chunk*

IF control

•*if*: This will cause the threads to parallelize the task only if a condition is met. Otherwise the code block executes serially.

Initialization

•*firstprivate*: the data is private to each thread, but initialized using the value of the variable using the same name from the master thread.
•*lastprivate*: the data is private to each thread. The value of this private data will be copied to a global variable using the same name outside the parallel region if current iteration is the last iteration in the parallelized loop. A variable can be both *firstprivate* and *lastprivate*.
•*threadprivate*: The data is a global data, but it is private in each parallel region during the runtime. The difference between *threadprivate* and *private* is the global scope associated with threadprivate and the preserved value across parallel regions.

Data copying

•*copyin*: similar to *firstprivate* for *private* variables, *threadprivate* variables are not initialized, unless using *copyin* to pass the value from the corresponding global variables. No *copyout* is needed because the value of a threadprivate variable is maintained throughout the execution of the whole program.
•*copyprivate*: used with *single* to support the copying of data values from private objects on one thread (the *single* thread) to the corresponding objects on other threads in the team.

Reduction

•*reduction(operator | intrinsic : list)*: the variable has a local copy in each thread, but the values of the local copies will be summarized (reduced) into a global shared variable. This is very useful if a particular operation (specified in *operator* for this particular clause) on a variable runs iteratively, so that its value at a particular iteration depends on its value at a prior iteration. The steps that lead up to the operational increment are parallelized, but the

threads updates the global variable in a thread safe manner. This would be required in parallelizing numerical integration of functions and differential equations, as a common example.

Others

•*flush*: The value of this variable is restored from the register to the memory for using this value outside of a parallel part
•*master*: Executed only by the master thread (the thread which forked off all the others during the execution of the OpenMP directive). No implicit barrier; other team members (threads) not required to reach.

User-level runtime routines[edit]

Used to modify/check the number of threads, detect if the execution context is in a parallel region, how many processors in current system, set/unset locks, timing functions, etc

Environment variables[edit]

A method to alter the execution features of OpenMP applications. Used to control loop iterations scheduling, default number of threads, etc. For example, *OMP_NUM_THREADS* is used to specify number of threads for an application.

Implementations[edit]

OpenMP has been implemented in many commercial compilers. For instance, Visual C++ 2005, 2008, 2010, 2012 and 2013 support it (OpenMP 2.0, in Professional, Team System, Premium and Ultimate editions), as well as Intel Parallel Studio for various processors.[19] Oracle Solaris Studio compilers and tools support the latest OpenMP specifications with productivity enhancements for Solaris OS (UltraSPARC and x86/x64) and Linux platforms. The Fortran, C and C++ compilers from The Portland Group also support OpenMP 2.5. GCC has also supported OpenMP since version 4.2.

Compilers with an implementation of OpenMP 3.0:

•GCC 4.3.1
•Mercurium compiler
•Intel Fortran and C/C++ versions 11.0 and 11.1 compilers, Intel C/C++ and Fortran Composer XE 2011 and Intel Parallel Studio.
•IBM XL compiler[20]
•Sun Studio 12 update 1 has a full implementation of OpenMP 3.0
Several compilers support OpenMP 3.1:

•GCC 4.7
•Intel Fortran and C/C++ compilers 12.1
•IBM XL C/C++ compilers for AIX and Linux, V13.1[24] & IBM XL Fortran compilers for AIX and Linux, V14.1[25]
•LLVM/Clang 3.7
•Absoft Fortran Compilers v. 19 for Windows, Mac OS X and Linux
Compilers supporting OpenMP 4.0:

•GCC 4.9.0 for C/C++, GCC 4.9.1 for Fortran
•Intel Fortran and C/C++ compilers 15.0[29]
•IBM XL C/C++ for Linux, V13.1 (partial) & XL Fortran for Linux, V15.1 (partial)
•LLVM/Clang 3.7 (partial)[26]
Several Compilers supporting OpenMP 4.5:

•GCC 6 for C/C++
•Intel Fortran and C/C++ compilers 17.0, 18.0, 19.0
Auto-parallelizing compilers that generates source code annotated with OpenMP directives:

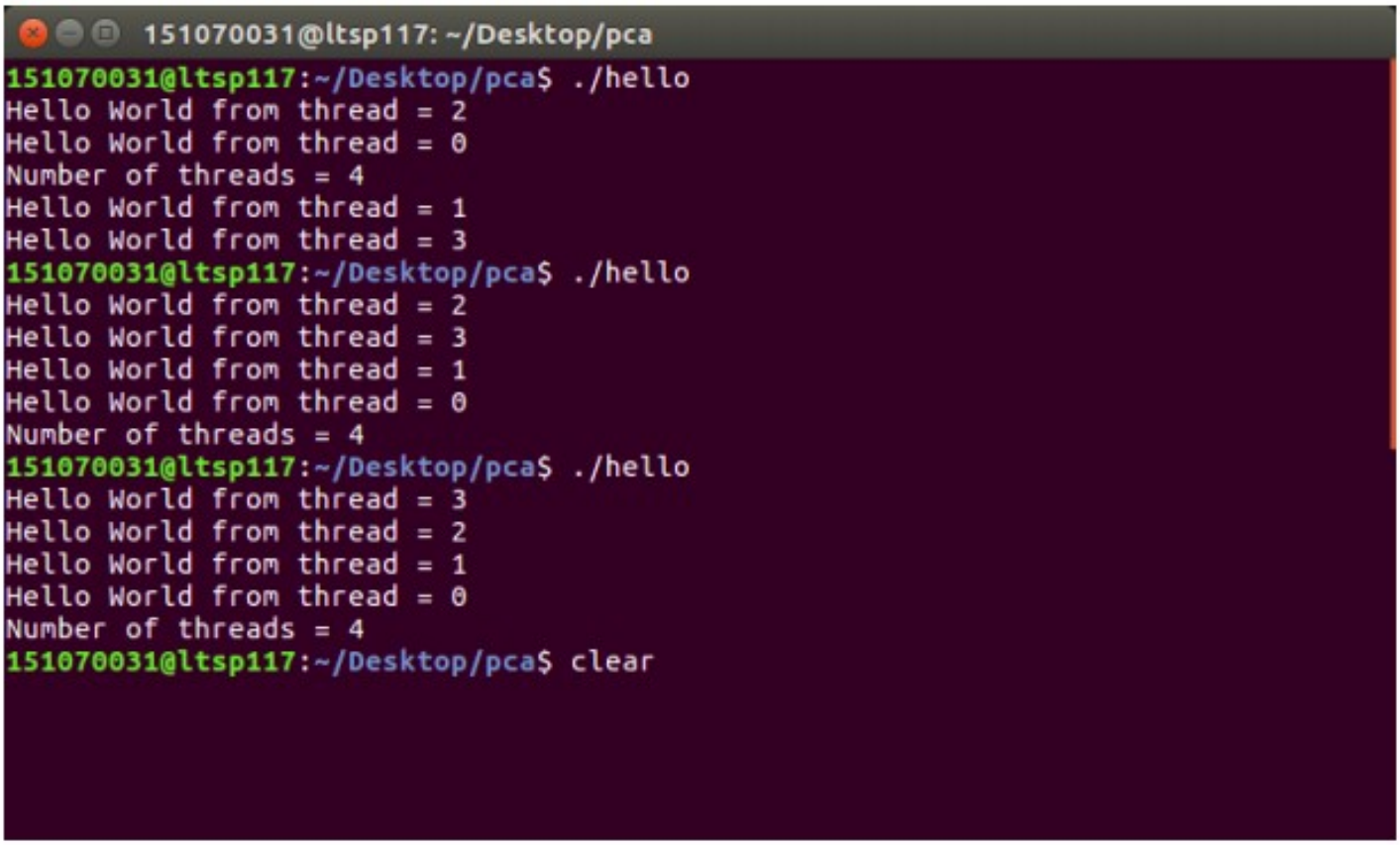
•iPat/OMP
•Parallware
•PLUTO
•ROSE (compiler framework)
•S2P by KPIT Cummins Infosystems Ltd.
Several profilers and debuggers expressly support OpenMP:

•Allinea Distributed Debugging Tool (DDT) – debugger for OpenMP and MPI codes
•Allinea MAP – profiler for OpenMP and MPI codes
•TotalView - debugger from Rogue Wave Software for OpenMP, MPI and serial codes
•ompP – profiler for OpenMP
•VAMPIR – profiler for OpenMP and MPI code

Code:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
int nthreads, tid ;
/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
{
/* Obtain and print thread id */
tid = omp_get_thread_num();
printf("Hello World from thread = %d\n", tid);
/* Only master thread does this */
if (tid == 0)
{
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
} /* All threads join master thread and terminate */
return 0;
}
```

Output:



Conclusion:

Thus,

1. Basics of OpenMP are now clear.
2. Various concepts such as parallelism, run-time libraries and environment variables were studied in detail.