Open MP Architecture on multi-platform shared memory multiprocessing programming.

Dr. M.Suresh Babu¹  D.Asha Devi²  A.Subramnayeswara Rao³

¹Professor & Head, Department of Computer Applications, Madanapalle Institute of Technology & Science, Madanapalle, AP, India – 517325
²Professor, Department of ECE, Madanapalle Institute of Technology & Science, Madanapalle, AP, India– 517325
³Assistant Professor, Department of MCA, Madanapalle Institute of Technology & Science, Madanapalle, AP, India – 517325

Abstract

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most processor architectures and operating systems, including Solaris, AIX, HP-UX, GNU/Linux, Mac OS X, and Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. OpenMP is managed by the non-profit technology consortium Open MP 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, Microsoft, Texas instruments, Oracle Corporation, and more. 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), or more transparently through the use of OpenMP extensions for non-shared memory systems.

Key words : OpenMP, Parallel Processing, Message passing interface

1.0 Introduction

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 a task is divided 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 preprocessor directive that will cause the threads to form before the section is executed. 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 number of threads can be assigned by the runtime environment based on environment variables or in code using functions. The OpenMP functions are included in a header file labelled omp.h in C/C++.

1.1 History

The OpenMP Architecture Review Board (ARB) published its first API specifications, OpenMP for Fortran 1.0, in October 1997. 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.

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.

Version 3.1 of the OpenMP specification was released July 9, 2011.

Version 4.0 of the specification was released in July 2013. 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.

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.

**1.2 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.

```c
#include <stdio.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, world.
Hello, world.

1.3 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

```c
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i, a[N];

    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

    return 0;
}
```

2.0 OpenMP 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 within 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 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, 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 datatype that runs iteratively so that its value at a particular iteration depends on its value at a prior iteration. Basically, the steps that lead up to the operational increment are parallelized, but the threads gather up and wait before updating the datatype, then increments the datatype in order so as to avoid racing condition. 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**

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.

**3.0Environment variables**

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.

In this section, some sample programs are provided to illustrate the concepts explained above.

**Hello World**

A basic program that exercises the parallel, private and barrier directives, and the functions *omp_get_thread_num* and *omp+get_num_threads* (not to be confused)

**C**

This C program can be compiled using gcc-4.4 with the flag -fopenmp

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

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

**C++**

This C++ program can be compiled using GCC:

```cpp
#include <iostream>
using namespace std;
#include <omp.h>
```

NOTE: The IOstreams library is not thread-safe. Therefore, for instance, *cout* calls must be executed in critical areas or by only one thread (e.g. masterthread).

```cpp
#include <iostream>
using namespace std;
#include <omp.h>
```
int main(int argc, char *argv[]) {
    int th_id, nthreads;
    #pragma omp parallel private(th_id)
    shared(nthreads)
    {
        th_id = omp_get_thread_num();
        #pragma omp critical
        {
            cout « "Hello World from thread " « th_id « 
        }
        #pragma omp barrier
        #pragma omp master
        {
            nthreads = omp_get_num_threads();
            cout « "There are " « nthreads « " threads" « 
        }
        #pragma omp end parallel
    }
    return 0;
}

Fortran 77

Here is a Fortran 77 version.

```fortran
PROGRAM HELLO
    INTEGER ID, NTHRDS
    INTEGER OMP_GET_THREAD_NUM, OMP_GET_NUM_THREADS
    CSOMP PARALLEL PRIVATE(ID)
    ID = OMP_GET_THREAD_NUM()
    PRINT *, 'HELLO WORLD FROM THREAD', ID
    CSOMP BARRIER
    IF ( ID .EQ. 0 ) THEN
        NTHRDS = OMP_GET_NUM_THREADS()
        cout « "There are " « nthreads « " threads" « 
    END IF
    CSOMP END PARALLEL
END
```

Fortran 90 free form

Here is a Fortran 90 free form version.

```fortran
program hello90
    use omp_lib
    integer:: id, nthreads
    !Somp parallel private(id)
    id = omp_get_thread_num()
    write (*,*) 'Hello World from thread', id
    !Somp barrier
    if ( id == 0 ) then
        nthreads = omp_get_num_threads()
        write (*,*) 'There are', nthreads, 'threads'
    end if
    !Somp end parallel
end program
```

Clauses in work-sharing constructs (in C/C++)

The application of some OpenMP clauses are illustrated in the simple examples in this section. The piece of code below updates the elements of an array \( b \) by performing a simple operation on the elements of an array \( a \). The parallelization is done by the OpenMP directive \#pragma omp. The scheduling of tasks is dynamic. Notice how the iteration counters \( j \) and \( k \) have to be made private, whereas the primary iteration counter \( i \) is private by default. The task of running through \( i \) is divided among multiple threads, and each thread creates its own versions of \( j \) and \( k \) in its execution stack, thus doing the full task allocated to it and updating the allocated part of the array \( b \) at the same time as the other threads.

```c
#define CHUNKSIZE 1 /*defines the chunk size as 1 contiguous iteration*/
/*forks off the threads*/
#pragma omp parallel private(j,k)
{
    #pragma omp for schedule(dynamic, CHUNKSIZE)
    for(i = 2; i <= N-1; i++)
        for(j = 2; j <= i; j++)
            for(k = 1; k <= M; k++)
                b[i][j] += a[i-1][j]/k + a[i+1][j]/k;
}
```

The next piece of code is a common usage of the reduction clause to calculate reduced sums. Here, we add up all the elements of an array \( a \) with
an \( i \)-dependent weight using a for loop, which we parallelize using OpenMP directives and reduction clause. The scheduling is kept static.

```c
#define N 10000 /*size of a*/
void calculate(long *); /*The function that calculates the elements of a*/
int i;
long w;
long a[N];
calculate(a);
long sum = 0;
/*forks off the threads and starts the work-sharing construct*/
#pragma omp parallel for private(w) reduction(+:sum) schedule(static,1)
for(i = 0; i < N; i++)
{
  w = i*i;
  sum = sum + w*a[i];
}
printf("\n%li",sum);
```

An equivalent, less elegant, implementation of the above code is to create a local sum variable for each thread ("loc_sum"), and make a protected update of the global variable \( sum \) at the end of the process, through the directive critical. Note that this protection is critical, as explained elsewhere.

```c
... long sum = 0, loc_sum;
/*forks off the threads and starts the work-sharing construct*/
#pragma omp parallel private(w,loc_sum)
{
  loc_sum = 0;
  #pragma omp for schedule(static,1)
  for(i = 0; i < N; i++)
  {
    w = i*i;
    loc_sum = loc_sum + w*a[i];
  }
  #pragma omp critical
  sum = sum + loc_sum;
}
printf("\n%li",sum);
```

4.0 Implementations

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. 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
- Nanos 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 C/C++ compiler\(^{[15]}\)
- Sun Studio 12 update 1 has a full implementation of OpenMP 3.0\(^{[16]}\)

Several compilers support OpenMP 3.1:

- GCC 4.7
- Intel Fortran and C/C++ compilers.

Auto-parallelizing compilers that generates source code annotated with OpenMP directives:

- iPAt/OMP
- Parallware
- PLUTO
- ROSE (compiler framework)
- S2P by KPIT Cummins Infosystems Ltd.

4.1 Pros and cons

Pros:
- Portable multithreading code (in C/C++ and other languages, one typically has to call
platform-specific primitives in order to get multithreading).

- Simple: need not deal with message passing as MPI does.
- Data layout and decomposition is handled automatically by directives.
- Scalability comparable to MPI on shared-memory systems.
- Incremental parallelism: can work on one part of the program at one time, no dramatic change to code is needed.
- Unified code for both serial and parallel applications: OpenMP constructs are treated as comments when sequential compilers are used.
- Original (serial) code statements need not, in general, be modified when parallelized with OpenMP. This reduces the chance of inadvertently introducing bugs.
- Both coarse-grained and fine-grained parallelism are possible.
- In irregular multi-physics applications which do not adhere solely to the SPMD mode of computation, as encountered in tightly coupled fluid-particulate systems, the flexibility of OpenMP can have a big performance advantage over MPI.
- Can be used on various accelerators such as GPGPU.

Cons:

- Risk of introducing difficult to debug synchronization bugs and race conditions.
- Currently only runs efficiently in shared-memory multiprocessor platforms (see however Intel's Cluster OpenMP and other distributed shared memory platforms).
- Requires a compiler that supports OpenMP.
- Scalability is limited by memory architecture.
- no support for compare-and-swap.
- Reliable error handling is missing.
- Lacks fine-grained mechanisms to control thread-processor mapping.
- High chance of accidentally writing false sharing code.
- Multithreaded Executables often incur longer startup times than single threaded applications, therefore if the running time of the program is short enough there may be no advantage to making it multithreaded.
- Often multithreading is used when there is no benefit yet the downsides still exist.

5.0 Performance expectations

One might expect to get an $N$ times speedup when running a program parallelized using OpenMP on a $N$ processor platform. However, this seldom occurs for these reasons:

- When a dependency exists, a process must wait until the data it depends on is computed.
- When multiple processes share a non-parallel proof resource (like a file to write in), their requests are executed sequentially. Therefore each thread must wait until the other thread releases the resource.
- A large part of the program may not be parallelized by OpenMP, which means that the theoretical upper limit of speedup is limited according to Amdahl's law.
- $N$ processors in a symmetric multiprocessing (SMP) may have $N$ times the computation power, but the memory bandwidth usually does not scale up $N$ times. Quite often, the original memory path is shared by multiple processors and performance degradation may be observed when they compete for the shared memory bandwidth.
- Many other common problems affecting the final speedup in parallel computing also apply to OpenMP, like load balancing and synchronization overhead.

Thread affinity

Some vendors recommend setting the processor affinity on OpenMP threads to associate them with
particular processor cores. This minimizes thread migration and context-switching cost among cores. It also improves the data locality and reduces the cache-coherency traffic among the cores (or processors).

Benchmarks

There are some public domain OpenMP benchmarks for users to try.
- NAS parallel benchmark
- OpenMP validation suite
- OpenMP source code repository
- EPCC OpenMP Microbenchmarks

6.0 Conclusion

We have evaluated the performance of various multicore processors in multisharing systems. The effect of OpenMP as a programming method: Several OpenMP directives were just inserted into the source code, and the performance became higher as the number of cores increased in the most of applications. Of course, Open MP directives are not always applicable easily to any applications, and we cannot always get the results we expected. The memory bandwidth and synchronization performance as an SMP multicore processor and their impact on the overall system performance is good. This system will improve the synchronization performance in our observation and will improve the power efficiency. In future internal memories to speed up synchronization and effect of spinlock for synchronization under multiple parallel workloads can be examined.

References


