Chapter -3
Technology Used

This chapter describes an overview of the technology and its features that have been used to evaluate the performance of parallel algorithms. The Intel C++ has support of OpenMP to implement the parallel algorithms and evaluate its performances (execution time) on shared memory multiprocessor system.

3.1 Introduction

OpenMP (open multiprocessing) is an open standard for platform-neutral parallel programming [12]. Since its introduction in 1997, OpenMP has steadily grown in popularity, especially among beginners to parallel programming. OpenMP uses the fork-join model of parallel execution using threads. OpenMP provides an excellent support for data and task parallelism.

OpenMP API (Application Programming Interface) was developed to enable portable shared memory parallel programming. The API has been designed to permit an incremental approach to parallelizing an existing sequential code, in which the portions of a code can be parallelized in successive steps. OpenMP provides a platform independent set of compiler directives, function calls, and environment variables that explicitly instruct the compiler to use the parallelism in the application. The software threading model on the computing platform plays an important role to understand and enhance the performance of the parallel applications.

The Idea of OpenMP

OpenMP builds on a large body of work that supports the requirement of programs for execution by a collection of cooperating threads [4]. A thread is a runtime entity that is able to independently execute a stream of instructions. The operating system creates a process to execute a program. If multiple threads collaborate to execute a program, they will share the resources including the address space of the corresponding
process. The individual threads need just a few resources of their own: a program counter and an area in memory to save variables that are specific to it. Threads running simultaneously on multiple processors or cores may work concurrently to execute a parallel program.

A Multithreaded program can be written in various ways, some of which permit complex interactions between threads. OpenMP supports fork-join programming model [17], which describes in Figure 3.1. Under this approach, the program starts as a single thread of execution, just like a sequential program. The thread that executes this code is referred to as the initial thread. Whenever a parallel construct is encountered by a thread while it is executing the program, it creates a team of threads (i.e. fork), it becomes the master of the team, and collaborates with the other members of the team to execute the code dynamically enclosed by the construct. At the end of the construct, only the original (master) thread of the team continues; all other threads terminated (i.e. join). OpenMP provides the notation for indicating the regions of a program that could be executed in parallel. Each portion of code enclosed by a parallel construct is called a parallel region.

![Figure 3.1 The fork-join programming model supported by OpenMP.](image)

3.2 OpenMP Features

The OpenMP API (Application Program Interface) comprises a set of compiler directives, runtime library routines, and environment variables to specify shared-memory parallelism in C/C++ programs. OpenMP routine generally affects only those threads that encounter it. Most of the directives are applied to a structured block of code,
a sequence of executable statements in C/C++ (which may be a compound statement with a single entry and single exit). **OpenMP** provides construct to:

1. Create teams of threads for parallel execution,
2. Specify how to share work among the members of a team,
3. Declare both shared and private variables, and
4. Synchronize threads, and enable them to perform certain operations exclusively.

### 3.2.1 Creating Threads

**OpenMP** have been used to parallelize the code by creating parallel regions. A team of threads have been created to execute the code in a parallel region of the program. Number of threads created is according to number of processors available in the system. To achieve this, we specify the parallel region by inserting a *parallel directive* immediately before the code that we want to execute in parallel. At the end of a parallel region there is implicit *barrier synchronization*: this means that no thread can progress until all other threads in the team reached to that point in the program. Afterwards, program execution continues with the threads that previously existed. If a team of threads executing a parallel region encounters another parallel directive, each thread in the current team creates a new team of threads and becomes its master.

### 3.2.2 Sharing Work among Threads

If we don’t specify how the work in a parallel region is to be shared among the executing threads, then they will each redundantly execute all of the code. This approach does not speed up the execution time. The **OpenMP** *work-sharing* directives have been used to distribute the computations in a structured block of code among the *threads*. Implicit *barrier synchronization* also exists at the end of a work-sharing construct. The use of work-sharing method has a huge performance increase of the program.

**Work Sharing and Loops:** The most common *work-sharing* approach is to distribute the work in *for* loop among the *threads* in a team. To accomplish this, we inserted the `#pragma parallel for` directive immediately before each loop within a parallel region. **OpenMP** strategy for sharing the work among threads in loops is to assign one or
more disjoint sets of iterations to each thread. We specified the strategy to partition the iterations and to assigns one contiguous chunk of iterations to each thread. If we don’t specify a strategy, then implementation-defined defaults have been used.

**Other Work-Sharing Strategies:** Other approaches may be used to assign work to threads within a parallel region. One approach consists of giving distinct pieces of work to the individual threads. This approach is suitable when independent computations are to be performed and the order in which they are carried out is irrelevant. It is also possible to specify that just one thread should execute a block of code in a parallel region.

### 3.2.3 Thread Synchronization

**OpenMP** has the *synchronization constructs* and *library routines* to coordinate tasks and data access in parallel regions. It also has *environment variables* to control the runtime environment of **OpenMP** parallel programs. **OpenMP** makes no guarantee that input or output to the same file is synchronous when executed in parallel. Synchronizing the actions of threads is necessary in order to ensure the proper ordering of their accesses to shared data and to prevent data corruption. Many mechanisms have been proposed to support the synchronization needs of a variety of applications [18, 19, 28]. Thread coordination is one of the toughest challenges of shared-memory programming. **OpenMP** provides for implicit *synchronization*. By default, **OpenMP** gets threads to wait at the end of a worksharing construct or parallel region until all threads in the team executing it have finished their portion of the work. Only then they can proceed. This is known as a *barrier*.

In the parallel code, it is required to guarantee that only one thread at a time works on a piece of code is called *synchronization*. **OpenMP** has several mechanisms that support *synchronization*. The Synchronization points are those places in the code where *synchronization* has been specified, either explicitly or implicitly. Synchronization points include *explicit* and *implicit barriers*, the start and end of critical regions, points where *locks* are acquired or released.
3.2.4 Other Features

Procedures, subroutines and functions complicate the use of parallel programming. In order to accommodate them, major changes are required to a program. One of the innovative features of OpenMP is that the directives may be inserted into the procedures that are invoked from within a parallel region.

**Number of Threads and Thread Numbers:** It is important to control the number of threads that execute a parallel region. OpenMP provides facility to specify this number prior to program execution via an environment variable, after the computation has begun via a library routine, or at the start of a parallel region. If this is not done, then the implementation chooses the default number of threads. OpenMP assigns consecutive numbers, starting from 0, to each thread in a team in order to identify them.

3.3 OpenMP Memory Model

The OpenMP is based on the shared-memory model shown in Figure 3.2. In this model, by default, data is shared among the threads and is visible to all of them. Sometimes, we required variables that have thread-specific values. When each thread has its own copy of a variable, so that it may potentially have a different value for each of them, this is called private variable. As an example, when a team of threads executes a parallel loop, each thread needs its own value of the iteration variable. Data declared may be shared or private with respect to a parallel region or worksharing construct. The use of private variables reduced the frequency of updates to shared memory. Thus, they help to avoid hot-spots, or competition for access to certain memory locations. Threads store their private data at run time in its own special region in memory known as the thread stack.

*Synchronization* is an enforcing mechanism used to impose constraints on the order of execution of threads. The features of the Shared memory model are:

1. All threads have access to the same global, shared memory.
2. Threads also have their own private data.
3. By synchronizing access to (protecting) globally shared data.
3.4 OpenMP Programming Model

OpenMP encourages structured parallel programming and based on distributing the work in loop among threads. OpenMP parallel programs can be written by assigning the work explicitly to different threads and this approach leads to highly efficient code. In a program, synchronization can be inserted manually to ensure that accesses to shared data are correctly controlled.

A style of programming that achieves high efficiency is based on method to create the sub-domains (a strategy called domain decomposition) and assigns them to the threads. Each thread then works on its own portion of the data. This strategy is referred to as SPMD (single program multiple data) programming.

3.4.1 Parallel Correctness

Ensuring the correctness of parallel algorithm is one of the major problems in shared-memory programming model. The important error in shared memory parallel programming is data race, which is very difficult to detect. The data race problem arises when two or more threads access the same shared variable without any synchronization,
and at least one of them making changes to variable. Each thread executes their instructions at slightly different speeds, and the work of the operating system sometimes affects the performance of parallel algorithm that varies from one run to another.

### 3.4.2 Performance Considerations

How much reduction of the execution time can be expected from OpenMP parallelization using shared memory? Let $T_1$ be the execution time of an application on 1 processor, then in an ideal situation, the execution time on $P$ processors should be $T_1/P$. If $T_P$ denotes the execution time on $P$ processors, then the ratio $S = T_1/T_P$ is known as the parallel speedup. Virtually all programs contain some regions that are suitable for parallelization and other regions that are not. By using an increasing number of processors, the time spent in the parallelized parts of the program is reduced, but the sequential section remains the same. Finally, the execution time is completely dominated by the time taken to compute the sequential portion, which puts an upper limit on the expected speedup. This effect known as Amdahl’s law, formulated as

$$S = 1/(f_{par}/P + (1 - f_{par})),$$

Where, $f_{par}$ is the parallel fraction of the code and $P$ is the number of processors.

In the ideal case when all of the code runs in parallel, $f_{par} = 1$, the expected speedup is equal to the number of processors used. To improve the performance (execution time) of parallel algorithm, it is require to select several regions of the sequential code for parallelization, and to ignore the rest, which remains sequential. Thus Parallel speedup is defined as the improvement in performance relative to the “best” sequential algorithm for the problem at hand. The obstacles along the way to perfect linear speedup are the overheads introduced by forking and joining threads, thread synchronization, and memory accesses. A measure of a program’s ability to decrease the execution time of the code with an increasing number of processors is referred to as parallel scalability.
3.5 OpenMP Execution Model

As we have described that OpenMP uses the *fork-join* model of parallel execution. It supports program to execute correctly both as parallel programs and as sequential programs. An OpenMP parallel program begins as a single thread of execution, called the *initial thread*. When any thread encounters a *parallel construct*, the thread creates a team of zero or more additional threads and becomes the master of the newly created team. A set of *implicit tasks* one per thread is generated. The code for each task is defined by the code inside the *parallel construct*. Each task is assigned to a different thread in the team and it is always executed by the thread to which it is initially assigned.

The task region of the task being executed by the encountering thread is suspended, and each member of the new team executes its implicit task. There is an implicit barrier at the end of the *parallel construct*. Beyond the end of the *parallel construct*, only the master thread resumes execution, by resuming the task region that was suspended upon encountering the *parallel construct*. When any team encounters a *worksharing construct*, the work inside the construct is divided among the members of the team, and executed cooperatively instead of being executed by every thread. This approach increases the performance of the application.