International Journal Of Research In Computer Engineering And Electronics. PAGE # 1 ISSN 2319-376X
VOL:3 ISSUE:2 (MARCH-APRIL'14)
ICV:4.08

Analysis of Parallel Computing, Focusing on Parallel Computing Models and Parallel Programming Models

Shweta Kumari

Abstract— Parallel computing has turn out to be an important subject in the field of computer science. It has proven to be critical when researching high performance solutions. The evolution of computer architectures towards a higher number of cores i.e. multi-core and many-core, can only confirm that parallelism is the means of choice for speeding up an algorithm. My goal is to present a set of theoretical and technical concepts that are frequently required to understand the parallel computing, its models and algorithms. In this paper I briefly discuss the design patterns in parallel computing. Focus is on a large variety of parallel computing and programming models. I talk about memory consistency models which provide the contract between software and hardware. I describe general Parallel Programming Methodologies and some parallel programming tools. In this article I study few implementation issues of parallel programming. By understanding above mentioned topics, readers can overcome many of the technical limitations found beside the way and can design better algorithms and achieve speedups.

Index Terms— Parallel computing, computing models, design patterns, implementation issues, programming models, parallel programming tools, evaluation.

1. INTRODUCTION

The aim of parallel computing is to boost an application's performance by executing the application on multiple processors. Traditionally parallel computing was associated with the high performance computing community, now it is becoming more common for the mainstream computing because of the recent progress of commodity multi-core architecture. For understanding parallel computing more closely we have discussed many factors related to it. We described some models like PRAM, UMP, and BSP. A few programming methodologies are used such as fosters PCAM method, incremental parallelization, automatic parallelization, and some more researches are done. This paper is organized in ten sections; the research begins with the design pattern for parallel computing in section two followed by a survey of the proposed models of parallel computation in section three. The fourth part examines the parallel programming models. Section five discusses some of the memory consistency models used in a parallel computing paradigm. Brief review of general parallel programming methodologies is given in section six, and in section seven I talk about parallel programming tools. This article presents a survey of parallel computing implementation issued in brief such as races, out of order execution, message buffering, and hardware errors. Ninth section put forward seven criteria to qualitatively evaluate parallel programming models. We conclude some observations at the end.

- *Shweta Kumari, currently pursuing masters degree program in computer science and engineering in Galgotias University, India.*

2. Design pattern for parallel computing

Design pattern for parallel computing is the result of two different directions of research. The goal of first direction of research was to identify key influences on computer architecture, aimed to analyze vast varieties of computing. This led to thirteen computation patterns of parallel computing. The second direction of research was on architecting large piece of software. This research led to the identification of a series of architectural styles or patterns [1].

A design pattern is a general solution to recurring problem, which occurs within a well-defined context. These are written in a highly structured format and capture essential elements of the required solution, such that a software designer can quickly and easily find what he or she needs to solve a problem. With the help of design patterns a software designer can develop many solution alternatives to a problem.

2.1. Patterns for parallel programming

For parallel programming in parallel computers, we combine the computational and structural patterns with

the parallel programming design pattern languages [2]. These parallel programming patterns define a distinct pattern language for parallel programming (PLPP). The PLPP emphasises on patterns relevant to cluster and shared-memory multiprocessor computers.

3. Models of parallel computing

A model of parallel computation is a parameterized description of a class of machines [3]. Some models of parallel computing are discussed below:

3.1. Parallel Random Access Machine (PRAM)

The PRAM model was proposed by Fortune and Wyllie [4]. It is a simple extension of the Random Access Machine (RAM) model used in the design and analysis of sequential algorithms. In PRAM a set of processors are connected to a shared memory, and a global clock feeds processors as well as memory. The execution of any instruction, takes exactly one unit of time and the shared memory can be accessed by any number of processers simultaneously. The memory model of the PRAM is the strongest consistency model known [5]. An EREW PRAM allows a memory location, exclusive read and exclusive write. CREW PRAM allows concurrent read but exclusive write. CRCW PRAM allows concurrent read and write to the same memory location in the same clock cycle. CROW PRAM is a little limited form, here each memory cell may be written by only one processor known as the cell's owner.

3.2. Unrestricted Message Passing (UMP)

A message-passing multicomputer also known as distributed memory machine consists of a number of RAMs which run asynchronously. They communicate via messages sent over a communication network. The send and receive commands can be of two types (i) blocking, i.e. the processors get synchronized and (ii) non-blocking, i.e. the sending processor puts the message in a buffer, the message is forwarded by the message- passing subsystem to the receiving processor. It get buffered there until the receiving processor executes the receive command. The operations performed locally are treated as in a RAM, Point-to-point non-blocking communications are modelled by the LogP model.

3.3. Bulk Synchronous Parallelism (BSP)

The BSP model, proposed by Valiant in 1990[6] enforces a structuring of message passing computations as a (dynamic) sequence of barrier-separated supersteps, where each superstep consists of a computation phase operating on local variables only, followed by a global interprocessor communication phase.

3.4. Data Parallel Models

In data parallel computing the same scalar computation involves the element wise application to several elements of one or many operand vectors creating a result vector. All element computations must be independent of each other hence may be executed in parallel, or in a pipelined way in any order. A special case of data parallel computing is single instruction multiple data (SIMD) computing.

3.5. Task Parallel Models and Task Graphs

Many applications can be considered as a set of tasks, each task solving a part of the problem. These tasks may communicate with each other during their execution. Such collection of tasks may be represented by a task graph, where arcs represent communication, i.e. data dependencies and nodes represent tasks. Task graphs can occur at several levels of granularity.

4. Parallel programming models

Parallel computing should be analyzed on the basis of the communication between the processors and their programs. PRAM uses shared memory model, whereas LogP and BSP use a message passing model. These two models are parallel programming models. A parallel programming model is an abstraction of the programmable aspects of a computing model. While computing models in section 3 are useful for algorithm design and analysis. Following are some parallel programming models which have been implemented by modern APIs.

4.1. Shared memory

In the shared memory model, read and write can be performed on a common memory. This programming model works good with the PRAM computing model, is useful for multi-core and GPU based solutions. When concurrent threads read and write on the same memory locations, we must supply an explicit synchronization and control mechanism such as monitors [7], and semaphores [8].

4.2. Message passing

In a message passing programming model processors communicate asynchronously or synchronously by sending and receiving messages containing words of data. In this model, emphasis is placed on communication and synchronization making distributed computing the main application for the model. This programming model works naturally with the BSP and LogP models.

4.3. Implicit

Implicit parallelism is a high-level tool, capable of achieving a degree of parallelism automatically from a sequential piece of source code. Its advantage is that all the hard work is done by the tool, achieving actually the same performance as a manual parallelization. Its disadvantage is that it only works for simple problems such as *for* loops with independent iterations.

4.4. Algorithmic skeletons

Algorithm skeletons provide an important abstraction layer for implementing a parallel algorithm. Algorithm skeletons, also known as parallelism patterns, were proposed by Cole in 1989 and published in 1991 [9]. The model is based on a set of parallel computing patterns known as skeletons that are available to use.

5. Memory Consistency Models

The memory consistency model provides the contract between software and hardware, the memory remains consistent and the program is guaranteed to execute correctly. It can be divided into following categories:

5.1. The Sequential Memory Consistency Model

The Sequential Consistency Model is simplest to understand since it extends the uniprocessor model, and hence, follows the basic assumptions which are made about sequential memory. The memory consistency is maintained through hardware, and therefore allows the programmer to write code which follows the intuitive memory model.

There are two requirements for maintaining sequential consistency [5]. First is program order requirement, here program order must be maintained among memory operations in a single processor. And second is write atomicity requirement, here a single sequential order must be maintained among all operations. There are three types of memory operation pairs: a read-after-write, a write-after-write, and a read/write-after-read. Following is an example of the first case: a read-after-write [5].

Initially:

```
flagA = flagB = 0;
```

Processor 1:

```
flagA = 1;
if(flagB == 0)
        // Enter Critical Section
        flagA = 0;
```

Processor 2:

```
flagB = 1;
if(flagA == 0)
        // Enter Critical Section
        flagB = 0;
```

This is Dekker's Algorithm for Critical Sections: This code is guaranteed to execute correctly on a sequentially consistent system due to the Program Order requirement. Now the second condition to ensure the appearance of write atomicity is to prohibit any reads from occurring on any memory location for which there is an outstanding write; this can be accomplished with an acknowledgment of invalidates or updates sent by all processors.

Initially:

```
varA = varB = varC = 0;
```

Processor 1:

```
varA = 1;
varB = 1;
```

Processor 2:

```
varA = 2;
varC = 1;
```

Processor 3:

```
while(varB != 1) {;} // Busy wait
while(varC != 1) {;} // Busy wait
reg1 = varA;
```

Processor 3:

```
while(varB != 1) {;} // Busy wait
while(varC != 1) {;} // Busy wait
reg2 = varA;
```

In a sequentially consistent system, this code [5] is guaranteed to run correctly due to the write atomicity requirement.

5.2. Relaxed Memory Consistency Models

The Relaxed Memory Consistency Models are the collective result of group of memory consistency models. These models relax one or more of the requirements of sequential consistency.

5.3. Transactional Memory Models

After the development of many relaxed consistency models, researchers invented a way to combine both cache coherency and memory consistency models in a single, software or hardware supported communication model for shared memory which is easy to use. This is known as Transactional Memory models. A transaction is a sequence of operations executed by a single thread. After completion of operations, the transaction does one of these: If any memory operation doesn't conflict with other memory operation of another transaction, the transaction commits and it takes effect; otherwise, it aborts and its effects are discarded.

6. General Parallel Programming Methodologies

Here we briefly review the general parallel programming methodologies (PPM).

6.1. Fosters Method

A researcher Foster [10] suggests that the design of a parallel program should start from an existing (possibly sequential) algorithmic solution to a computational problem by partitioning it into many small tasks and identifying dependences between these that may result in communication and synchronization, for which suitable structures should be selected. These first two design phases, partitioning and communication, are for a model that puts no restriction on the number of processors.

6.2. Incremental Parallelization

Parallel programming languages such as HPF and OpenMP, are designed as a semantically consistent extension to a sequential base language, such as Fortran and C, which allow to start from sequential source code and can be parallelized incrementally.

6.3. Automatic Parallelization

Automatic parallelization is of high importance to industry but is very difficult. It is of two forms: static parallelization supported by a smart compiler, and run-time parallelization supported by the run-time system or the hardware.

6.4. Skeleton Based and Library Based Parallel Programming

Skeleton programming also known as structured parallel programming [11, 12] restricts many ways of expressing parallelism to compositions of only a few, predefined patterns, so-called skeletons. Skeletons [11, 13] are generic, portable, and reusable basic program building blocks for which parallel implementations may be available.

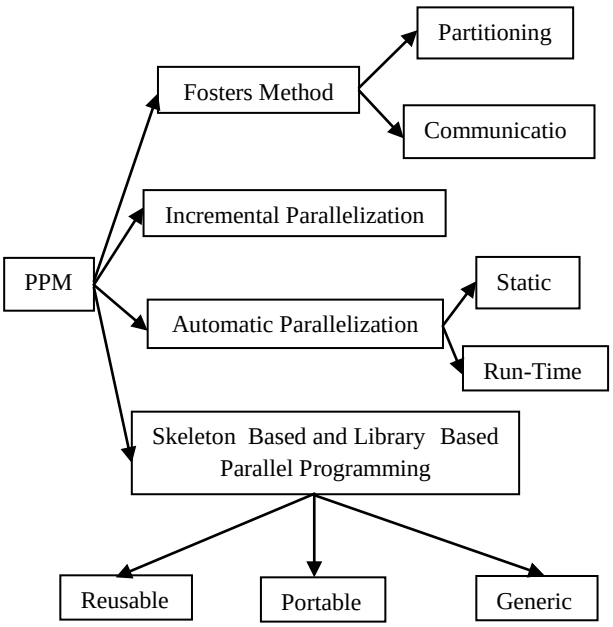

Fig1: Pictorial Representation of Parallel Programming Methodologies

7. Parallel programming tools

There are many programming tools are available for the implementation of parallel programs. The selection of parallel programming tool to be used depends on the characteristics of problem to be solved. Some of these are discussed below.

7.1. PVM

PVM (Parallel Virtual Machine) is extensively used for message passing library, fashioned to support the development of parallel programs executed on a set of interconnected heterogeneous machines. A set of tasks is contained in PVM program, which performs communication in a parallel virtual machine by exchanging messages. To control the sending and receiving of messages, a managing process is executed in every machine. The parallel programs can be written in C, C++, or FORTRAN.

7.2. MPI

MPI (Message Passing Interface) enables program portability among different parallel machines. It cannot handle issues like debugging and program structuring, because it just defines a message passing programming interface, not a complete parallel programming environment. For point-to-point communication between pairs of tasks MPI contains routines, which can be in two modes, blocking and non-blocking. Communication is available in three modes (i) ready, (ii) standard, and (iii) synchronous.

7.3. Linda

Linda is based on the idea of associative shared memory, it is a language which offers a set of primitives meant for process construction and communication. The shared memory is the tuplespace, it contains a group of tuples(or data registers). Information is accessed by its content, not by its address in case of associative access to memory. Every time when a process wants to communicate with another one, it generates a new tuple and spaces it in the tuplespace, and the receiver process may access this tuple since it is written in the shared memory space. The programmer develops a parallel program implementation by writing a C, C++, or FORTRAN code by means of the basic Linda operations to access the tuplespace: write a tuple, read a tuple, take a tuple from the tuplespace, place a new tuple in the tuplespace, and create a new process.

7.4. HPF

Processor mapping and data distribution onto the physical processors can be controlled by the programmer using six HPF compiler directives: ALIGN, DISTRIBUTE, PROCESSORS, TEMPLATE, REDISTRIBUTE, and REALIGN. At the start the

programmer has to relate the group of arrays by ALIGN. After that these aligned substances are distributed to the processors using DISTRIBUTE directive. REALIGN and REDISTRIBUTE are the vibrant forms of the DISTRIBUTE and ALIGN directives, allowing data mapping to change throughout the program execution. To identify the shape and the size of a set of abstract processors, the PROCESSORS directive is used. The TEMPLATE directive describes a conceptual object to be used for alignment and distribution operations.

7.5. Threads

Threads are not specially related to parallel program, they are common operating systems concept. Due to their significance in providing support for concurrent programming, it is very important to understand them. Several threads may perform in the context of single process, and can communicate by means of global memory allocated to the associated process. Programming with threads is very useful in shared-memory machines.

8. Implementation issues

Most programming models for parallel computing present chances for good performance, but at the cost of a greater chance for program error. This section briefs a few of the most frequent programming errors for programming models.

8.1. Races

One of the most hazardous and frequent errors of parallel programming is a race condition. This takes place in parts of the code where a race among two or more threads of execution concludes the behaviour of the program. The program starts behaving incorrectly, if the wrong thread wins the race.

8.2. Out-of-Order Execution

In writing algorithms for parallel programming model, it should be ensured that data on one thread is not accessed by any other, until a few conditions is satisfied. Locks are often used for this purpose. Unluckily, locks are generally quite expensive to execute, hence programmers are often appealed to use simple flag variables to mediate access to the data. With flag variables it is very tough to guarantee that either the compiler or the processor will protect the order of the operations inside a single thread, which appears to be independent statements. Hence within a single thread, the order of operations is not guaranteed and it is known as out of order execution.

8.3. Message Buffering

Message passing combines data transfer and notification into a single routine and there is no direct access to the memory of another process. Hence message passing programs are very immune to race conditions. The real risk in message passing begins by using buffered send operations in unsafe means. The programme may function correctly for few inputs but may laid to deadlock for others.

8.4. Hardware Errors

Parallel computers are frequently used for the most difficult problems. And another source of problems in them is probability of an error in the computer hardware, it is low but not zero. This occurs mainly with high-performance interconnect networks.

9. Qualitatively Evaluation of Parallel Programming model

In this section, we describe seven criteria to qualitatively evaluate a parallel programming model.

9.1. System Architecture

Two architectures are considered: shared memory and distributed memory. In shared memory architecture systems such as an SMP/MPP node, all processors share a single address space. In such models, applications can run and make use of only processors inside a single node. Whereas in distributed memory architecture systems such as a cluster of compute nodes, there is one address space per node.

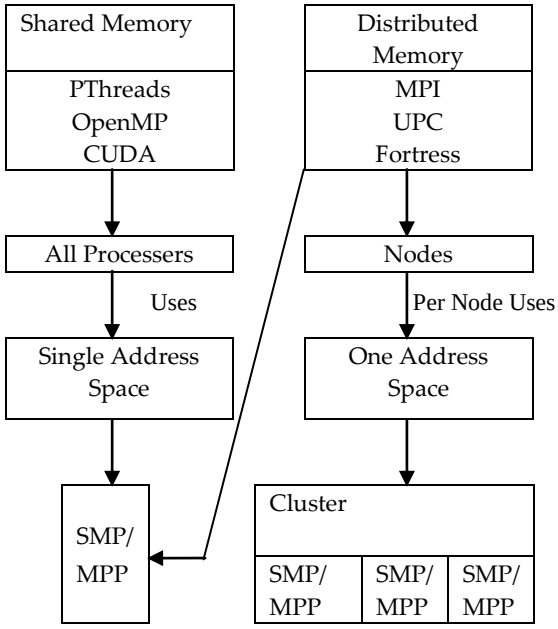

Fig2: Six Programming Models and their Supported System Architecture

Fig2 illustrates the supported system architecture of the six programming models. As can be seen, Pthreads, OpenMP and CUDA support shared memory

architecture, and thus can only run and utilize processors within a single node. On the other hand, MPI, UPC and Fortress also support distributed memory architecture so that applications developed with these model can run on single node (i.e. shared memory architecture) or multiple nodes.

9.2. Programming Methodologies

Focus should be at how parallelism abilities are exposed to programmers. Some examples are: API, special directives, new language specification, etc.

9.3. Worker Management

This monitors at the creation of the unit of worker, threads or processors. Worker management is implied when there is no need for programmers to manage the lifetime of workers. They need to only specify, the number of unit of workers required or the section of code to be execute in parallel. In explicit approach, programmer needs to code the construction and destruction of workers.

9.4. Workload Partitioning Scheme

Worker partitioning describes how the workload are divided into smaller chunks which is called tasks. In implicit approach, programmers require to only specify that a workload can be processed in parallel. How the workload is actually partitioned into tasks need not be managed by programmers. In contrast, with the explicit approach, programmers require to manually decide how workload is partitioned.

9.5. Task-to-Worker Mapping

Task-to-worker mapping defines how tasks are map onto workers. In the implicit approach, programmers do not need to specify which worker is responsible for a particular task. In contrast, the explicit approach requires programmers to manage how tasks are assigned to workers.

9.6. Synchronization

Synchronization defines the time order in which workers access shared data. In implicit synchronization, there is no or little programming effort done by programmers: either no synchronization constructs are needed or it is sufficient to only specify that synchronization is needed. In explicit synchronization, programmers are required to manage the worker's access.

9.7. Communication Model

This aspect looks at the communication concept used by a model.

10. Conclusion

Parallel computing has evolved significantly from being a matter of high equipped data centres and supercomputers to almost every electronic device. Today, the field of parallel computing is having one of its best moments in history of computing and its importance will only grow as long as computer architectures keep evolving to a higher number of processors. Speedup and efficiency are the most important measures in a parallel solution and will continue to be in the following years. At the end of this review of parallel computing models, their programming models and many more issues we may observe some current trends and speculate a bit about the future of parallel programming models. There is still much work to be done in the field of parallel computing. The challenge for massive parallel architectures in the following years has become more flexible and energy efficient. At the same time, the challenge for computer science researchers
will be to design more efficient algorithms by using the features of these new architectures.

11. References

[1] M. Shaw and D. Garlan, "Software architecture: perspectives on an emerging discipline," 1996.

[2] T. G. Mattson, B. A. Sanders, and B. L. Massingill, "Patterns for parallel programming," Addison-Wesley Professional, 2004.

[3] Ferri Abolhassan, Reinhard Drefenstedt, Jorg Keller, Wolfgang J. Paul, and Dieter Scheerer, "On the physical design of PRAMs," Computer J.,36(8):756-768, December 1993.

[4] S. Fortune and J. Wyllie, "Parallelism in random access machines," In Proc. 10th Annual ACM Symp. Theory of
Computing, pages 114-118, 1978.

[5] Sarita V. Adve and Kourosh Gharachorloo, "Shared Memory Consistency Models: a Tutorial," IEEE Comput.,29(12):66-76, 1996.

[6] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman, "LogGP
Incorporating long messages into the LogP model for parallel computation," Journal of Parallel and Distributed Computing, 44(1):71-79, 1997.

[7] C. A. R. Hoare, "Monitors: an operating system structuring concept," Commun. ACM, 17(10):549–557,October 1974.

[8] N. Dunstan, "Semaphores for fair scheduling monitor conditions," SIGOPS Oper. Syst. Rev., 25(3):27–31,May 1991.

[9] M. Cole, "Algorithmic skeletons: structured management of parallel computation," MIT Press, Cambridge, MA, USA, 1991.

[10] Ian Foster, "Designing and Building Parallel Programs," Addison Wesley, 1995.

[11] Murray I. Cole, "Algorithmic Skeletons: Structured Management of Parallel Computation," Pitman and MIT Press, 1989.

[12] Susanna Pelagatti, "Structured Development of Parallel Programs," Taylor&Francis, 1998.

[13] J. Darlington, A. J. Field, P. G. Harrison, P. H. B. Kelly, D. W. N. Sharp, and Q. Wu, "Parallel Programming
Using Skeleton Functions," In Proc. Conf. Parallel Architectures and Languages Europe, pages 146-160. Springer LNCS 694, 1993.