A Comparative Study of Parallelization Paradigms

Imran Ahmad, Aneel Rahim, Adeel Javed, Khalid Haseeb* and G.Qasim**

Abstract

Parallel programming is a challenging job in distributed community and depends upon the availability of suitable software tools and environment. And consequently, software developers have to face constraints such as non-determinism, communication, synchronization, data partitioning and distribution, load balancing, fault-tolerance, heterogeneity, shared or distributed memory, deadlocks, and race conditions. In this communication, a comparative study of parallelization paradigms is presented which facilitates the parallel programmer to choose the appropriate and suitable paradigms in different scenarios.

Keywords: Parallel programming, Parallelization, Parallelization paradigms, Master worker/slave, Single program multiple data, Divide and conquer, Data pipelining, Speculative

1. Introduction

A concurrent program specifies two or more processes that cooperate in performing a task. In a parallel program, concurrent processes are executed in parallel on multiple processors. For an application program, decreasing execution time, increasing fault-tolerance, explicitly exploiting the inherent parallelism of an application, overcoming memory constraints, cost saving and taking the advantage of non-local resources are the motivations for parallel processing [7]. Concurrent programming is the activity of constructing a program containing multiple processes that cooperate in performing some task. Given a specification of the problem to be solved, decisions have to be made about what and how many processes to use and how they should interact. These decisions are affected by the application and by the underlying hardware on which the program will run. Whatever choice is made, a critical problem is ensuring that communication between processes is properly synchronized [8].

Parallelism may be implicit and explicit. In implicit parallelism, the user does not specify, and thus cannot control, the scheduling of calculations and/or the placement of data; in explicit parallelism, the programmer is responsible for most of the parallelization effort such as task decomposition, mapping tasks to processors, and the communication structure. Explicit parallelism obtains a better efficiency than parallel languages or compilers that use implicit parallelism. Parallelism makes available more computational performance than is available in any single processor, although getting this performance from parallel computers is not straightforward [9].

The main software issue in parallel programming is to decide between either porting existing sequential applications or developing new parallel applications from scratch. There are three strategies for creating parallel applications: the first strategy is based on automatic parallelization, the second is based on the use of parallel libraries, while the third strategy-major recording-resembles from-scratch application development. Levels of parallelism can also be based on the lumps of code (grain size) that can be a potential candidate for parallelism. Table 1.1 lists categories of code granularity for parallelism.

The different levels of parallelism are depicted in Figure 1.1. Among the four levels of parallelism, the first two levels are supported
transparently either by the hardware or parallelizing compilers. The programmer mostly handles the last two levels of parallelism. The three important models used in developing applications are shared-memory model, distributed memory model (message passing model), and distributed-shared memory model.

<table>
<thead>
<tr>
<th>Grain Size</th>
<th>Code Item</th>
<th>Parallelised by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Fine</td>
<td>Instruction</td>
<td>Processor</td>
</tr>
<tr>
<td>Fine</td>
<td>Loop/Instruction block</td>
<td>Compiler</td>
</tr>
<tr>
<td>Medium</td>
<td>Standard One Page Function</td>
<td>Programmer</td>
</tr>
<tr>
<td>Large</td>
<td>Program-Separate heavyweight process</td>
<td>Programmer</td>
</tr>
</tbody>
</table>

The methodological approach for designing parallel algorithms provides maximum the range of options, that provides mechanisms for evaluating alternatives, and that reduces the cost of backtracking from wrong choices, allows the programmer to focus on machine-independent issues such as concurrence in the early stage of design process and machine-specific aspects of design are delayed until late in the design process.

3. Parallel Programming Paradigm:

The computation resources define the level of granularity which are efficiently supported on the system. The type of parallelism reflects the structure of either the application or the data and both types may exist in different parts of the same application. Parallelism arising from the structure of the application is named as functional parallelism. In this case, different parts of the program can perform different tasks in a concurrent and cooperative manner. But parallelism may also be found in the structure of the data. This type of parallelism allows the execution of parallel processes with identical operation but on different parts of the data. In parallel applications, popular paradigms includes Task-Farming (or Master/Slave), Single Program Multiple Data (SPMD), Data Pipelining, Divide and Conquer, and Speculative Parallelism [2].

3.1 Tasks-Farming (or Master/Slave)

The master splits up the problem(s) into small tasks, distributes them among a farm of slave processes and gathers the partial results to produce the ultimate results of the computation. The slave processes execute in a simple cycle i.e. gets a message with the task, processes it and then to send result to the master. Generally, the communication takes place between master and slaves only. The master-slave model is used to model task and data parallelism both. The master/slave paradigm has been successfully used for a wide range of parallel applications. It is a good programming model for applications for distributed and heterogeneous Grid resources [3]. Task farming may either use static load-balancing or dynamic load-balancing.

3.1.1 Characteristics of Master Slave:
Adaptable: To change the conditions of the system, not only the load of the processors but also the possible reconfiguration of the system resources. Master slave model is adaptable if it utilizes the dynamic load balancing.

Robustness: Paradigm can respond quite well due to failure of some processors. It simplifies the creation of robust applications which is capable of surviving the loss of slaves or even the master.

Efficient and Scalable: Paradigm can achieve high computational speeds and degree of scalability.

Less Communication overhead: Parallel algorithm can be executed on different processors with different data inputs simultaneously. In such applications, as there is no dependence between different runs so there is no need for communication or coordination between the processes.

Centralized Control: For a large number of processors, centralized control of the master process can become a bottleneck to the applications. The scalability of the paradigm by extending the single master to a set of masters, each of them controlling a different group of process slaves is possible.

Modular Structure: The master-slave model separates control and computation into two distinct programs which leads to a better software structure.

Data Consistency: The usage of the shared data in the master-slave model usually has simple and clear fixed patterns. So, the data consistency of this model is less complex and is easier than other distributed computing scenarios.

3.2 Single-Program Multiple-Data (SPMD)

In SPMD paradigm, each process executes basically the same piece of code but on a different part of the data. This involves the splitting of application data among the available processors. This type of parallelism is also referred to as geometric parallelism, domain decomposition, or data parallelism. Figure 1.3 presents a schematic representation of this paradigm. Using a suitable load balancing algorithm can enhance the efficiency of SPMD model.

An SPMD application can typically be structured into three major components: (a) the single code which is replicated for the execution of a task; (b) the load balancing strategy; and (c) a skeleton (which initializes and terminates the application, manages the tasks, and controls the execution of the other parts). The first component is specific to each application, but the other two usually are not. SPMD applications can thus be modeled by a framework [4].

3.2.1 Characteristics of SPMD:

More Communication overhead. Processors communicate with neighboring processors and the communication load will be proportional to the size of the boundary of the element, while the computation load will be proportional to the volume of the element.

Global Synchronization. It may also be required to perform some global synchronization periodically among all the processes. The communication pattern is usually highly structured and extremely predictable.

Self-Generation of data. The data may initially be self-generated by each process or may be read from the disk during the initialization stage.

Load-Balancing. SPMD applications can be very efficient if the data is well distributed by the processes and the system is homogeneous. If the processes present different work loads or capabilities, then the paradigm requires the support of some load-balancing scheme able to adapt the data distribution layout during run-time execution.

Loss Sensitive. This paradigm is highly sensitive to the loss of some process. Usually, the loss of a single process is enough to cause a deadlock in the calculation in which none of the processes can advance beyond a global synchronization point.

Simple Programming Process. Coding and debugging is usually simpler under this model than in arbitrary MIMD programs.

3.3 Data Pipelining

Data pipelining is based on a functional decomposition approach: the tasks of the algorithm, which are capable of concurrent operation, are identified and each processor executes a small part of the total algorithm. Figure 1.4 presents the structure of this model. Processes are organized in a pipeline; each process corresponds to a stage of the pipeline and is responsible for a particular task. This paradigm is often used in data reduction or image processing applications.
3.3.1 Characteristics of Data Pipelining:

Communication Pattern. The communication pattern can be very simple since the data flows between the adjacent stages of the pipeline. For this reason, this type of parallelism is also sometimes referred to as data flow parallelism.

Asynchronous Communication. The communication may be completely asynchronous.

Load Balancing. The efficiency of this paradigm is directly dependent on the ability to balance the load across the stages of the pipeline.

Robustness. The robustness of this paradigm against reconfigurations of the system can be achieved by providing multiple independent paths across the stages.

3.4 Divide and Conquer

In divide and conquer approach a problem is divided up into two or more sub problems. Each of these sub problems is solved independently and their results are combined to give the final result. In parallel divide and conquer, the sub problems can be solved at the same time, given sufficient parallelism. The popular techniques for parallel in-core divide-and-conquer are task parallelism, data parallelism, concatenated parallelism and mixed parallelism. Divide-and-conquer paradigm can be used to efficiently construct decision trees [5]. The divide-and-conquer paradigm can be used for problems which are recursive in nature. The execution of a problem instance is represented by a divide-and-conquer tree where each node represents a task or subtask. There are three generic computational operations for divide and conquer: split, compute, and join. Figure 1.5 presents this execution.

3.4.1 Characteristics of Divide & Conquer:

Communication Overhead. The splitting and recombining process also makes use of some parallelism, but these operations require some process communication. The sub problems are independent; no communication is required between processes working on different sub problems.

Referential transparency. In a divide-and-conquer application, function execution will always produce the same outputs if given the same input, a property also known as referential transparency.

Fault tolerance. Exploiting referential transparency of the divide-and-conquer paradigm makes it possible to create a fault tolerance mechanism based on redoing the work lost in crashes of processors. Because of referential transparency fault tolerance mechanism can have very low overhead.

Synchronization. The sub problems are independent so no synchronization between processes is needed.

3.5 Speculative Parallelism

Some problems have complex data dependencies, which reduces the possibilities of exploiting the parallel execution, to solve these problems speculative paradigm is used because it is quite difficult to obtain parallelism through any other paradigms. Speculative parallelism is the initiation of parallel computation on the basis of speculation about the usefulness of its result [6]. In other words, a task is spawned in the hope that its result will later be of use. There are two issues to be addressed in speculative computing. First, resources of the system are limited. Those computations which have been determined to be unwanted should be removed from the system as soon as possible to free resources for useful work. Second, computation which is deemed to be more promising should be favored in getting system resources.

3.5.1 Characteristics of Speculative:

Efficient. Networks of workstations and parallel computers are not always fully in use - some of processors are idle. These “wasted” computing resources can be utilized for speculative work.

Quality of Service. While parallel execution has been usually used to speed up a certain amount of task, speculative parallel execution of interactive programs is used to enhance amount of task during a certain interval and, consequently, enhance quality of interactive services.

Rapid Response. Much more computation can be applied with keeping good response time as the optimum technique is under utilization.

Quality of Results. Quality of results vastly depends upon the cycles the system gets which results in the accuracy in the output.

Feedback. Speculative parallelism can yield performance improvements over conventional approaches to parallel computing because when the result is determined to be required, it is
Suitable paradigm can be selected for given parallel problem to solve. In selection appropriate paradigm different factor are considered e.g. Decomposition types, Distribution, Decomposition Pattern, Adaptability, Fault tolerant, Communication, Synchronization, Efficiency, Robustness, Overhead, Load Balancing, Acceptability of Results, Loss Sensitive, Centralized Control, Scalability, Heterogeneity, Deadlock Happen. The comprehensive comparison of the Master and Slave, SPMD, Data Pipelining, Divide and Conquer and Speculative paradigms is given in the Table 1.2 below.

4. Comparison of Paradigms

Table 1.2. Comparison of parallelization Paradigms Features Master and Slave SPMD Data Pipelining Divide and Conquer Speculative

<table>
<thead>
<tr>
<th>Features</th>
<th>Master and Slave</th>
<th>SPMD</th>
<th>Data Pipelining</th>
<th>Divide and Conquer</th>
<th>Speculative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition Types</td>
<td>static</td>
<td>static</td>
<td>static</td>
<td>Dynamic</td>
<td>Algo Dependent</td>
</tr>
<tr>
<td>Distribution</td>
<td>Dynamic</td>
<td>static</td>
<td>static</td>
<td>Dynamic</td>
<td>Algo Dependent</td>
</tr>
<tr>
<td>Decomposition Pattern</td>
<td>Iterative</td>
<td>Geometric</td>
<td>Functional</td>
<td>Recursive</td>
<td>Speculative</td>
</tr>
<tr>
<td>Adaptability</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Fault tolerant</td>
<td>Yes</td>
<td>Less</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Communication</td>
<td>Less</td>
<td>High</td>
<td>Less</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Synchronization</td>
<td>No</td>
<td>Yes</td>
<td>Mostly Asynchronous</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Robustness</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Overhead</td>
<td>Less</td>
<td>High</td>
<td>Less</td>
<td>Less</td>
<td>Very High</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Acceptability of Results</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>May be</td>
</tr>
<tr>
<td>Loss Sensitive</td>
<td>Moderate</td>
<td>Very High</td>
<td>High</td>
<td>Very High</td>
<td>Less</td>
</tr>
<tr>
<td>Centralized Control</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Scalability</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Heterogeneity</td>
<td>Allowed</td>
<td>Allowed</td>
<td>Not Allowed</td>
<td>-</td>
<td>Allowed</td>
</tr>
<tr>
<td>Deadlock Happen</td>
<td>Never</td>
<td>May be</td>
<td>May be</td>
<td>May be</td>
<td>Never</td>
</tr>
</tbody>
</table>
5. Summary and Future Work:

The choice of paradigm is determined by the available parallel computing resources and by the type of parallelism inherent in the problem.

The computing resources may define the level of granularity that can be efficiently supported on the system. The type of parallelism reflects the structure of either the application or the data and both types may exist in different parts of the same application.

If the problem’s domain can be broken up into smaller domains and each process can execute the algorithm on each part of it then problem is decomposed geometrically and thus SPMD is the best choice. If the problem is based on loop execution, where each iteration can be done in an independent way. This approach is implemented through a central queue of runnable tasks, and thus corresponds to the task-farming paradigm. If the problem can be broken into several sub-problems to solve it in a parallel way. It clearly corresponds to a divide and conquer approach. If the problem can be broken down into many distinct phases, where each phase executes a different algorithm within the same problem then most used topology is the process pipelining. If the problem is quite difficult to obtain parallelism through any one of the previous paradigms then speculative paradigm is the only choice to solve the problem.

Which programming paradigm is the best depends on the nature of the given problem, the hardware components of the parallel system, the network and parallel execution model? This paper summarizes the selection of paradigm mainly on the nature of given problem. A comparative study of network or parallel execution models can help more in selection of appropriate paradigm for some given problem.

References: