International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 3 Issue 1, January - 2014

Various Ways of Parallelization of Sequential Programs

Ankita Bhalla
M.Tech (CSE)
GNDU Amritsar

Abstract

Parallelization is becoming necessity of parallel computing field. The main reason of parallelization is to compute large and complex program as fast as possible. However it is difficult to parallelize the sequential program. This paper describes about loop parallelization that allows parallelizing the loops of the programs as we know loops take most CPU time. This paper discusses about speculative parallelism in which program is parallelized while maintaining its sequential order. Some tools are briefly discussed like HydraVM and PTRAN. This paper describes about variant based parallel execution of sequential programs which introduces concept of multiple variants in a sequential programs.

Keywords: HydraVM, Manual Parallelism, Speculative Parallelism, Loop Parallelism, PTRAN, parallelization tools, Variant based competitive parallelism.

1. Introduction

Sequential program is defined as a set of instructions in a serial fashion. Instructions are executed line by line one after another. It takes so much time to compute large sequential programs which leads to performance degradation of computer systems. Parallelism is one of the key means for improving the performance of computer systems. Parallelism in programs is a fundamental characteristic of the program that denotes the independence of computations in a program. Parallelism provides the ability to perform several computations (instructions) in a program concurrently because of their independence. Parallelism exploits the concurrency.

2. Need of Parallelization

The main reason of parallelization a sequential program is to run the program faster. The need of parallel approach arises because some problems were too costly to be solved with the classical approach and we need to find the results as soon as possible. The criteria for evaluating the performance of a parallel program is the speedup used to express how many times the parallel program works faster than the sequential one, where both programs are solving the same problem. [1] The speedup formula is

$$S= T_s/T_p$$

Where
T_s is the execution time of the fastest sequential program for our problem
T_p is the execution time of the parallel program used to solve the same problem. [1]

2.1 Amdahl's law

According to the Amdahl law, if we execute the parallel program on parallel processor, some portion of it cannot be executed parallely. That portion has to be executed sequentially by single processor. Let α be the portion which cannot be parallelized. The rest of $(1 - \alpha)$ will be executed in parallel.Here N is number of processing elements. Speed up is:
$S= 1 / (\alpha + ((1- \alpha)/N))$
In Amdahl's law, problem size remains fixed. If we increase the number of processors then speed up as well as efficiency decreases. This law focuses on execution of program must be completed as fast as possible.

2.2 Gustafson's Law

This law is correlated with Amdahl's law. It basically works on limitation of Amdahl's law. Amdahl's law is limited to fixed problem size. Gustafson's law states that as we increase the number of processing elements in parallel system problem size must increases but the execution time of the problem remain constant. This law focuses on scalability of problem size having constant execution time. Speed up given by this law is
$S(P) = P- \alpha(P-1)$

P is number of processing elements and α is portion that has to be executed sequentially.

2.3 Sun Ni's Law

Sun Ni's law focuses on memory, which is associated with every processor, should be scaled up. This law treats Amdahl's law and Gustafson law as special cases in order to find memory bounded speed up. In this law degree of parallelism plays very important role. Here works performed by processors are considered very carefully. This law also includes communication overhead associated with processor in its formula.

3. Parallelization of a Sequential Program

In this section various techniques which are used to parallelize the sequential programs are briefly described with their limitations.

3.1 Manual Parallelization

If the user decides to manually parallelize his program, he can freely decide which parts have to be parallelized and which not. This is called manual parallelization. Here user must be expert programmer that significantly involves in partitioning of the computations into multiple threads. He has to explicitly define the desired communication mechanisms (message passing or shared memory) and synchronization methods (locks, barriers, semaphores and so on) in the parallel program. When multiple threads must concurrently operate on the same data, it is often necessary to use locks to avoid race conditions between the threads [2].

Limitations

- This technique requires experts. So it is restricted techniques.
- It is good for small programs but for large and complex programs or applications it fails at some points. As we all know human are more prone to do mistakes.

3.2 Complier Automatic Parallelization

The easiest way to parallelize a sequential program is to use a compiler that detects, automatically or based on the compiling directives specified by the user, the parallelism of the program and generates the parallel version by finding interdependencies in the source code. The automatically generated parallel version of the program could be executed on a parallel system. The user does not have to be concerned about which part of the program is parallelized because the compiler will take such a decision when the automatic parallelization facility is used. [2]

Limitations

- User has a very limited control over parallelization.
- In a complex program that contains nested loops, procedure calls etc., it is very difficult to find and to analyze the dependencies which restricts the capabilities of such compilers.

3.3 Loop Parallelization

Parallelizing loops is one of the most important challenges because loops usually spend the most CPU time even if the code contained is very small. A loop could by parallelized by distributing iterations among processes. Every process will execute just a subset of the loop iterations range. Usually the code contained by a loop involves arrays whose indices are associated with the loop variable. This is why distributing iterations means dividing arrays and assigning chunks to processes.
If data affected by the inner loop are then referenced in the main loop, we need to synchronize data just after the end of the inner loop in order to be sure that the values accessed by the main loop are the updated one. The data synchronization was added just after the second loop end and the inner loop was distributed over multiple processes and processors. Every process will execute just a subset of the inner loop iterations range. Using the partial parallelization of a loop, we can minimize the load imbalance but the communication overhead will be higher as result of synchronization. [1]

For example in case of single loop, if I want to print "hello World" say 500 times or even more in MATLAB environment, it takes more time if it execute sequentially, but if it execute parallely on multiple processor it take very less time . Suppose it executes on MATLAB Parallel Computing toolbox which solves the problem using multicore processors, GPUs and computer clusters. However this toolbox provides 12 workers by default that works concurrently.

In MATLAB, Sequential program code for print “hello World” is:

```
For i: 1 : 500

Disp(‘Hello World’)

End
```

In MATLAB Parallel Computing toolbox, code for print “hello World” on multiple workers is:

```
Matlabpool local 10    \\ to start 10 workers

Parfori=1 : 500       \\ parfor is parallel for loop

Disp(‘Hello World’)

End

Matlabpool close  \\ close matlab workers
```

However program running on multiple works take less time. In this loop is distributed among multiple workers. Workers run concurrently and produce result frequently as compared to run same program sequentially.

Limitations

- Loop Parallelism concentrate mainly concentrate on for loops.
- This approach is not good if sequential programs consist of very complex nested loops.
- Here we need to take care of the partials results of the loops carefully. So synchronization plays vital role in this approach.

4. Parallelization Tools

Nowadays trend is moving from sequential programs to parallel programs. As sequential programs are time consuming so we need to convert them into parallel programs by rewriting them. As we know in order to create parallel programs, parallel programming is required. Parallel programming must be portable as it has to run efficiently on heterogonous systems. To do this compiler of parallel programming has to do restructuring of program which is time consuming task. Here I m going to discuss briefly about some ready-made tools that convert sequential programs into parallel programs by extracting parallelism from sequential programs. These are explained as follows:

PTRAN: PTRAN (Parallel TRANslator) is parallelizing system at IBM's T.J. Watson Research Center which converts the legacy sequential Fortran program into parallel Fortran programs. It is source to source complier. This system mainly concern with Fortran written programs which make this tool very specific for converting the legacy sequential program into parallel program. But nowadays frotran is used very less as we are now moved towards 4GL. PTRAN is seldom used for parallelization of sequential program. [3]

HydraVM: HydraVM is a virtual machine that extracts parallelism automatically from sequential programs. A set of techniques including code profiling, data dependency analysis, and execution Analysis are applied on sequential code (at the byte level). HydraVM is built by extending the Jikes RVM. Jikes RVM (Research Virtual Machine) provides a flexible open testbed to prototype virtual machine technologies. HydraVM works in three phases.

1. The first phase focuses on detecting parallel patterns in the code by monitoring code execution and determining memory access and execution patterns. This may lead to slower code execution due to inspection overhead. Information collected here is stored in Knowledge Repository.
2. The second phase starts after collecting enough information in the Knowledge Repository about which blocks were executed and how they access memory. The Builder component uses this information to split the code into superblocks, which can be executed in parallel. New version of the code is generated and is compiled by the Recompilation component. The TM Manager manages memory access of the execution of the parallel version, and organizes transaction commit according to the original execution order. The manager collects profiling data including commit rate and conflicting threads.
3. The last phase is tuning the reconstructed program based on thread behavior (i.e., conflict rate). The Builder evaluates the previous reconstruction of superblocks by splitting or merging some of them, and reassigning them to threads. The last two phases work in an alternative way till the end of program execution, as the second phase represents a feedback to the third one. [4]

MATLAB Parallel Computing Toolbox: MATLAB Parallel Computing toolbox solves the problem using multicore processors, GPUs and computer clusters. Parallel computing toolbox creates multiple workers (called 'labs' or computations engines) on local machine. By default Matlab provide 12 workers to execute applications locally on a multicore desktop. Without changing the code, we can run the same application on a computer cluster or a grid computing service (using MATLAB Distributed Computing Server which we need to purchase). Matlab parallelize the programs or applications without using CUDA or MPI Programming. This toolbox allow to us to run sequential as well as parallel program. However to run parallel program on multicore processors firstly we must use matlabpool command to open as many workers we want (maximum workers are 12).

5. Speculative Parallelization Verses Variant based Competitive Parallel Execution

First of all speculative parallelization is discussed and then variant based parallelism is discussed briefly.

5.1 Speculative Parallelization

Speculative parallelization attempts to use the many processing cores by creating concurrency from a program but also maintaining the sequential program order. It overcomes the limitations of traditional parallelization by creating threads that are composed from the program and speculatively executing them in parallel. Additional hardware support is used to determine threads that violate dependencies and squash them, and to enforce sequential program order of concurrently executed speculative threads. As threads are use here it is also known as Thread Level Speculative parallelism.

Program Demultiplexing (PD, in short) is an execution paradigm based on speculative parallelization, for sequential programs. In sequential execution, the call site of a method represents the beginning of execution of that method, and happens on the same processing core as the program. However, in PD, the execution of a method occurs on another available processing core, speculatively, before the call site is reached in the program. Several such speculative executions of methods create concurrency in a program. The speculative execution is usually invoked after the method is ready, i.e. after its data dependencies are satisfied for that execution instance. Speculative threads in PD are composed of methods. Methods allow programmers to decompose a problem into several subtasks and enable them to write a complex and lengthy program.

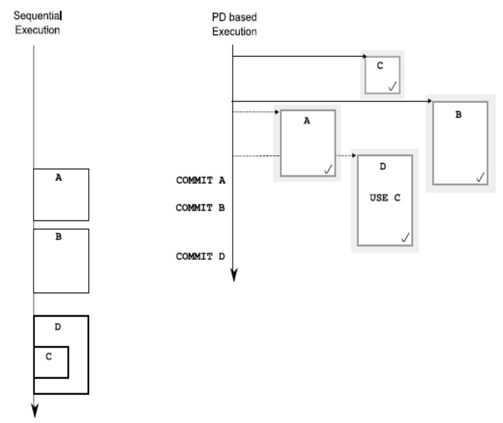

Figure 1. Program demultiplexing overview

The sequential execution on the left represents execution of methods A, B, and D. Method C is called inside D. On the right side is the PD based execution. The methods are spawned for speculative execution. Method D uses the speculative execution of method C. Methods A, B, and D are committed when the call site in the program is reached. The speculative executions do not violate any data dependencies and this is indicated by the tick mark on the bottom right of the method's box. [5]

Limitations

- It may happen that the core gets destruct on which any thread speculatively executing. It may harm the execution of the thread where it requires.
- This approach is difficult to apply for complex programs which involve nested loops, interdependent code and so on.

5.2 Variant-based Competitive Parallel Execution

Competitive parallel execution (CPE) is a model to adapt and execute existing sequential programs to increase their performance on multi-processor and multi-core systems. The fundamental idea of CPE is to include variants of one or multiple regions of a sequential program and to let these variants compete at program execution time.CPE is a technique for modifying and executing existing sequential applications to increase their performance on parallel systems. Competitive parallel execution (CPE) is a simple yet attractive technique to improve the performance of sequential programs on multi-core and multi-processor systems. The central idea of CPE is to facilitate the introduction of multiple variants for parts of a program, where different variants are suited for different run-time conditions.

A sequential program is transformed into a CPE-enabled program by introducing multiple variants for parts of the program. The performance of different variants depends on runtime conditions, such as program input or the execution platform, and the execution time of a CPE-enabled program is the sum of the shortest variants. The purpose of creating variants is to make the program adaptive to different run-time conditions. Variants compete at run-time under the control of a CPE aware run-time system. The run-time system ensures that the behavior and outcome of a CPE-enabled program is not distinguishable from the one of its original sequential counterpart.

Figure 2 illustrates the general execution model of a CPE enabled program with an example. The execution alternates between sequential phases, where only a single variant is running, and competitive phases, where multiple variants are running in parallel. The example program in Figure 2 executes two sequential and three competitive phases. Variants compete against each other in every competitive phase. At the conclusion of a competitive phase the program state of the winning variant is synchronized with all its peers. The execution then proceeds to the succeeding competitive or sequential phase. [6]

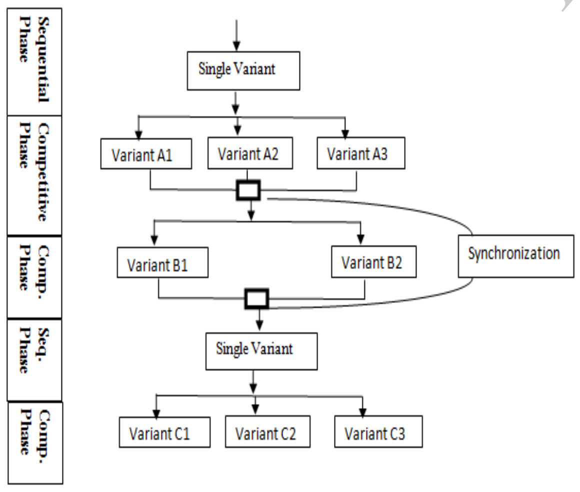

Figure 2. Execution Control of CPE

Figure 2: Example execution control flow of a CPE enabled program with two sequential and three competitive phases. Two or three variants compete in each competitive phase. A competitive phase ends upon completion of a variant, and the program state is synchronized to the state of this winner. The behavior and semantics of a CPE-enabled program must not be distinguishable from a sequential execution, in which only a single variant runs in each phase. The run-time system must provide two isolation properties to guarantee the semantical equivalence with the original sequential program:

1. The effects of a variant must be contained with respect to competing variants. A change in program state of one variant must thus not be observable by other variants.
2. The set of I/O operations performed by the CPE enabled program and the order in which they are performed must not have any side-effects that differ from a sequential execution of the program.

CPE model is not only restricted to flat competitive phases but also supports nested competitiveness.

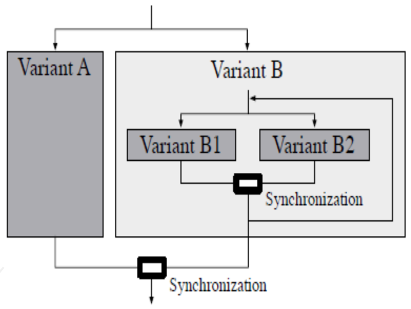

Figure 3. Example for a nested competitive execution

Variants A and B compete at the outer nesting level. Variant B in turn executes a loop where two variants B1 and B2 repeatedly compete with each other for each loop iteration. The inner competitive execution between B1 and B2 has no effect on the execution of variant A. In turn, if A reaches the outer synchronization point before B, the latter, including potentially executing variants B1 and B2, will be interrupted, and the program execution proceeds from the state of A.[6]

Two different approaches to transform an existing sequential program into a CPE-enabled program:

- Computation-Driven Competitiveness
- Compiler-Driven Competitiveness

Computation-Driven Competitiveness: Here variants which are to be executed are present in the program. It is a simple and straightforward parallelization of heuristic algorithms. In this the process of enhancing a sequential program is a straightforward process. Here variants are executed in isolation with respect to each other. Thus process does not require any reasoning about data sharing, dead-locks and other difficulties intrinsic to concurrent programming. Also, the process does not require detailed knowledge of the inner-workings of the original program with all its data structures and algorithms. [6]

Compiler-Driven Competitiveness: Here variants of parts of a program are generated by selecting different optimization strategies during compilation. Compiler-driven CPE exploits the fact that many optimizing compilers are unable to identify the best optimization settings for many programs. Compiler-driven CPE therefore employs the compiler to generate variants for frequently executed parts of the program by applying different optimization strategies upon compilation. [6]

6. Conclusions

This paper briefly discusses the ways of parallelizing the sequential programs. As compared to other approaches of parallelizing the sequential program, variant based competitive parallel execution is better. It can be beneficial for small programs as well as for large and complex programs. Unlikely in Speculative Parallelism where speculatively execute the independent part but that part may not be optimized and also its execution. But in this approach by using Complier driven competitiveness automatic select the best and optimized independent part called variant and its execution gives optimized results. This approach gives more correct results as compared to other approaches. Also, instead of writing the parallel version of sequential program from scratch, the corresponding sequential program is converted into parallel program by using various parallelization techniques. We can correlate with re-engineering.

6. References

[1] AlecuFelician, “How To Parallelize A Sequential Program”, 2006

[2] Ben Hertzberg, “Runtime Automatic Speculative Parallelization Of Sequential Programs”, November 2009

[3]Wilson Cheng-Yi Hsieh “Extracting Parallelism From Sequential Programs”, 1988

[4] Mohamed M. Saad, Mohamed Mohamedin, AndBinoyRavindran “Hydravm: Extracting Parallelism From Legacy Sequential Code Using STM”, 2010

[5] SaisanthoshBalakrishnan “Program Demultiplexing: Data-Flow Based Speculative Parallelization Of Methods In Sequential Programs”, 2007

[6] Oliver Trachsel, Thomas R. Gross “Variant-Based Competitive Parallel Execution Of Sequential Programs”, 2010