IMPACT OF DEPENDENCY AND LOAD BALANCING IN MULTITHREADING REAL-TIME CONTROL ALGORITHMS

M A Hossain and M O Tokhi

1Department of Computing, The University of Bradford, UK.
2Department of Automatic Control and Systems Engineering, The University of Sheffield, UK.

INTRODUCTION

The concept of concurrent multithreading in multiprocessor computing is currently a subject of widespread interest to the scientist and professional software developer. Concurrent multithreading in a multiprocessor domain is a growing technology with the potential to enable the development of much higher performance application as compared to the traditional method of software development in uniprocessor or multiprocessor environment. Threads reduce overheads by sharing fundamental parts, for instance, code (text), data, stack, file I/O etc. By sharing these parts switching happens much more frequently. Although sharing information is not difficult, however, sharing with a dependency between threads cause degradation of overall computing performance. A single thread program possesses a single execution thread or library not designed to be used in a multithread environment. Whereas, a multithread program uses multiple execution threads or a library that allows concurrent access from multiple threads. Concurrent multithreading separates a process into many execution threads, each of which runs independently. Typically, applications that express concurrency requirements with threads need not take into account the number of available processors. The performance of the application improves transparently with additional processors. A single thread program possesses a single execution thread or library not designed to be used in a multithread environment. Whereas, a multithread program uses multiple execution threads on a multiprocessor Gray (1), Lo et al (2).

Data dependency is one of the key issue in multithreading for real-time high performance computing. During implementation of an algorithm, data dependency between two blocks or two statements requires memory access time. In practice, increasing dependencies implies increasing access time or more inter-process communication in case of concurrent multithread implementation in multiprocessor domain. This, accordingly, degrades the real-time performance. Thus, it is essential to study and analyse data dependencies in an algorithm intended for real-time concurrent thread implementation. Such a study must address the essential question, such as, how to reduce the block or statement dependencies, and, what the impact of data dependencies in real-time inter-process communication will be. Detection of multithreading scope in an application involves finding sets of computations that can be performed simultaneously. The approach to parallel multithreading is thus based on the study of data dependencies. The presence of dependence between two computations implies that they can not be performed in parallel. In general, the fewer the dependencies, the greater the parallelism, Moldovan (3). Many algorithms have regular data dependencies that is, dependencies that repeat over the entire set of computations in the algorithm. For such algorithms, dependencies can be concisely described mathematically and can be manipulated easily. However, there are algorithms for which dependencies vary from one computation to another and these algorithms are more difficult to analyze. When two or more algorithms have similar dependencies, it means that they exhibit similar parallel properties, Hossain et al (4).

Performance of the synchronization mechanism of a multi-processor determines the granularity of parallelism that can be exploited on that machine. Synchronization on a multiprocessor carries a high cost due to the hardware levels at which synchronization and communication must occur, Tullsen et al (5). Therefore, the study of concurrent multithreading also includes inter-process communication, issues of granularity of the algorithm and of the hardware and regularity of the algorithm. Hardware granularity can be defined as the ratio of computational performance over the communication performance of each processor within the architecture. Similarly, task granularity can be defined as the ratio of computational demand over the communication demand of the task, Nocetti and Fleming (6). More clearly,

\[
\text{Task granularity} = \frac{\text{Run-time length of a task}}{\text{Communication overhead}} = \frac{R}{C}
\]

Typically a high compute/communication ratio is desirable. The concept of task granularity can also be viewed in terms of compute time per task. When this is large, it is a coarse-grain task implementation. When it is small, it is a fine-grain task implementation. Although, large grain tasks may ignore potential parallelism, partitioning a problem into the finest
possible granularity does not necessarily lead to the fastest solution, as maximum parallelism also has maximum overhead, particularly due to increased communication requirements. Therefore, when partitioning the algorithm into subtasks and distributing these across the processing elements, it is essential to choose an algorithm granularity that balances useful parallel computation against communication and other overheads, Tokhi and Hossain, (7), Liau and Prasanna (8).

In this paper, an active vibration control algorithm for a flexible beam system is employed to study and explore real-time concurrent multitread implementation issues in high performance multiprocessor domain. A dual processor based machine comprises of two Intel processors (1GHz) is considered as the computing domain to demonstrate the critical issues for concurrent multithreading in real-time computing. This investigation also addresses the issues of synchronization, task granularity, load balancing and inter-process communication problem due to data and control dependencies in the algorithm. Finally, a comparison of the results of the implementations is made on the basis of real-time performance to explore systems design incorporating multithreading techniques in multiprocessor domain for real-time control.

ACTIVE VIBRATION CONTRO ALGORITHM

Consider a cantilever beam system with a force \( U(x, t) \) applied at a distance \( x \) from its fixed (clamped) end at time \( t \). This will result in a deflection \( y(x, t) \) of the beam from its stationary position at the point where the force has been applied. In this manner, the governing dynamic equation of the beam is given by,

\[
\mu^2 \frac{\partial^4 y(x, t)}{\partial t^4} + \frac{\partial^2 y(x, t)}{\partial t^2} = \frac{1}{m} U(x, t)
\]

(1)

where, \( \mu \) is a beam constant and \( m \) is the mass of the beam. Discretising the beam in time and length using the central FD methods, a discrete approximation to equation (1) can be obtained as, Virk and Kournoulis (9),

\[
Y_{k+1} = -Y_{k-1} - \Delta^2 S Y_k + \frac{(\Delta t)^2}{m} U(x, t)
\]

(2)

where, \( \Delta^2 = \left[ (\Delta t)^2 / (\Delta x)^2 \right] \mu^2 \) with \( \Delta t \) and \( \Delta x \) representing the step sizes in time and along the beam respectively, \( S \) is a pentadiagonal matrix (the so called stiffness matrix of the beam), \( Y_i \) \( (i = k + 1, k, k - 1) \) is an \( (n-1) \times 1 \) matrix representing the deflection of end of sections \( i \) to \( n \) of the beam at time step \( i \) (beam divided into \( n-1 \) sections). Equation (2) is the required relation for the simulation algorithm that can be implemented on a computing domain easily.

A schematic diagram of an AVC structure is shown in Figure 1. A detection sensor detects the unwanted (primary) disturbance. This is processed by a controller to generate a cancelling (secondary, control) signal so that to achieve cancellation at the observation point. The objective in Figure 1 is to achieve total (optimum) vibration suppression at the observation point. Synthesising the controller on the basis of this objective yields, Tokhi and Hossain (10)

\[
C = \left[ 1 - \frac{Q_1}{Q_0} \right]^{-1}
\]

(3)

where, \( Q_0 \) and \( Q_1 \) represent the equivalent transfer functions of the system (with input at the detector and output at the observer) when the secondary source is \textit{off} and \textit{on} respectively.

![Figure 1: Active vibration control structure](image)

To investigate the nature and real-time processing requirements of the AVC algorithm, it is divided into two parts, namely control and identification. The control part is tightly coupled with the simulation algorithm, and both will be described in an integral manner as the control algorithm. The simulation algorithm will also be explored as a distinct algorithm. Both of these algorithms are predominately matrix based. The identification algorithm consists of parameter estimation of the models \( Q_0 \) and \( Q_1 \) and calculation of the required controller parameters according to equation (3). However, the nature of identification algorithm is completely different as compared with the simulation and control algorithms, Tokhi and Hossain (7). Thus, for reasons of consistency only the simulation and control algorithms are considered in this investigation.

The simulation algorithm forms a major part of the control algorithm. Thus, of the two algorithms, the simulation algorithm is considered here to address the issue of data dependency. Consider the simulation algorithm in equation (2). This can be rewritten, for
computing the deflection of segments 8 and 16, as follows, assuming no external force applied at these points:

\[
\]

\[
\]

It is also noted that computation of deflection of a particular segment is dependent on the deflection of six other segments. These heavy dependencies could be major causes of performance degradation in real-time sequential computing, due to memory access time. On the other hand, these dependencies might cause significant performance degradation in real-time concurrent multithreading due to inter-process communication overheads, Hossain et al (11).

**HARDWARE AND SOFTWARE RESOURCES**

To demonstrate the issues of real-time multithread implementation in multiprocessor environment, a computing domain with dual Pentium III processors was used. The domain comprises of dual PIII (1GHz) processors, 1000MB RDRAM (800MHz clock speed) as main shared memory and a 256 KB L2 cache per processor. This is a DELL multitasking machine (Brimstone) with Linux operating system version 2.2.16. The g++ (using gcc) compiler version 2.96 was used to develop source code for real-time implementation.

**IMPLEMENTATION AND RESULTS**

To investigate the parallel multithread implementation of the flexible beam simulation algorithm of a cantilever beam of length \( L = 0.635 \) m, mass \( m = 0.0375 \) Kg and \( \lambda = 0.3629 \) was considered. For an adequate accuracy to be achieved in representing the first few (dominant) modes of vibration of the beam, the beam was divided into 20 segments or above.

Figure 2 shows the execution time for a single thread and two threads with synchronization. It is noticed that due to the synchronization, inter-process communication overheads and other operating system overheads, a significant level of performance degradation is observed for two threads as compared to a single thread. In an ideal situation, it is expected that the execution time in implementing the algorithm with two threads within dual processors domain should be half. However, the actual performance achieved for two threads is even much lower than a single thread and this trend continues to increase with the increment of the number of iterations.

Table 1 shows the multithread execution time with and without synchronization. It is noticed that a significant level of performance is achieved for two threads as compared to a single thread execution time without synchronization. However, this improvement is not consistent or evident with the further level of increments of threads. This could be due to the limitation of the number of processors in the domain. This demonstrates that the domain which comprises of two processors, two threads might be the best mapping rather further level of granularity. In addition, operating system's overhead, scheduling and mapping problem could cause a further level performance degradation with the increments of threads.
Figure 3: Execution time for a single thread and two threads without synchronization (for 20 segments)

Figure 4: Execution time for two threads with and without synchronization (for 20 segments)

Table 2 shows the communication overhead and task granularity for 90,000 iterations. It is noticed that the level of communication overhead is significantly higher as compared to actual computation time which, in turn, reduces the task granularity ratio. This is further demonstrated that the impact of overhead due to a significant level of thread dependencies, in turn, inherent data dependencies of the simulation algorithm.

To explore the capabilities of concurrent multithreading further, an I/O work was inserted within the flexible beam simulation algorithm. The I/O work was involved for file saving into the secondary memory. The algorithm was implemented in accordance with the design mechanism shown in Figure 5 enforcing both concurrently executing threads to wait for each other.

The first and most important observation that can be made from this graphical representation in Figure 6 is that the performance of the program actually increased beyond a certain size of segments. This could be due to balanced loading of the threads which improves the overall CPU utilisation of the program and reduces the OS overhead generated by threads being suspended when trying to acquire a lock on a locked mutex variable. Once the calculation thread went sufficiently busy, the probability of triggering a synchronisation lock method was drastically improved, thus resulting in higher performance. It appeared odd but intriguing that performance could be improved by adding more tasks to one thread.

Finally, the control algorithm was implemented to explore the impact in real-time implementation. It was noted earlier that the control algorithm is tightly coupled with simulation algorithm and computational load for calculating control response in each iteration is equivalent to the computation of one segment of the beam. It is observed from the experiment that the execution time for control algorithm is similar level of simulation algorithm and the complexity in terms of multithreading remains similar. Therefore, detail description in implementing control algorithms will remain similar and is not provided here to avoid duplication.

**CONCLUDING REMARKS**

An investigation into the dependency and load balancing of multithreading in multiprocessor domains for real-time control, emphasising thread synchronization, has been presented, and demonstrated the real-time implementation of the algorithm. It has been demonstrated that data dependency and load balancing are the critical constraints of multithreading algorithm and the complexity in terms of multithreading remains similar. Therefore, detail description in implementing control algorithms will remain similar and is not provided here to avoid duplication.
into domain. It has been noticed that fine grain threading with higher level of data dependency could cause significant level of performance degradation. In this particular application, increment of threads increases the synchronization overhead, which, in turn, relates with inter-process dependencies, operating system’s overhead for run-time scheduling, and memory management.

Finally, it is noted that the impact of synchronization, data dependencies, load balancing and granularity of algorithm are critical issues for real-time multithreading in multiprocessor domain.

Figure 5: Algorithm design mechanism (MUTEX represents mutual exclusion)

Figure 6: Performance of the algorithm with file I/O for 100,000 iteration

REFERENCES