Programming and Execution for the DDM-CMP System

Kyriakos Stavrou*
Paraskevas Evripidou* Pedro Trancoso*

* Department of Computer Science, University of Cyprus,
75 Kallipoleos Ave., P.O.Box 20537 - 1678 Nicosia,Cyprus,
E-mail: {tsik,skevos,pedro}@cs.ucy.ac.cy

ABSTRACT

DDM-CMP is a single-ISA, homogeneous Chip Multiprocessor that operates under the Data-Driven Multithreading (DDM) model of execution. DDM-CMP requires no modifications of the CPU or the OS allowing it to execute both conventional and DDM binaries. This paper presents the structure of DDM programs and explains their execution on the DDM-CMP system.

KEYWORDS: Data-Driven Multithreading, Chip Multiprocessors, DDM-CMP

1 Introduction

To deliver performance as predicted by Moore’s Law, computer architects have relied on the advancements of process technology, as well as improvements of the computer architecture and organization. While this approach has worked well in the past, it is currently only resulting in diminishing returns [Arvi05]. This is due to the inability of traditional architectures in surpassing two major obstacles: the memory and power walls. The former is due to the imbalance between the speed of microprocessors and main memory, while the later is due to the high frequencies and complexity in modern microprocessors. Both walls can be traced back to the von Neumann model of execution that has dominated the computer architecture field since the advent of digital computers.

Data-Driven Multithreading (DDM) [Kyri05], is an alternative model of execution that does not suffer from the previously mentioned limitations. DDM tolerates the latencies by allowing the computation processor to produce useful work while a long latency event is in progress. In this model, the synchronization part of the program is separated from the computation part allowing it to hide the synchronization and communication delays. While such computation models usually require the design of dedicated microprocessors, Kyriacou et al. [Kyri05] showed that the DDM benefits may be achieved using commodity microprocessors. The only additional requirement is a small hardware structure, the Thread Synchronization Unit (TSU).

DDM-CMP [Stav05a] is a single-ISA, homogeneous Chip Multiprocessor that supports
the DDM model of execution. As such, it is able to combine the advantages of the DDM model with those of the CMP architecture. Preliminary studies have shown that DDM-CMP is able to deliver high performance speedup combined with significant power reduction [Stav05a, Stav05b, Tran05].

In this paper, we present the structure of DDM programs and explain their execution. We also outline the DDM-Runtime Support System that allows execution of DDM programs with conventional, unmodified CPUs and OS.

The rest of this paper is organized as follows. Section 2 outlines the DDM-CMP architecture. Section 3 presents the structure of DDM programs whereas Section 4 explains their execution. Finally, Section 5 concludes this work.

2 The DDM-CMP Architecture

DDM-CMP is a single-ISA, homogeneous chip multiprocessor able to support the Data-Driven Multithreading model of execution. DDM-CMP combines the benefits of the DDM model together with those of the CMP architecture without requiring any modification to the OS or the CPUs. As such, it enables the execution of both DDM and non-DDM applications.

To operate under the DDM model, each core of the CMP must have its own Thread Synchronization Unit (TSU). The TSU is a small, on-chip, memory-mapped device responsible for scheduling DDM-threads for execution in a dataflow manner, that is, when all their input data have been produced. Taking into consideration all extra hardware structures required to support the DDM model of execution, we found that in the same hardware budget of a modern high-end, single-chip microprocessor (that of P4), a DDM-CMP chip with 16 embedded PowerPC 405 cores can be build. For this chip, the DDM-overhead is less than 18%.

3 A program in DDM

A program in DDM is an arbitrary collection of serial (non-DDM) and parallelizable (DDM) portions of code. A significant benefit of the DDM-CMP system is that it provides a very intuitive way for representing these portions of code. In particular, portions of DDM code are represented as DDM Code-Blocks whereas the computation threads within a DDM-Block are represented as DDM threads.

During compilation, a program is partitioned into code threads and Data-Driven Synchronization Graph (SG). The former refers to the actual code of the program and describes how it is split into DDM-threads. SG describes the dependencies of these DDM-threads.

To clarify the above mentioned concepts, we will use the Multiple Linear Regression program presented in Figure 1-(a). This program evaluates the expression $(X^T \cdot X)^{-1} \cdot X^T \cdot Y$ where tables X and Y are defined from the input data. The DDM portion of the program is represented by a DDM-Block (lines 04-11) that is located between the two non-DDM sections, those of lines 02 and 14. The boundaries of DDM-threads and DDM-blocks can be easily defined through simple pragma directives (Figure 1-(b)). A special, DDM-CMP preprocessor handles these directives and produces the final DDM-Code, which is converted to an executable binary with conventional compilation.

Figure 1-(c) shows the dependencies of the threads of this program. As an example, notice the arc between threads 5 and 3 which models the fact that the calculation of $X^T \cdot Y$
(thread 5) must wait for the completion of the thread that calculates $X^T$ (thread 3).

Figure 1: (a) The pseudocode of the Multiple Linear Regression (MLR) program. (b) Parallelization using DDM pragma directives. (c) The Synchronization Graph. (d) Execution using 2 DDM-Kernels.

Two other important issues regarding DDM programming, not shown in the example of Figure 1(a), are worth mentioning. The first, is the ability to handle programs with multiple DDM and non-DDM portions of code arbitrarily interleaved. The second, concerns the ability to have nested DDM portions of code. Both features are necessary for easy and intuitive modeling of large-scale, real-life, parallelizable programs.

4 Execution of a DDM program

A primary target of the DDM-CMP architecture is to be able to execute not only DDM applications, but also conventional, non-DDM binaries. To meet this goal, execution of DDM binaries must be transparent to the OS and the CPUs. This is achieved by a specially developed Runtime Support System (RSS) that does not require modifications of the OS or the CPUs. RSS enables execution under the DDM model through simple, lightweight, user level processes, the DDM-Kernels.

To execute a DDM application on the DDM-CMP system, the Synchronization Graph (SG) of the program must be first loaded on the TSUs. The TSUs schedule threads for execution following the order directed by the program’s SG, i.e. they schedule a thread for execution only after the threads on which it depends have executed.

The RSS provides the necessary mechanisms for initializing the TSU. However, as the TSU space is limited, only a portion of the program’s SG can exist on the TSU. Consequently, a mechanism that allows dynamic TSU loading and unloading is also required.

A DDM-application starts its execution by launching $n$ DDM-Kernels. Each kernel is run by a different process on a different CPU. The application completes its execution when all its kernels have done so. To dynamically load and unload the TSUs, two special types of threads are used: the Inlet and Outlet threads (a pair of such threads exists for each DDM-Block for each DDM-Kernel). The Inlet thread is the first thread of each DDM-Block and its function is to load the TSU with all threads of that block for the corresponding kernel. The necessary information is inserted in the application’s code during compilation. The last thread of a DDM-Block is the Outlet thread whose main operation is to clear the resources allocated on the TSU. The operation of the Inlet and Outlet threads provides the required mechanisms to dynamically handle the TSUs.
When a thread completes its execution, it informs the TSU and transfers the execution to the DDM-Kernel. Acknowledging its completion, allows the TSU to find the next ready threads. At the same time, the DDM-Kernel reads from the TSU the address of the first instruction of the next ready thread. If such a thread exists, execution is transferred to it, otherwise this TSU query process is repeated.

Figure (d) shows how the program of Figure (a) is executed on a DDM-CMP machine with 2 DDM-Kernels. After the initialization phase, the program creates the DDM-Kernels. Each kernel, in turn, loads its TSU with the corresponding Inlet thread (I1 for kernel 1 and I2 for kernel 2) and transfers the execution to it. Each Inlet thread, loads its TSU with the threads that will be executed by its kernel, i.e. I1 loads threads 1, 3, 4 and O1, while I2 loads threads 2, 5, 6 and O2. When the Inlet threads complete their execution, they notify the TSU and transfer control back to the DDM-Kernel. This allows the TSUs to find the next ready threads and the DDM-Kernel to transfer execution to them. When, for example, I1 completes and notifies its TSU, thread 1 is deemed as executable. When queried for the next ready thread, the TSU will provide the address of the first instruction of thread 1 and the DDM-Kernel will transfer the execution to it.

All but one Outlet threads of a DDM-Block, after clearing their TSUs, transfer the execution to the corresponding Inlet thread of the next DDM-Block (Figure (e)). The remaining Outlet thread, will execute the non-DDM code, if such code exists. However, the Outlet threads of the last DDM-Block operate in a slightly different way. Specifically, these threads are statically set by the compiler to exit their DDM-Kernels.

5 Conclusions

In this paper we have presented the structure of DDM programs and showed the inherent relationship that exists between sections of parallelizable programs and DDM primitives. Additionally, we outlined the DDM Runtime Support System that allows the architecture to operate without requiring modifications of the OS or the CPU. This characteristic of the DDM-CMP system enables it to execute both conventional, non DDM, and DDM binaries.

References


