Data-flow Concurrency on Distributed Multi-core Systems

George Michael[1], Samer Arandi[2] and Paraskevas Evripidou[1]
[1]Department of Computer Science, University of Cyprus, Nicosia, Cyprus
[2]Department of Computer Engineering, An-Najah National University, Nablus, Palestine

Abstract— *The Dynamic Data-Flow model of execution has many inherit properties, such as tolerance to latencies and distributed concurrency, which make it suitable for distributed execution. Data-Driven Multithreading (DDM) is a hybrid Data-flow/Control-flow model that implements the Data-Flow principles at the Thread level on sequential processors. In this paper we demonstrated that the Data-Driven Multithreading Virtual Machine (DDM-VM), can achieve high performance in Distributed Nodes (multi-core systems). A shared Global Address Space is supported across all the Nodes in the system to facilitate data movement. We have evaluated our work on both Homogeneous and Heterogeneous systems. The performance evaluation shows that the distributed execution achieves 80-84% of the maximum possible speedup using off-the-shelf networking.*

Keywords: multi-core, data-driven multithreading, heterogeneous, homogeneous, distributed

1. Introduction

Data-Driven Multithreading (DDM) has demonstrated that Data-Flow concurrency can be implemented in commercial control-flow processors in an efficient manner [16]. The Data-Driven Multithreading Virtual Machine (DDM-vm) is a parallel software platform that supports Data-Flow concurrency on conventional control-flow multi-core systems. The DDM model combines the latency tolerance and distributed concurrency of the dynamic data-flow model of execution with the efficient execution of the control-flow model. The DDM-vm targets both homogeneous and heterogeneous multi-core architectures.

In this work we advance the state-of-the-art by supporting DDM execution across Distributed multi-core systems. Previous implementation either supported distributed DDM execution across single-processor Nodes [16] or DDM execution within a multi-core Node [21]. The results of this work further demonstrates that data-flow concurrency can be utilized to tolerate synchronization and network latency efficiently.

First, we present the DDM-vm that is executing on symmetric multi-core architectures. Then we highlight the extensions and modifications of the DDM-vm design to support distributed DDM execution across multiple computers. Concluding, with the evaluation of single- and distributed-Node DDM-vm using a suite of benchmark applications and multiple, different clusters.

The evaluation of the single-Node DDM-vm shows that it achieves an overall average speedup of 9.6 out of 11. The evaluation of the distributed execution shows that it achieves an average of 80% to 84% of the maximum possible speedup when utilizing various number of cores per Node. The results are stable across different cluster configurations. These results demonstrate the efficiency and scalability of the DDM-vm.

The rest of the paper is organized as follows: An overview of Data-driven Multithreading is presented in Section 2. Section 3 describes the DDM-vm. Section 4 presents the support for distributed DDM-vm execution. The evaluation is presented in Section 4. The related work is presented in Section 5 and Section 6 concludes this paper.

This work was supported in part by the EU FP7 TERAFLUX (249013) project

2. Data-Driven Multithreading

DDM [16] is a multithreaded model that applies dynamic Data-flow principles for the communication among threads and exploits highly efficient control-flow execution within a thread. Programming constructs such as loops and functions are mapped into DDM threads (D-Threads).

DDM decouples the synchronization part of a program from the execution part and allows them to execute asynchronously, thus shortening the critical path. At the core of the DDM model is the Thread Scheduling Unit (TSU) [11] which schedules threads at run-time based on data-availability. DDM utilizes data-driven caching policies, called Cacheflow, to implement deterministic data prefetching which can substantially improve the locality of sequential processing [15].

DDM has shown that it can exploit efficiently Data-flow concurrency on commercially available multi-core systems [1], [2]. DDM does not need traditional memory coherence because it enforces the single-assignment semantics for data exchange among threads.

In DDM all D-Threads that are ready for execution are held in the Firing queue of each core. Thus, DDM prefetching [15] is deterministic and can be very near to optimal. The ability of DDM to deterministically prefetch data allows high performance without the need for complex and expensive modules such as Out of Order Execution (OOE). Thus, DDM inspired processors had the potential

to have simple and low power designs and at the same time achieve high performance.

DDM programs are partitioned at compile time into a number of threads. Each thread is associated with its meta-data: Instruction Frame pointer (IFP), Data-Frame Pointers (DFP), Consumer threads and the Ready Count (RC) which is the number of producer threads.

DDM supports dynamic Data-flow concurrency based on the U-Interpreter [3] principles. It uses the tagging system of the U-Interpreter to distinguish between different instantiations of a static code template. It maps the tag of the U-Interpreter into a unique integer, called the *context* in DDM. Consequently it maps the entire unravelled Data-flow graph into the virtual space of the machine.

3. The Data-Driven Multithreading Virtual Machine

The Data-Driven Multithreading Virtual Machine (DDM-vm) software model is composed of the:

- Thread Scheduling Unit (TSU), that is implemented as a software module executing on one of the cores
- Network Interface Unit (NIU) is implemented as a software module that is executing on the same core as the TSU
- Runtime support system that (with the help of the TSU) handles the tasks of thread scheduling, execution instantiation and data management implicitly on the rest of the cores

The DDM-vm can be used both on heterogeneous multi-cores with software-managed memory (the Cell B.E. Processor [12]), and on symmetric multi-cores. DDM-vm support for symmetric multi-cores is described in the following sections. A detailed description and performance results of the DDM-vm on the Cell B.E. Processor can be found in [1].

The *synchronization graph* contains the *meta-data* of the threads which mainly convey the consumer/producer dependency relationships of each thread. The virtual machine uses the *meta-data* to schedule threads based on data-availability; a thread is scheduled for execution when all its producers finish execution. The scheduling of threads is interleaved with their execution, thus shortening the critical path of the application. In the case of architectures with a software-managed memory hierarchy the DDM-vm prefetches the thread data from the main memory to the cache of the processor before starting its execution.

The overall architecture of DDM-vm is depicted in Figure 2. The support of Data-Driven execution on control-flow processors communication is achieved by changing the state of the TSU structures allocated in main memory.

Next, we describe the structures and operations of the TSU and the TSU-Processor interface for symmetric multi-cores.

3.1 The Thread Scheduling Unit (TSU)

a) The TSU Memory Structures: are allocated in main memory. Some of the structures are common for all the cores: Synchronization Memory (SM) and and Graph Memory (GM) and the rest: (1) Acknowledge Queue (AQ), (2) Waiting queue and (3) Firing Queue (FQ)are local in each core. The common TSU structures are:

The **Graph Memory (GM)**: holds the *synchronization template* of each thread:

- **Thread identifier** (ThreadId),
- **Instruction Frame Pointer** (IFP),
- **Consumers List** number of Data Frame Pointers,
- **Ready Count** (RC): number of producer threads
- **Data Frame Pointers** (DFPs) which are retrieved at runtime by calling a helper-function.
- **Thread attributes**: (i) Scheduling policy , (ii) Type of Synchronization Memory (More information can be found in [1]), and the (iii) *Arity* of the specifying the loop nesting level.
- **Consumer List** (CL): contains two fields, *Cons1* and *Cons2*, which can hold the thread identifier for one or two consumers. If the thread has more than two consumers, a CL entry is created that holds the list of consumer threads. In that case, the GM entry is modified so that *Cons1* is set to zero and *Cons2* is set to point to the entry in the CL.

The Synchronization Memory (SM): holds the Ready Count (RC) values for each invocation of a DDM thread. The SM entries are uniquely indexed using the *context* of the invocations. The RC value in the GM entry is used to initialize the RC entries in the Synchronization Memory. As the performance of the SM is critical to the overall system performance, we have utilized three different implementations of the SM. Details of the implementations and their performance is presented in [1].

b) The per-core TSU structures: are also allocated in main memory:

The Acknowledgement Queue (AQ): holds requests to decrement the RC of one or more invocations of consumer threads. The AQ requests are enqueued when a producer thread finishes execution. The request include the consumer identifier, *context* and the decrement value by which the consumer(s) RC is decremented. A sample of this request can be seen on figure 1 as the command Dec.

The Waiting Queue (WQ): holds the information of threads which with RC=0 and are waiting for prefetching to be completed.

The Firing Queue (FQ): holds the information of threads that are ready to execute. This includes the IFP, *context* and the DFP list. The latter is needed for supporting distributed DDM execution, in which part of the thread data could be allocated dynamically by the TSU (described in the

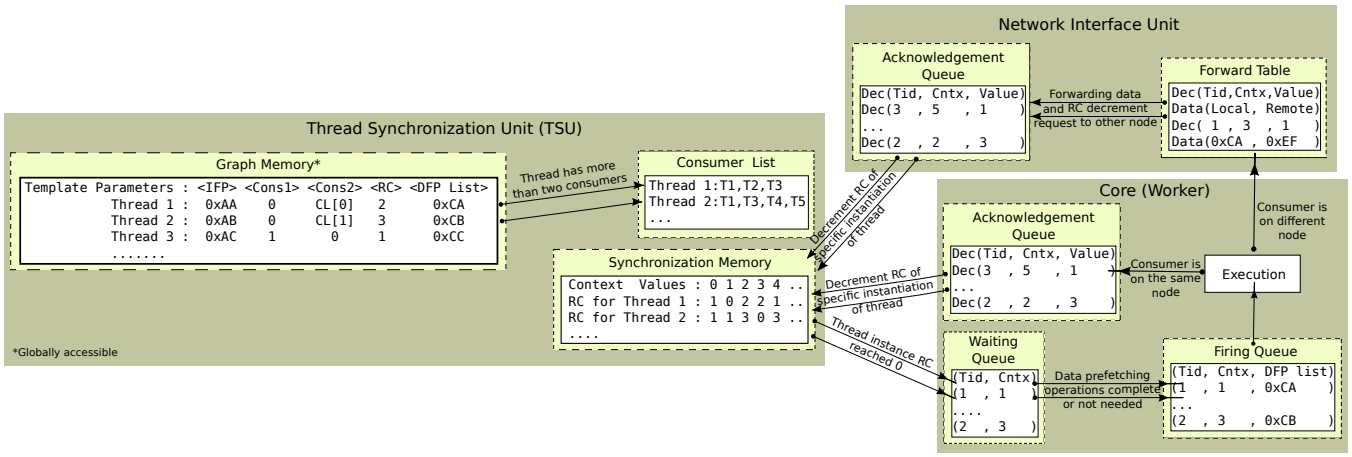

Fig. 1: TSU organization and transitions

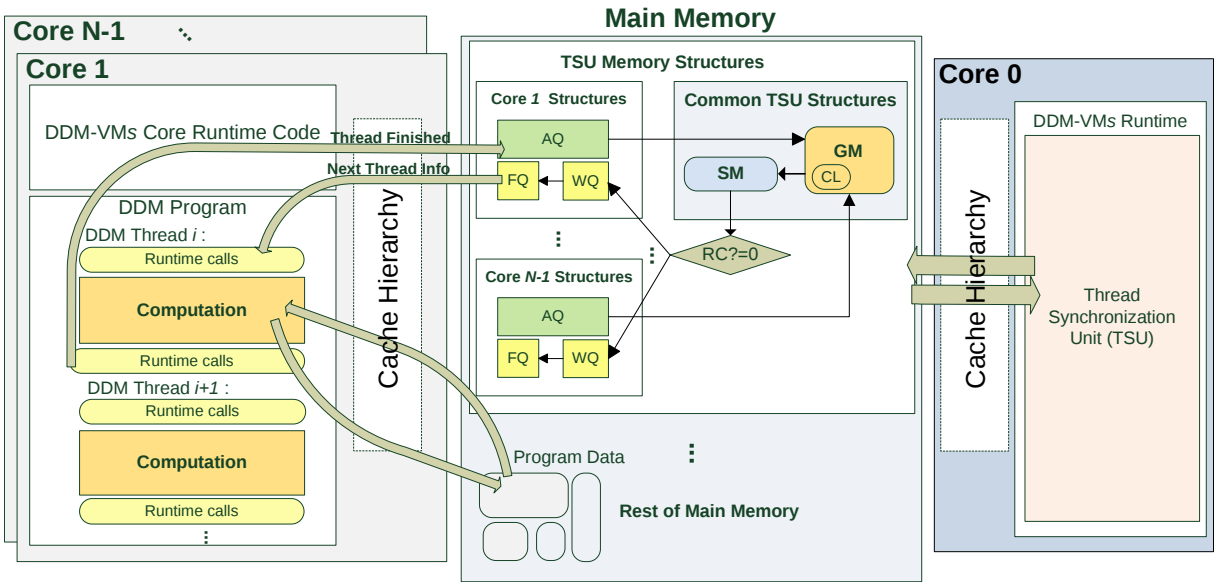

Fig. 2: The architecture of the DDM-vm

following section).

Figure 2 depicts the structures of the TSU.

3.2 Thread Execution and the TSU-Processor interface

The DDM thread execution takes place on the cores and consists of two types of operations or phases: computation and synchronization. The synchronization operations are performed by the runtime, which communicates with the TSU via the TSU-Processor interface. The purpose of the communication is to:

1) Inform the TSU to decrement the RC of the consumers of the thread that has just completed execution by inserting requests in the AQ
2) Providing the execution cores with the information of the next ready thread to execute. This is achieved by accessing the meta-data of ready threads in the FQ.

The two tasks are implemented by accessing the related TSU structures in main memory directly: The AQ in the first task and the FQ in the second task.

The TSU operations:

1) The TSU running on one of the cores processes the AQ requests to decrement the RC of consumer threads.
2) If any RC reaches zero, the corresponding thread invocation is scheduled for execution on a core that is selected by a scheduling policy. This is done by inserting the thread information into the Waiting Queue (WQ) were it waits from Data prefetching.
3) Finally the metadata of the threads are then moved

from the WQ into the Fire Queue (FQ) indicating they are ready for execution.

Figure 2 depicts the various operations performed by the TSU and the runtime.

4. Distributed DDM-vm Execution

Tolerating the Network and Synchronization latencies is the key for the efficiency of Distributed systems. The Data-Flow model provides tolerance to such latencies; an operation starts only after all it s data has been produced. Furthermore, Data-flow concurrency enforces the minimal ordering of objects as dictated by the true data-dependencies. Constructs like critical session and barriers do not exists in Data-Flow systems. Cache coherence is also a major challenge of the distributed systems. This is also not needed in Data-Flow inspired systems such as DDM.

The main difference between single-Node and distributed/multi-Node DDM execution is the introduction of remote memory accesses resulting from producer and consumer threads running on different Nodes. To this end, we employ data *forwarding* [19], [13], to the Node where the consumer is scheduled to run. We facilitate this by supporting a shared Global Address Space (GAS) across all the Nodes. A Network Interface Unit (NIU) has been implemented in the TSU to handle the low-level communication operations.

In terms of the distribution of threads across the cores of the system Nodes, this work explores a *static* scheme, in which the mapping is determined at compile time and does not change during the execution. This simplifies the scheduling and data management tasks and, in the presence of an accurate knowledge of the threads execution loads, can lead to a very efficient and balanced parallel execution. It is important to note that a static distribution only specifies *where* the thread will be scheduled once its ready, however, *when* the thread is ready is decided based on data-availability. The benefit of this approach extends to programmability. *Distributed* DDM-vm programs are fundamentally the same as *single-Node* ones. Aside from the distribution of program data in the GAS across the Nodes at startup and gathering the results after the program execution.

Next, we highlight the additions and modifications to the TSU structures & operations that are required for supporting distributed execution.

4.1 Modifications to the Thread Scheduling Unit

The DDM-vm runtime adopts a distributed organization consisting of multiple TSU units (one per Node[1]) communicating across the network to coordinate the overall DDM execution. As shown in Figure 3.

[1]Node: multi-core processor

4.1.1 The TSU structures

In the Graph Memory (GM) of each Node, we only load the *metadata* of the threads that are expected to execute on that Node. The rest of the TSU structures remain unchanged. Two new structures to support distributed execution have been added:

1) **The Distributed Acknowledgement Queue (AQ)** This queue holds the decrement RC requests coming from the TSUs on the remote Nodes.
2) **Forward Table (FT)** This table holds the address and size of the data that will be forwarded to remote Nodes.

4.1.2 The TSU operations

When the TSU is notified that a thread finished execution, it determines by the scheduling policy (ThreadId,*context*) weather it will on the producer Node or at a remote Node. If the core is on the same Node, an entry is inserted in the AQ of the local Node TSU. However, if the core belongs to a remote Node, a message containing the invocation (ThreadId,*context*) is sent to that Node. When the message is received, a request to decrement the RC of that invocation is enqueued in the *distributed* AQ on that Node. In addition to the request message, the data produced by the thread is also forwarded to the remote Node.

The TSU on each Node continuously checks the local AQ(s) and the distributed AQ in order to decrement the RC of threads invocations.

Once the RC of a specific thread invocation reaches zero, its metadata is moved into the Waiting Queue (WQ). The rest of the activities proceed as described in 3.1. The additional steps required for managing the forwarding of produced data to consumers running on remote Nodes,are described in Section 4.3.

4.2 The Network Interface Unit (NIU)

The NIU, is responsible for the handling the inter-Node communication. NIU was used in a previous DDM implementation [16] to support communication among distributed single-processor Nodes, however it was implemented as a hardware module. In this work the NIU is implemented as a software module that relies on the underlying network hardware interface. Initially, we have consider using MPI [9] for handling the low-level network connectivity in the NIU. However, our evaluation has shown that the expected overheads of invoking an external library were very high. This combined and our need for customized communication, we decided to develop our own optimized connectivity layer using non-blocking TCP sockets.

The NIU is responsible for managing the network initialization, establishing connections with the other Nodes in the system and providing communication services to the TSUs during the execution. The NIU also supports

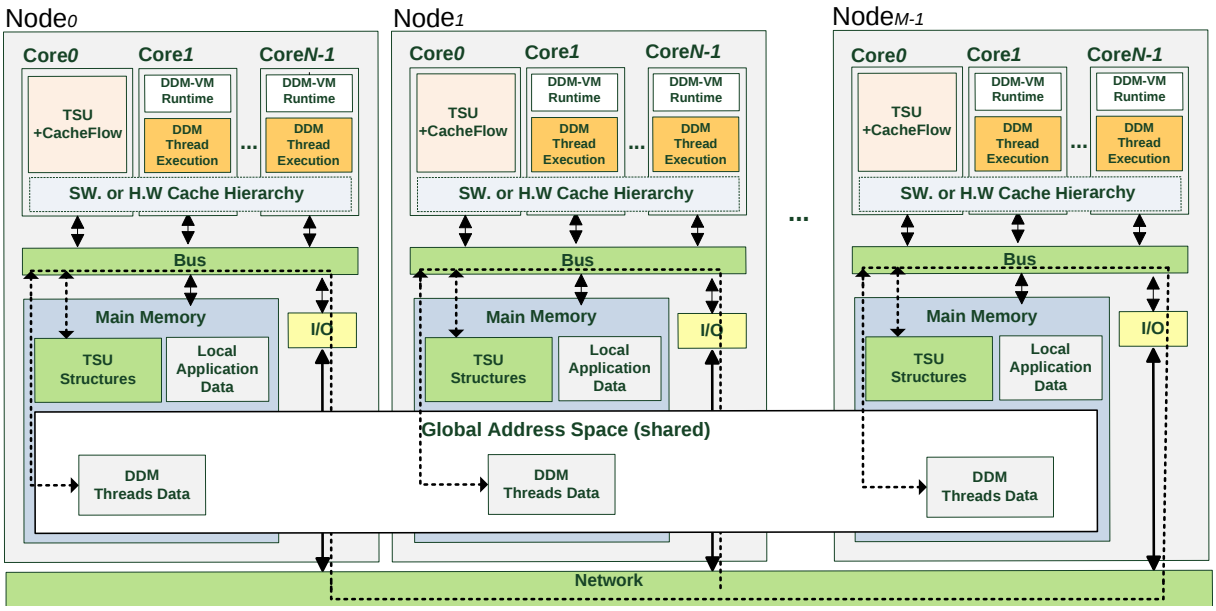

Fig. 3: The Distributed DDM-vm architecture

distributing/gathering data across the global address space in the system at start-up and post-execution of the DDM-vm program.

In the initialization stage the NIU on each Node establishes connections with all the other Nodes. For each Node two non-blocking sockets are allocated, one for sending (*outgoing* socket) and one for receiving (*incoming* socket). Once the connections are established through the sockets, the NIUs exchange information related on the number of cores utilized for DDM execution on each Node. This information is maintained in a table, which used later by the TSU to determine on which Node each core is located.

The NIU abstracts the underlying network and provides the TSU with a simple communication interface. The TSU uses this interface to exchange:

- **Synchronization commands or messages**: the most important one is the request to decrement the *RC* of a specific consumer invocation.
- **Data *forwarding***: when a thread produces data that is needed by a consumer on a remote Node, the TSU passes the data to the NIU to *forward* it to the remote Node.

The NIU tolerates the latencies of network communication by overlapping its work and data transfers with threads execution and the rest of the TSU work. The NIU module is naturally split into two main independent sub-units:

- **The *send* sub-unit**: responsible for sending commands and forwarding data to remote Nodes. Both commands and data are first encapsulated in a simple message with a header describing the content, before they are sent to the remote Node. The sending operation returns when the messages have been stored in the O.S. network layer buffers.
- **The *receive* sub-unit**: responsible for receiving and processing the messages sent from remote Nodes. It continuously polls the *incoming* network connections in a round-robin fashion. The received messages are processed according to their type. In the case of decrement RC request commands, the metadata of the message is inserted as an entry in the *distributed* AQ.

On symmetric multi-cores the *receive* sub-unit is launched in an auxiliary thread (that is pinned to the same core running the TSU), taking advantage of the fact that the processor is an SMT supporting two hardware threads [12]. Furthermore, we further distribute the work of the *send* sub-unit across the cores. This is possible because the services of the *send* sub-unit are invoked by the DDM-vm runtime threads on the cores. Thus, removing the tasks of the *send* sub-unit from the critical path of the TSU.

4.3 Distributed Shared Memory

The DDM-vm supports a Distributed Shared Memory (DSM) [20] abstraction in which part or all of the main memory address space on each Node is mapped to the Global Address Space (GAS) of the DSM. The GAS is a collection of ordered pairs **(Node_id, local_address)** that is shared among all Nodes. The first component refers to the Node identifier and the second refers to a conventional main memory address on that Node. Figure 3 illustrates the GAS across the system Nodes.

Coherence-management operations typically associated with DSM systems [20] are not required between the Nodes,

because produced data is *forwarded* to consumers running on remote Nodes. Coherence operations are only required within each Node's memory hierarchy and so it is managed by the hardware on symmetric multi-cores.

The mapping of the program data into the GAS depends on the assignment of the program threads. The data of each specific invocations of a thread is mapped to the part of the GAS belonging to the Node where this invocations is scheduled to run. The movement of data between producers and consumers running on different Nodes during the execution is managed automatically by the DDM-vm without the Need for programmer intervention of the.

A number of runtime calls facilitate the allocation and release of the data in the GAS. The calls also abstracts the distribution and gathering of data among the Nodes. These calls invoke the services of the NIU to move the data between the main memories of the Nodes.

Data forwarding: DDM-vm employs *Data Forwarding* [19], [13] for tolerating data communication latency across the Nodes. When a *producer* thread executes *Data forward command* the produced data is send immediately to the remote Node where the *consumer* thread is scheduled to execute. This increases the chances of tolerating the communication latency and eliminates the need for remote read operations. Thus, resulting in reducing the total communication cost since remote read operations are usually more costly that remote write operations [14].This also eliminates coherence management operations as previously mentioned.

The forwarding of data completes before decrementing the RC of the consumer thread. Consequently, when a thread RC reaches zero, its data is **guaranteed** to reside in the main memory of remote Node.

The DDM-vm supports threads that produce data consumed by multiple remote consumers running on different Nodes. In this case a *list* of output addresses is provided (instead of only one) and the DDM-vm *forwards* the data to all the locations in the list by creating an FT entry for each remote address.

5. Evaluation

In this section we present the evaluation of the single-Node execution on symmetric multi-cores and the distributed/multi-Node execution both for symmetric multi-cores and the Cell processor. The evaluation of the single-Node execution on the Cell processor is presented in [1].

5.1 Experimental Setup

For the evaluation of the distributed execution we used three clusters each connected using an off-the-shelf Gigabit Ethernet switch. Technical information are listed in Table 1.

The characteristics of the benchmark suite are shown in Tables 2 and 3. The applications used for evaluating the two implementations are the same, However, some of their characteristics differ, such as the granularity (the average dynamic execution time of the application threads) due to the vectorization of the computational kernels in the case of the Cell and the fact that the code is compiled with different compilers. Moreover, the symmetric multi-core system has more main memory compared to that on the PS3, which enables us to use larger input sizes.

5.2 The symmetric multi-core single-Node evaluation

We executed the benchmarks for the *large* input size and the 64x64 granularity. Figure 4 depicts the speedup results. The baseline for the speedup is the best sequential (non-DDM) execution among all the granularities. Note that the maximum possible speedup is 11, since we reserve one core out of the 12 cores for the execution of the TSU.

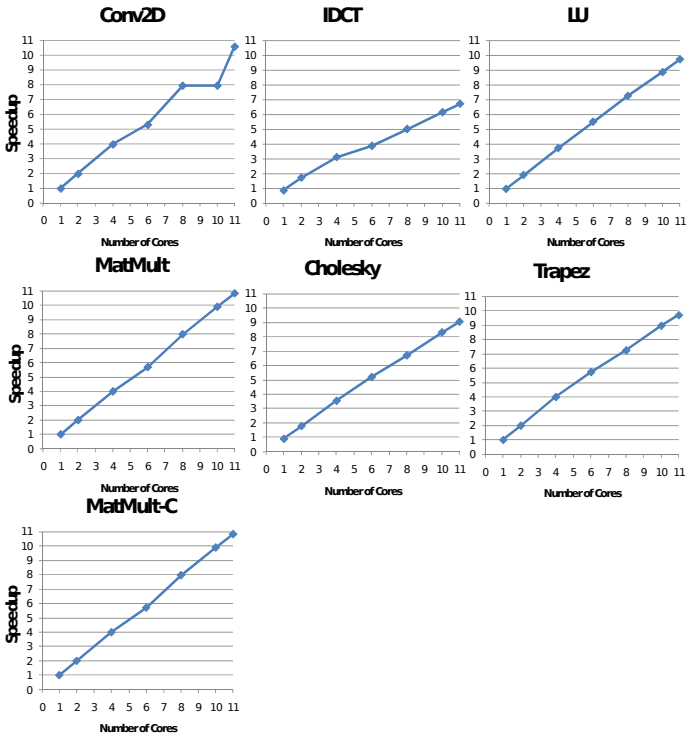

Fig. 4: Symmetric multi-core single-Node speedup

The results demonstrate that overall, the system scales well over the range of the benchmarks and achieves - when utilizing all the cores - an average speedup of 9.6 out of 11, which indicates the efficiency and scalability of the system.

5.3 Distributed symmetric multi-core execution

The benchmarks we executed contains applications with little communication during the execution (Conv2D, IDCT and MatMult, Trapez), and ones with heavy inter-Node communication (LU and Cholesky). For all the benchmarks

Table 1: Experimental setup

Configuration	Nodes	Processors per Node	Memory per Node	O.S.	Network
Homogeneous System 1	2	2 x Six-Core AMD Opteron(tm) Processor 2427	32 GB	Ubuntu Linux 2.6.31	Giga-bit Ethernet
Homogeneous System 2	4	Four-Core AMD Phenom(tm) II X4 B95 Processor	4 GB	Ubuntu Linux 2.6.31	Giga-bit Ethernet
Heterogenous Sony PS3	4	One+Six-Core Cell Broadband Engine	256MB	Fedora Linux 2.6.23-r1	Giga-bit Ethernet

Table 2: The benchmarks suite characteristics - DDM-vm on Cell

Benchmark	Description	Average Granularity of Benchmark Threads		Problem Size		
		Granularity	Execution Time	Large	XLarge	XXLarge
MatMult	Blocked Matrix Multiplication	64x64 block	22.1μs	2048x2048	3072x3072	-
Cholesky	Blocked Cholesky Factorization (vectorized)	64x64 block	22μs	2048x2048	3072x3072	-
Cholesky-S	Blocked Cholesky Factorization (scalar)	64x64 block	8.2ms	2048x2048	3072x3072	-
LU	Blocked LU Decomposition	64x64 block	1.82ms	2048x2048	3072x3072	-
Conv2D	9x9 convolution filter	64x64 block	48.11μs	2048x2048	3072x3072	4096x4096
		96x96 block	107μs			
IDCT	Inverse Discrete Cosine Transform	64x64 block	98.8μs	2048x2048	3072x3072	4096x4096
Trapez	Trapezoidal rule for integration	variable	variable	675K steps	5400K steps	10800K steps

Table 3: The benchmarks suite characteristics - DDM-vm AMD processor

Benchmark	Description	Average Granularity of Benchmark Threads			Problem Size	
		Granularity	Execution Time		XLarge	XXLarge
			System-2	System-1		
MatMult	Blocked Matrix Multiplication	64x64	387 μs	528 μs	4096x4096	8192x8192
		128x128	3333 μs	4540 μs		
MatMult	Blocked Matrix Multiplication - Coarge-grained	64x64	7250 μs	8436 μs	4096x4096	8192x8192
		128x128	28500 μs	36350 μs		
Cholesky	Blocked Cholesky Factorization	64x64	134 μs	182 μs	4096x4096	8192x8192
		128x128	916 μs	1240 μs		
LU	Blocked LU Decomposition	64x64	380 μs	520 μs	4096x4096	8192x8192
		128x128	2918 μs	3975 μs		
Conv2D	9x9 convolution filter	64x64	626 μs	855 μs	4096x4096	8192x8192
		128x128	2500 μs	3416 μs		
IDCT	Inverse Discrete Cosine Transform	64x64	12 μs	17 μs	8192x8192	16384x18384
		128x128	49 μs	68 μs		
Trapez	Trapezoidal Rule of Integration	variable	variable	variable	675M steps	1350M steps

working on matrices we have used blocks of 128x128. In our experiments we utilized 1, 4, 8 and 11 cores per Node for System-1 cluster, which resulted in 2, 8, 16 and 22 total cores in the system, respectively. One core per Node is used as the TSU. For the System-2 cluster we utilized 1, 2 and 3 cores per Node, which resulted in 4, 8 and 12 total cores, respectively (remember that we always reserve one core for the TSU execution and thus the maximum number of utilized cores is 11 on the 12-core machine and 3 on the 4-core machine). We have used two input sizes per benchmark. Figure 5 illustrates the speedup results for both clusters.

The results show that for the largest input size the system achieves an average of 80% and 84% of the maximum possible speedup for the System-1 and System-2 clusters, respectively, which is a very good result. Analysing the results further, it is clear that as the input size increases the system scales better. The average speed-up utilizing all the cores is 13.1 out of 22 for the smaller input size and 16 out of 22 for the larger input size for the System-1 cluster. The average speedup on the System-2 cluster is 8.9 out of 12 for the smaller input size and 9.5 out of 12 for the larger

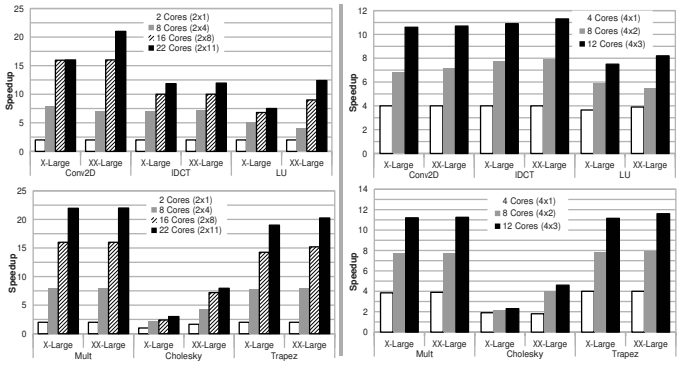

Fig. 5: Distributed symmetric multi-core execution (System-1 left, System-2 right) - Speedup

input size. This is expected as larger problem sizes allow for amortizing the overheads of the parallelization. The results are summarized in Table 4.

Input sizes and granularities (compared to single-Node execution) need to scale as the distributed system scales.

Table 4: Distributed symmetric multi-core execution results - Summary

	System-1 Cluster		System-2 Cluster	
	Smaller Input Size	Larger Input Size	Smaller Input Size	Larger Input Size
Average Speedup Percentage	74%	80%	79%	84%
Average Speedup (utilizing all cores)	13.1/22	16/22	8.9/12	9.5/12

The Cholesky benchmark yields the least performance as its threads exchange data heavily across the Nodes and so it is affected to a great extent by the large latency of the the network. The LU benchmark similarly has a heavy inter-Node data exchange, however, because its threads have a larger granularity compared to Cholesky's (for the same block size), the TSU has a better chance of overlapping the network latencies, thus yielding better performance.

5.4 Distributed execution on the Cell processor

For the evaluation of distributed on the Cell processor execution we used a cluster of four PS3 Nodes. We used the same benchmarks as in 5.3. For all the benchmarks working on matrices we have used blocks of 64x64 except for the Conv2D benchmark in which we used 96x96 blocks. For the Cholesky benchmark we used scalar computational kernels instead of the vectorized ones as the latter proved too fine-grained for the application to scale. We denote the version using the scalar kernels as Cholesky-S. In our experiments we have utilized 1, 2, 4 and 6 SPEs per Node, which resulted in 4, 8, 16 and 24 total SPEs in the system, respectively. Moreover, we have used two input sizes per benchmark. Figure 6 illustrates the speedup results.

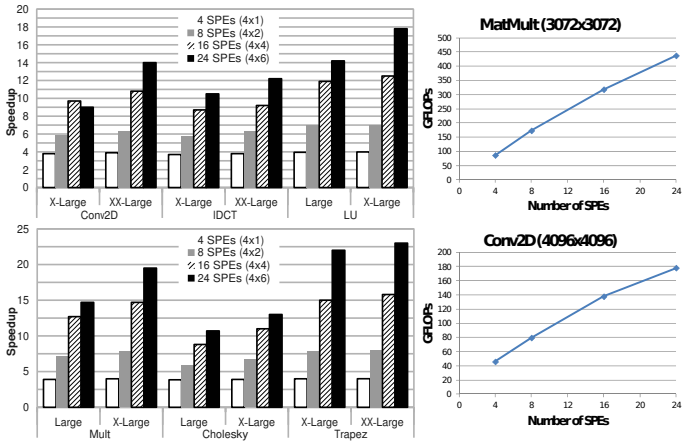

Fig. 6: Distributed execution on the Cell processor - Speedup (left), GFLOPs performance results for MatMult and Conv2D (right)

The results show that for the largest input size the system achieves an average of 80% of the maximum possible speedup for all the benchmarks, which is a very good result. Similar to the results in 5.3, the system scales better as the input size increases as this allows for amortizing the overheads of the parallelization. The average speedup (on all the benchmarks) utilizing all the SPEs is 13.4 out of 24 for the smaller input size and 16.54 out of 24 for the larger input size.

As noted in 5.3, compared to single-Node execution, larger input sizes (on all the benchmarks) and larger granularities (on Conv2D and Trapez) are needed for the system to scale due to the additional latencies introduced by the network data and synchronization messages transfer.

Figure 6 reports the GFLOPs performance results for the two computationally intensive benchmarks MatMult and Conv2D.

The results illustrate that utilizing all the SPEs on the four Nodes the system delivers an impressive 0.44 TFLOPs for the MatMult benchmark and 178 GFLOPs for the Conv2D benchmark, which demonstrates the efficiency of the distributed execution on the Cell.

6. Related Work

Star Superscalar (StarSs) [4], [17], [18] is a parallel programming platform that targets symmetric multiprocessors and multi-cores, the Cell processor and GPUs. It schedules annotated tasks at run-time based on data-dependencies. StarSs focuses on the ease of programmability and portability and utilizes a source-to-source compiler and a number of runtime libraries. Unlike the approach adopted by our work, where we build the dependency graph statically if possible, StarSs always builds its task dependency graph at run-time which incurs extra overheads. Performance comparison between the DDM-vm and the Cell implementation of the StarSs platform can be found in [2].

Open Multi-Processing (OpenMP) [5] is a widely-utilized parallel programming API that supports shared-memory programming. OpenMP traditionally targets loop-based parallelism, but the standard was recently extended with the concept of tasks to accommodate irregular applications.

OmpSs [6] is a variant of OpenMP that incorporates ideas from StarSs to support asynchronous task parallelism on clusters of heterogeneous architectures. It emphasizes programming productivity and uses a compiler/runtime approach to move data across a disjoint address space. The programmer annotates the sequential code with compiler directives that are translated into calls to a runtime system that manages the parallelism extraction and data coherence and movement. The information provided by the programmer is

used by the runtime to distribute the work across the cluster while optimizes communications using affinity scheduling and caching of data.

The Message Passing Interface (MPI) [9] has been traditionally the *de facto* for programming clusters and distributed systems. MPI allows achieving high-performance but sacrifices the ease of programmability. As the programmer has to handle all the low-level tasks of parallelism (partitioning of data and computations, movement of data during program execution, coherence, etc.). This is in contrast with our approach which automates most of these tasks.

A number of programming models have emerged, which try to facilitate the programmability of distributed systems by providing a global address space view of the aggregated memories of all the Nodes in the system. Such a view facilitates porting sequential applications and reduces the complexity of distributed programs. Examples of such models include UPC [10], Chapel [7] and X10 [8]. Although such systems ease the burden of the programmer, achieving good performance still requires a non-trivial effort from the programmer especially that the distribution of data and the coherency are still handled by the programmer. Moreover, such systems require the support of special compilers and libraries.

7. Conclusion and Future Work

In this paper we have demonstrated that Data-flow concurrency can be efficiently implemented on distributed multi-core systems. We have implemented DDM in heterogeneous and homogeneous systems utilizing off-the-shelf networking (Gigabit Ethernet). The evaluation analysis have shown that the achieved results are consistent across multiple different platforms and configurations. This further confirms that the hybrid Data-Flow models, such as DDM, are very promising alternative to the sequential model for distributed systems.

We believe that the DDM model can inspire the evolution of the micro-architecture of the next generation multi-core systems with the addition of a hardware TSU on-chip and the use of a data-driven hierarchy of scratch-pad memories that can replace the traditional multi-level cache hierarchy. Such memory hierarchies will be deterministic and smaller in size than current cache hierarchies. We have built and evaluated a software implementation of such a system for the Cell processor. We are currently developing an FPGA-based distributed multi-core system in which we are introducing these micro-architectural changes.

References

[1] S. Arandi and P. Evripidou, "Programming multi-core architectures using data-flow techniques," in *SAMOS '10: Proceedings of the 10th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, Samos, Greece, July 2010.

[2] ——, "DDM-VMc: the data-driven multithreading virtual machine for the cell processor," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, ser. HiPEAC '11. New York, NY, USA: ACM, 2011, pp. 25–34.

[3] Arvind and K. P. Gostelow, "The u-interpreter," *Computer*, vol. 15, no. 2, pp. 42–49, 1982.

[4] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta, "CellSs: a Programming Model for the Cell BE Architecture," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 86.

[5] O. A. R. Board, "Openmp 3.0 specification," May 2008, http://www.openmp.org.

[6] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with ompss," in *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 555–566.

[7] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005.

[9] J. J. Dongarra, R. Hempel, A. J. Hey, and D. W. Walker, "A proposal for a user-level, message-passing interface in a distributed memory environment," Knoxville, TN, USA, 1993.

[10] C. W. W. et al, "Introduction to upc and language specification," University of California-Berkeley, Tech. Rep., 1999.

[11] P. Evripidou, "Thread Synchronization Unit (TSU): A Building Block for High Performance Computers," *In: Proceedings of the International Symposium on High Perfomance Computing, Fukuoka, Japan.*, 1997.

[12] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[13] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas, "Data forwarding in scalable shared-memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, pp. 1250–1264, December 1996. [Online]. Available: http://portal.acm.org/citation.cfm?id=245565.245581

[14] C. Kyriacou, "Data driven multithreading using conventional control flow microprocessors," Ph.D. dissertation, Dept. of Computer Science, University of Cyprus, 2005.

[15] C. Kyriacou, P. Evripidou, and P. Trancoso, "Cacheflow: A Short-Term Optimal Cache Management Policy for Data Driven Multithreading,," *Proc. EuroPar-04*, pp. 561–570, Aug. 2004.

[16] ——, "Data-Driven Multithreading Using Conventional Microprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 10, pp. 1176–1188, 2006.

[17] J. M. Pérez, P. Bellens, R. M. Badia, and J. Labarta, "Cellss: Making it easier to program the Cell Broadband Engine processor," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 593–604, 2007.

[18] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with starss," *Int. J. High Perform. Comput. Appl.*, vol. 23, pp. 284–299, August 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1572226.1572233

[19] D. K. Poulsen and P.-C. Yew, "Data prefetching and data forwarding in shared memory multiprocessors," in *Proceedings of the 1994 International Conference on Parallel Processing - Volume 02*, ser. ICPP '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 280–280. [Online]. Available: http://dx.doi.org/10.1109/ICPP.1994.81

[20] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: concepts and systems," *Parallel Distributed Technology: Systems Applications, IEEE*, vol. 4, no. 2, pp. 63 –71, summer 1996.

[21] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso, "TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems," in *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 25–34.