Computer Science Department

University of Crete

Runtime support for programming explicit communication chip multiprocessors.

Master Thesis

Michail Zampetakis

April 2010

Heraklion, Greece

University of Crete

Computer Science Department

Runtime support for programming explicit communication chip multiprocessors

Thesis submitted by

Michail Zampetakis

In partial fulfillment of the requirements for the

Master of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________

Michail Zampetakis

Committee Approval: ______________________

Dimitrios Nikolopoulos

Associate Professor, Thesis Supervisor

Committee Approval: ______________________

Manolis Katevenis

Professor

Committee Approval: ____________________________

Dionisios Pnevmatikatos

Professor

Department Approval: ____________________________

Panos Trahanias

Professor, Director of Graduate Studies

Abstract

Modern chip multiprocessors (CMP) with explicit managed local memories offer robust and efficient development systems. Explicitly managed memories allow programmers to control the locality and the exchange of the data of the programs they develop. Using this immediate control of data exchange programmers can develop applications that achieve high performance by optimizing data transfers and apply proper data distribution between local and global memories. Programmers have to develop applications that must be specific for each system in order to fully exploit the available resources and achieve high performance.

In this work we develop several applications using a modern multicore development system based on multiple processors and local memories managed by explicit and implicit communication mechanisms. In order to achieve high performance we exploit the available communication mechanisms to explicitly manage memories and apply data exchange patterns that maximize the resource utilization of the system and achieve high performance. For each application, we measure its performance for various cases and analyze their performance under various circumstances.

We develop a Fast Fourrier Transform (FFT), a bitonic sort algorithm, three applications based on the MapReduce framework and a stream application that measures the communication mechanisms' performance by stressing the system. The system we use is a system that was developed at the

CARV (Computer Architecture and VLSI Systems) laboratory of FORTH (Foundation of Research and Technology) and is based on a modern development platform FPGA (Field Programmable Gate Array).

In this thesis we introduce modules and functionalities in system software libraries, to exploit explicit on-chip communication mechanisms in parallel programming models. Moreover, we port and analyze the performance of the applications for the development system and report techniques on how to exploit the available communications mechanisms in order to achieve high performance using explicit communication mechanisms. We measure the performance and the minimum granularity at which the parallel applications can gain speedup under various cases. And finally we identify the difficulties and the limitations of the applications' porting to the prototype system.

We achieve speedup at parallel execution of the Bitonic sort application that takes even 700 cycles to be executed in sequential execution. In MapReduce applications we achieve speedup almost up to 2 and 4 for two and four processors respectively and in Stream application we stress the communication mechanisms of the prototype system and achieve up to 3200MB/s on-chip data transfer rate.

Περίληψη

Τα σύγχρονα πολυεπεξεργαστικά συστήματα με διαχείριση αποκλειστικών τοπικών μνημών προσφέρουν μια αποτελεσματική πλατφόρμα ανάπτυξης παράλληλων προγραμμάτων. Η ρητή διαχείριση μνημών επιτρέπει στους προγραμματιστές να ελέγχουν άμεσα την τοπικότητα και τη μεταφορά των δεδομένων ενός προγράμματος. Η χρήση αυτού του άμεσου ελέγχου επιτρέπει τη δημιουργία εφαρμογών οι οποίες επιτυγχάνουν υψηλές επιδόσεις αφού οι μεταφορές δεδομένων βελτιστοποιούνται και τα δεδομένα διαμοιράζονται κατάλληλα ανάμεσα σε τοπικές και κοινές μνήμες. Η εκμετάλλευση, όμως, τέτοιων συστημάτων απαιτεί την ύπαρξη κατάλληλων εφαρμογών οι οποίες θα είναι σε θέση να χρησιμοποιούν τους διαθέσιμους πόρους με τέτοιο τρόπο ώστε να πετύχουν την αδιάλειπτή τους χρήση.

Σε αυτή την εργασία αναπτύσσουμε διάφορες εφαρμογές χρησιμοποιώντας ένα πολυεπεξεργαστικό σύστημα ανάπτυξης βασισμένο σε πολλαπλούς πυρήνες με αποκλειστικές τοπικές μνήμες οι οποίες διαχειρίζονται είτε με σαφής είτε με έμμεσους τρόπους επικοινωνίας. Προκειμένου να επιτύχουμε τη μέγιστη επίδοση εκμεταλλευόμαστε τους μηχανισμούς ρητής επικοινωνίας που το σύστημα προσφέρει ώστε να διαχειριστούμε τις μνήμες και να ανταλλάξουμε δεδομένα επιτυγχάνοντας τη μέγιστη δυνατή χρήση των διαθέσιμων πόρων του συστήματος. Ακόμα, μετράμε και αναλύουμε τις επιδόσεις κάθε εφαρμογής για διάφορες περιπτώσεις και αναφέρουμε τις μεθόδους βελτιστοποίησης για κάθε μια.

Οι εφαρμογές που αναπτύσσουμε είναι ο γνωστός μετασχηματισμός Fourrier, ένας διτονικός αλγόριθμος ταξινόμησης, τρεις εφαρμογές Map-Reduce και, τέλος, μια εφαρμογή stream μέτρησης επιδόσεων της μεταφοράς δεδομένων στο σύστημά μας. Το σύστημα το οποίο χρησιμοποιούμε αναπτύχθηκε στο εργαστήριο CARV (Computer Architecture and VLSI Systems) του ITE (Ίδρυμα Τεχνολογίας κι Έρευνας) και βασίζεται σε μια σύγχρονη πλατφόρμα ανάπτυξης FPGA (Field Programmable Gate Array).

Σε αυτή την εργασία προσθέτουμε επιπλέον υπομονάδες στο σύστημα και λειτουργικότητες στις βιβλιοθήκες, ώστε να εκμεταλλευτούμε την ρητή επικοινωνία στα παράλληλα προγραμματιστικά μοντέλα. Επιπλέον, μεταφέρουμε και αναλύουμε τις επιδόσεις των εφαρμογών και αναφέρουμε τεχνικές εκμετάλλευσης των διαθέσιμων μηχανισμών επικοινωνίας ώστε να επιτύχουμε υψηλές επιδόσεις με τη χρήση των ρητών μεθόδων επικοινωνίας. Μετράμε την επίδοση και τον ελάχιστο κόκκο προγράμματος όπου μπορούμε να επιτύχουμε επιτάχυνση της παράλληλης εκτέλεσης μιας εφαρμογής συγκρινόμενη με τη σειριακή σε διάφορες περιπτώσεις. Τέλος, αναφέρουμε τις δυσκολίες και τους περιορισμούς της ανάπτυξης των εφαρμογών στο πρωτότυπο σύστημα.

Μετράμε επιτάχυνση της παράλληλης εκτέλεσης του αλγόριθμου της διτονικής ταξινόμησης ο οποίος απαιτεί μόλις 700 κύκλους σειριακής εκτέλεσης. Στις εφαρμογές MapReduce μετράμε επιτάχυνση της εκτέλεσης μέχρι περίπου 2 και 4 για δυο και τέσσερις επεξεργαστές αντίστοιχα και στην Stream εφαρμογή πιέζουμε τους μηχανισμούς επικοινωνίας του συστήματος επιτυγχάνοντας ρυθμούς μεταφοράς on-chip δεδομένων με ταχύτητες μέχρι και 3200MB/s.

Acknowledgments

I feel grateful to my supervisor, Prof Dimitris Nikolopoulos as also Prof. Manolis Katevenis and Prof. Dionisios Pnevmatikatos for their valuable assistance and guidance during my postgraduate studies.

My warmest appreciation to the members of the CARV Laboratory of the FORTH-ICS and especially to the members of SARC group.

Last but not least, I would like to thank my family, my parents Fanouris and Anthi and my brother Nikos for the support and encouragement they provided me with.

This work was carried out with the financial and technical support from FORTH - ICS and the Europoean Commission in the context of the SARC integrated, and with the assistance of the HiPEAC Network of Excellence.

Michail Zampetakis
April 2009

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As applications become more and more demanding in processing recourses, simple processors have been replaced by sophisticated multicore processors or multiprocessor systems. Such systems are used to accomplish heavy tasks, experiments, even everyday tasks in desktop computers. Multicore processors provide lower power consumption with higher performance and low design complexity. Several systems embed this kind of processors. These include high performance computers, desktop computers or even embedded processors in mobile devices. These systems, however, demand high performance memory systems, fast data exchange mechanisms and efficient processing units in order to achieve high performance.

There are two dominant schemes of memory hierarchies of modern multicore computing; either multi-level cache (with coherence support), or scratchpads (with DMA functionalities). General purpose systems usually use the case of caches due to the transparent (implicit) way of handling data locality and communication. Data are located and then moved not under the direct control of the application software; instead, data copies are placed and moved as a result of cache misses or cache coherence events, which are indirect only results of application software actions. The benefit is simplicity: the application programmer does not need to worry about where data should reside and how and when they should be moved. The disadvantage is inability to optimize for the specific data transfer patterns that occur in specific applications. Scratchpads are on-chip SRAM, which are a small, high-speed data memory that is connected to the same address and data buses with off-chip memory. This makes them efficient for storing data to process. One main difference between the scratchpad SRAM and data cache is that the SRAM guarantees a single-cycle access time, whereas an access to cache is subject to compulsory, capacity, and conflict misses.

However, in order to fully exploit a system with explicit communication mechanisms, programmers should create applications with awareness of the available resources of the system and the advantages and disadvantages between different communication schemes. Programmer needs to manage scratchpad for software caching of data and implement data communication between cores as efficient as possible. Applications should exploit all of the available processing elements without any significant overhead and implement efficient communication between memories.

In order to study all the above, we use a FPGA development board with a complete multiprocessor system. The system contains four processors, each with a local scratchpad memory and a cache hierarchy, an external DDR Memory, a

NoC and other peripherals. These modules, connected with buses and point to point connections, provide a complete development environment for writing and studying parallel applications. The complete architecture is described in detail in Chapter 2.

In order to achieve high performance, programmers should take care of several programming issues, especially when using systems with explicit communication mechanisms. We take these issues into consideration and present several techniques to fully exploit these mechanisms. In order to achieve high performance on such systems, we use communication mechanisms in particular ways. We use remote stores for small data transfers as these perform better than DMAs which are faster for big data transfer sizes. Moreover, communication time should be overlapped with computation time in order processors not to idle wait. We also use multiple buffering when possible, in order to maximize the memory's throughput to achieve faster data transfers.

These techniques stress the communication mechanisms of the system and achieve high performance. We are able to achieve speedup for parallel execution of programs that take even 700 clock cycles at the sequential execution. Bitonic sort is an application that can achieve speedup for such small task size. Moreover, data transfers can be overlapped with computations by using DMAs for small sizes depending on the application. Stream application shows up that communication can be fully overlapped by computations even with DMAs as small as 512B and double-buffering.

1.1 Thesis Contribution

The contributions of this thesis are the following:

1. Introduce modules and functionalities in system software libraries, to exploit explicit on-chip communication mechanisms in parallel programming models.

2. Port and analyze the performance of several applications for the development system.

3. Report techniques on how to exploit the available communications mechanisms in order to achieve high performance using explicit communication mechanisms.

4. Measure the performance and the minimum granularity at which the parallel applications can gain speedup under various cases.

5. Identify the difficulties and the limitations of the applications' porting to the system.

1.2 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 describes the development platform we use in order to develop the applications. We present and analyze the FFT, the bitonic sort, the Map Reduce and the stream benchmark with their results in Chapter 3 to 6 respectively. Chapter 7 refers to related work. We summarize our work and conclude with Chapter 8.

Chapter 2

Development Platform

The system we use is based on a Xilinx Virtex-5 FPGA XUPV5-LX110T board [1]. It contains four processors, each with a level one data cache, a runtime configurable level two data cache, a global off chip DDR RAM memory, a crossbar connecting the above modules and several other peripherals that help accomplish common tasks. These are described in the next chapters.

2.1 Processors

The system integrates four soft core MicroBlaze processors. The MicroBlaze processor is a reduced instruction set computer (RISC) optimized for implementation in Field Programmable Gate Arrays (FPGAs). Figure 2.1 shows a functional block diagram of the MicroBlaze core. The fixed feature set of the processor includes 32-bit general purpose registers, 32-bit instruction word with three operands and two addressing modes, 32-bit address bus, single issue pipeline. The MicroBlaze soft core processor is highly configurable, allowing us to select a specific set of features required by our design. So, in addition to these fixed features, we parameterized all MicroBlaze processors with additional features. Some of the most fundamental additional functionalities we use are the instruction cache over Cache Link (IXCL) interface, the 32-bit integer multiplier and the processor version register (PVR) which is unique for each processor in the system. The instruction cache is direct mapped (1-way associative) with user selectable cacheable memory address range, configurable cache and tag size and an option to use 4 or 8 word cache-line. We use caches of size 4KB on each processor with 8 word cache-line size. The code was in the external DDR RAM and is caching through an IXCL bus to the instruction cache. Moreover we added the PVR to distinguish different processors at runtime in order each processor to accomplish the appropriate tasks. Each processor runs at 75MHz.

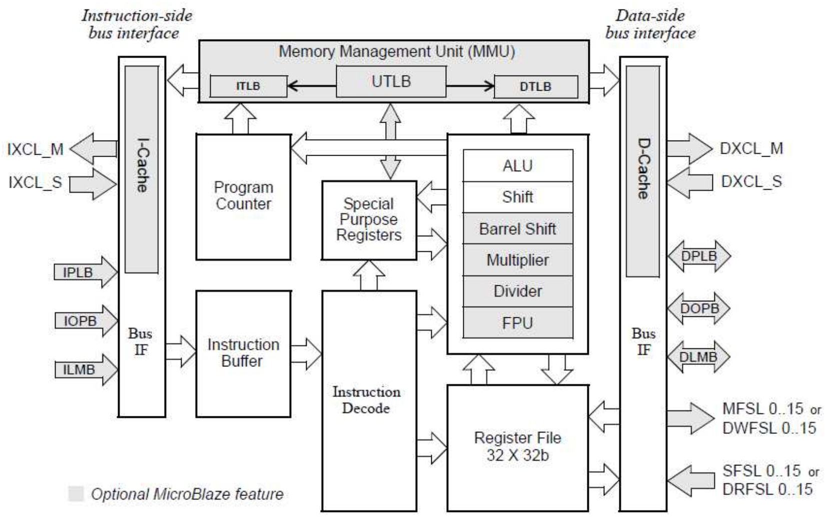


Figure: 2.1: MicroBlaze core block diagram

2.2 Memories

Each processor Node contains a MicroBlaze, a L1 cache and a unified L2 with a NI controller. L1 cache is 8KB direct map with 32bytes cache line size and the L2 cache is a 4-way, phased, 64KB with 32bytes cache line size. Each L2 cache line can be configured at runtime to behave as a command line, a control line or as simple memory [3]. This means that it can be a scratchpad line, a queue, a counter or a completion notification space of an event. At least one of these ways should be left unconfigured in order to allow accesses to the external memory.

The system contains one external DDR2 SDRAM of size 256MB which we use for just storing data to process and for storing the necessary segments of code and data for each program. These segments are the stack, the heap, the text, the rodata, the ini, the fini and several other common segments that are used by the processor to execute the code. These segments are read by each processor and are stored in the instruction cache or the data cache.

2.3 NoC

Each processor and the external memory are connected to a network interface (NI) device in order to communicate with each other through a centralized crossbar. The NI is tightly-coupled to the L2 cache and serves all data transfers from/to tile's configurable memory and the NoC. NI supports special packets formats for communication purposes. The NoC is consisted of one arbiter for each NI and the Data of each NI_Out module are distributed to every arbiter of the NoC. The figure 2.2 shows the block diagram of the NoC.

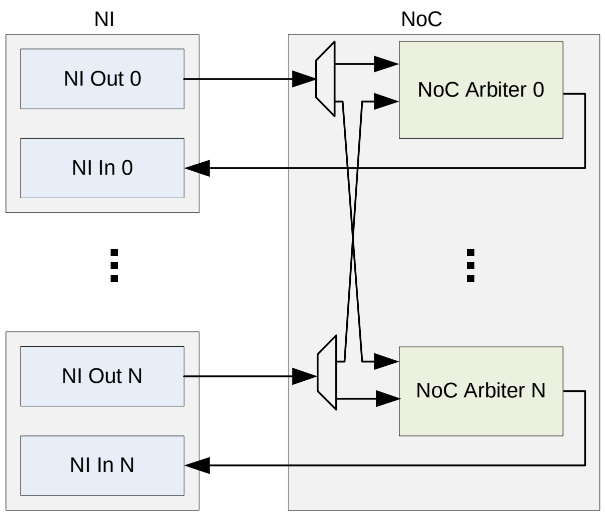


Figure 2.2: NoC block diagram

2.4 Peripherals

Except from the above basic modules, the system contains several peripherals that provide programmers an integrated development environment. There is a mutex, a RS232 UART controller and a global accessible counter module embedded in the system. The mutex module provides a lock mechanism for mutual exclusion and the RS232 UART module provides support for performing console I/O, debugging, etc.

In order to measure the performance of an application it is desirable to have a common basis among all processors. In the prototype system we use a global common counter which is 64-bit wide and increases at each clock cycle. This provides the programmers with a basis of measuring the performance of their applications on the real system during runtime.

2.5 Overall View

Composing all the above modules creates the system we use. The block diagram of the system is presented in figure 2.3. It contains all of the modules connected with several busses and point to point connections. Each node is consisted of the MicroBlaze processor, the instruction cache, the L1 cache memory, the arbiter, the configurable L2 cache memory and the network interface. MicroBlaze is connected through the PLB bus to the Mutex, the UART and the DRAM controller modules. Each instruction cache is directly connected to the DRAM controller in order to read the requested code segments each time. DRAM controller is connected to a network interface in order to serve requests from the processors to or from the DRM. Finally the Mutex and the UART peripherals are connected to the PLB bus and are accessible by every core in the system.

A previous version of the prototype was presented in [2] and [3]. The current version is a major rewrite of the code, optimized for logic reuse, implementing event responses, three levels of NoC priority and some other features not present in the previous versions.

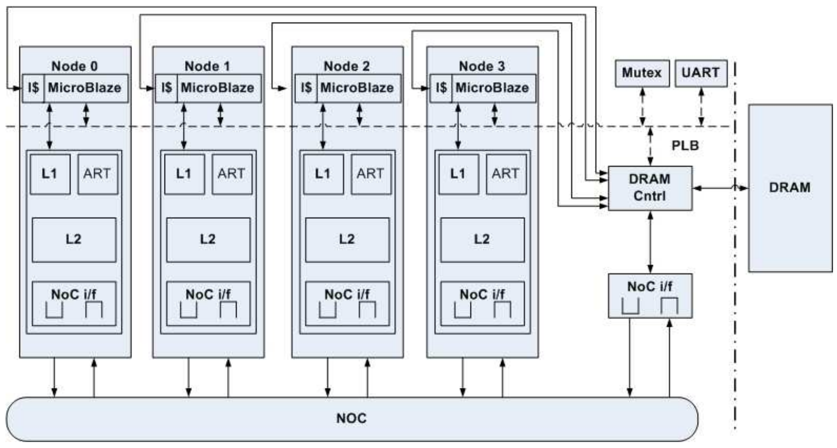


Figure 2.3: System block diagram

The described system has fully implemented in a hardware prototype based on Xilinx Virtex-5 FPGA XUPV5-LX110T board [1]. A view of this board is presented in figure 2.4. The XUPV5-LX110T Development System features a Xilinx Virtex-5 XC5VLX110T FPGA, a Xilinx System ACE Compact Flash configuration controller, a 64-bit wide 256Mbyte DDR2 small outline DIMM (SODIMM) module compatible with EDK supported IP and software drivers, a 10/100/1000 tri-speed Ethernet PHY supporting MII, GMII, RGMII, and SGMII interfaces, a USB host and peripheral controllers, a RS-232 port, a 16x2 character LCD, and many other I/O devices and ports.

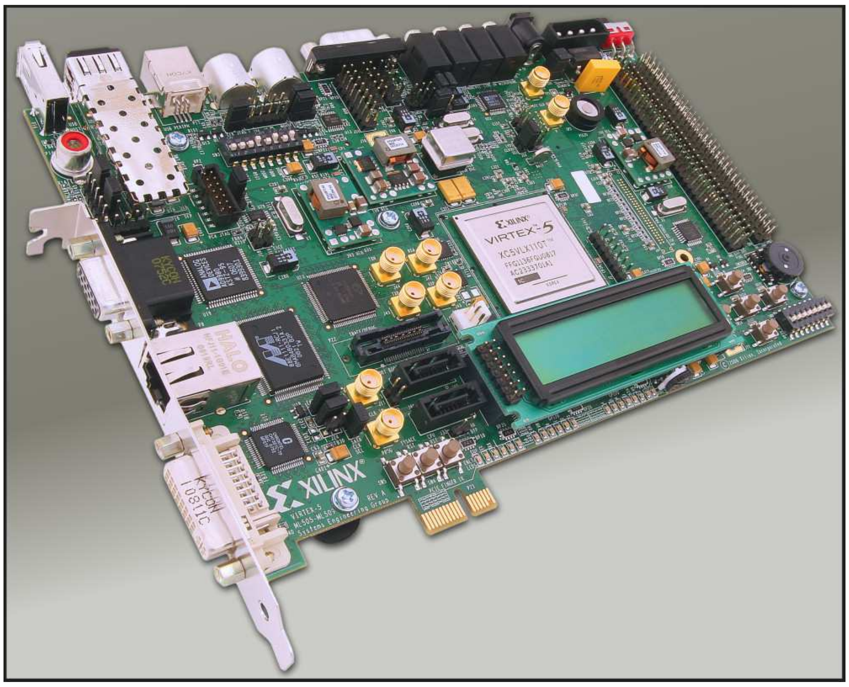


Figure 2.4: Xilinx Virtex-5 FPGA XUPV5-LX110T board

2.6 Communication

As the number of processing cores per chip increases, so does the need for efficient and high-speed communication and synchronization support, so that applications can exploit the numerous available cores. A sophisticated system must support at least some basic communication mechanisms such as DMAs and simple memory accesses. The prototype system, apart from these basic functions, supports interprocessor communication mechanisms with rDMAs and remote stores to scratchpad memories between processors.

The system provides mechanisms to transfer data with DMAs from any scratchpad or the DRAM to any other scratchpad or the DRAM. It is also possible to direct access the DRAM through the cacheable path through the NoC or through a direct link with uncacheable accesses. There is also the capability of rDMAs (remote DMAs) where a processor is able to initiate a DMA transfer from one processor to another without being necessary one of the participants. The local scratchpad memories can be also accessed directly as usual, however, the remote scratchpad memories of other processors can be accessed directly only with store commands (remote stores).

In order to achieve more efficient communication between processors the prototype system provides Remote Stores, Remote queues, Messages, and Counters that offer additional flexibility to the programmers [3]. Remote Stores to scratchpad regions of remote processors, optimize the latency of single-word data transfers. Remote Queues is an appropriate level of abstraction for multiprocessor synchronization where fast multi-word Messages, e.g. data up to cache-line size, from multiple sources can perform atomic remote enqueues. Queues are hosted inside scratchpad regions and their configuration (size and pointers) can be programmed in special control lines, marked in the tags of the cache-scratchpad. Messages are initiated through NI command buffers, already used for DMAs, where data are provided directly by the processor – no source address is needed. Finally Counters have implemented, also hosted in scratchpad space, as a primitive to support RDMA completion detection, barriers, and other synchronization primitives. Counters are initialized with a value (transfer size in bytes) and trigger writing to notification addresses when they expire (reach zero). The software can specify an acknowledgement address in NI commands to identify a counter that will gather all partial acknowledgements for DMA segments; acknowledgement addresses are allowed to be “null” to deactivate the mechanism. As for the remote stores, there is a special register which holds the number of pending remote stores, issued by each processor, and allows each processor to check whether all remote stores have been completed.

All of the provided communication mechanisms have advantages and disadvantages compared to the other mechanisms. Scratchpad loads have a latency of 4 clock cycles while stores take 3 clock cycles to be committed to memory. The observed processor latency for stores is 1 clock cycle, since all stores are “posted” and pipelined in the prototype system. Remote-Stores of 4-bytes cost 27 cycles and are faster than the equivalent messages and DMAs, since the initiation is implicit. Minimum-sized messages and DMAs of 4-bytes have the same end-to-end latency of 30 clock cycles. Large DMAs cost a significant amount of cycles, e.g. a 128-byte DMA costs 76 cycles and this is attributed mostly to latency enforced by the “store-and forward” operation at the receiver.

2.7 Libraries

In earlier works [16] [17] several libraries developed for this specific system in order to support the basic functionality and to provide the programmers some fundamental primitives. The libraries are separated in four categories, the system library, the NI library, the scratchpad library and the synchronization library.

The system library contains the most essential functionalities of the design. It implements locks, barriers, memory allocation, and basic timing and I/O facilities; it provides alternative implementations of locks and barriers, thread-safe memory allocation, thread-safe I/O functions, and basic mechanisms for getting a core ID and the value of a global system timer. In table 2.1 we illustrate the most fundamental mechanisms with a short description that the system library provides.

Function	Arguments	Returns	Description
sys_getcpuid	-	A processor ID	Returns the P.V.R. of the current processor
sys_init	-	-	Initializes the mutex, the NI and the caches of the system.
sys_timer_low	-	A timestamp	Returns the value of the global counter
sys_malloc	Size in bytes	An address	Thread safe malloc function for the external DRAM
sys_printf	The message	-	Thread safe printf function.

Table 2.1: Functions of the system library

The NI library implements the basic functions of the network interface. It contains functions for preparing and issuing DMAs, for managing command buffers, notifications, and queues, and for sending messages to remote scratchpad memories. Table 2.1 reviews the most fundamental mechanisms with a short description that the system library provides.

Function	Arguments	Returns	Description
ni_cmd_alloc	-	Allocated Address	Allocates a command line.
ni_cmd_alloc_Wnotif	-	Allocated Address	Allocates a command line with notification.
ni_cmd_wait_complete	A command buffer	-	Blocks till notification arrives.
ni_queue_alloc	The queue size	An address	Returns an address to an allocated queue.
ni_queue_size	A queue's address	The size	Returns the size of a given queue.
ni_queue_get_item	A queue's address	A queue element	Dequeues an item from the specific queue.
ni_cmd_dma	A DMA command buffer, a source & a destination address, a size	-	Initiates a DMA transfer of the given size, from the source to the destination addresses.
ni_cmd_msg_data_1-5	A command handle address, a destination address and 1 to 5 words to send	-	Sends contiguously 1 up to 5 words to an address space.

Table 2.2: Functions of the NI library

The scratchpad library manipulates scratchpad memory: allocate a part of the L2 cache memory as scratchpad space at runtime, convert local addresses to remote addresses, and check if an address is local or remote. The scratchpad library also implements primitives for marking a cache line as a queue, a counter, a register, or a control line. Table 2.3 shows the most commonly used mechanisms with a short description that the scratchpad library provides.

Function	Arguments	Returns	Description
scr_get_way_addr	A way number	A new address	Returns the local scratchpad address of the way.
scr_make_addr	An offset	A new address	Returns the local base scratchpad address plus the offset.
scr_make_addr_ remote	A processor's ID, an offset	A new Address	Returns the base scratchpad address of the processor with that ID plus the offset.
scr_mark_line	A line address, a tag value	-	Marks that cache line with the given tag
scr_malloc	Size in bytes	An address	Returns the first scratchpad address that allocated.
scr_is_local	An address	If it is local	-
scr_mark_mrQ	An address, the size	-	Allocates a multiple readers queue.

Table 2.3: Functions of the scratchpad library

Finally the synchronization library provides synchronization methods that are commonly used by the programmers based on the mutex module and the queues. The first one uses the hardware mutex peripheral to implement the locking mechanism and the barrier, and the second one uses the hardware queues and counters for the mutex and the barrier accordingly. The fundamental functions for these mechanisms are presented at table 2.4.

Function	Arguments	Returns	Description
sys_mutex_init	A mutex variable	-	Initializes the mutex.
sys_mutex_lock	A mutex variable	-	Blocks till mutex lock is acquired.
sys_mutex_unlock	A mutex variable	-	Releases the mutex lock.
sys_barrier_init	A barrier variable	-	Initializes the barrier.
sys_barrier_wait	A barrier variable, the amount of participants	-	Blocks till all participants join the barrier.
Barrier_Init	A counter barrier address	-	Initializes the counter barrier.
Barrier	A counter barrier address	-	Blocks till all participants join the barrier.

Lock_Init	A queue mutex variable	-	Initializes the queue mutex.
mrQ_Lock	A queue mutex variable	-	Blocks till queue mutex lock is acquired.
mrQ_Unlock	A queue mutex variable	-	Releases the queue mutex lock

Table 2.4: Functions of the synchronization library

For better apprehension of the provided mechanisms we present here the way to use some of the functions that libraries provide. We present the methods to send a message, to initiate and wait for completion a DMA and a Remote Store and how to create and manipulate a queue.

```
int nBytes=4, id=1;
/* allocate a command buffer for the message */
ni_cmd_handle cmd_buf;
/* allocate 4 Bytes scratchpad memory at Base_Scr_Addr
address */
u32 Base_Scr_Addr = scr_malloc(nBytes);
/* Send to the remote scratchpad of Processor 1, the local
Scratchpad base address*/
ni_cmd_msg_data_1(cmd_buf ,  scr_make_addr_remote(id,
Base_Scr_Addr) , Base_Scr_Addr);
```

Figure 2.5: NI message example

```
int nBytes =4, id=1, data=1234;
/* REM_STORE_CNT_BASE0 address contains the pending
remote stores counter */
volatile u32 RS_cnt_addr=REM_STORE_CNT_BASE0;
/* allocate 4 Bytes scratchpad memory at Base_Scr_Addr
address */
u32 Base_Scr_Addr = scr_malloc(nBytes);
/* create a remote address to make the remote store*/
u32 Remote_Scr_Addr = scr_make_addr_remote(id,
Base_Scr_Addr)
/* Initiate the remote store */
*Remote_Scr_Addr=data;
/* poll the counter till all pending remote stores arrive */
while (*RS_cnt_addr!=0) ;
```

Figure 2.6: Remote Store example

```
int line=4;
ni_cmd_handle dma;
/* define DMA’s size equals to 16 bytes */
int DMA_SIZE=16;
u32 remote, local;
/* allocate a command buffer with notification for the DMA*/
ni_cmd_alloc_Wnotif(&dma);
/* Poll_Addr will be the address that the notification will
arrive at the DMA completion */
volatile unsigned long *Poll_Addr;
/* allocate space for the notification and initialize it */
Poll_Addr=scr_malloc(line);
*Poll_Addr=0'
/* Update the notification counter of the DMA */
ni_notif_update(dma.notif, DMA_SIZE, Poll_Addr);
/* Assume a local and a remote address for the DMA */
local = &Scr_Base;
remote = &Scr_Remote;
/* Initiate the DMA */
ni_cmd_dma(dma, local, remote, DMA_SIZE);
/* Wait for completion */
ni_cmd_wait_completeL(dma);
```

Figure 2.7: DMA example

```
u32 TOKEN=0xCAFECAFE;
/* allocate a command buffer with notification for the queue*/
ni_cmd_handle cmd_buf;
ni_cmd_alloc(&cmd_buf.handle);
/* allocate a scratchpad region for the queue */
uint32 addr =
scr_aligned_array_malloc(QUEUE_SIZE*LINE_SIZE);
/* mark scratchpad region as queue */
scr_mark_mrQ(addr, QUEUE_SIZE_BITS);
/* enqueue the TOKE to the queue */
ni_cmd_msg_data_1(cmd_buf.handle, addr, TOKEN);

/* allocate a scratchpad line */
u32 Base_Scr_Addr = scr_malloc(nBytes);
/* dequeue the TOKEN from the queue and write it to
Base_Scr_Addr */
ni_cmd_read_msg(cmd_buf.handle, addr, Base_Scr_Addr,
0x14);
```

Figure 2.8: Multiple Reader Queue example

2.8 Tools

For the hardware and software synthesis we use the ISE design suite and the Embedded Development Kit (EDK) tools. They provide a complete flow for RTL-based designs and Intellectual Property (IP) components. For compiling software, we use a version of gcc, mb-gcc, targeted to Microblaze processors and the Xilinx Microprocessor Debug (XMD) engine, for debugging.

Tools offer to the programmers a lot of options to develop, run and debug theirs code. A program can run from 1 up to all four processors, debug is available at runtime and it is possible to generate custom linker scripts for different purposes. Different codes can be downloaded to each processor. However, we use the same code, to all participating processors, parameterized according to the processor’s PVR. This code resides at the same memory (DRAM) but it is stored in different address space. During compilation tools add extra code segment that are specific for each processor. As a result we have to download and execute the code at different segments whereas all processors execute exactly the same code.

Chapter 3

Stream

3.1 Benchmark Structure

The STREAM triad benchmark [4] stresses the bandwidth at different layers of the memory hierarchy. The benchmark copies three arrays from a “remote” to a “local” memory, conducts one addition and one multiplication on each array element, and sends the results back to original “remote” memory.

We develop two configurations of STREAM for stressing on-chip and off-chip memory bandwidth respectively. In the on-chip configuration, the data streams from scratchpad memories to scratchpad memories and backwards, whereas in the off-chip configuration, data streams from DRAM to scratchpad memories. In both cases we have developed versions using the DMAs and the remote stores communication mechanisms.

3.2 Application Analysis

For these two configurations (on-chip and off-chip) we test single to multiple buffering using remote stores and DMAs for data exchange. As remote loads are not supported by the system, we are not able to develop the application without using the DMA mechanism. In the remote stores configuration, all processors execute remote reads with DMA operations and writes by remote store operations. We use multiple buffering because it stresses the memories bandwidth as it overlaps communication with computation. Moreover, we can parameterize the size of all buffers in order to observe the impact of it on the total performance of the system. By increasing the buffer size and the number of buffers, we are able to observe whether we are able to achieve the maximum bandwidth of the system.

For on-chip communications, initial data are stored in a scratchpad of a single processor. All other processors request with DMAs to fill their buffers from that remote scratchpad to their local one. Then, they calculate the results and send them back to the initial processor using DMAs or remote stores. Calculations are one addition and one multiplication each time. For off-chip communication, processors follow the same scheme but with the difference that the initial data are stored in the external memory.

In figure 3.1we illustrate the block diagram of the Stream Triad benchmark for a single processor and for double buffering. We present the data transfers only for the “a” buffers but “b” and “c” are the same as well. Initially each processor copies to its local scratchpad a portion of each array a, b and c. With this way it fills the a1, b1 and c1 buffers. Then, processors request to fill the data of the next set of buffers, a2, b2 and c2 and starts calculating the results using the 3 first buffers a1, b1 and c1. Processors write the results to the a1’, b1’ and c1’ buffers and then they send back the results to the DRAM. Processors request again to fill the buffers a1, b1 and c1, which just processed, with data from the DRAM. If the next set of buffers a2, b2 and c2 has filled with the data that processors requested, they start calculating this new and fill the result buffers a2’, b2’ and c2’ with the results. This is done for all the multiple buffers till data finish. In the on-chip version, transfers take place between scratchpad memories of different processors. This means that DRAM does not participate and the processors read and store data from a remote scratchpad of another processor following the same scheme.

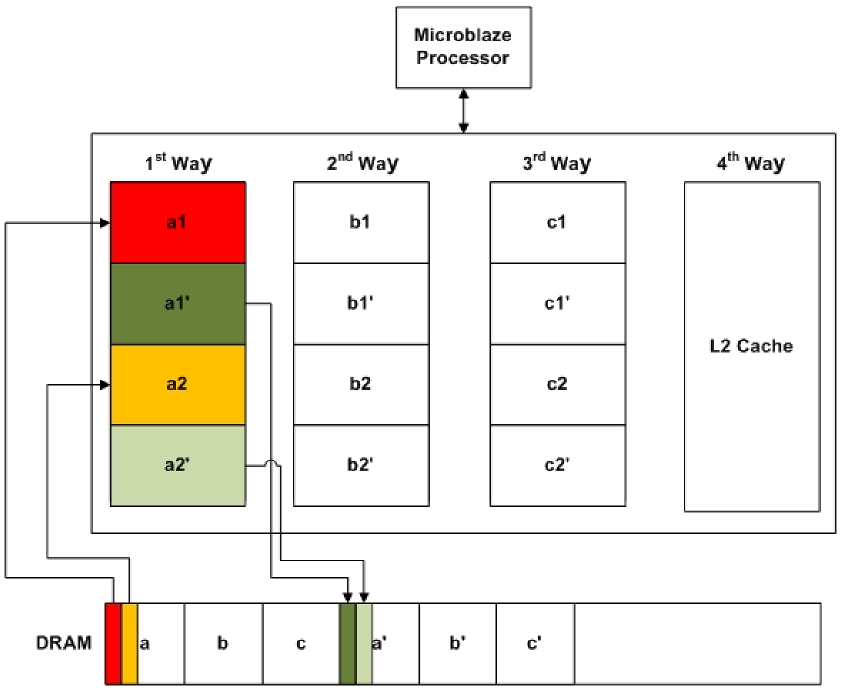


Figure 3.1: The STREAM Triad benchmark block diagram

3.3 Results

We parameterize the application in order to run it for one, two or four processors in parallel, to use remote stores or DMAs for data transfers, for various buffer sizes and to use or not multiple buffering of various number of buffering. In each case we measure the total time of the application in order to observe the performance when using each combination of the above.

We compile the Stream benchmark with medium (-02) optimizations for gcc and with the flags "-funroll-loops -fmodulo-sched" in all cases. The "-funroll-loops" optimization option will perform the optimization of loop unrolling and will do it only for loops whose number of iterations can be determined at compile time or run time which is done in most of the cases for our applications. The "-fmodulo-sched" optimization option will perform swing modulo scheduling immediately before the first scheduling pass. This pass looks at the innermost loops and reorders their instructions by overlapping different iterations. As Microblaze processor has a very simple branch predictor, these two optimizations offer a general speed improvement to our applications. Moreover, we apply warm up in order to take advantage of spatial and temporal locality of instruction and data caches of the system.

In figure 3.2 we analyze the aggregate bandwidth of the STREAM application with off-chip transfers using DMAs. We plot the maximum feasible bandwidth of the system (horizontal line) for off-chip DMAs and the realizable bandwidth while we vary the buffer size, the number of participating processors and the number of the used buffers for overlapping computation with memory latency. We observe that when we use 3-buffering bandwidth saturates in all cases (4-buffering curves are overlapped with 3-buffering curves). As participating processors increase so the aggregate bandwidth does. This aggregate bandwidth, however, does not double when we double the processors because the controllers and the memories saturate by the big amount of requests. Moreover, as buffer size increases we observe that bandwidths increases significant from 512B to 1KB but from 1KB to 2KB buffer size, bandwidth increases by a small amount. The maximum aggregate bandwidth that the benchmark manages to achieve for off-chip transfers using DMAs is about 180MB/s.

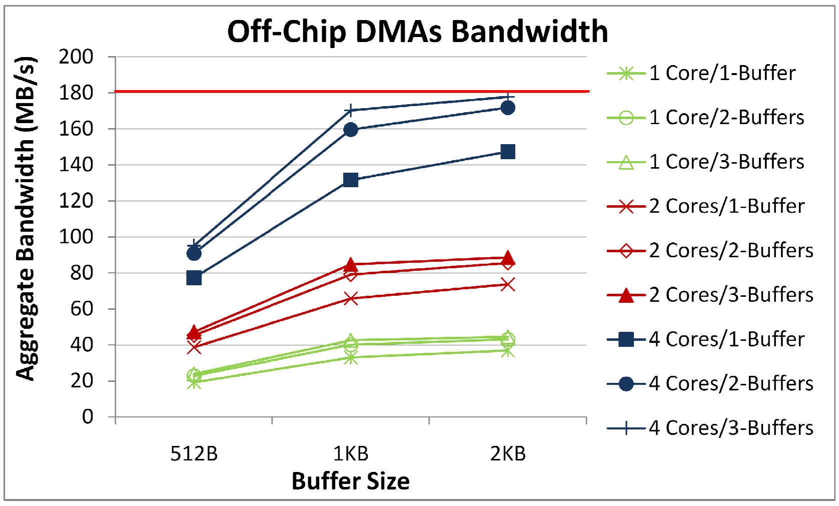


Figure 3.2: Performance of STREAM benchmark with off-chip DMAs

We present in figure 3.3 the bandwidth that the STREAM benchmark achieves for on-chip transfers using DMAs. We plot the maximum feasible bandwidth on the prototype (horizontal line) for on-chip DMAs and the realizable bandwidth while we vary the buffer size, the number of participating processors and the number of the buffers we use for overlapping computation with memory latency. We observe that bandwidth saturates when we use 3-buffering in all cases (4-buffering curves are overlapped with 3-buffering curves). As participating processors increase so the aggregate bandwidth does. And as buffer size increases we observe that bandwidths increases significant from 512B to 1KB but from 1KB to 2KB buffer size bandwidth has minor increase. The maximum aggregate bandwidth that the benchmark manages to achieve for off-chip transfers using DMAs is about 320MB/s.

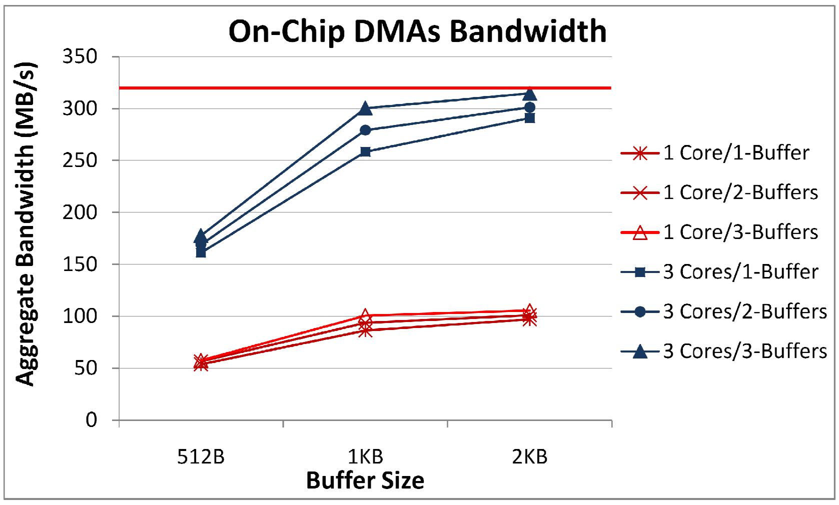


Figure 3.3: Performance of STREAM benchmark with on-chip DMAs

In figure 3.4 we present the aggregate bandwidth of the STREAM application with off-chip transfers using remote stores. Once more we plot the maximum feasible bandwidth of the system (horizontal line) for off-chip remote sores and the realizable bandwidth while we vary the buffer size, the number of participating processors and the number of the buffers we use for overlapping computation with memory latency. We observe that bandwidth saturates when we use 3-buffering in all cases (4-buffering curves are overlapped with 3-buffering curves). As participating processors increase so the aggregate bandwidth does. And as buffer size increases we observe that bandwidths increases significant from 512B to 1KB but from 1KB to 2KB buffer size bandwidth has minor increase. The maximum aggregate bandwidth that the benchmark manages to achieve for off-chip transfers using remote stores is about 21MB/s.

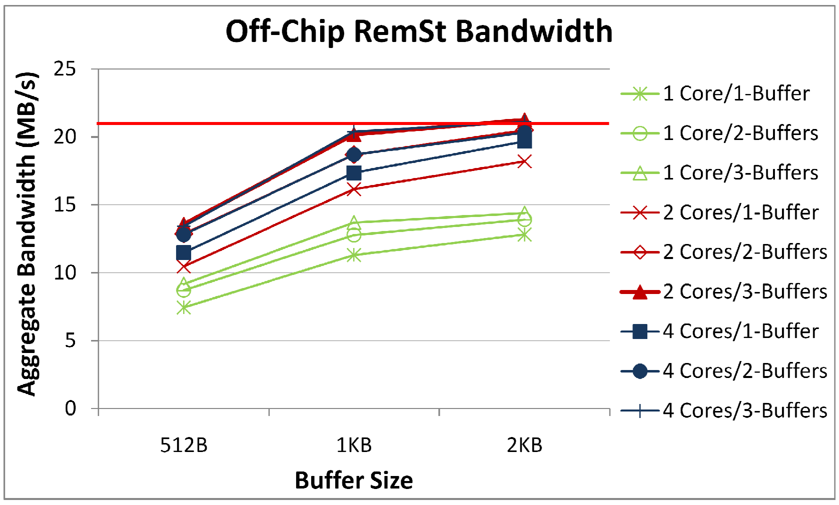


Figure 3.4: Performance of STREAM benchmark with off-chip remote stores

Finally, in figure 3.5 we discuss the aggregate bandwidth that the STREAM benchmark achieves for the off-chip transfers using remote stores. We plot the maximum feasible bandwidth of the system (horizontal line) for on-chip remote sores and the realizable bandwidth while we vary the buffer size, the number of participating processors and the number of the used buffers for overlapping computation with memory latency. We observe that, when we use 3-buffering, bandwidth saturates in all cases (4-buffering curves are overlapped with 3-buffering curves). As participating processors increase so the aggregate bandwidth does, and as buffer size increases we observe that bandwidths increases significant from 512B to 1KB but from 1KB to 2KB buffer size bandwidth has minor increase. The maximum aggregate bandwidth that the benchmark managed to achieve for on-chip transfers using remote stores is about 71MB/s.

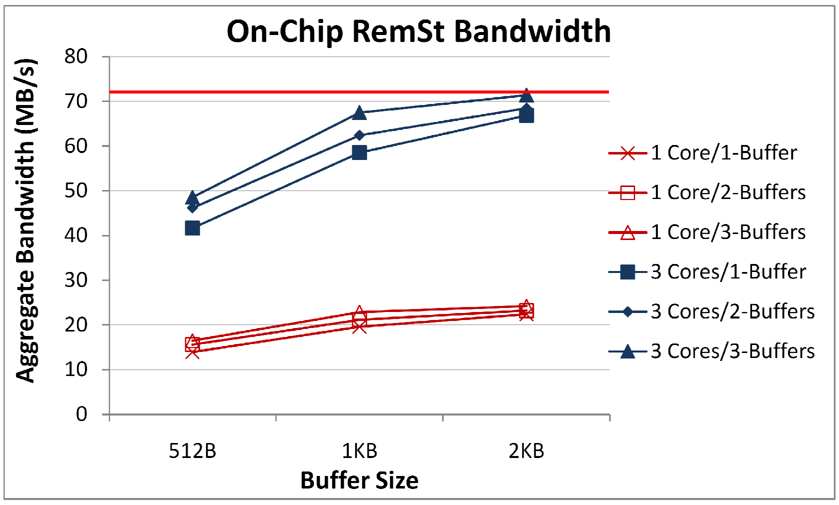


Figure 3.5: Performance of STREAM benchmark with on-chip remote stores

3.4 Observations

It is obvious that multiple buffering improves overall performance. As buffering increases so does the bandwidth. Three-buffering is the upper limit where bandwidth reaches its limits at the system in all the above cases. The maximum buffer size of 2KB seems to be marginally enough to fully stress the system's communication mechanisms in all of the cases.

On-chip communication achieves higher aggregate bandwidth compared to the off-chip transfers as expected. Aggregate bandwidth for on-chip transfers is limited to 3200MB/s and for off-chip to 180MB/s. DMAs offer a more effective way for data transfers for both on and off-chip communication compared to remote stores. However, this comparison is acceptable only for big transfer sizes. DMAs offer an aggregate bandwidth limit at 320MB/s for on-chip communication while remote stores offer only 800MB/s. Remote stores achieve lower performance than DMAs in big sizes because they have higher overhead to initiate and higher latency. Processors need a single instruction to initiate a remote store for 4 bytes. DMAs take 9 processor's cycles to initiate a transfer of any size. This means that remote stores have lower initiation time per transfer size for small sizes compared to the DMAs. However, DMAs have a fixed initiation time per transfer size and achieve better as they access the memory back to back.

We are not able to get the peek bandwidth of the system in all cases. When not all of the four processors participate we cannot fully utilize the system as many links stay idle and memories for some periods of time might be also idle. As buffering size increases we can overlap communication with computations which is necessary to fully utilize the system. We observe that when we use 3 or more buffers we can fully exploit the communication mechanisms in each case. Having fewer buffers than 3 occurs to not fully occupy the system. Moreover, buffer size effects the communication performance. Smaller buffers need more DMA initiations, extra code to check their completion and more control code which lead to higher communication overhead.

Chapter 4

Bitonic Sort

4.1 Benchmark Structure

Bitonic sort [5] is one of the fastest sorting networks. A sorting network is a special kind of sorting algorithm, where the sequence of comparisons is not data-dependent. This makes sorting networks suitable for implementation in hardware or in parallel processor arrays. The bitonic sort sorting network consists of $\Theta\left(n \cdot \log(n)^2\right)$ comparators. It has the same asymptotic complexity as odd-even mergesort and shellsort.

Bitonic sort is based on repeatedly merging two bitonic sequences to form a larger bitonic sequence. On a bitonic sequence we can apply the operation called bitonic split which halves the sequence in two bitonic sequences such that all the elements of one sequence are smaller than all the elements of the other sequence. Thus, given a bitonic sequence we can recursively obtain shorter bitonic sequences using bitonic splits, until we obtain sequences of size one at which point the input sequence is sorted. This procedure, of sorting a bitonic sequence using bitonic splits, is called bitonic merge and it is easy to implement on a network of comparators (known as bitonic merging network). By using this divide-and-conquer strategy bitonic sorting produces the desirable results.

First, a comparator network BitonicMerge is built which sorts a bitonic sequence. It produces two bitonic subsequences, where all elements of the first are smaller or equal than those of the second. Therefore, BitonicMerge can be built recursively as shown in Figure 4.1.

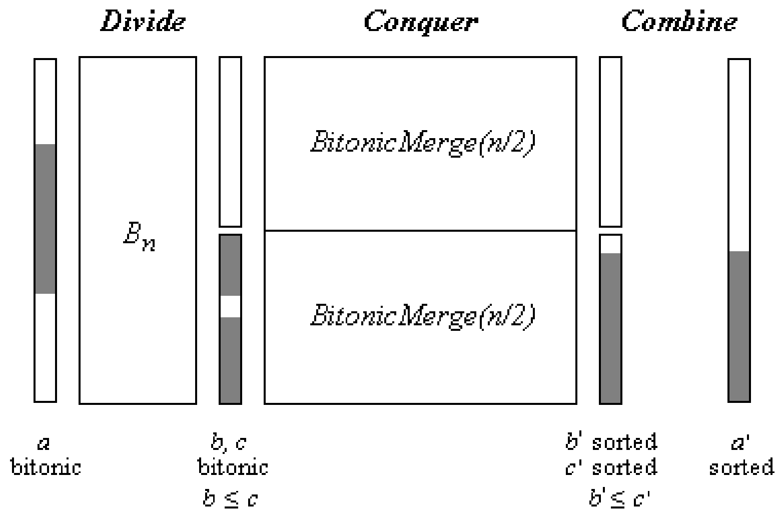


Figure 4.1: Bitonic Merge of size n

The bitonic sequence, necessary as input for BitonicMerge, is composed of two sorted subsequences, where the first is in ascending and the other in descending order. The subsequences themselves are sorted by recursive application of BitonicSort which is presented in figure 4.2.

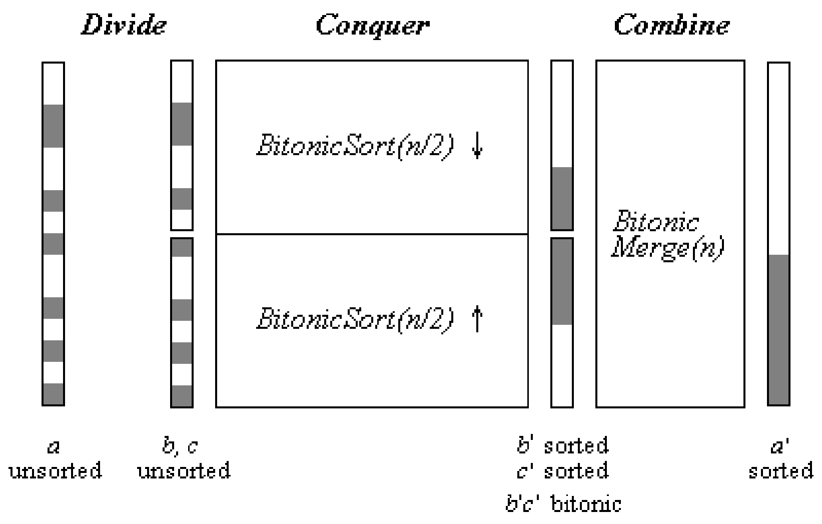


Figure 4.2: Bitonic Sort of size n

4.2 Application Analysis

The bitonic sort benchmark we use originates from the StreaMIT language benchmarks [6]. Bitonic-sort is computation bound and we use it to measure the minimum granularity of exploitable parallelism on the architecture. We configure the benchmark so that sorting and any associated data exchanges between processors perform entirely on-chip and we explore the trade-off between DMAs and remote stores in the implementation of the benchmark.

Bitonic sort was initially designed for parallel processors as a result we ported it to our prototype system without any significant changes to its basic algorithm. Each processor sorts a part of the initial array send that to another processor that merges the smaller arrays to a bigger and goes on.

Initially an array of size N (N should be a power of 2) separates into as many parts as the participating processors (assume 4 processors here). Each processor calls a recursive bitonic sort function with ascending or descending order if its PVR is even or odd accordingly. If its PVR is odd, it sends the results to the processors with PVR equal to its PVR-1 and exits. Processors with even PVR poll for incoming data from another processor. When processors receive the data they call the bitonic merge function in order to merge the data they sorted in the previous step and the data they received from the other processor just before. If only two processors are participating the execution stops here as the total array is sorted. Otherwise, Processor 0 calls again the bitonic sort function for the half of the initial data with ascending order and the other processor calls the bitonic sort function with descending order, sends the results to processor with PVR equals to 0 and exits. When processor with PVR equal to 0 receives the data it merges them using again the bitonic merge function and the result gives the total sorted array. If only one core is participating it is obvious that we use only a bitonic sort function in order to sort the complete array. In figure 4.3 we present the steps of the bitonic sort algorithm's phases.

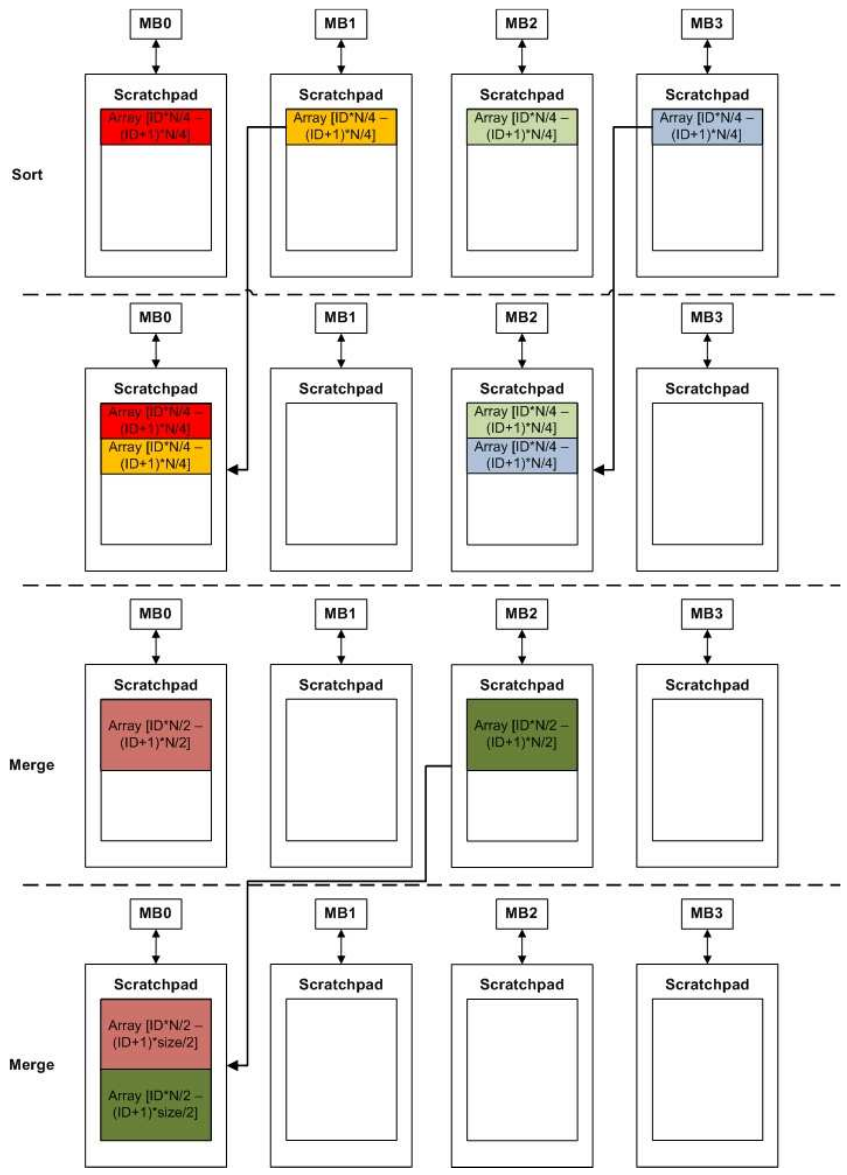

MB0
MB1
MB2
MB3
Scratchpad
Array [ID*N/4 – (ID+1)*N/4]
Sort
Merge
Array [ID*N/2 – (ID+1)*N/2]
Array [ID*N/2 – (ID+1)*size/2]

Figure 4.3: The bitonic sort algorithm flow chart

It is obvious that there is a lot of control code during the execution of the bitonic sort in our system. We simplify these code segments, in our application, in such a way that they do not affect the overall performance. Simple bitmask checks take the place of control code, unrolled loops replace recursive function calls and reuse of common code segments applied in order to achieve higher instruction locality.

4.3 Results

We can parameterize the application in order to run for one, two or four processors in parallel, to use remote stores or DMAs for data transfers and for various sizes of the input array. In this section we present the results from the application and discuss them. On each case we measure the total time, the computation and the communication time separately in order to observe the application's performance when using different communication mechanisms.

We compile the bitonic sort application with medium (-02) optimizations for gcc and with the flags "-funroll-loops -fmodulo-sched" in all cases. Moreover, we apply warm up in order to take advantage of spatial and temporal locality of the instruction and the data caches of the system.

In figure 4.4 we present the execution time of the bitonic sort application using DMAs for one, two and four processors and for 4 elements up to 4K elements. The scale is logarithmic on the vertical axis otherwise small sizes would be overlapped. As participating processors increase we can achieve speedup up to 2.7 for four processors compared to the execution time of a single processor.

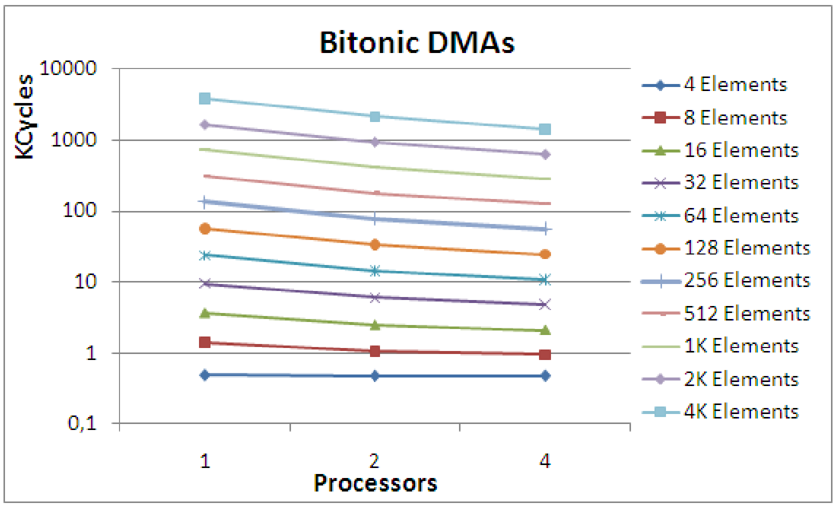


Figure 4.4: Bitonic Sort execution time with DMAs

In figure 4.5 we present the minimum granularity that we can achieve speedup with more than one core for this application. This figure is a more detailed execution time chart than the previous one. It contains only the four smallest problem sizes from 4 up to 32 elements with the common axis scale. We can view that we can achieve speedup even with 8 elements or more. The benchmark for 4 elements runs in less than 1400 cycles for one processor and in less than 700 for four processors. This shows that the prototype system can achieve speedup even for small programs which is essential for achieving high performance with multiple processors for all applications. This means that the granularity of the system that we can achieve speedup is acceptable even for programs with less than 700 cycles.

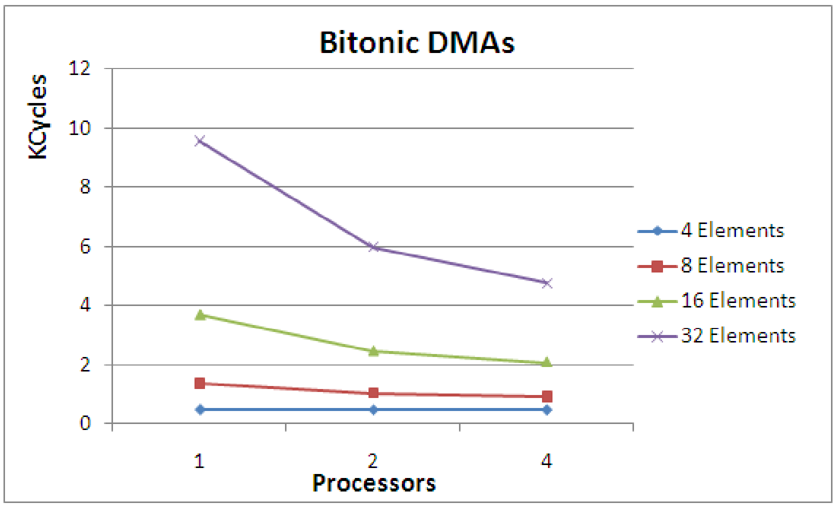


Figure 4.5: Bitonic Sort execution time with DMAs - Granularity

In figures 4.5 and 4.6 we present the execution time of the FFT application using remote stores and the minimum granularity that we can achieve speedup with more than one core for this application for 4 elements up to 4K elements. We can observe once more that we have gain when number of participating processor increases and the minimum granularity that we achieve speedup is from array sizes of 4 elements.

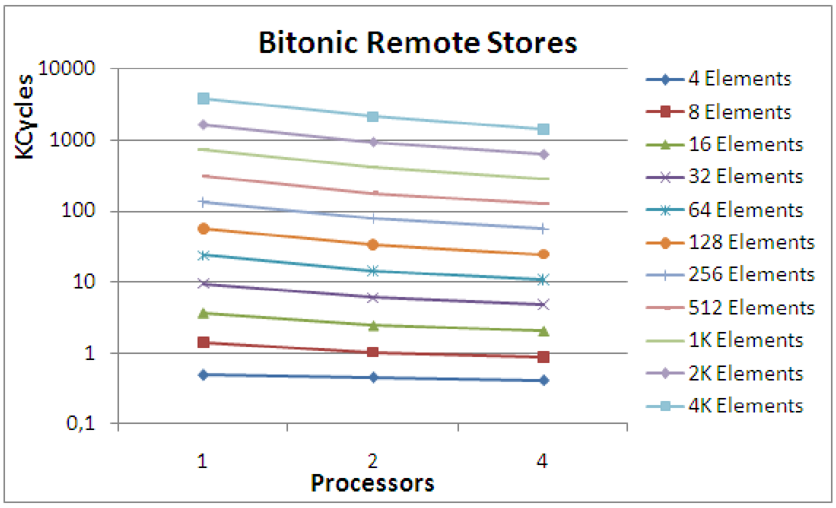


Figure 4.6: Bitonic Sort execution time with remote stores

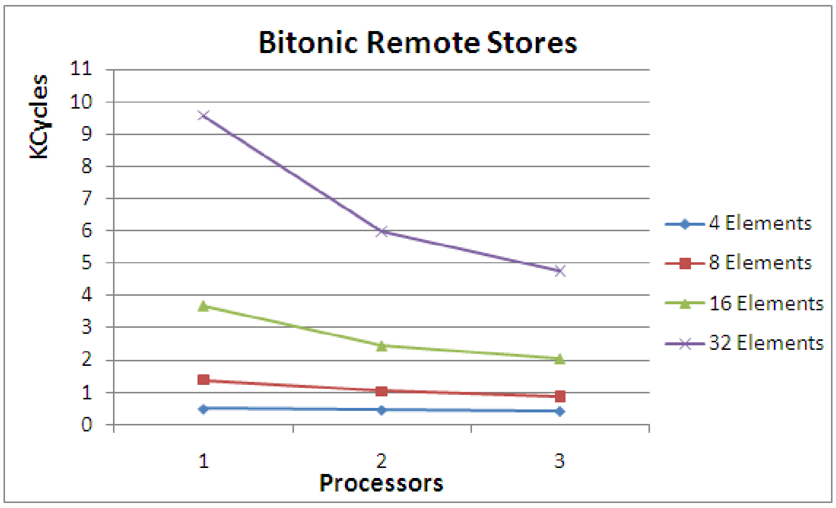
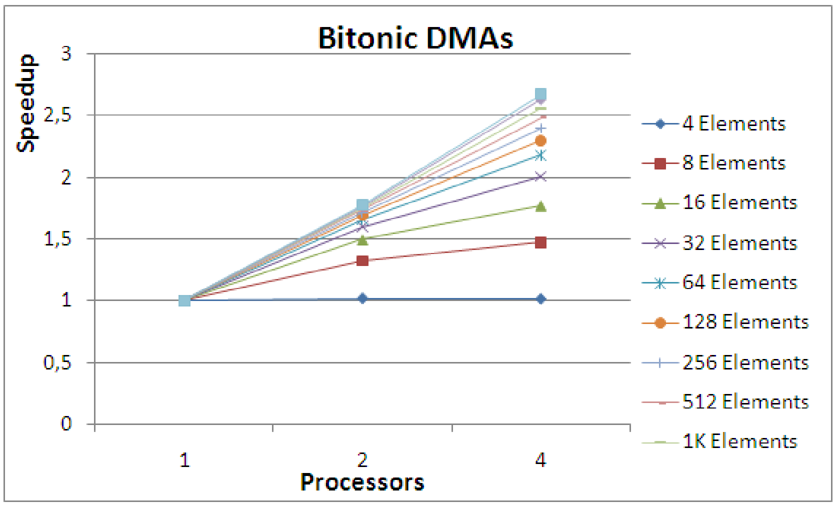


Figure 4.7: Bitonic Sort execution time with remote stores – Granularity

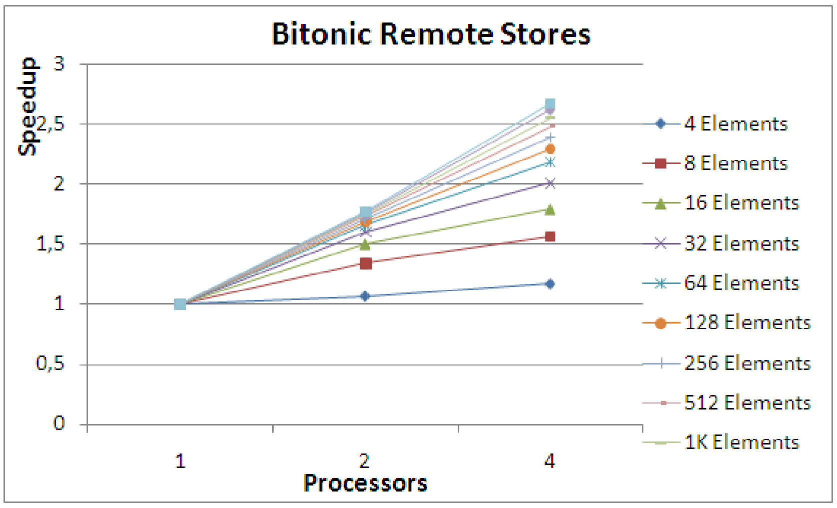
In order to measure the speedup that the application achieves, we measure the execution time of the applications compared to the execution time of the application running on a single processor. Once more, we measure this speedup for 2 and 4 processors, for various array sizes and for versions of the application using DMAs and remote stores for data transfers.

Figure 4.8 presents the speedup we can achieve when using DMAs. We observe that we have speedup even with array size of 8 elements and we achieve speedup up to 1.8 for two processors and up to 2.7 for four processors.



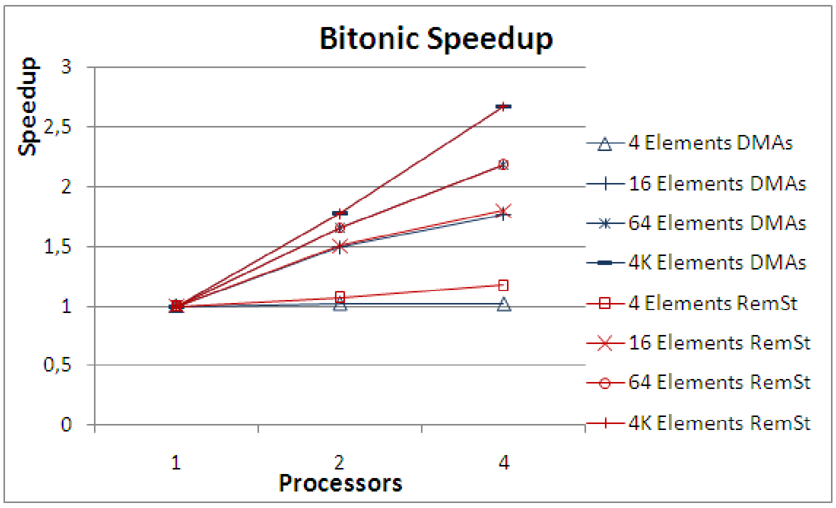
4.8: Bitonic Sort with DMAs speedup

Figure 4.9 showsthe speedup we achieve for the version of the application that uses remote stores for data transfers. As before, we have significant speedup from array size of 8 elements and we achieve speedup up to 1.8 for two processors and up to 2.7 for four processors.



4.9: Bitonic Sort with remote stores speedup

In order to compare the DMA and remote store version straightforward, we plot figure 4.10 where we present achieved speedup, when using DMAs and remote stores for communication. We sort arrays of 4, 16, 64 and 4K elements and present the difference between the speedup we achieve. We observe for small sizes that when exchanging data using remote stores we achieve more speedup that when using DMAs. This occurs as remote stores mechanism was designed especially for small data transfers while DMAs for bigger as we presented in chapter 2. However, hardware optimizes back to back remote stores and we have almost the same speedup in big array sizes for this application.



4.10: Bitonic Sort with DMAs & remote stores speedup

Computation and communication ratios for the smallest sizes are presented in figure 4.11. We normalize each bar to the execution time of the remote store version of the equal size. Computation time is the same between versions of remote stores and DMAs for equal array sizes, as expected. However, communication time when using DMAs is more than when using Remote stores for these sizes.

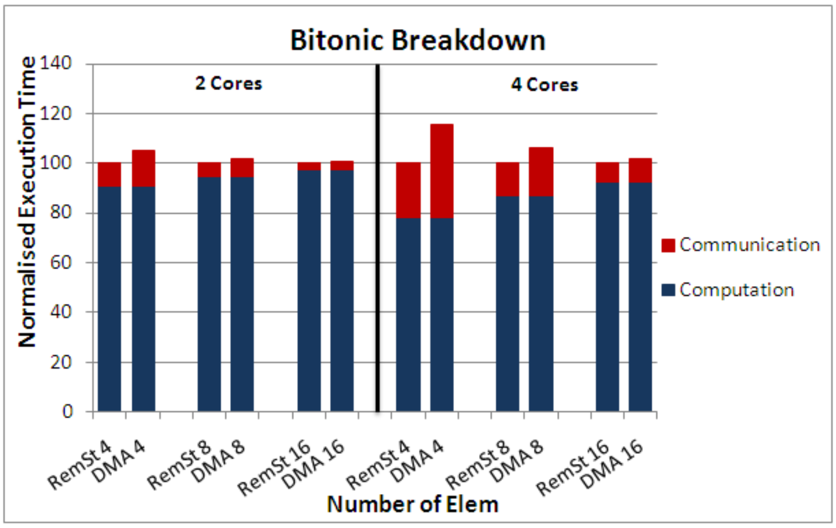


Figure 4.11: Breakdown of Bitonic Sort

4.4 Observations

Bitonic sort, a sorting network application, originally made for sorting in parallel array processors, proved a suitable application to exploit the explicit communication mechanisms of the system. It achieves good scalability and high speed up from just a few array elements and clock cycles. Remotes store communication mechanism is faster than DMAs in small sizes but in big sizes both of them are fast enough and able to achieve speedup up to 1.8 and 2.7 for two and four processors accordingly.

We observe that when using remote stores to exchange data between processors we achieve higher performance for arrays up to 64 elements. This is acceptable as remote stores achieve higher performance compared to DMAs for small data exchanges as we presented at Chapter 2 This application does not fully stresses the communication mechanisms of the prototype system as not all processors exchange data and processors must idle wait in some cases for data. As a result in bigger array sizes we observe that DMAs and remote stores perform the same for this application.

Chapter 5

FFT

5.1 Benchmark Structure

The Fast Fourier Transform (FFT) application is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse [7]. An FFT computes the DFT and produces exactly the same result as evaluating the DFT definition directly; the only difference is that an FFT is much faster.

We use a FFT benchmark that originates from the StreaMIT language benchmarks [6] and uses butterfly portions to calculate the results. In the context of fast Fourier transform algorithms, a butterfly is a portion of the computation that combines the results of smaller discrete Fourier transforms (DFTs) into a larger DFT, or vice versa (breaking a larger DFT up into sub transforms). The name "butterfly" comes from the shape of the data-flow diagram. Most commonly, the term "butterfly" appears in the context of the Cooley–Tukey FFT algorithm, which recursively breaks down a DFT of composite size n = rm into r smaller transforms of size m where r is the "radix" of the transform. These smaller DFTs are then combined with size-r butterflies, which themselves are DFTs of size r pre-multiplied by roots of unity.

5.2 Application Analysis

The FFT benchmark we use originates from the StreaMIT language benchmarks [6] and includes all to all data exchange patterns between processors. The benchmark is configured so that it performs the entire computation and all-to-all data exchanges on-chip, in order to stress the performance of the cache-integrated NI mechanisms that the system provides. We implement data exchanges using DMAs and remote stores to explore trade-offs between the two communication mechanisms.

Each processor undertakes a part of the total signal at each repetition and calculates its results. If we assume that CPU_NUM processors participate in the FFT computations, and there are k signals at a specific point of time, each processor will calculate k/CPU_NUM signals. These signals are divided by each processor till they become signals of single point. Then each processor makes the basic transformation and composes the transformation. On each step all processors exchange all the data they calculate to all other processor, as these processors will need these at the next steps. We present the procedure using pseudo-code that each processor executes in figure 5.1.

```
for each FFT stage
if there are enough groups for all processors
{
        for each group of butterfly
                for each butterfly in the group
                        compute the butterfly
        send the results to the other processors
        wait for the results from the other processors
}
else
{
        for each group of butterfly
        {
                split group to create groups for all processors
                for each butterfly in the group
                        compute the butterfly
        }
        send the results to the other processors
        wait for the results from the other processors
}
```

Figure 5.1: The FFT algorithm

At each iteration of butterfly group, each processor sends the whole group to the rest of the processors. If that group is big enough, processor splits it into two smaller groups and sends it to the other processors. This mechanism overlaps computations with communication as the second part of the group is able to arrive while the processor is calculating elements of the first part. Moreover, processors do not need all of the data at each step of the algorithm. So, each time, the algorithm checks what data each processor needs, in order to send only them. Finally, when the existing signals become less than the available processors, each processor undertakes a part of the same signal with another processor in order not to idle waiting till there are enough signals for every processor.

5.3 Results

We can easily parameterize the application in order to run for one, two or four processors in parallel, to use remote stores or DMAs for data transfers and for various sizes of the input array. In this section we present the results from the application and discuss them. On each case we measure the total time, the computation and the communication time separately in order to observe the application's performance when using different communication mechanisms.

We compile the bitonic sort application with medium (-02) optimizations for gcc and with the flags "-funroll-loops -fmodulo-sched" in all cases. Moreover, we apply warm up in order to take advantage of spatial and temporal locality of instruction and data caches of the system.

In figure 5.2 we present the execution time of the FFT application using DMAs for one, two and four processors and for 4 elements up to 4K elements. The scale is logarithmic on the vertical axis otherwise small sizes would be overlapped. As participating processors increase we can view that we achieve speedup but only in big array sizes. The reason that we do not achieve speedup for small sizes is explained later on this section.

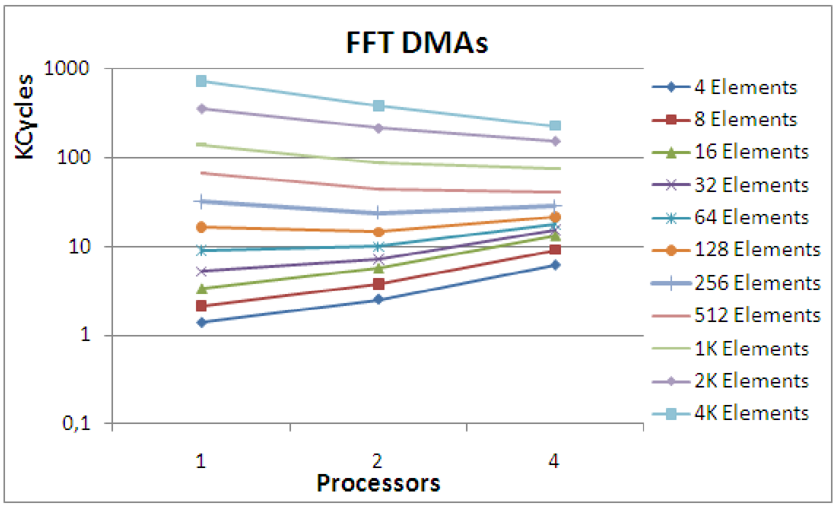


Figure 5.2: FFT execution time with DMAs

In figure 5.3 we present the minimum granularity that we can achieve speedup with more than one core for this application. This figure is a more detailed execution time chart than the previous one. It contains only the eight smallest problem sizes from 4 up to 512 elements with the common axis scale. We can view that we can achieve better performance by adding more processors with medium array sizes.

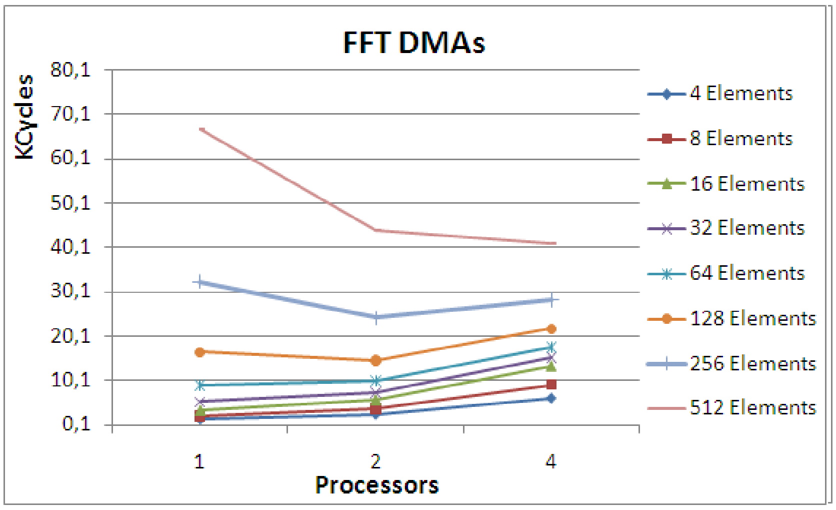


Figure 5.3: FFT execution time with DMAs – Granularity

In figures 5.4 and 5.5 we present the execution time of the FFT application using remote stores and the minimum granularity that we can achieve speedup with more than one core for this application for 4 elements up to 512 elements. We can observe this time that we have gain when amount of participating processor increases and the minimum granularity that we achieve speedup is from array sizes of 128 elements or more.

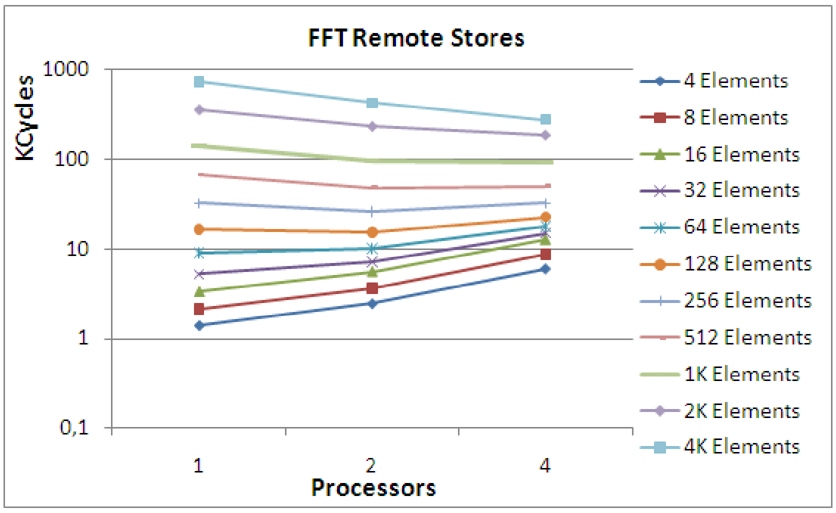


Figure 5.4: FFT execution time with remote stores

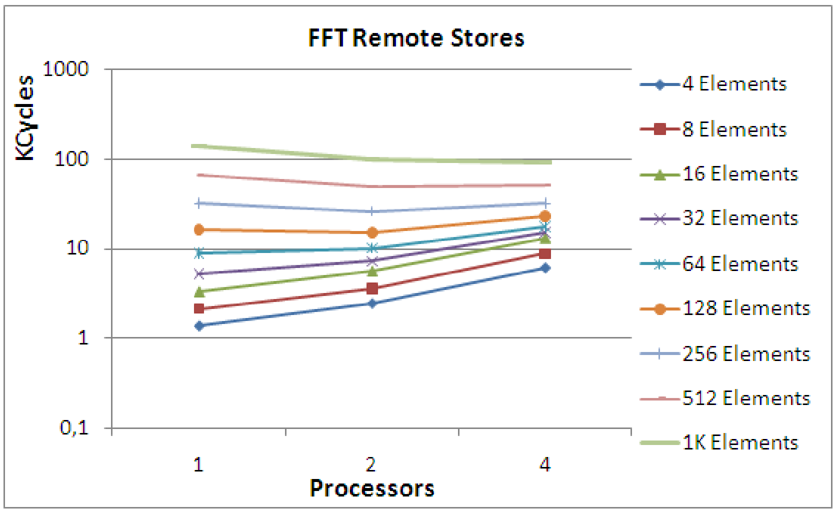


Figure 5.5: FFT execution time with remote stores – Granularity

In order to measure the speedup that the application achieves, we measure the execution time of the application compared to the execution time of the application running on a single processor. Once more, we measure this speedup for 2 and 4 processors, for various array sizes and for versions of the application using DMAs and remote stores for data transfers.

In Figure 5.6 we show that we can achieve speedup when using DMAs from array sizes of 128 elements for two processors and for array sizes of 256 elements for four processors using DMAs. Moreover, we achieve maximum speedup up to 1.9 for two processors and up to 3.2 for four processors.

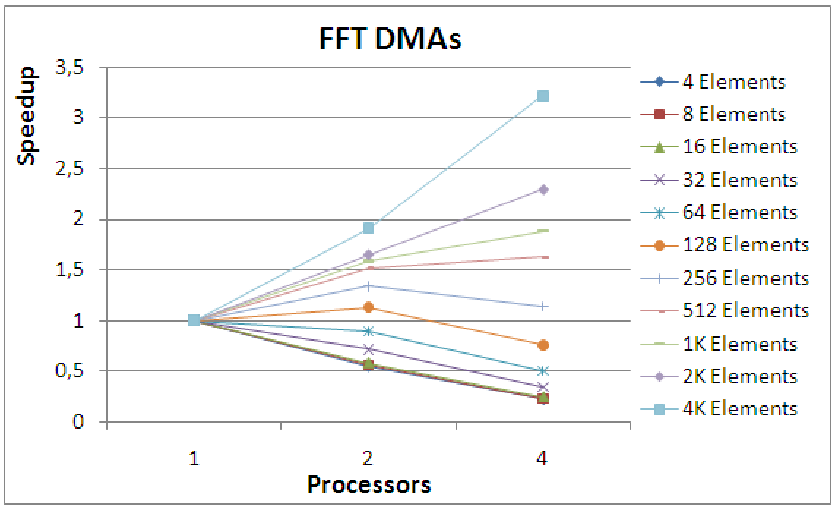


Figure 5.6: FFT with DMAs speedup

As for the version of the application that uses remote stores for data transfers, we can view the speedup we achieve in the figure 5.7. We observe that we have speedup from array of size 128 elements for two processors and for size of 512 elements for four processors using remote stores. Moreover, we achieve maximum speedup up to up to 1.7 for two processors and up to 2.7 for four processors.

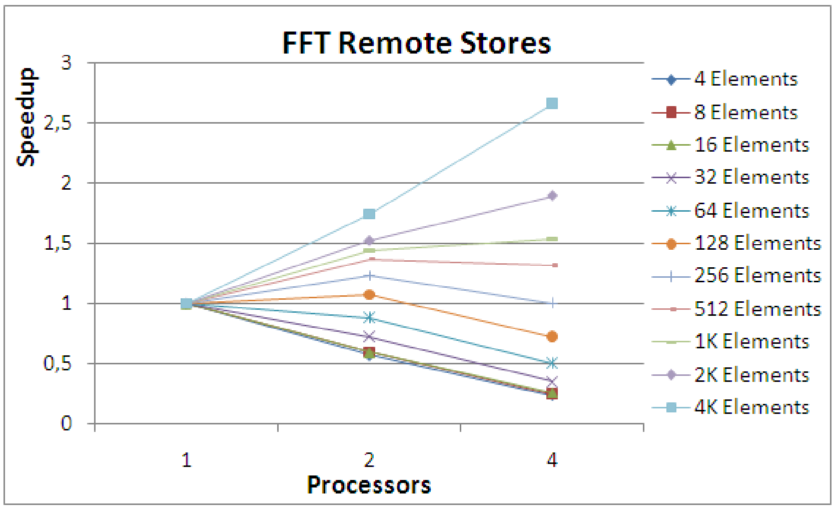


Figure 5.7: FFT with remote stores speedup

In order to compare the DMA and remote store version straightforward, we present figure 5.8 where we plot the speedup we achieve, when using DMAs and remote stores for communication. We use 4, 256, 1K and 4K elements of array size to present the difference between the speedup we achieve. We observe that as array sizes become grater, so the DMAs versions achieve higher speedup than the remote stores versions. As array sizes become bigger so the transfer sizes do. As a result DMAs, that are designed to efficiently transfer big sizes, they perform faster transfers than remote stores for this application which is communication intensive.

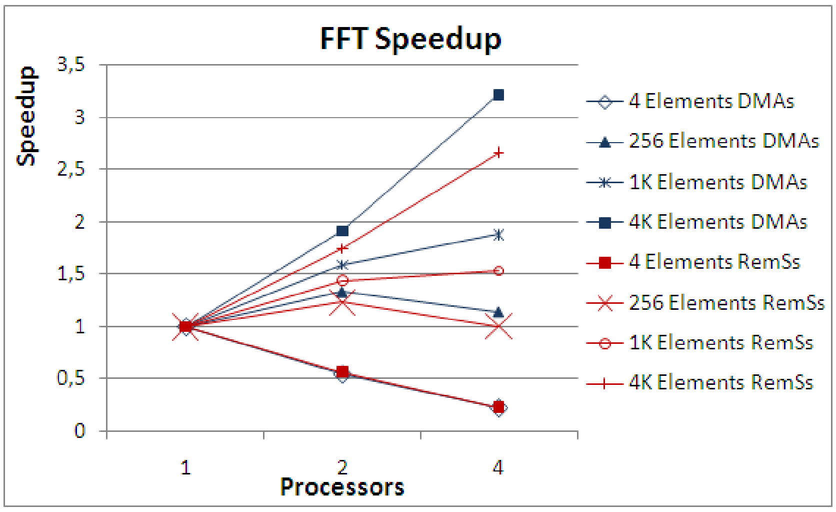


Figure 5.8: FFT with DMAs & remote stores speedup

We present computation and communication ratios for the smallest sizes in figure 5.9. Each bar is normalized to the execution time of the remote store version of the equal size. Computation time is the same between versions of remote stores and DMAs for equal sizes as expected. However, communication time when using DMAs is more than when using Remote stores for these sizes.

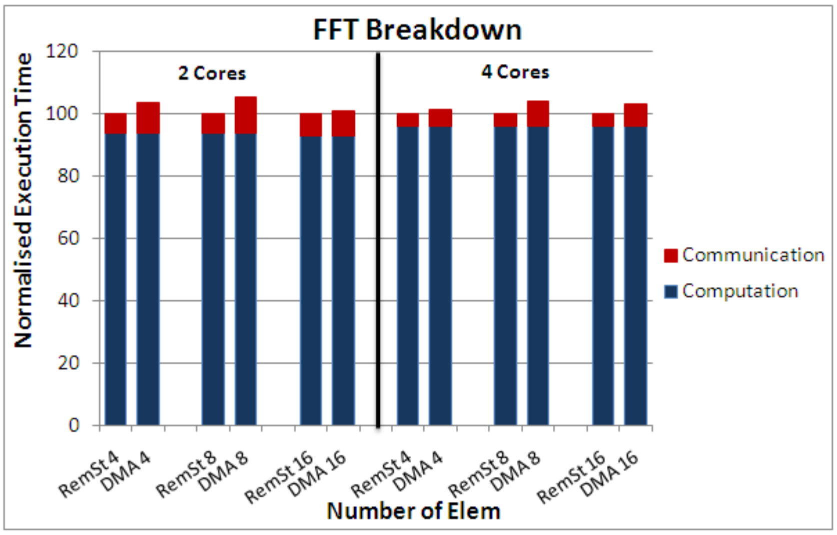


Figure 5.9: Breakdown of FFT – Small sizes

For array sizes of 4, 32 and 256 elements breakdown seems different. As array sizes grow DMAs achieve better performance than remote stores which are faster for small data transfers. In figure 5.10 we can see that for 4 elements remote stores need less time than DMAs. However in 256 elements DMAs are pretty faster than remote stores and that contributes to achieve better performance.

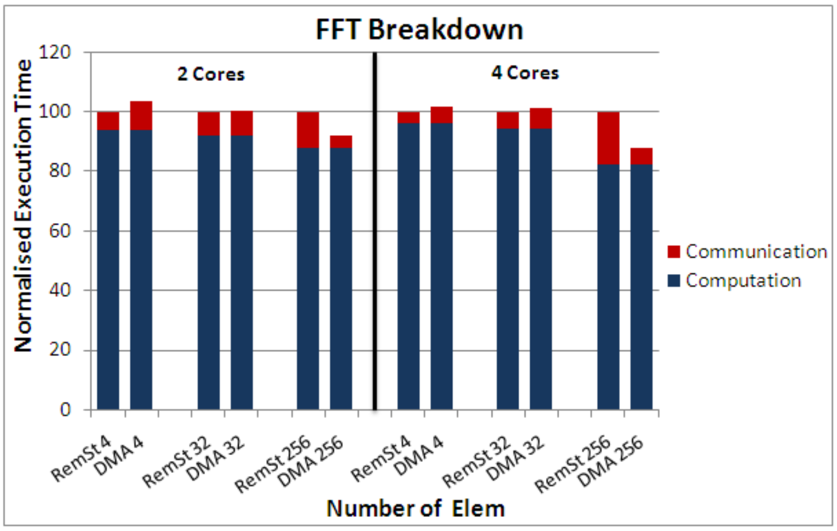


Figure 5.10: Breakdown of FFT – Various sizes

5.4 Observations

FFT algorithm, an efficient algorithm to compute the discrete Fourier transform gives us important information about the system, the available communication mechanisms and the application itself. FFT application achieves good scalability and low execution times for more than 256 array elements. Remote stores communication mechanism performs faster than DMAs in small sizes but in big sizes DMAs achieve higher performance due to the communication demands of the application. Both of them are fast enough and achieve speedup up to 1.9 and 3.2 for two and four processors accordingly.

We observe that when using remote stores to exchange data between processors we achieve higher performance for arrays up to 64 elements. This application stresses the communication mechanisms of the prototype system as all processors exchange data with all other processors at each step of the execution. As a result in bigger array sizes we observe that when using DMAs application achieves higher performance compared to the version with the remote sores. This is acceptable as DMAs perform better compared to remote stores when we transfer big data segments.

Chapter 6

Map-Reduce

6.1 Benchmark Structure

MapReduce is a software framework introduced by Google to support distributed computing on large data sets on clusters of computers [8]. The framework is inspired by map and reduce functions applied to data sets. MapReduce libraries have been written for many programming languages such as C++, C#, Erlang, Java, Python, Ruby, F#, R and many others.

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible through this model such as word count, histogram production, k-means clustering algorithm, distributed sort, linear regression and many other.

At the Map step the master node takes the input, chops it up into smaller sub-problems, and distributes those to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes that smaller problem, and passes the answer back to its master node. During the Reduce step the master node then takes the answers to all the sub-problems and combines them in a way to get the output - the answer to the problem it was originally trying to solve.

The advantage of MapReduce is that it allows for distributed processing of the map and reduction operations. Provided each mapping operation is independent of the other, all maps can be performed in parallel - though in practice it is limited by the data source and/or the number of CPUs near that data. Similarly, a set of 'reducers' can perform the reduction phase - all that is required is that all outputs of the map operation which share the same key are presented to the same reducer, at the same time. While this process can often appear inefficient compared to algorithms that are more sequential, MapReduce can be applied to significantly larger datasets than that which "commodity" servers can handle. The parallelism also offers some possibility of recovering from partial failure of servers or storage during the operation: if one mapper or reducer fails, the work can be rescheduled - assuming the input data is still available.

MapReduce was initially proposed by Google for large scale data processing in a distributed computing environment [8] and the model has recently been ported to shared memory multiprocessor systems [10] and to the Cell broadband engine architecture [9].

For our purposes we implement three different applications based on the MapReduce programming model. Each one is based on the same model but differs on application specific details. Firstly, we present the base MapReduce application model for the prototype system and then we present each application separately with its own details and any necessary additions. The applications we develop and present in the next sections are a MapReduce word count application, a MapReduce histogram application and a MapReduce k-means clustering algorithm.

6.2 Map-Reduce programming Model

For our system purposes we modify MapReduce programming framework in order to work efficiently on the specific system. Scratchpad memories are small, so local data do not fit. Moreover, we implement algorithm phases that are not efficient, or able to run to multiple processors, to run by a single core. There is no coherence between scratchpads or caches and the global memory so for every access all processors must be aware and much more that we present at the next sections.

We present the dataflow of MapReduce framework for our architecture in figure 6.1. At the dataflow we assume that all four processors participate at the overall procedure. Moreover, all orange process boxes are executed by all processors in parallel while single core processes are in blue boxes. An initial array of keys is stored at the global memory, and each processor undertakes an equal portion of that array. Processor copies this portion to its local scratchpad memory and then it maps, it sorts the keys and copies them back to the global memory. During this phase processors use double buffering at the local scratchpads and DMAs to exchange data between local scratchpad and the global memory. Buffers have maximum size of 16KB each which is the same with the DMA’s transfer size. When every processor finishes, these sorted arrays must be combined to one big sorted array in order to efficiently apply the reduction. This is done by a single processor which merges all the sorted arrays to one totally sorted array. The processor brings to its local scratchpad memory portions of all the sorted arrays and merges them gradually by checking if any of the buffers gets empty. During this phase processors use double buffering with 16KB buffer size and DMAs to exchange data between local scratchpad and the global memory. The reduction phase initiates after this phase. Each processor undertakes once more a portion of the sorted array, and applies the reduction phase. During this phase each processor combines all the same keys to a single pair with the key and a value showing the amount of time that the key has appeared. Due to the sorted keys in the array, reduction has to check only the portions of the array after the first unique key till it finds a new one key. This method avoids checking for the same keys all over the array. At the end, there will be four separate reduced arrays. However, there must be only one reduced array. This total reduction is done by only one processor. This processor checks the borders between the four arrays and reduces the keys. The processor brings to its local scratchpad memory portions of the borders of each array using DMAs and compares the keys of the borders. If they are same it applies a reduce phase for only that key and copies the pairs key-value back to the global memory. At this time the procedure finishes and the final array is consisted of sorted key-value pairs.

We use a system that integrates small scratchpad memories, compared to the original MapReduce array. As a consequence, even all four processors participate in the overall process the part that each processor should undertake cannot fully fill in the local scratchpad memories. So, at each phase of the algorithm that a processor must store a quarter, or a half or even the whole original array, it gets just a part that fills at its own local scratchpad, copy it back, get a next one and so one. For performance purposes each part should be half of the size of the processor's scratchpad in order to apply double buffering of these parts and overlap communication with computation to achieve higher performance.

We compile each MapReduce application with medium (-02) optimizations for gcc and with the flags "-funroll-loops -fmodulo-sched" in all cases. Moreover, we apply warm up in all cases in order to take advantage of spatial and temporal locality of instruction and data caches of the system.

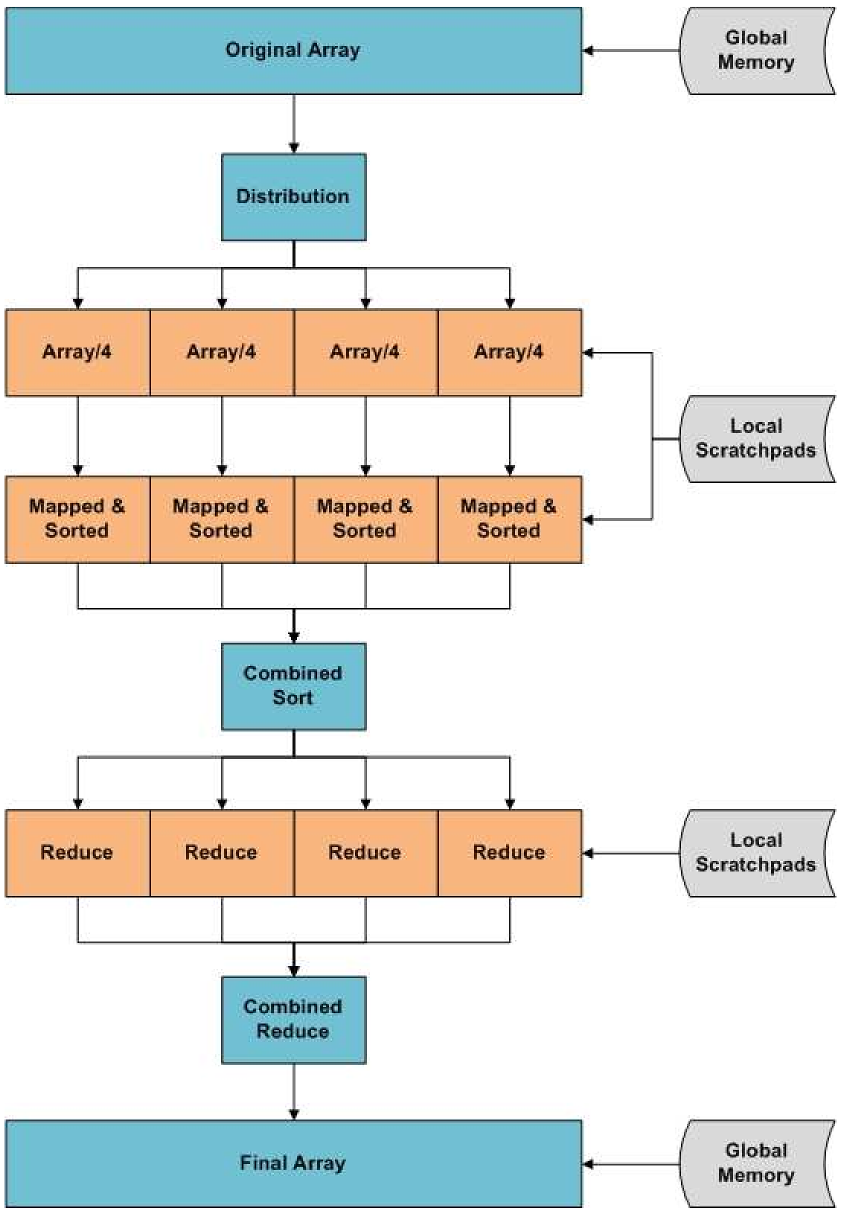


Figure 6.1: MapReduce Data Flow

6.3 Map-Reduce Word Count

6.3.1 Application Analysis

A word count application counts the frequency of occurrence of each word in a given text file. It is reasonable that processors do not easily collaborate with words. Rather than this, it is preferable to use numbers. For this reason processors transform the given text file of this application to a file with numbers where each number indicates a different word. This is easily done by a hash function with which, each number represents a unique word and vice versa.

The overall process starts by converting a text file containing numbers using an appropriate hash function. Then processors apply the main MapReduce procedure in this data as presented in the previous section in order to count the words. The results of this procedure is a vector containing pairs of keys (numbers that represent words) and values (frequency of the appearance of the specific key). Using the reverse hash function that used at the first step, processors are able to convert the results to the initial words.

6.3.2 Results

We can parameterize the application in order to run for one, two or four processors in parallel and for various sizes of the input array. In this section we present the results from the application and discuss them. On each case we measure the total time and the time that each phase of the algorithm takes in order to observe the application's behavior for different problems.

In figure 6.2 we present the execution time of the MapReduce word count application for one, two and four processors and for vector sizes from 4K elements up to 256K elements. The scale is logarithmic on the vertical axis otherwise small sizes would be overlapped. As participating processors increase we can view significant gain on the execution time in all cases, while as array size decreases we observe proportional performance gain.

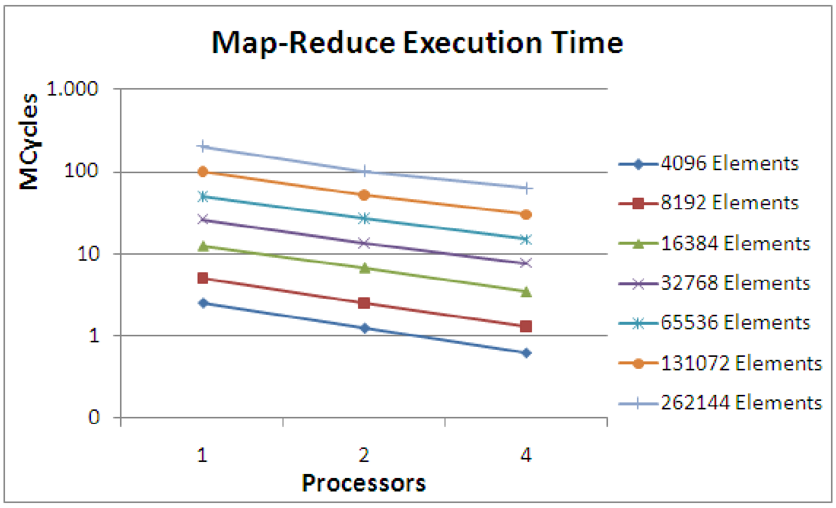


Figure 6.2: MapReduce word count execution time

In order to observe the speedup that we can achieve for the application as participating processors increase and the input array size increases we plot figure 6.3. In this figure we mark that we can achieve speedup almost up to 2 and up to 4 for two and for four processors accordingly. This means that the specific application achieves very high scalability for our architecture. We observe that as array size increases we get less speedup compared to smaller arrays sizes speedup. The reason for this is that during combine phase a single core has to calculate the size of the mapped array that each one of the other cores has to reduce. This means that a single core has to run through a bigger array each time and calculate the amount the distinct keys that exist in the array in order to assign the same amount of keys to each processor for reduction and allocate the appropriate space. This phase of the algorithm is not necessary when only one core participates as this core will do the reduction of all the keys.

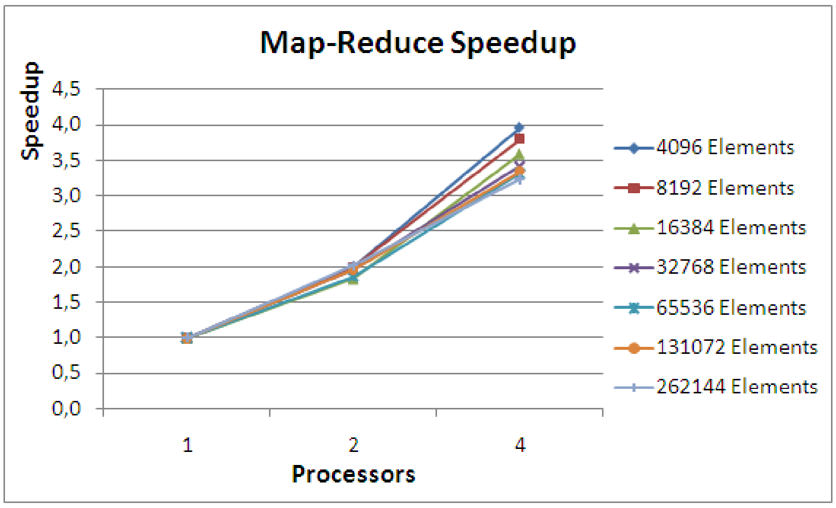


Figure 6.3: MapReduce word count speedup

In figures 6.4 and 6.5 we present two breakdowns of the application, indicating where the overall execution time is spent. The first one contains the results for the three smallest array sizes and the second one for the rest of them. We observe at both that as the input array size increases so does every phase of the algorithm. It is clear that when the array size doubles, the overall time of each phase duplicates, but when more processors participate in the procedure, the time becomes the half. These are the reasons that the specific application achieves a high scalability at the system. Moreover, we observe that the word count application on our system takes more time to complete the map phase and the combine phase, compared to the reduce phase, with the combine phase being the dominant one.

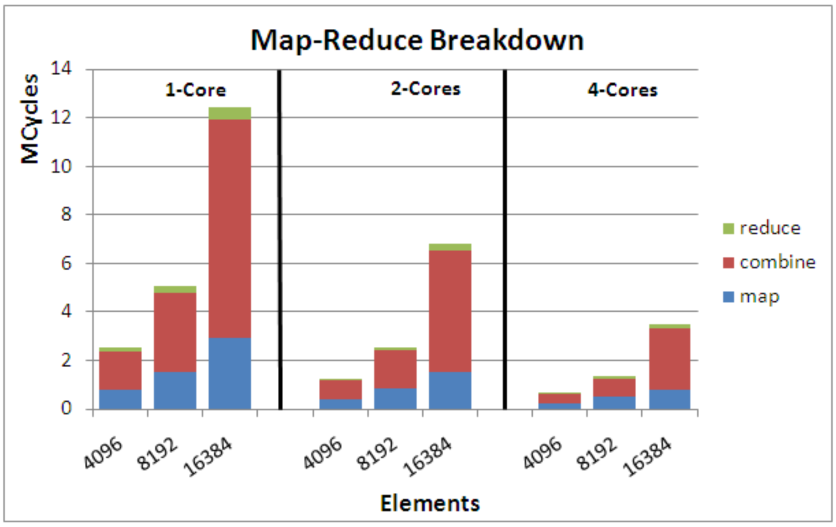


Figure 6.4: MapReduce word count breakdown – Small sizes

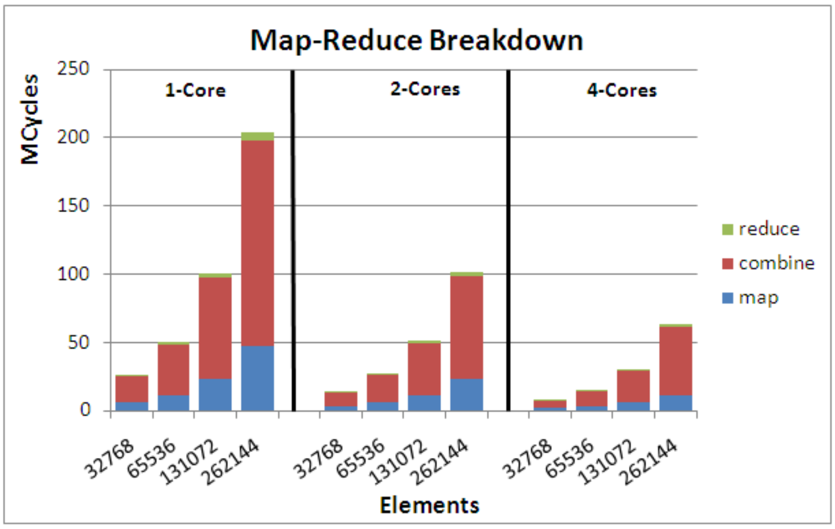


Figure 6.5: MapReduce word count breakdown – Big sizes

6.4 Map-Reduce Histogram

6.4.1 Application Analysis

The MapReduce histogram application counts the frequency of occurrence of each color component for a given bitmap image file. Map counts the occurrences of each color component and Reduce gathers the intermediate sums to produce a final sum for each component. The array that this procedure results contains sorted pairs of keys and values which indicate the time of appearance of each value in the original array. This is the same representation as the representation of a histogram. As a result this application is one of the few that can run as it is by the MapReduce base algorithm. This means that the input array does not need any processing, before we apply the MapReduce and the results are ready for use immediately after the base MapReduce algorithms finishes.

6.4.2 Results

We can parameterize the application in order to run for one, two or four processors in parallel and for various sizes of the input array. In this section we present the results from the application and analyze them. On each case we measure the total time and the time that each phase of the algorithm takes in order to observe the application's behavior for different problems.

In figure 6.6 we present the execution time of the MapReduce word count application for one, two and four processors and for vector sizes from 4K elements up to 256K elements. The scale is logarithmic on the vertical axis otherwise small sizes would be overlapped. As participating processors increase we can view significant gain on the execution time in all cases, while as array size decreases we observe proportional performance gain.

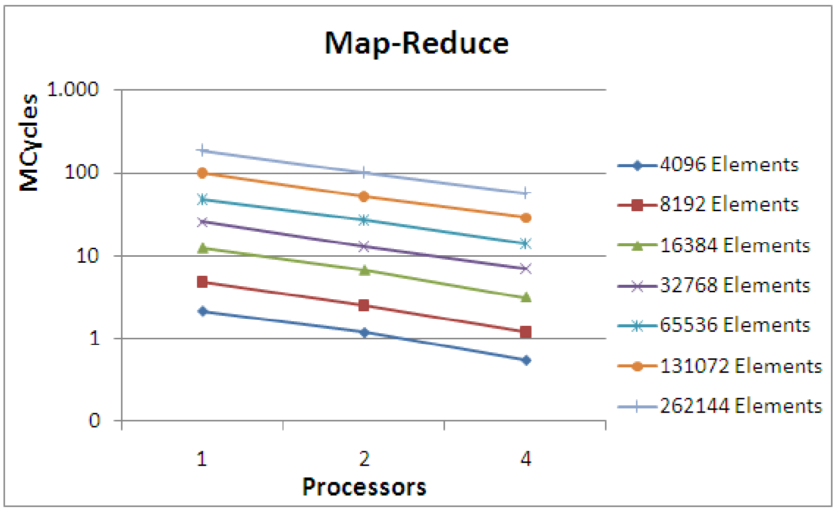


Figure 6.6: MapReduce histogram execution time

In order to observe the speedup we achieve for the application as participating processors increase and the input array size increases we plot figure 6.7. In this figure we observe that we achieve speedup almost up to 2 and up to 4 for two and for four processors. This means that the specific application achieves very high scalability for our architecture. We observe that as array size increases we get less speedup compared to smaller arrays sizes speedup. The reason for this is that during combine phase a single core has to calculate the size of the mapped array that each one of the other cores has to reduce.

This means that a single core has to run through a bigger array each time and calculate the amount the distinct keys that exist in the array in order to assign the same amount of keys to each processor for reduction and allocate the appropriate space. This phase of the algorithm is not necessary when only one core participates as this core will do the reduction of all the keys.

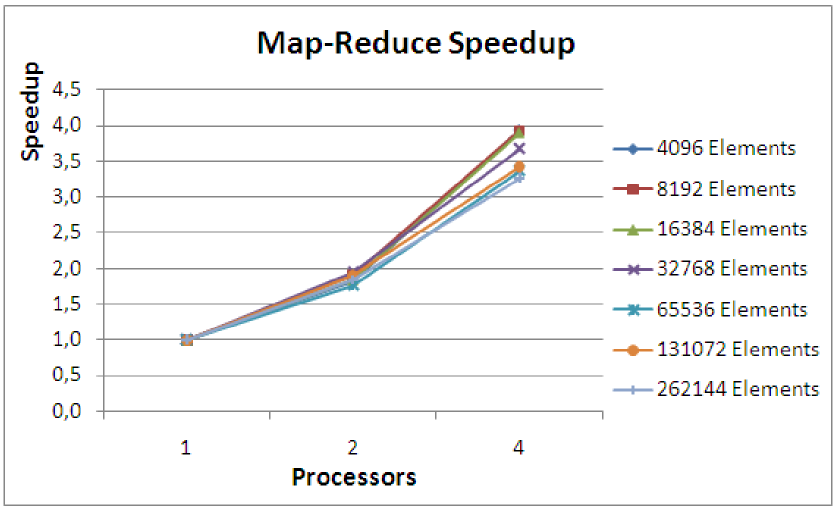


Figure 6.7: MapReduce histogram speedup

In figures 6.8 and 6.9 we present two breakdowns of the application indicating where the overall execution time is spent. The first one contains the results for the three smallest array sizes and the second one for the rest of them. We observe in both of them that as the input array size increases so does every phase of the algorithm. It is clear that when the array size doubles, the overall time of each phase duplicates, but when more processors participate in the procedure, the time becomes the half. These are once more the reasons that the specific application achieves a high scalability at the system. Moreover, we observe that the histogram application takes more time to complete the map phase and the combine phase, compared to the reduce phase, with the combine phase being the dominant one as before.

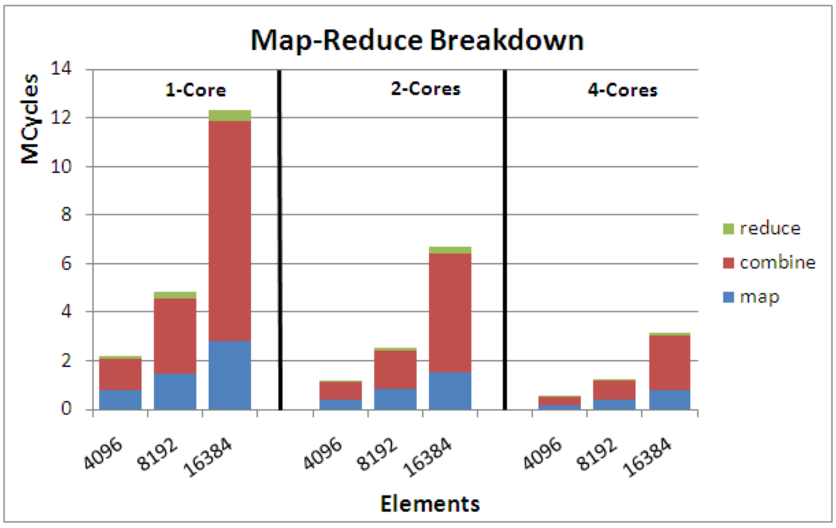


Figure 6.8: MapReduce histogram breakdown – Small sizes

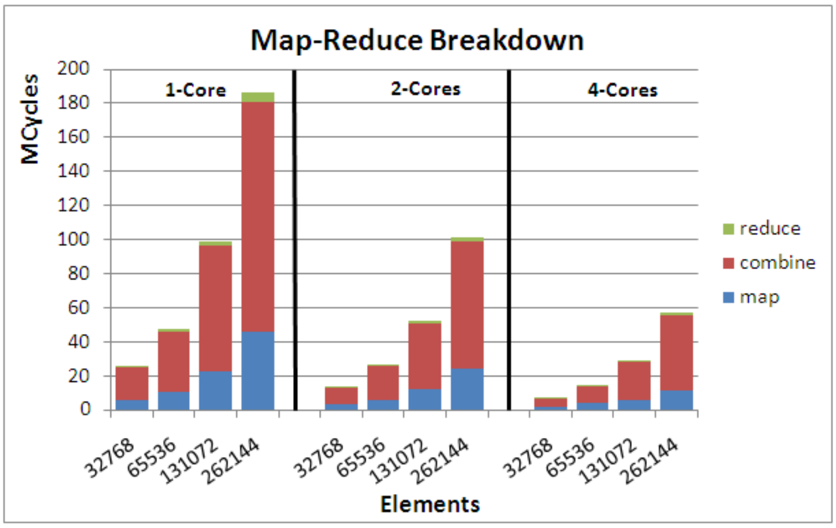


Figure 6.9: MapReduce histogram breakdown – Big sizes

6.5 Map-Reduce k-means

6.5.1 Application Analysis

MapReduce k-means application clusters a set of data points. Map takes as input a point, finds the distance between the point and each cluster, and assigns the point to the closest cluster. Reduce computes the new cluster means by averaging the coordinates of all points assigned to the given cluster. The algorithm iterates until it converges.

The algorithm is as follows. Given a data set where all the data are numeric, the algorithm for k-means clustering starts with k cluster centers (chosen randomly or according to some specific procedure), assigns each data to its nearest cluster center re-calculates the cluster centers as the "average" of the data of each cluster. This procedure is repeated until some criteria are met.

This repetition is sensitive to the criteria that must be met for the algorithm to stop, the initial centers of the selected clusters and the data set. For these reasons we take measurements for one complete repetition and for 4 initial cluster centers.

6.5.2 Results

We can parameterize the application in order to run for one, two or four processors in parallel and for various sizes of the input array. In this section we present the results from the application and discuss them. On each case we measure the total time and the time that each phase of the algorithm takes in order to observe the application's behavior for different problems.

In figure 6.10 we present the execution time of the MapReduce k-means application for one, two and four processors and for vector sizes from 4K elements up to 256K elements. The scale is logarithmic on the vertical axis otherwise small sizes would be overlapped. As participating processors increase we can view significant gain on the execution time in all cases, while as array size decreases we observe proportional performance gain.

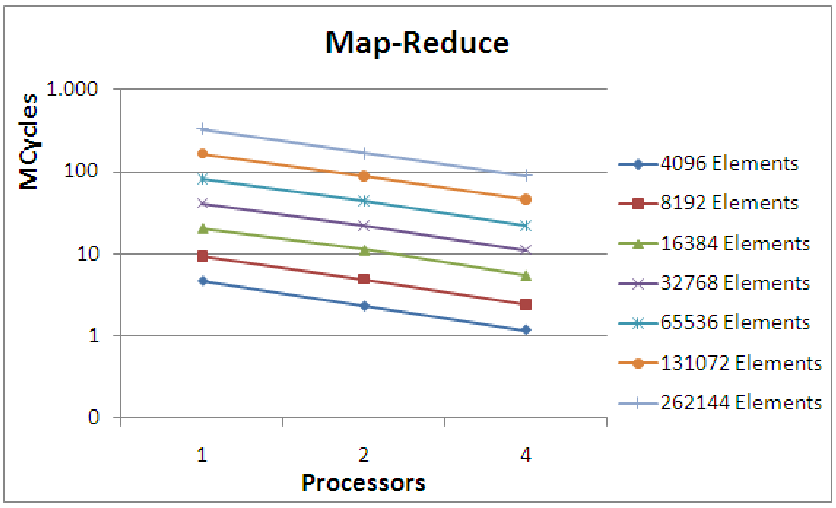


Figure 6.10: MapReduce k-means execution time

In order to observe the speedup we can achieve for the application as participating processors increase and the input array size increases we plot figure 6.11. In this figure we can mark that we can achieve speedup almost up to 2 and up to 4 for two and for four processors. This means that the specific application achieves high scalability for our architecture. We observe that as array size increases we get less speedup compared to smaller arrays sizes speedup. The reason for this is that during combine phase a single core has to calculate the size of the mapped array that each one of the other cores has to reduce. This means that a single core has to run through a bigger array each time and calculate the amount the distinct keys that exist in the array in order to assign the same amount of keys to each processor for reduction and allocate the appropriate space. This phase of the algorithm is not necessary when only one core participates as this core will do the reduction of all the keys.

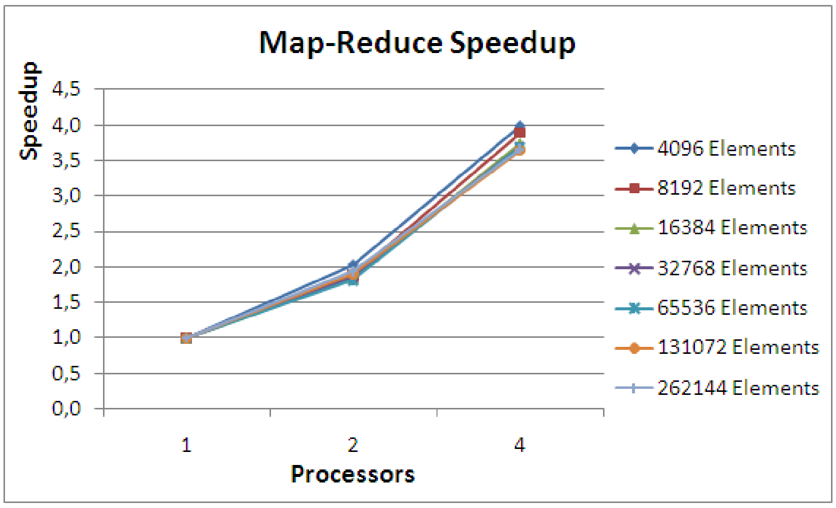


Figure 6.11: MapReduce k-means speedup

In figures 6.12 and 6.13 we present two breakdowns of the application indicating where the overall execution time is spent. The first contains the results for the three smallest array sizes and the second one for the rest of them. We observe in both of them that as the input array size increases so does every phase of the algorithm. It is clear that when the array size doubles, the overall time of each phase duplicates, but when more processors participate in the procedure, the time becomes the half. These are the reasons that the specific application achieves a high scalability at the system. Moreover, we observe that the k-means application takes more time to complete the map phase and the combine phase, compared to the reduce phase, with the map phase being the dominant one.

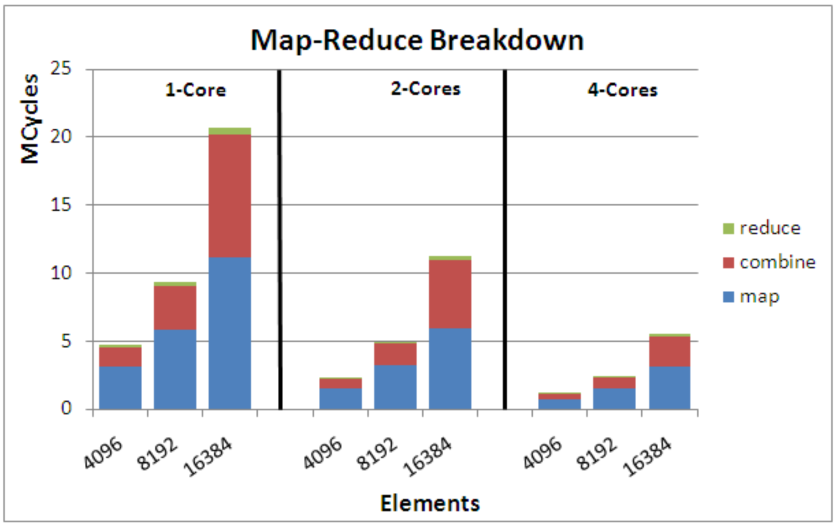


Figure 6.12: MapReduce k-means breakdown – Small sizes

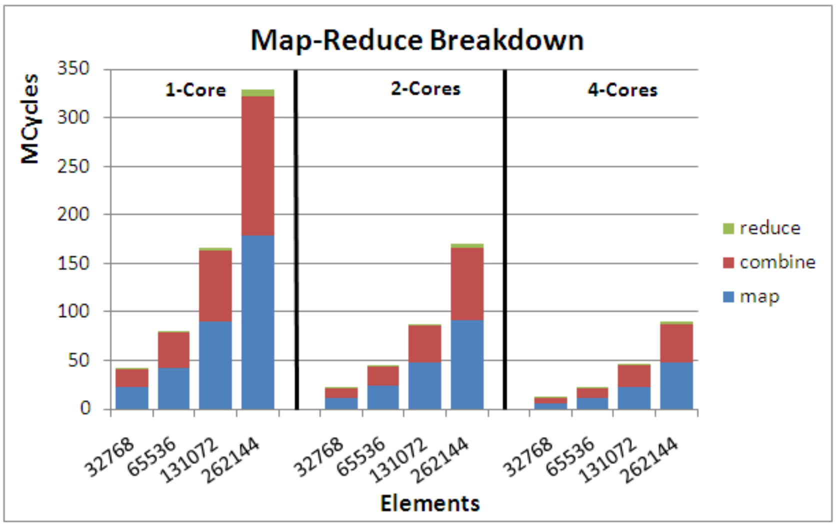


Figure 6.13: MapReduce k-means breakdown – Big sizes

6.6 Observations

All of the MapReduce applications achieve high performance and good scalability as array size or participating processors increase. These lead to achieve speedup up to 2 and up to 4 for two and four processors accordingly. However, as the input array's size increases we get less speedup compared to the speedup we get for small array sizes. This is caused by a part of the algorithm that cannot be parallelized and runs to a single processor. The lower speedup we achieve is 1.7 and 3.3 for two and four processors accordingly which is adequate for a parallel application.

For the word count and the histogram applications it is not necessary to change lots of thing to the base MapReduce algorithm, however, we have to add more functionalities for the k-means to support the necessary data processing for the clustering. As a result k-means algorithm takes more execution time to the map phase compared to the map phase of the word count and the histogram applications.

Chapter 7

Related Work

7.1 Related Work

A significant amount of research and literature is available on the topic of runtime support for programming chip multiprocessors. However, only few of them exploit explicit communication mechanism that the systems support.

For the most known high end architectures, there have been implemented sophisticated SDKs that provide some primitives to the programmers by exploiting the available recourses of the system. These offer high performance mechanism to transfer data between memories, synchronize processors, and manage hardware modules of the system through software.

The Cell Broadband Engine - or Cell as it is more commonly known - is a microprocessor designed to bridge the gap between conventional desktop processors and more specialized high-performance processors. In a simple analysis, the Cell processor can be split into an external input and output structures, the main processor called the Power Processing Element (PPE), eight fully-functional co-processors called the Synergistic Processing Elements, or SPEs, and a specialized high-bandwidth circular data bus connecting the PPE, input/output elements and the SPEs, called the Element Interconnect Bus or EIB. This processors offers a lot of challenging in parallel high performance application development. The Cell Broadband Engine software development kit [12] offers a variety of sophisticated mechanisms to exploit the available resources of the Cell multiprocessor. These contain mechanisms to transfer data through DMAs, to move data from local storage to effective addresses, apply barriers, fences, manage mailboxes, atomically execute tasks and several other mechanism that provide programmers with sophisticated task in order to achieve high performance.

CUDA (for Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA [14]. CUDA is the computing engine in NVIDIA graphics processing units or GPUs that is accessible to software developers through industry standard programming languages. Programmers use 'C for CUDA' (C with NVIDIA extensions), compiled through a PathScale Open64 compiler to code algorithms for execution on the GPU. CUDA has several advantages over traditional general purpose computation on GPUs (GPGPU) using graphics APIs. These contain scattered reads where code can read from arbitrary addresses in memory, a fast shared memory region that can be shared amongst threads and be used as a user-managed cache, enabling higher bandwidth than is possible and much more. The CUDA library offers mechanisms to allocate memory, copy data, to stream data from memories, and of course to execute fast calculations.

Apart from SDKs for known architectures there have been developed common interfaces for various systems in order to easily port applications from a system to another. One of the most commonly used interfaces is the MPI (Message Passing Interface) [15]. MPI is a specification for an API that allows many computers to communicate with one another. It is used in computer clusters and supercomputers. MPI's goals are high performance, scalability, and portability. MPI remains one of the dominant models used in high-performance computing today. MPI defines routines for synchronization, data movement, collective computations, blocking and non-blocking send and receive operations and several other primitives that provide programmers a variety of routines to exploit computer clusters and supercomputers.

Apart from the SDKs and the developed API a lot of studies have been done to effectively port several applications in several high end systems.

In [6] authors present the StreamIt language and compiler for streaming applications. The StreamIt language provides novel high-level representations to improve programmer productivity and program robustness within the streaming domain. At the same time, the StreamIt compiler aims to improve the performance of streaming applications via stream-specific analysis and optimizations. We motivate, describe and justify the language features of StreamIt, which include a structured model of streams, a messaging system for control, and a natural textual syntax. Several applications have been developed based on the StreamIt language. Some of them are the bitonic sort, the DES encryption algorithm, the FFT, the filter bank, an MP3 decoder and several others.

In [9] authors presented a design and implementation of MapReduce for the Cell architecture that provides a simple machine abstraction to users, hiding parallelization and hardware primitives. This runtime automatically manages parallelization, scheduling, partitioning and memory transfers. They showed that the model is well suited for many applications that map well to the Cell architecture, and that the runtime sustains high performance on several MapReduce applications.

MapReduce has also been ported for multi-core and multiprocessor systems. In [10] authors describe Phoenix, an implementation of MapReduce for shared-memory systems that includes a programming API and an efficient runtime system. The Phoenix runtime automatically manages thread creation, dynamic task scheduling, data partitioning, and fault tolerance across processor nodes. And in [11] authors optimize the Phoenix runtime on a quad-chip, 32-core, 256-thread with shared-memory system with NUMA characteristics. They show that efficient execution on a large-scale system requires a multi-layered optimization approach where runtime developers must carefully select the runtime algorithms and optimize their implementations around NUMA challenges.

At last but not least, in the work presented at [15], authors describe efficient algorithms for the FFT application that perform well in cases where problem fits or not in data caches. Problem sizes that fit in the data cache do not face significant difficulties. However, problems that exceed cache size perform poorly. In order to reduce cache misses authors exhibit appropriate data replacement and twiddle multiplies.

Chapter 8

Conclusions

8.1 Limitations

The system we use for the application development is a prototype system based on a modern development board that contains a FPGA platform, with several commonly used peripherals. These tools do not offer the capabilities of modern ASIC (Application-Specific Integrated Circuit) or integrated circuit systems as they have a maximum capacity that does not allow developers to add as many modules as the might would need.

For the system we use, we have the limitation to use four processors as it is impossible to add more due to lack of space. More processors would give us more clear results for the scalability of the existing system and the applications we develop. Moreover, these processors have poor performance in some simple tasks, such as loops and control statements as they lack branch predictors.

Another limitation that the system has is the small scratchpad memories. Having big enough scratchpad memories, could offer higher performance as multiple buffering does not always achieves the highest performance with small buffer sizes. Moreover we would be able to execute application with bigger problem sizes on-chip.

However, even by scaling down the data sets of benchmarks to fit in the small scratchpad memories and to be able to be executed by only four processing units we exploit fine-grain parallelism and achieve speedup for all of the applications.

Cache coherence support for the system could also improve the performance of some application. Cache coherence support could improve applications with irregular and input-dependent communication patterns, where it is hard for the programmer to perform timely data prefetching and implement the required communication with bulk data transfers. However, coherence might lead to lower performance in some cases where explicit communication is more appropriate for an application due to the coherence protocol overhead.

8.2 Future Work

There is even more work that can be done with the software running in the system we use. First of all the system libraries have to be updated with more basic tasks that provide programmers a more integrated and robust system. Moreover, a runtime support for task management of the system would allow programmers to manage the system and develop parallel applications more easily. Another possible addition to the system's software could be an API porting such as the MPI. This development would allow porting of more applications, which already have been ported to MPI, to the specific system. At last but not least, more application should be developed for the specific system in order to observe the system's performance under various circumstances.

As far as the hardware system we use is conserved, it would be desirable to have bigger scratchpad memories, more processors, and even remote read capability. Moreover, coherence among memories could improve the performance of some applications and make the system more complete. Another possible upgrade of the system could be a multi-board system, where multiple boards will be connected through links in order to have more resources in one system.

8.3 Conclusion

In this work we use a complete prototype chip multiprocessor system with explicitly managed local memories in order to develop several applications. This system is robust and offers programmers a platform to develop and execute parallel application from scratch. It offers various sophisticated implicit and explicit communication mechanism to exchange data and synchronization methods and high performance.

We present the available communication mechanism and the system's capabilities. We port several applications to the specific system and exploit effectively the explicit communication mechanism that the system provides. First of all we develop a stream application to stress the performance of all the communication mechanisms that the system provides. This application shows that on-chip DMAs achieved maximum aggregate bandwidth 320MB/s, off-chip DMAs 180MB/s, on-chip remote stores 71MB/s and final off-chip remote stores achieved 21MB/s. These results give us the limits of the communication mechanism of the system which are enough for a system of such a scale.

The bitonic sort and the FFT applications give us important results for the performance of the system and the minimum problem granularity that we can achieve speedup when using multiple cores. The bitonic sort achieves speedup up to 1.8 for two processors and up to 2,7 for four. This application achieves performance for multiple processors even with problem sizes of 4 elements and 700 clock cycles. When using the remote stores mechanism. FFT is more communication intensive application compared to the bitonic sort application as the first one demands all to all data exchanges whether the bitonic sort demands one to one communication. Due to these facts the FFT application achieves speedup greater than one, compared to the performance of the application running on a single processor, for 256 elements or more. FFT application achieves maximum speedup up to 1.9 and 3.2 for two and four processors accordingly.

Moreover, we develop three application based on the MapReduce programming model. These applications are a word count, a histogram production and the k-means clustering algorithm. All of them achieve high scalability as participating processors or problem size increase. As a result, applications manage to achieve maximum speedup almost up to 2 and up to 4 for two and four processors respectively.

To conclude with, system can achieve high performance and good scalability for various applications if we effectively exploit the provided explicit communication mechanisms. Libraries offer full support of the communication mechanisms that the system provides with high performance. We use properly these features and report techniques to exploit these in order to achieve high performance and high speedup rates on the prototype system.

Chapter 9

Bibliography

[1] Xilinx Inc. Xilinx University Program XUPV5-LX110T Development System. http://www.xilinx.com/univ/xupv5-lx110t.htm.

[2] George Nikiforos, George Kalokairinos, Vassilis Papaefstathiou, Stamatis Kavadias, Dionisios Pnevmatikatos and Manolis Katevenis, "A run-time Configurable Cache/Scratchpad Memory with Virtualized User-Level RDMA Capability," in the 6th HiPEAC Industrial Workshop on Embedded Computing, 26 November 2008, THALES Research and Development - Palaiseau, Paris, France.

[3] George Kalokairinos, Vassilis Papaefstathiou, George Nikiforos, Stamatis Kavadias, Manolis Katevenis, Dionisios Pnevmatikatos, and Xiaojun Yang, "FPGA Implementation of a Configurable Cache/Scratchpad Memory with Virtualized User-Level RDMA Capability,"

Proc. IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS2009), 20-23 July 2009, Samos, Greece.

[4] J. McCalpin, “Memory Bandwidth and Machine Balance in Current High Performance Computers,” IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, Dec. 1995.

[5] M. Ajtai, J. Komlos, E. Szemeredi, “An O(n log n) Sorting Network,” proceedings of the 25th ACM Symposium on Theory of Computing, 1 September 1983.

[6] Saman P. Amarasinghe and Michael I. Gordon and Michal Karczmarek and Jasper Lin and David Maze and Rodric M. Rabbah and William Thies, “Language and Compiler Design for Streaming Applications,” International Journal of Parallel Programming, vol. 33, no. 2-3, pp. 261–278, 2005.

[7] J.W. Cooley and J.W. Tukey, “An algorithm for the machine computation of the complex Fourier series,” Mathematics of Computation, vol. 19, pp. 297–301, April 1965.

[8] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[9] M. de Kruijf and K. Sankaralingam, "MapReduce for the Cell BE architecture," IBM Journal of Research and Development, vol. 53, no. 5, 2009.

[10] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis, “Evaluating MapReduce for Multi-core and Multiprocessor Systems,” proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture (HPCA), Phoenix, AZ, February 2007.

[11] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System," proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), pp. 198-207, Austin, TX, October 2009.

[12] The Cell Broadband Engine resource center, http://www.ibm.com/developerworks/power/cell/index.html

[13] Gropp William, Lusk Ewing, Skjellum Anthony, “Using MPI: portable parallel programming with the message-passing interface,” MIT Press in Scientific And Engineering Computation Series, Cambridge, MA, USA. pp. 307, 1994.

[14] Compute Unified Device Architecture (CUDA), http://www.nvidia.com/object/cuda_home.html.

[15] Kevin R. Wadleigh, Hewlett-Packard Company, High Performance Systems Division, Richardson, Texas, U.S.A,

"High Performance FFT Algorithms for Cache-Coherent Multiprocessors," international Journal of High Performance Computing Applications, Volume 13, Issue 2, pp. 163 – 171, May 1999.

[16] The Scalable computer ARChitecture (S.A.R.C.) project. http://www.sarc-ip.org.

[17] The S.A.R.C. architecture manual. https://hardbox.ics.forth.gr/svn/sarc/archManNI