Implementing Scalable Parallel Programming Models with Hybrid Address Spaces

Anastasios Papagiannis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisor: Prof. *Manolis Katevenis*

This work has been performed at the **Institute of Computer Science (ICS) Foundation for Research and Technology – Hellas (FORTH), Heraklion, Crete, GREECE.**

The work is partially supported by the **European Community's Seventh Framework Programme [FP7/2007-2013], I-CORES project** under contract number n^o **224759.**

University of Crete
Computer Science Department

Implementing Scalable Parallel Programming Models with Hybrid Address Spaces

Thesis submitted by
Anastasios Papagiannis

in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Anastasios Papagiannis

Committee approvals: _____
Manolis Katevenis
Professor Thesis Supervisor

_____
Dimitrios S. Nikolopoulos
Professor Committee Member

_____
Angelos Bilas
Professor Committee Member

Departmental approval: _____
Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, February 2013

Abstract

This thesis introduces hybrid address spaces as a design methodology for implementing scalable runtime systems on many-core architectures without hardware support for cache coherence. We demonstrate hybrid address spaces in an implementation of MapReduce, a well-established programming model for large-scale, fault-tolerant data processing. Using the Intel Single-Chip Cloud Computer as an experimental testbed, we present HyMR, a staged MapReduce runtime system whereby different stages alternate between a distributed memory address space and a shared memory address space to improve performance and scalability. In exploring hybrid address spaces, we introduce four improvements in the implementation of MapReduce: (1) Lock-free data distribution algorithms, using user-defined splitter functions. (2) A scalable, interrupt-less implementation of work-stealing for non-coherent architectures using exclusively on-chip communication to minimize latency. (3) Optimized implementation for on-chip barrier algorithms for non-coherent many-core processors. (4) A new mechanism to enable fast access from a core to the private memory of another core on-chip, which accelerates global exchange operations.

We compare HyMR to an optimized reference implementation using exclusively distributed address spaces and find that hybrid address spaces improve performance by a factor of 1.71× (geometric mean). We also compare HyMR with Phoenix++, a state-of-art implementation for systems with hardware-managed cache coherence in terms of scalability and sustained to peak data processing bandwidth, where HyMR demonstrates improvements of a factor of 3.1× and 3.2× (geometric mean) respectively.

Περίληψη

Η εργασία αυτή εισάγει τους υβριδικούς χώρους διευθύνσεων ως τεχνική για την υλοποίηση κλιμακώσιμων συστημάτων χρόνου εκτέλεσης σε πολυπύρινες αρχιτεκτονικές χωρίς την υποστήριξη υλικού για την συνοχή των κρυφών μνημών. Παρουσιάζουμε τους υβριδικούς χώρους διευθύνσεων για την υλοποίηση του MapReduce, ενός καθιερωμένου μοντέλου προγραμματισμού για μεγάλης κλίμακας, με ανοχή σε σφάλματα, επεξεργασία δεδομένων. Χρησιμοποιώντας τον επεξεργαστή Intel Single-Chip-Cloud ως πειραματική πλατφόρμα δοκιμών παρουσιάζουμε το HyMR, ένα σύστημα χρόνου εκτέλεσης MapRecuce για το οποίο διαφορετικά στάδια εκτέλεσης εναλλάσονται μεταξύ κατανεμημένης μνήμης χώρου διευθύνσεων και κοινόχρηστης μνήμης χώρου διευθύνσεων για την βελτίωση της επίδοσης και της κλιμακωσιμότητας. Στην εξερεύνηση των υβριδικών χώρων διευθύνσεων εισάγουμε τέσσερις βελτιώσεις στην υλοποίηση του MapReduce: (1) Παράλληλους αλγόριθμους, χωρίς την χρήση συγχρονισμού (lock-free), για τον διαμοιρασμό δεδομένων, χρησιμοποιώντας συναρτήσεις που ορίζονται από τον χρήστη. (2) Μια κλιμακώσιμη, χωρίς διακοπές (interrupts), υλοποίηση του αλγορίθμου έργου-κλοπή (work-stealing) για αρχιτεκτονικές χωρίς την υποστήριξη υλικού για την συνοχή των κρυφών μνημών, χρησιμοποιώντας μόνο μνήμη που βρίσκεται μέσα στον επεξεργαστή για την ελαχιστοποίηση της αδράνειας (latency). (3) Βελτιστοποιημένη υλοποίηση αλγορίθμων φραγμάτων (barriers) χρησιμοποιώντας μνήμη που βρίσκεται μέσα στον επεξεργαστή για πολυπύρινες αρχιτεκτονικές χωρίς την υποστήριξη υλικού για την συνοχή των κρυφών μνημών. (4) Ένα νέο μηχανισμό που επιτρέπει την γρήγορη πρόσβαση από έναν πυρήνα στην τοπική μνήμη κάποιου άλλου πυρήνα, το οποίο επιταγχύνει την καθολική ανταλαγή δεδομένων.

Συγκρίνουμε το HyMR με μία βελτιστοποιημένη υλοποίηση αναφοράς η οποία χρησιμοποιεί μόνο κατανεμημένους χώρους διευθύνσεων και βρίσκουμε ότι οι υβριδικοί χώροι διευθύνσεων βελτιώνουν την επίδοση κατά 1.71× (γεωμετρικός μέσος). Επίσης συγκρίνουμε το HyMR με το Phoenix++, την καλύτερη υλοποίηση για συστήματα με υποστήριξη υλικού για την συνοχή των κρυφών μνημών, σε όρους κλιμακωσιμότητας και αποδοτικότητας σε σύγκριση με τον μέγιστο ρυθμό επεξεργασίας δεδομένων, όπου το HyMR αποδεικνύει βελτιώσεις κατά 3.1× και 3.2× (γεωμετρικός μέσος) αντίστοιχα.

Acknowledgements

There are so many people that I would like to thank, each one helped me with their own special way. First of all, I would like to thank my advisor Professor Dimitrios S. Nikolopoulos. He has been a tireless source of inspiration, encouragement and ideas during my undergraduate and graduate studies. He introduced me in computer architecture and systems research. For all that and much more, I am grateful.

I would also like to thank Professor Manolis Katevenis and Professor Angelos Bilas for contributing as members of my Masters committee. I would like also to thank them for the background, I gain during my undergradute and graduate studies from courses and several discussions. They are always willing to help me about my studies.

I need to express my gratitude to the University of Crete and the Department of Computer Science for providing me with proper education; as well as the Institute of Computer Science of the Foundation for Research and Technology (ICS-FORTH) for supporting me.

Moreover I would like to give my appreciation to my friends, Vassilis Papakonstantinou and Antonis Papaioannou for the encouragement and the support during my undergraduate and graduate studies.

Last but not least I am grateful to my father Eleftherios, my mother Eleni as well as my sister Anna–Maria for the encouragement and the support in every single aspect of my life.

Herakleion–Crete, Anastasios Papagiannis
February 2013

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many-core processors use diverging memory architectures. Some processors use a memory hierarchy with local multi-level caches per core and a hardware protocol to keep those caches coherent [1]. This memory architecture resembles earlier shared-memory multi-processors from a programmer's standpoint. However, some processors use memory hierarchies without a coherence protocol. On the other hand, Graphics Processing Units (GPUs) [2], the Intel SCC [3] and the Cell processor [4] are representative examples of non-coherent architectures. Programming a non-coherent architecture requires explicit communication between local address spaces, through message passing or Direct Memory Access (DMA). Explicit communication increases the programmer's burden, as it requires a high level of expertise in parallel programming and architectures to master. Runtime systems ease this burden to a certain extent by implementing high-level communication primitives and packaging them in user-level libraries (e.g MPI). Alternatively, non-coherent architectures can be programmed with a shared address space model. In this case, the runtime system implements a virtual shared memory abstraction. In all cases, the runtime system is a critical component that largely defines

performance and programmability.

Runtime systems for non-coherent architectures are currently implemented on top of distributed address spaces, typically using one address space per core. The runtime system itself implements all necessary inter-core communication operations for scheduling and synchronization, as well as all application-level communication through explicit message passing or DMAs. These operations flow either exclusively between local memories or between local memories and DRAM. This implementation paradigm has been used on the Cell processor, for implementing shared-memory programming models such as OpenMP [5], COMIC [6] Sequoia [7], and CellSs [8] and the Intel SCC for the implementation of X10 [9] and Shared Virtual Memory models [10]. Intuitively, using explicit communication in the runtime system yields a scalable implementation. In particular, explicit communication leverages on-chip data transfer paths and a scalable NoC interconnect for passing data between cores without paying the cost of off-chip memory accesses. This approach works particularly well for exchanges of messages that fit in on-chip local memories. However, this approach is not necessarily optimal in other cases. Applications often need to transfer large amounts of data between processes or threads in a program with littler or no processing on the data itself. If these streaming data transfers flow through the on-chip memory hierarchy, they will incur cache pollution, without offering an opportunity for data reuse. Such operations should be best left uncached to maximize performance. A shared, global address space model suits these operations best.

This thesis introduces *hybrid address spaces* as a fundamental design and implementation methodology for scalable runtime systems. The intuition behind our proposition is that a runtime system uses on-chip communication paths for small data transfers, such as those needed to exchange control

data for scheduling, and off-chip communication paths for large, streaming data transfers. To verify this intuition, we present HyMR, an implementation of the MapReduce programming model [11] on the Intel Single-Chip Cloud Computer [3]. The MapReduce runtime implements a staged execution model. We show that while certain stages are best implemented with message passing over a distributed address space, other stages are best implemented with in-place memory copying in a single, global address space. In demonstrating the concept of hybrid address spaces in runtime systems, we make several more contributions towards improving performance and scalability of MapReduce on non-coherent many-core architectures. These contributions include:

- Scalable, application-specific data splitters.
- A scalable, interrupt-less implementation of work-stealing on non-coherent architectures using exclusively on-chip communication to minimize latency.
- A new exploration of on-chip barrier algorithms for non-coherent many-core processors.
- A new mechanism to enables fast access from a core to the private memory of another core, which enables fast implementations of global exchange operations.

Our implementation of HyMR provides design guidelines for latency and throughput critical runtime system operations that are common to many, if not all, programming models. These include scheduling and load balancing, data distribution, and various point-to-point and group communication operations.

We compare HyMR to a reference runtime system implemented using exclusively message passing. HyMR outperforms the baseline in all tests. We also compare HyMR with Phoenix++, a state-of-art MapReduce implementation for hardware-managed cache-coherence systems [12]. HyMR achieves, on average, 3.1× improvement in speedup and 3.2× improvement of bandwidth efficiency, on the same number of cores.

The rest of this thesis is organized as follows: Section 2 provides background on MapReduce and the Intel SCC processor. Section 3 presents the design and implementation of DiMR, a reference implementation of the MapReduce runtime for SCC processor, which uses exclusively distributed address spaces. Section 4 presents the design and implementation of HyMR. Section 5 presents our experimental analysis and results. Section 6 discusses related work and Section 7 concludes the thesis.

Chapter 2

Background

Hardware support for cache coherence on a processor with many cores increases complexity and power [13]. Although many efforts attempt to address the scaling and power limitations of cache coherence on systems with many cores [1], several vendors of many-core processors opt for a non-cache-coherent architecture. On such an architecture, a programmer writes parallel code using either explicit communication mechanisms or a shared virtual memory layer implemented in software. In this section we provide background on non-cache-coherent many-core processors and programming models providing a shared memory abstraction on such processors. We discuss in more detail the architecture of the Intel Single Chip Cloud Computer (SCC), a processor prototyped to explore the performance, programmability and power-efficiency of non-coherent architectures. We use the SCC as an implementation vehicle for implementing scalable runtime systems with hybrid address spaces. We then provide background on MapReduce, a parallel programming model for large-scale data processing, inspired by functional languages.

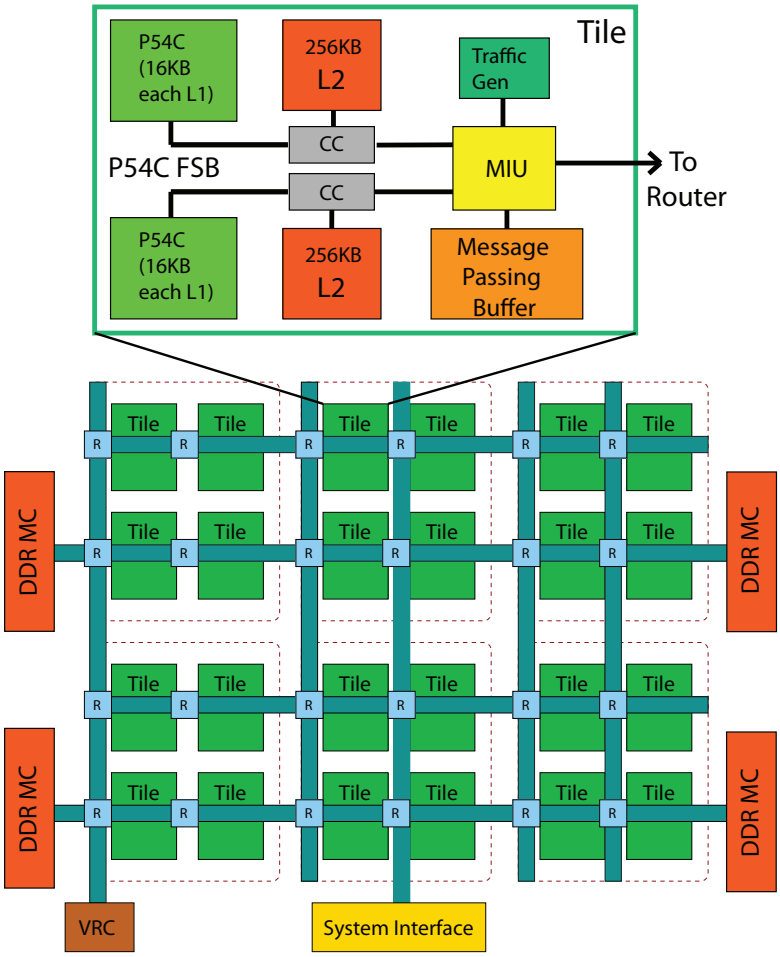

Figure 2.1: SCC processor diagram.

2.1 Intel Single-Chip-Cloud-Computer (SCC)

The Intel SCC[1] [14] (Figure 2.1) is a many-core processor with 24 tiles and two IA cores per tile. The tiles are organized in a 4×6 mesh network with 256 GB/s bisection bandwidth. The processor has four integrated DDR3 memory controllers, one for each group of six tiles. Each core has a private L1 instruction cache of 16 KB, a private L1 data cache of 16 KB and a pri-

[1]The SCC is not a stand-alone computer thus to get it running, a management PC (MCPC) needs to be used. The SCC connects to the MCPC through external PCIe.

vate unified L2 cache of 256 KB. Each dual-core tile has a 16 KB message passing buffer (MPB). The MPB is the only component of the SCC on-chip memory hierarchy that is shared between cores. The SCC does not implement any cache-coherence mechanism between MPB and caches. The MPB provides space for direct core-to-core communication. Data used in on-chip communication is read from the MPB, bypassing the L2 cache. For writes, a no-allocate policy is used, in conjunction with a write combining buffer in the L1 cache. Software needs to maintain coherence between the MPB and the L1 caches by using an L1 cache invalidation instruction (CL1INVMB), when data is stored in the MPB. According to the processor specifications [15], the latency to read a cache line from MPB buffers and off-chip DRAM are:

$$Local\ MPB = 45C_c + 8C_m \tag{2.1}$$

$$Remote\ MPB = 45C_c + 4 \cdot n \cdot 2C_m \tag{2.2}$$

$$DRAM = 40C_c + 4 \cdot n \cdot 2C_m + 46C_r \tag{2.3}$$

where C_c, C_m and C_r denote the clock cycles of the core, the mesh network and the DRAM respectively and n denotes the number of mesh network hops required to reach the destination ($0 < n \leq 8$). Although the difference to access MPB and DRAM is 46 DRAM cycles, accesses to the MPB bypass the L2 cache, which can not be flushed or invalidated from hardware. The obvious drawback of using the MPB is its small size (8KB per core).

2.1.1 SCC Address Spaces

The SCC uses 32-bit Pentium cores. A programmable, software-managed translation table (called Look-Up Table or LUT) enables the system to extend

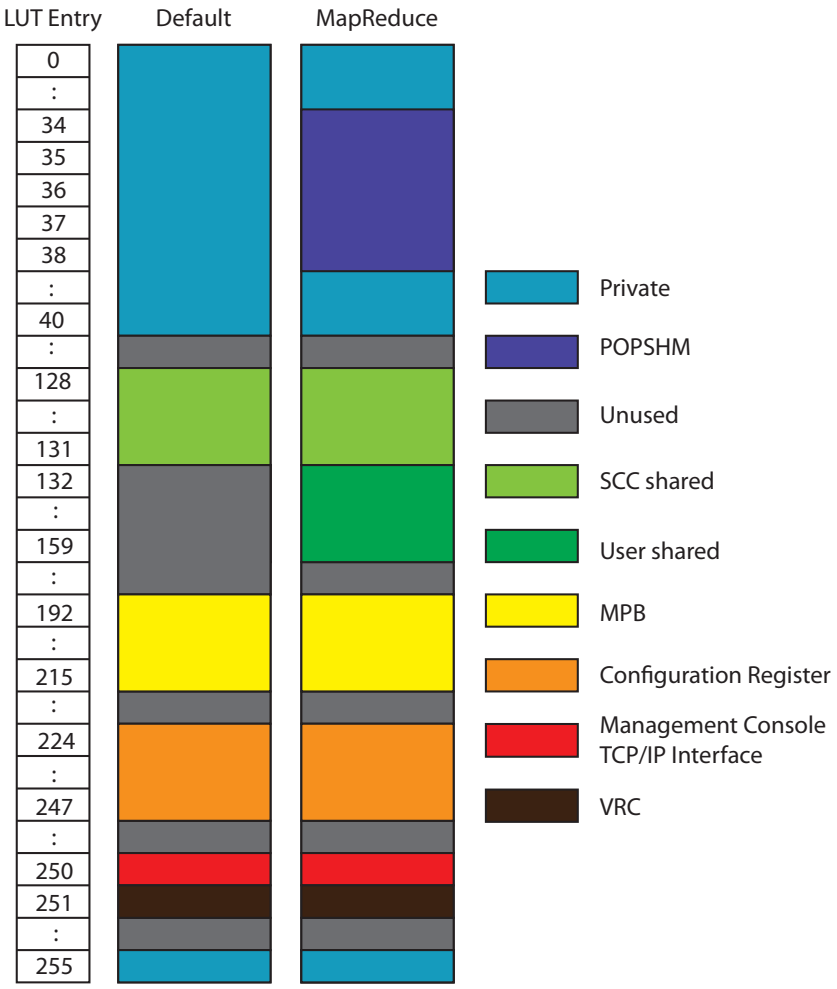

Figure 2.2: Defauly mappings of LUT entries at runtime.

the width of physical addresses to 34 bits, allowing system configurations with to up to 64 GB of off-chip memory (specifically, up to 16 GB for each of four groups of six tiles). The LUT has 256 entries, each mapping 16MB of DRAM. Software control of LUT mappings provides a convenient tool for implementing hybrid private and shared address spaces in the system.

Figure 2.2 shows the default configuration of LUT entries. The SCC reserves 41 (0–40) entries at the bottom of the LUT to map up to 656 MB

of private physical memory for each core. The operating system running on the core uses part of this memory, while the user can use the rest. Intel provides a custom Linux kernel that during the boot process, allocates 5 (34–38) contiguous entries from each core's private address space. This new address space called *POPSHM*. Four entries (128–131) in the LUT are shared among all cores. Some parts of this shared memory are used by system services[2]. Entries 192–215 in the LUT map MPBs and entries 224–247 map configuration registers of cores. Entry 250 addresses the system interface; access to this memory is confined to the PCIe driver. Entry 251 addresses the voltage regulator control (VRC) registers. There is no restriction in reprogramming LUT entries to translate to a different address space during the execution of a program.

2.1.2 SCC System Software

From the programmer's point of view, SCC resembles a cluster with portions of memory shared between cores. Each core runs its own image of the Linux kernel. Cores communicate through messages and several libraries that provide message passing primitives are available to programmers like RCCE [3] and RCKMPI [16]. Small messages can be exchanged directly on-chip using the MPBs. Large messages on the other hand can be exchanged via a memory copy in DRAM. Figure 2.3 shows the flow of messages in both cases, using an example where core 0 sends a message to core 47. When sending a small, less than 8KB (MPB size), message, the sender writes the message in its local MPB. The L2 cache is bypassed and the L1 cache is configured as write no-allocate. The sender stores flags in the MPB to synchronize this operation with the receiver. When the data is ready, the receiver can read

[2]e.g. MCPC and the on-die network driver that allows TCP traffic from core to core.

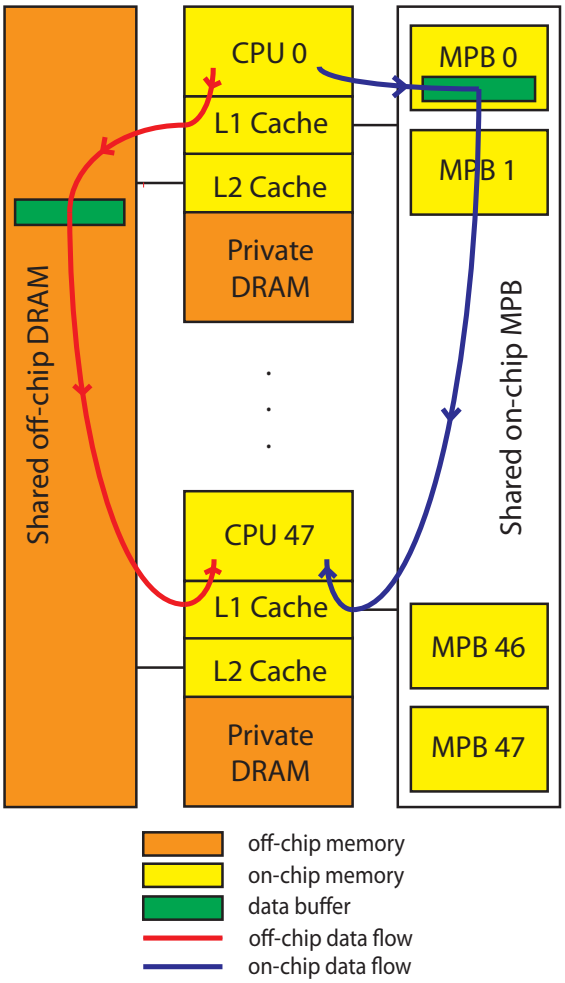

Figure 2.3: Message flow using off-chip DRAM and on-chip MPB.

the data to its private memory through its own L1 cache. The MPB provides higher bandwidth and lower latency than the available shared memory. In spite of this advantage, message passing for messages larger than 8KB can be faster through DRAM, due to protocol overheads related to the small size of the MPB and the necessity to split and reassemble parts of the message into the whole message. The alternative solution is to use shared DRAM to exchange messages greater than 8KB . The L2 cache can still be bypassed in this case, to avoid severe cache pollution. When transmitting a large mes-

sage, the sender writes the whole message in shared memory. The L1 caches need to be flushed to maintain coherence and consistency. The receiver can read the whole message from shared memory through the L1 cache. The SCC provides a facility to invalidate all MPB data with a single instruction (CL1INVMB), flush all L1 cache data with a single instruction (INVFLUSH), or invalidate all L1 cache data with a single instruction (INV). Due to the lack of a hardware flush/invalidate mechanism, the processor can use a software memory driver to flush the L2 cache, if needed. Selective use of the L1 and L2 caches is critical for performance and we revisit this issue while discussing the implementation of HyMR on the SCC.

2.2 The MapReduce Programming Model

2.2.1 Programming Model

MapReduce is a set of language abstractions, inspired by Lisp [11], to express data-parallel computations and aggregations. The MapReduce programming model is widely popular among developers of algorithms for Big Data analytics. MapReduce is commonly employed for running crawling and machine learning algorithms on large volumes of text and image data, as well as processing large graphs [11, 17, 18, 19]. Practical implementations provide MapReduce abstractions as a library API or embed MapReduce in a high-level language, such as Java [20, 21, 12, 22, 23, 24, 25, 26, 27, 28].

A MapReduce application applies a parallel operator, the *map* function, on input data structured as a sequence of <key,value> pairs. The output of the map function is a set of intermediate <key,value> pairs. A user-defined reduction operator, the *reduce* function, aggregates the intermediate pairs according to their keys. Finally, the aggregated pairs are sorted by key

value. Aggregation and sorting are optional in MapReduce applications. The language or library may provide standard aggregators and sorting functions for high performance and ease of programming.

Figure 2.1 shows a classic MapReduce example that counts the number of occurrences of each word in a collection of documents [11]. The map function emits each word from the documents with a temporary count of occurrences set to 1. The reduce function measures the total number of occurrences for each unique word. MapReduce is an extremely simple programming model. The programmer applies operators on data lying in a single logical address space, albeit the actual implementation may distribute data between physically separate memories and disks. The operators adhere to a share-nothing model, which virtually eliminates races, deadlocks, and most complexities that render correctness checking hard on conventional parallel programming models. On the flip side, the performance of MapReduce programs is heavily dependent on the implementation efficiency and scalability of the runtime system.

2.2.2 MapReduce Runtime Systems

To MapReduce runtime system (Figure 2.4) splits input pairs into work units. Tasks executing the map function (mappers) process work units in parallel across multiple nodes, processors, or cores. The runtime system partitions the intermediate pairs produced from mappers into buckets with each bucket holding pairs with the same key. These buckets, called partitions, are distributed between tasks executing the reduce function (reducers). The runtime system finally merges and sorts the output pairs produced by reducers.

A MapReduce runtime system must optimize execution-time parameters such as the size of work units, the number of mappers and reducers,

Listing 2.1: WordCount algorithm in MapReduce

```
// input: a document
// intermediate output: key=word; value=1
Map(String input) {
    for each word w in input
        EmitIntermediate(w, 1);
}

// intermediate output: key=word; value=1
// output: key=word; value=occurences
Reduce(String key, Iterator values) {
    int result = 0;
    for each v in values
        result += v;
    Emit(key, result);
}
```

the assignment of work units to nodes, processors or cores and the allocation and management of buffer space between stages of the computation. The runtime can perform several additional optimizations: eliminate global synchronization between stages of MapReduce, using a dataflow execution model [29]; eliminate function call overheads by increasing the granularity of work units [20, 21]; reduce load imbalance also by adjusting the granularity of work units and/or the number of mappers and reducers [30]; optimize locality and overlapping computation with data transfers by prefetching work units [31]; and conserve bandwidth and cache space via hardware compression [32]. The runtime system can also provide scalable, application-specific fault tolerance, which is beyond the scope of this work.

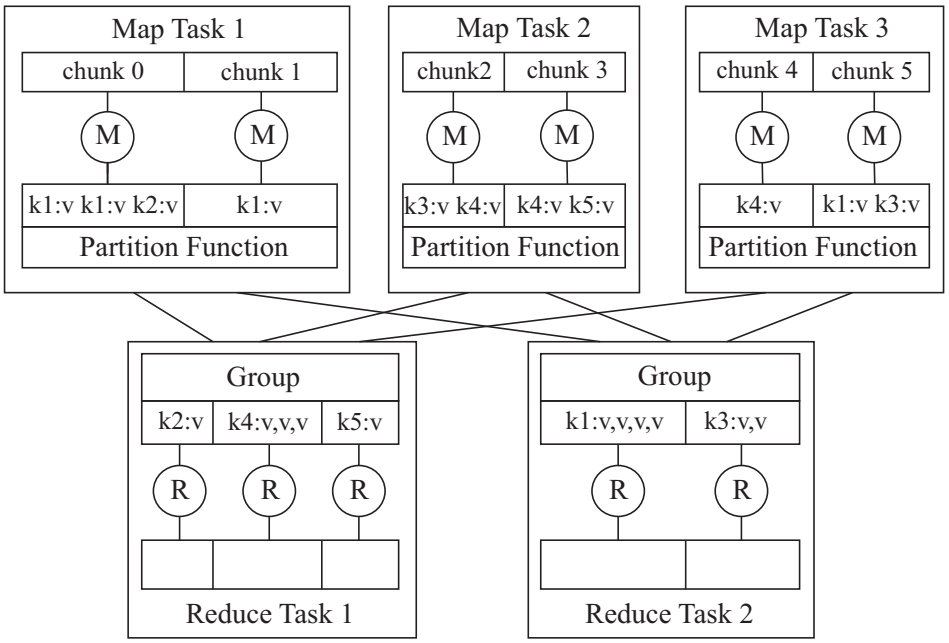

Figure 2.4: Typical MapReduce workflow.

Chapter 3

DiMR Design and Implementation

To place HyMR in context, we first discuss present a reference implementation of the MapReduce runtime system using exclusively message passing over distributed address spaces. This design views the SCC as a cluster of single-core nodes, each with its own Linux image. Cores exchange messages using the RCCE library [3]. A detailed description of the reference design is available in [33].

The reference design implements a seven-stage runtime system for MapReduce. We refer to the seven stages as *map*, *combine*, *partition*, *group*, *reduce*, *sort* and *merge*. The *combine* and *merge* stages are optional in typical MapReduce setups, whereas the *group* stage replaces an intermediate sorting stage of MapReduce to reduce computational complexity [24, 26, 20, 23]. Figure 3.1 shows the stages and what messages are exchanged between cores in each of them. We use the *WordCount* benchmark as an example to explain the details of these stages.

In the *map* stage, the runtime system divides the input evenly to as many

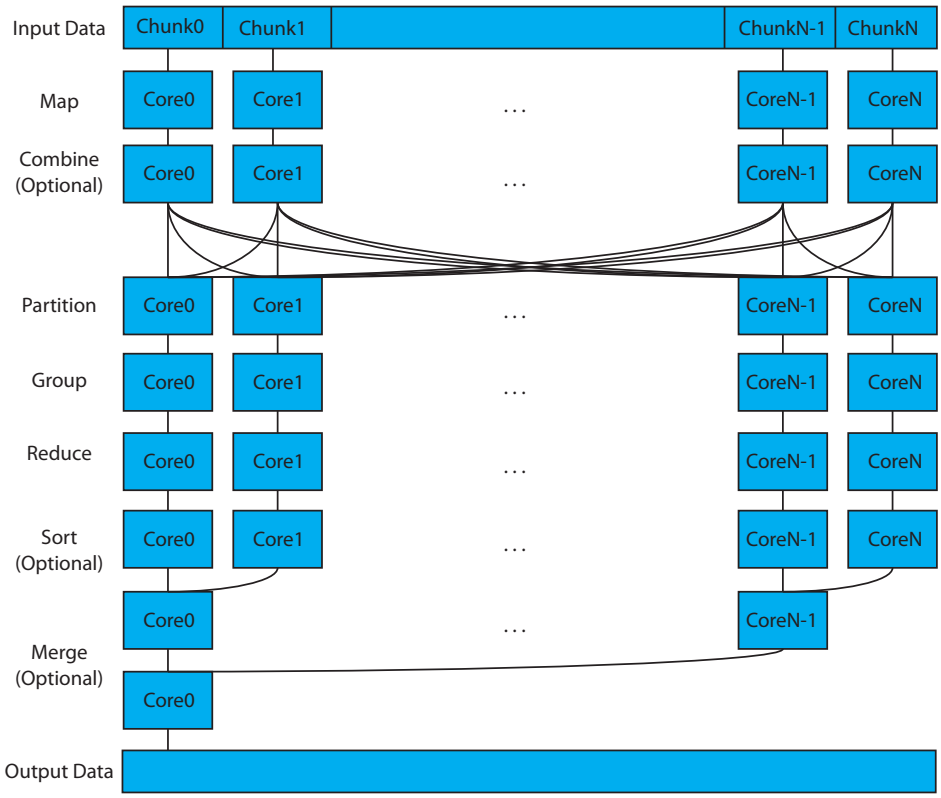

Figure 3.1: The flow of MapReduce Runtime using Message Passing.

partitions[1] as the number of cores. Each core then executes the user-defined map function over the data in its private partition. During this stage the runtime does not exchange any messages between cores. This function takes a key-value pair as input and produces one or more intermediate key-value pairs. The volume of the intermediate output is unknown until runtime. To reduce memory management overhead, the reference design preallocates a large chunk of memory (64 MB in our implementation) to hold intermediate data and allocates more space on demand, if the intermediate data overflows the preallocated chunk. To split intermediate data between different partitions, the reference implementation provides an option between a user-

[1]Not to be confused with the *partition* stage of the MapReduce runtime system.

defined hash function and a generic hash function, the latter implemented in the MapReduce runtime system. The hash function takes a key as an argument and returns the ID of a partition to store the generated intermediate key-value pair. Each core emits keys and values in a contiguous buffer.

The *combine* executes if and only if the user provides a combiner function. This stage is executed locally, as does *map*, and does not exchange messages between cores. The purpose of this stage is to reduce locally the size of each partition produced by a given core during *Map*. The combiner function takes a key and a list of partially aggregated intermediate values associated with the same key, as input. It produces a single key-value pair where the value is an updated partial aggregation of the values associated with the key, as output. After *Combine* stage we synchronize the cores using a barrier.

The *partition* stage performs an all-to-all exchange between cores. Data partitions generated during *Map* may differ in size. The reference implementation uses a custom all-to-all exchange algorithm for the SCC to achieve scalable data partitioning. The algorithm first executes an all-to-all exchange of the intermediate partition's sizes, followed by an all-to-all exchange of the intermediate data [33]. The algorithm implements the all-to-all exchange using pairwise exchanges. Let p be the number of available cores and $rank$ the core ID. This algorithm uses $p - 1$ steps and in each step k the core ranked as i receives data from core $i - k$ and sends data to core $i + k$. We use the *RCCE_ {send, recv}* functions to implement this all-to-all exchange.

The *group* stage groups together all key-value pairs with the same key, taken across all intermediate data partitions. All the data needed is in each core's private memory. There is no need to exchange any messages between cores in this stage. In prior research [24, 26, 20, 23], generic sorting with a user-defined comparator was used to perform grouping in MapReduce. Our

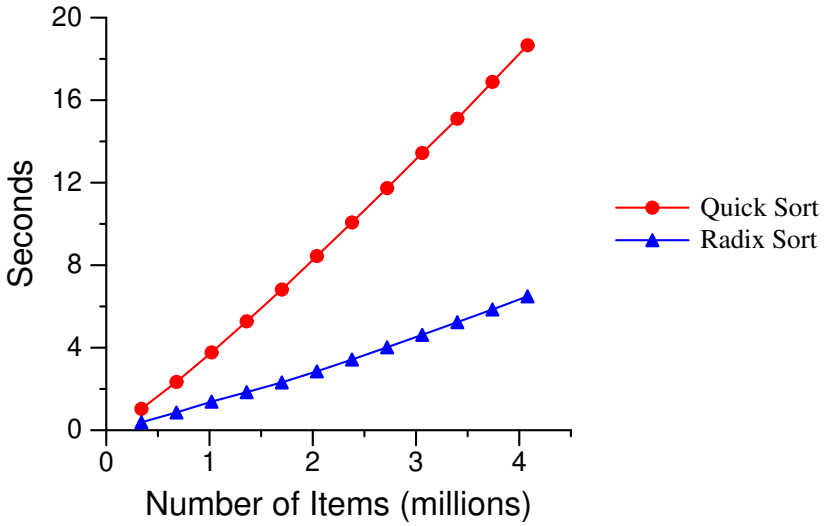

Figure 3.2: Libc qsort vs. radix sort, for a variable number of word-size elements

reference implementation uses a variant of radix sort [34] for grouping on the SCC. The quicksort algorithm employed in prior MapReduce implementations on multi-core systems has complexity $O(nlogn)$, whereas radix sort has complexity $O(kn)$ where k is the size of the key in bytes. Figure 3.2 shows a comparison of the libc quicksort implementation and our radix sort implementation for different input sizes. This measurements are taken from one core on the SCC. Radix sort outperforms quicksort but with one caveat. Radix sort sorts strings of bytes and can not use a user-defined comparator for sorting. This caveat implies that in applications where the key data type is not a string, radix sort may produce unsorted sequences that need to be processed further in the following stages of MapReduce. In the common case, the data produced before the *reduce* stage is more than the data produced after the execution of *reduce* stage. This happens because key duplication in the data generated before the *reduce* stage. Following the *reduce*, there are only distinct keys and a single value associated with each key. We choose to

run the actual sorting algorithm after the *reduce* stage.

The *reduce* stage executes a user-defined key aggregation function. The prior *group* stage exports an array of distinct keys, each containing the number of occurrences of the key and a pointer to an array of its values. The output size of the reduction stage can be statically identified, therefore the implementation preallocates the stage's output buffers. In the *sort* stage, the implementation sorts the key-value pairs produced following the reduction, using sequential quicksort and a user-specified comparison operator. Both *reduce* and *sort* stages locally on private memory and do not exchange any messages between cores. Finally an optional *merge* stage merges the output of all cores in one core. The reference implementation uses the binomial merge algorithm for this stage [35], which completes in $logn$ steps. In each of these steps the cores exchange the previously merged output data.

Chapter 4

HyMR Design and Implementation

In a hybrid address space design, a runtime system uses on-chip communication paths for small data transfers, such as the data transfers needed to pass pointers for the purposes of scheduling computation and data transfers between cores, and off-chip communication paths through shared memory, for performing transfers of large volumes of application data. HyMR implements a staged execution model. We elaborate why while certain stages are best implemented over a distributed address space, other stages are best implemented over a shared address space.

4.1 HyMR Stages

Figure 4.1 shows the stages of HyMR. HyMR has four stages compared with DiMR which has seven stages. In HyMR we merge *Map* and *Combine* into a single stage. We remove *Group* stage from DiMR. The new implementation of *Map* stage allows us to have intermediate data grouped before *Reduce*

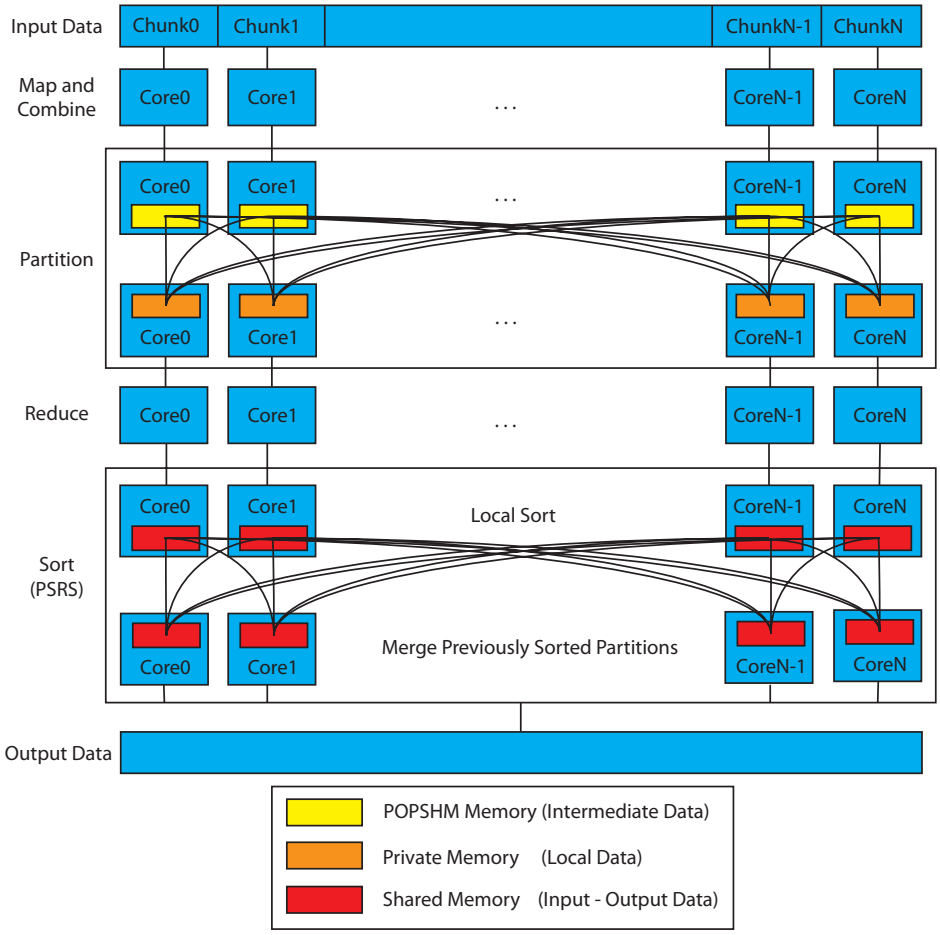

Figure 4.1: The flow of MapReduce Runtime using Hybrid Address Spaces.

stage. We merge *Sort* and *Merge* stage into a singe *Sort* stage. This stage implemented using shared memory and there is no need to further merge the sorted partitions. We keep *Partition* and *Reduce* stages the same as reference design.

We guarantee coherence at the granularity of stages in HyMR. We flush the L2 cache following the execution of mappers and combiners, as the privately owned *POPSHM* address space is cacheable and the SCC has no native hardware support for cache coherence. The flush completes with a memory

barrier. In *Partition* and *Reduce* stages there is no synchronization because both of them executed in private address space. This allows us also to remove the need to flush the caches in order to guarantee coherence between these two stages. In order to make sure that all cores finished with *Reduce* stage we execute a barrier before *Sort* stage. As *Sort* stage contains four sub-stages we have to synchronize this execution. We provide more details in Section 4.1.5.

4.1.1 Scalable Custom Splitters

HyMR uses scalable input splitters over a shared address space. The input is stored in shared memory and accessible from all cores. The input is read-only so there is no need for synchronization in accessing the input during splitting. Each core retrieves a private partition of the input without communicating with other cores, using a local, sequential prefix-scan algorithm. Therefore, splitting can be implemented entirely in parallel. Following splitting, each core allocates a queue in it's own MPB buffer for the input key-value pairs. The runtime executes a user-specified *map* function on each item in the queue. The split function distributes the input evenly between cores, although application-specific splitters can be used in the same context for better load balancing. HyMR provides three application-specific *splitters*, a *text splitter*, a *line splitter* and a *generic splitter*. Users may also implement a *custom splitter* to divide the input size in a different way than the three provided splitters. The generic splitter uses the prefix-scan algorithm running independently on each core, to identify the beginning of each core's chunk in the input. The text and line splitters divide characters or text lines as evenly as possible between cores.

4.1.2 Map

Map tasks have no side effects and no dependencies between them [11]. Therefore, they are suitable for running in a distributed address space without cache coherence. The runtime system stores the output of each mapper task running on a core in the core's POPSHM addess space.

Each core executes mappers that process a queue of inputs provided from splitters. Mappers emmit intermediate key-value pairs, using user-specified hash function to distribute their intermediate outputs between as many partitions as the number of cores. These partitions are aggregated in following MapReduce stages. Each core uses a private *POPSHM* address space for mapping data. This space is represented by five LUT entries, or 80MB. The output of mappers is held in containers, implemented as an array of lists of values, with one list per key. We use a hash table with open addressing, which is faster than separate chaining, Red-Black trees and AVL trees, which we also evaluated on the SCC. The hash table contains 4096 buckets. We implement dynamic resizing of the hash table if a core exports more than 4096 intermediate key-value pairs. We double the size of the table when the fraction of used buckets in the table exceeds a predefined threshold (currently set to 0.8). We use quadratic probing to resolve collisions. Each core cannot export more than five LUT entries, or 80MB of intermediate data. The *POPSHM* implementation in the Linux kernel defines this limit. The runtime system performs no deallocation of *POPSHM* address space. We implement a low latency custom memory allocator to avoid the overheads of storing information that needs on deallocation. This allocates memory in an ascending manner.

HyMR combines the output of mappers, by reducing the data with a user-defined aggregator. The distributed memory implementation uses an

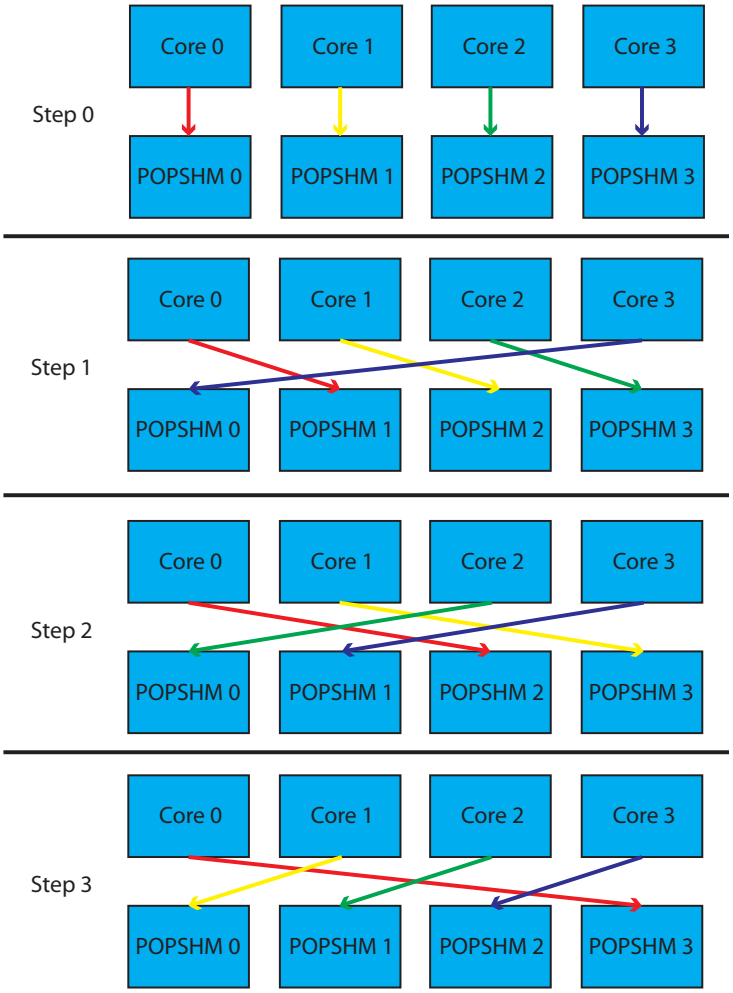

Figure 4.2: The algorithm to avoid memory contention in Rearrange stage.

all-to-all exchange at this stage. We optimize reductions by performing an in-place aggregation of intermediate data in private memory, as the data is produced by mappers. This minimizes space and time overhead by avoiding redundant memory allocation and storing only aggregated data.

4.1.3 Partition

HyMR uses shared memory to implement an all-to-all exchange of the, typically voluminous data, emitted from mappers. We merge all intermediate containers of each core in a single container stored in private memory. This container contains <key, list-of-values> pairs. We store distinct keys and for each key we assign a list of all values produced by all cores during *Map* stage. We then go through an iterative process where in each iteration, we modify the LUTs of a core to map to the *POPSHM* private address space of another core. The runtime system knows at execution time the starting physical address of each *POPSHM* segment. We use an Intel driver to map the physical addresses of each *POPSHM* segment to the virtual address space of user programs. We manage coherence explicitly, by marking the pages in *POPSHM* address space as non-cacheable by L2 cache. The SCC does not provide a hardware mechanism to flush or invalidate all cache lines in the L2 cache and a software implementation is prohibitively expensive. Given that all *POPSHM* pages are read-only in this stage, there is no need to flush the L2 caches. Therefore, the runtime just invalidates the MPB pages after each remapping. The remapping process requires as many iterations as the number of cores. To avoid contention when two or more cores access DRAM through the same memory controller, each core begins remapping from its local core's *POPSHM* and increases POPSHM index in a circular way. Figure 4.2 shows this algorithm using 4 cores as an example. This proces guarantess that memory traffic and contention are balanced between the memory controllers. Remapping *POPSHM* address spaces requires no synchronization.

4.1.4 Reduce

HyMR uses both the private and the shared address spaces to implement *reduce* stage. The input data of this stage stored in private memory of each core. We store the output data in the shared memory to execute the next stage. Before the execution of this stage each core has in its private memory a hash table of all <key, list-of-values> pairs on which it has to execute the user-defined *reduce* function. We iterate through each <key, list-of-values> pair and call the user specified reduce function on it. HyMR provides an iterator interface for the list-of-values that the user can use inside the *reduce* function. The result of each *reduce* call is an output key-value pair. We use shared memory to store these pairs in order to all cores can access these in the next stage.

4.1.5 Sort

The distributed address space implementation of MapReduce uses a binomial merge algorithm based on message passing. In HyMR, the output is stored in shared memory instead and all cores execute parallel sorting using regular Sampling (PSRS) [36]. The authors in [36] claim that if the input has no duplicate keys this algorithm has good load-balancing properties compared to the other parallel sorting algorithms. In MapReduce the input of this stage has no duplicate keys.

In PSRS, each core exports in shared memory an array of output key-value pairs. In this step, the runtime has to merge as many arrays as the number of cores into a single array, which is also sorted. Parallel sorting algorithms choose $c-1$ pivots and split the input into c partitions, c the number of cores. The cores exchange data to retrieve their respective partitons and sort each

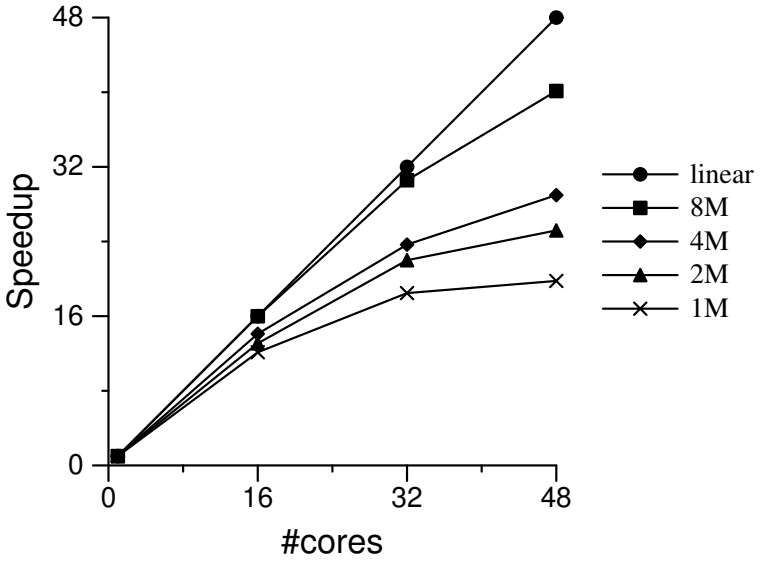

Figure 4.3: Speedup of PSRS implementation over sequential libc qsort.

partition locally. The selection of pivots is critical for load balancing.

PSRS has four stages. Assume that the runtime system must sort n keys on c cores. In the first phase, each core uses quicksort to sort its share of the elements, which is no more than $\lceil n/c \rceil$. Each core selects the data items with indices $0, n/c^2, 2n/c^2, ..., (c-1)(n/c^2)$ as a regular sample of its locally sorted block. In the second phase of the algorithm, one core gathers and sorts the local regular samples. It selects $c-1$ pivot values from the sorted list of regular samples. The pivot values are at indices $c + \lfloor c/2 \rfloor - 1, 2c + \lfloor c/2 \rfloor - 1, ..., (c-1)c + \lfloor c/2 \rfloor$ in the sorted list of regular samples. At this point each core partitions its sorted sublist into c partitions, using the pivot values as separators between partitions. In the third phase of the algorithm, cores exchange partitions. During the fourth phase, each cores merges its $c-1$ partitions with its private partition into a single list. The values on this list are disjoint from the values on the lists of other cores. At the end of this phase the elements are sorted in a single array.

We implement a hybrid address space version of PSRS using on-chip MPB buffers for synchronization data instead of using shared memory to minimize latency and achieve simple coherence maintenance. The authors in [36] propose that only one core (without loss of generality, core 0) can choose the samples and sort them to find the actual pivots. This method requires however 2 barriers. Since input data is read-only and PSRS is not in-place, we can lift the restriction that only one core chooses the pivots. All cores choose the pivots with the same PSRS algorithm, without synchronization. As all data reside in off-chip shared memory and all cores can access the data through LUTs, there is no need to execute an all-to-all exchange. The runtime system allocates space for the output array in shared memory and stores the sorted partitions into this array.

Figure 4.3 shows the speedup of the hybrid address space implementation of PSRS over the sequential libc qsort implementation. We use the same qsort implementation in the first phase of PSRS.

4.2 MapReduce Optimizations

HyMR uses several additional optimizations that leverage hybrid address spaces.

4.2.1 Optimizing On-Chip Barriers

We revisited several scalable barrier algorithms presented in [37], to explore how these algorithms perform and should be revised in the presence of a private, on-chip address spaces with fast communication paths that do not involve off-chip memory. We implemented the algorithms with on-chip data transfers, keeping shared data (e.g. counters) in the MPB buffers and using

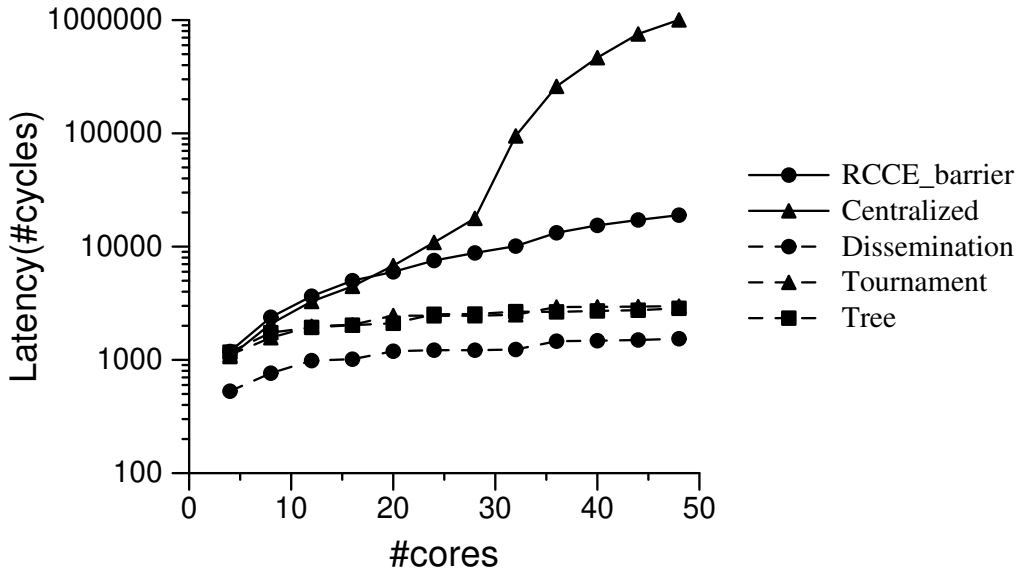

Figure 4.4: Comparison of barrier algorithms in SCC.

the cacheable private address space of each core otherwise. We leverage the on-chip shared memory because the shared data needed to implement synchronization algorithms has a very small memory footprint. Furthermore, the runtime system can bypass the L2 cache and use the *CL1INVMB* instruction to invalidate data before reads, or the write no-allocate policiy with a write combining buffer for writes.

We experimented with the *Centralized Barrier*, *Tournament Barrier*, *Tree Barrier* and *Dissemination Barrier* from [37]. We compare these algorithms against the barrier implementation provided with RCCE named *RCCE_ barrier*. This is a simple, similar to a centralized, counter-based barrier with local sensing but instead of a single counter, each core has its own local counter stored in MPB buffers. This reduces the contention in MPB memory compared with *Centralized Barrier*. Figure 4.4 compares the barrier implementations. In the *Centralized Barrier* all shared data is stored in a single MPB.

The latency that each core expends to access that MPB depends on the number of hops in the SCC 2D mesh interconnect. The *Centralized Barrier* algorithm is ill-suited for many-core processors with distributed on-chip memory. The *RCCE_barrier* has the disadvantage that a single root core must update a flag on each other core that participates in the barrier. All other algorithms distribute shared data between MPB buffers in a way that minimizes accesses to remote MPB buffers. Figure 4.4 indicates that the *Dissemination Barrier* algorithm fits the SCC best. In [37] the authors show that *Dissemination Barrier* has the lowest latency compared with the other algorithms. We show that *Dissemination Barrier* also fits on SCC distributed on-chip memory architecture. The *Dissemination Barrier* algorithm executes $\log c$ rounds to propagate arrival and departure notifications between cores using point-to-point communication between MPBs.

4.2.2 Interrupt-less Work-Stealing

On the SCC, the latency for accessing DRAM depends on the number of hops that the access must traverse in the chip's 2D mesh until it reaches a specific memory controller that serves all accesses from the issuing core. In memory-intensive applications this architectural feature can introduce load imbalance. We implement a work stealing algorithm inspired by Cilk [38], using however the MPB to implement fast, on-chip communication between the local core schedulers. We implement scheduling dequeues as non-cacheable queues and preserve coherence for the state of dequeues using explicit invalidation of entire MPB buffers. We use this work-stealing only in *Map* stage. The other stages can be balanced using a good hash function in *Map* stage. Although we implement *Map* stage using distributed memory we choose to implement work-stealing using on-chip shared-memory (MPB buffers). Using shared-

memory the thief can get a portion of work from the victim without interrupt it's execution. The thief choose victims randomly.

Chapter 5

Experimental Analysis

We compare HyMR to the reference distributed address space implementation, which we refer to as DiMR. We perform further comparisons with Phoenix++, a state-of-art implementation of MapReduce for multi-core systems with hardware-supported cache coherence [12]. We use four benchmarks which are representative of MapReduce applications with a varying number of distinct intermediate keys:

- *WordCount* counts the number of occurrences of each word in text files. The map function splits the input text into words, whereas the reduce function sums the number of occurrences of each word to produce a final count. The number of distinct intermediate keys is the number of distinct words in the text files.

Application	**Input size**
WordCount	400 MB
Histogram	1.6 GB
LinearRegression	400 MB
MatrixMultiply	$2048 * 2048$ Matrices

Table 5.1: MapReduce application workloads

- *Histogram* counts the frequency of occurrences of each RGB color component in an image file. The map function emits the occurrences of each color component in pixels and the reduce function produces the sum of occurrences of each component. The maximum number of distinct intermediate keys is 3×256.

- *LinearRegression* computes a line of best fit for a set of points, given their 2D coordinates. Map computes intermediate summary statistics for the points like the sum of squares, while reduce gathers all data of each of the summary statistics and calculates the best fit. This benchmark exports 5 intermediate keys.

- *MatrixMultiply* multiplies two dense matrices of integers. In this benchmark the *Map* function implements the matrix multiplication kernel and does not emit any intermediate data. The runtime splits the input and each chunk is a row of each input matrix. The runtime also uses work-stealing to balance the load between the available cores.

We choose benchmarks that vary in the number of distinct intermediate keys that they produce, to stress different stages of the MapReduce runtimes. *WordCount* represents one extreme case, by exporting as many number of intermediate keys as the number of words in the input text files. *MatrixMultiply* represents the other extreme, since it does not produce any intermediate keys. *Histogram* and *LinearRegression* are between these limits. *Histogram* exports from 0 to 768 distinct intermediate keys depending on the input. *LinearRegression* exports 5 distinct intermediate keys for every input. Benchmarks that emit a large number of intermediate keys stress the *Combine*, *Rearrange* and *Merge* stages. In the other hand of no intermediate keys we stress the *Map* stage.

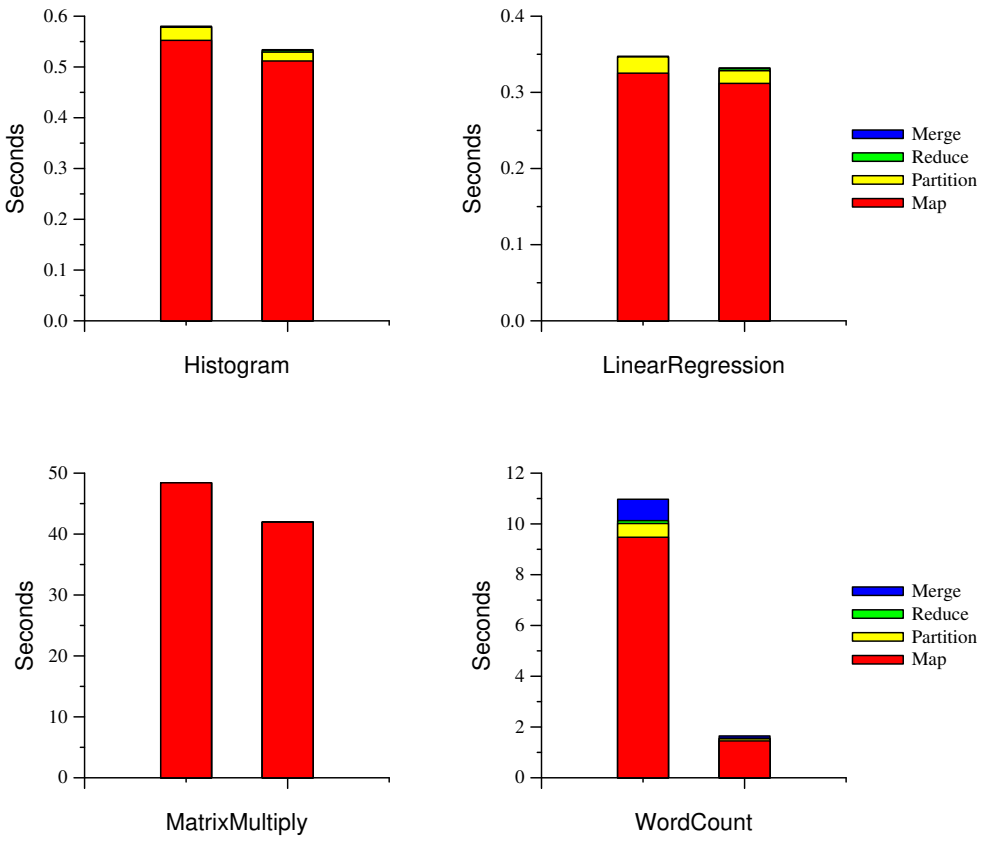

Figure 5.1: DiMR (left bar) vs. HyMR (right bar) performance

Table 5.1 lists the MapReduce application workloads that we used for experiments. We use an SCC node, where each tile of cores runs at a frequency of 800MHz, the mesh interconnect runs at a frequency of 800MHz and DRAM runs at a frequency of 800MHz. We use sccKit 1.4.1.3 and each core runs Linux kernel version 2.6.38. We use version 4.5.2 of GCC and G++ compilers.

Application	Partition Speedup	Merge Speedup
WordCount	6.64×	9.61×
Histogram	1.48×	0.69×
Linear Regression	1.28×	0.78×
Matrix Multiply	1.00×	1.00×
GeoMean	1.88×	1.50×

Table 5.2: Speedup for partition and merge stages computed using DiMR execution time over HyMR execution time using 48 cores.

5.1 Message-Passing vs. Hybrid-Adress-Spaces

We first compare DiMR (Section 3) to HyMR (Section 4), in terms of aboslute performance. *WordCount* generates the largest number of distinct intermediate key-value pairs among the benchmarks, thus stressing the *Combine*, *Partition* and *Merge* phases of MapReduce. Figure 5.1 shows the breakdown of execution time of each benchmark with DiMR (left) and HyMR (right). For these results, we use 48 cores of the SCC. We note that in all cases, execution time is dominated by the *Map* stage. This indicates that both DiMR and HyMR have been heavily optimized to avoid bottlenecks during communication-intensive stages, such as partitioning and sorting [24]. The *Map* stages includes the *Map* and *Combine* phases in our implementation for both runtimes. With hybrid memory, we use work stealing and the HyMR's optimized combiner. These two optimizations justify why HyMR *Map* is faster than the DiMR *Map*. HyMR also uses a global address space in shared memory for the *Partition* stage. This allows the runtime system to use hash table with open addressing to store intermediate data. This data-structure enable the implementation of a more efficient combiner. In DiMR, the runtime system stores intermediate data as raw data and the processing of this data adds several overheads.

The workload of tasks in the *Map* stage is not the same across tasks.

Tasks exhibit variation in their execution time for different chunks of input data, thus load-balancing is necessary in a MapReduce runtime system. A shared address space enables an efficient implementation of interupt-less load-balancing in HyMR using work-stealing.

The *Partition* stage is based on a all-to-all exchange, implemented with message passing in DiMR, but on shared memory, through LUT remapping, in HyMR. Table 5.2 shows the speedup that shared memory all-to-all exchange achieves over message passing all-to-all exchange for all benchmarks, using 48 cores. These results illustrate that a cache-bypassing, all-to-all exchange in place in shared memory performs better in all cases. Benchmarks with many intermediate keys have larger performance gains. In *MatrixMultiply*, the only exception, neither runtime executes the *Partition* stage.

HyMR and DiMR have identical implementation of the *Reduce* stage. In the *Merge* stage, DiMR uses the binomial merge algorithm whereas HyMR uses sorting with regular sampling. Table 5.2 shows the speedup that HyMR achieves over DiMR during the *Merge* stage, for all benchmarks using 48 cores. *WordCount* has the largest number of output keys and the performance gain is the most significant in comparison to other benchmarks. *Histogram* and *LinearRegression* indicate a small slowdown from using hybrid address spaces. *MatrixMultiply* does not execute a *Merge* stage.

5.2 Scalability

Overall, HyMR consistently outperforms DiMR on the SCC. To compare HyMR with Phoenix++, we evaluate the latter on a 48-core cache-coherent multi-processor, with 4 AMD Opteron 6172 processors running at 2.1GHz and 64GB of DRAM. This system runs Linux version 2.6.32 and the 4.7.0

version of GCC and G++ compilers. Our comparison is not a direct one, as the SCC and AMD systems have fundamentally different processors, memory management units and communication substrates. While the cache-coherent AMD system would support distributed memory and hybrid address space implementations, these implementations would all be underpinned by the hardware coherence protocol, which would render message passing with direct core-to-core communication, as in the SCC, infeasible. Conversely, a shared memory implementation of the runtime system on SCC would require a software virtual memory coherence protocol, which is hard to scale on many cores. It is for these reasons that we compare MapReduce implementations on different platforms and use two metrics that partially neutralize the underlying architecture: scalability, percentage of peak data processing bandwidth (bandwidth utilization) achieved by each implementation.

Figure 5.2 indicates that in all cases HyMR achieves almost linear speedup whereas Phoenix++ encounters scalability bottlenecks, usually at 32 cores. In both HyMR and Phoenix++, the execution time dominated by the *Map* stage (Figure 5.1), which includes the *Combine* stage in both implementations. These stages are fully parallel, with no application data communication and low synchronization activity between cores. The actual problem of Phoenix++ is false sharing, as an effect of data structure layout and the the hardware-supported cache-coherence protocol. We use distributed memory during *Map* and *Combine* stages. This allows us to remove the false sharing problem. Interestingly, the scaling gap between HyMR and Phoenix++ increases with the number of cores.

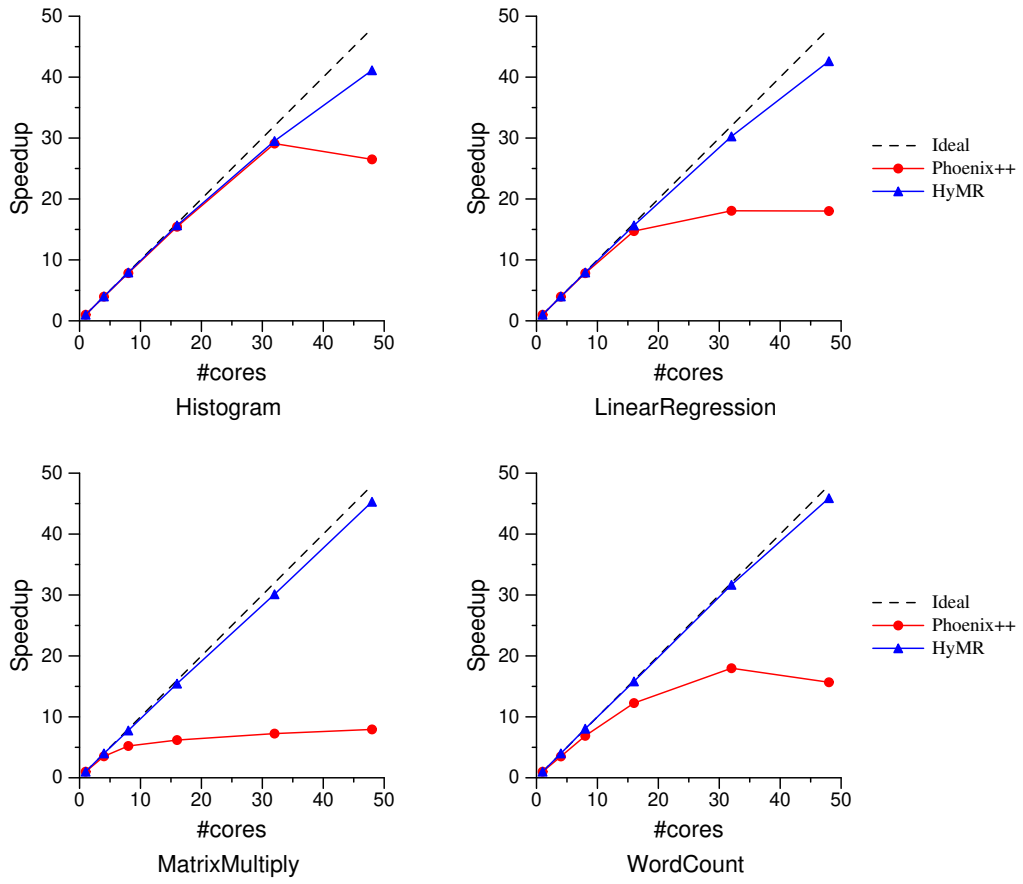

Figure 5.2: Speedup of benchmarks on the SCC (using HyMR) and AMD (using Phoenix++) systems.

5.3 Sustained to Peak Bandwidth

As MapReduce fundamentally targets data-intensive applications, the data processing bandwidth of the MapReduce runtime system is a proper metric for evaluation. We compare the bandwidth that each benchmark achieves normalized to the peak data streaming bandwidth in each of our two platforms. In both cases we measure the peak bandwidth using the STREAM benchmark [39, 40] (Triad case). Figure 5.3 shows the peak bandwidth that each system achieves, as reported by the STREAM benchmark. AMD

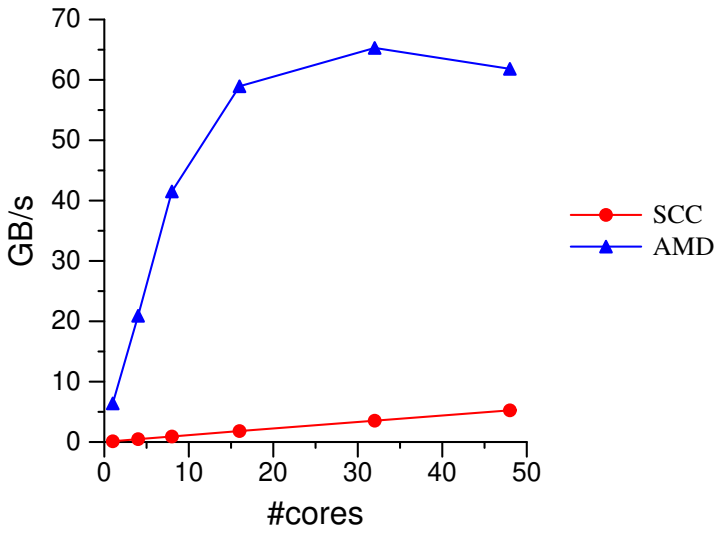

Figure 5.3: Comparison between HyMR (SCC) and Phoenix++ (AMD) bandwidth utilization.

Opteron cores run in 2.1GHz and use 64GB DRAM clocked at 1333MHz, while and SCC cores in 800MHz and use 32GB DRAM clocked at 800MHz. AMD Opteron processors also have a significantly more efficient ALU than the outdated Pentium-class cores used on the SCC. These differences justify the gap in available memory bandwidth between the two architectures. Despite this difference, we note that available bandwidth scales well with the number of cores on the SCC but reaches a point of saturation at 32 cores on the AMD system.

We measure the bandwidth that each benchmark achieves with HyMR and Phoenix++. We normalize the measurements with the peak bandwidth of the platform on which each runtime executes. This is an efficiency metric with an ideal value of 1. Figure 5.4 shows that in WordCount, Histogram and LinearRegression the bandwidth efficiency of HyMR exceeds the efficiency of Phoenix++. Phoenix++ achieves higher bandwidth efficiency only in *MatrixMultiply*, where the required memory bandwidth is at any rate low, as

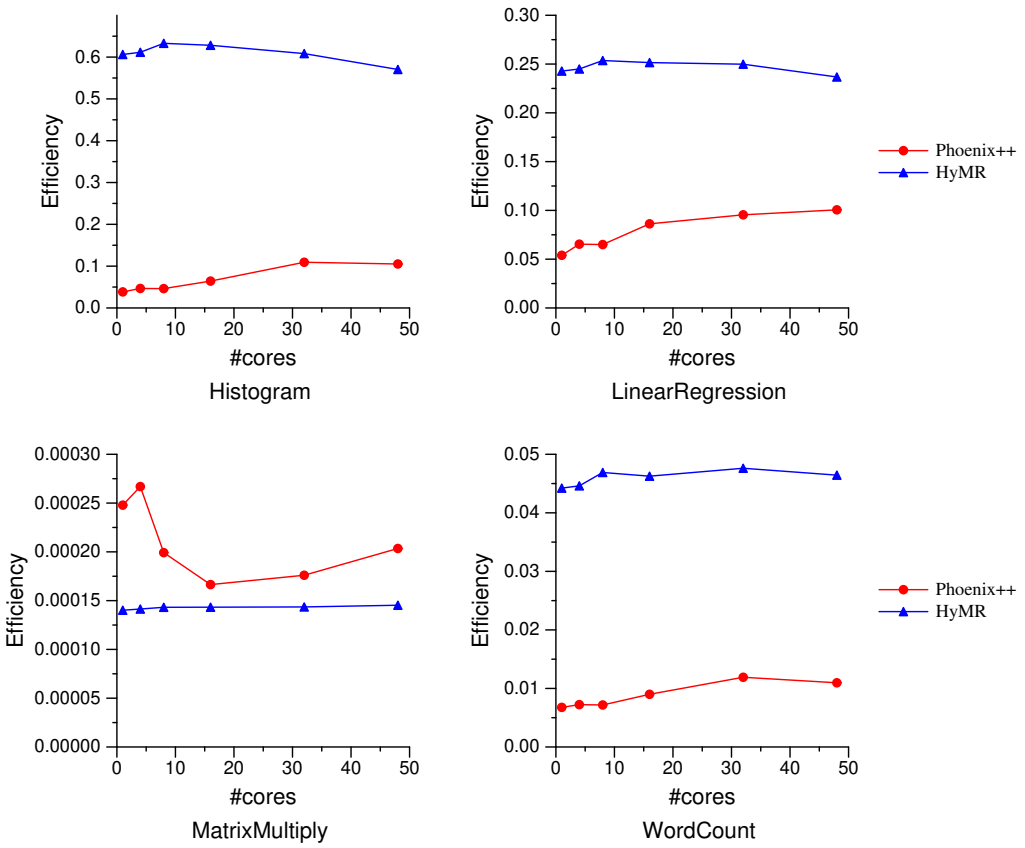

Figure 5.4: Bandwidth efficiency for our benchmarks.

the benchmark exhibits excellent locality. On average HyMR achieves 3.18× better bandwidth efficiency than Phoenix++ on 48 cores.

Chapter 6

Related Work

Several prior research efforts ported MapReduce to prominent hardware platforms for high-performance computing, including cache-coherent multicore processors [20, 21, 12, 22, 30] and non cache-coherent multicore processors [23, 24].

Phoenix, a port of MapReduce for cache-coherent shared-memory multicore systems [20, 21, 12], exploits locality implicitly by controlling the granularity of tasks and the assignment of tasks to cores. Phoenix performs dynamic assignment of map and reduce tasks to cores. It controls task sizes so that the working set of each task fits in the L1 cache of each core. Phoenix also provides an option to perform prefetching in the L2 data cache. The main focus in the design of Phoenix is on achieving scalability through NUMA-aware memory management. Each map thread emits intermediate results on a space allocated on the closest memory module to the CPU the thread is scheduled on. In [21], the authors use a multi-layer approach to optimize the runtime system. These layers include the algorithm, the implementation and and the runtime-OS interaction. In the most recently published version of Phoenix [12] the authors provides a modular, flexible pipeline that can

be easily adapted by the user to the characteristics of a particular workload while allowing users to write simple, strict MapReduce code. In [30] the authors explore the design of the MapReduce data structures for grouping intermediate key/value pairs. A different approach to optimize Phoenix is proposed in [22] where the authors use "tiling strategy" to minimize task memory footprints and improve cache locality. HyMR differs from Phoenix in that it leverages both distributed and shared address spaces on-demand, to improve scalability. However, the design and implementation of HyMR do not prevent the horizontal (cache-level) or vertical (NUMA DRAM-level) locality optimizations implemented in Phoenix++.

High-performance implementations of MapReduce have also been available on systems with distributed address spaces, most notably the Cell BE processor [23, 24]. In these implementations, the runtime system controls locality explicitly, using DMAs and software prefetching via multi-buffering in the map, merge and sort stages. Contrary to Phoenix, the runtime system does not hash and does not partition keys in per-core buffers, thereby eliminating memory copies, while still allowing a balanced distribution of work during the sort and reduce stages. HyMR, contrary to the prior implementations of MapReduce on Cell, leverages both distributed and shared address spaces. The use of shared address space with cache bypassing in HyMR enables more efficient exchanges of large volumes of data between cores.

Chapter 7

Conclusions

This thesis presents the design and implementation of MapReduce runtime system using *hybrid address spaces*. Many-core processors such as the SCC processor provide communication pathways through distributed address spaces or shared address spaces, both on-chip and off-chip. The idea elaborated in this work as to use distributed address spaces in runtime system stages where cores share nothing in terms of application data and only need to exchange control messages for the purposes of scheduling and load balancing. The absence of a hardware cache coherence protocol allows runtime systems to scale almost perfectly in share-nothing stages. On the contrary, runtime stages where cores exchange singificant volumes of application data are best implemented in an off-chip shared address spaces. Where data is streamed and there is no opportunity for data reuse, bypassing caches is the most performant implementation option.

This thesis further argued that in staged runtime systems, an application-specific implementation of cache coherence is scalable and performant. In MapReduce specifically, the *Map* and *Reduce* stages are embarrassingly parallel and running them over a cache coherence protocol results in a perfor-

mance hit.

The techniques presented in this thesis can be used to implement domain-specific scalable runtime systems and scalable applications in future homogeneous many-core processors without hardware cache coherence, such as Intel's Runnemeede [41].

Bibliography

[1] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Commun. ACM*, vol. 55, no. 7, pp. 78–89, Jul. 2012. [Online]. Available: http://doi.acm.org/10.1145/2209249.2209269

[2] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn, "GPGPU: general purpose computation on graphics hardware," in *ACM SIGGRAPH 2004 Course Notes*, ser. SIGGRAPH '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1103900.1103933

[3] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core scc processor: the programmer's view," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.53

[4] J. Kahle, "The Cell Processor Architecture," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 38. Washington, DC, USA: IEEE Computer Society, 2005, pp. 3–. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2005.33

[5] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang, "Supporting openmp on cell," *Int. J. Parallel Program.*, vol. 36, no. 3, pp. 289–311, Jun. 2008. [Online]. Available: http://dx.doi.org/10.1007/s10766-008-0073-6

[6] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han, "COMIC: a coherent shared memory interface for Cell BE," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 303–314. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454157

[7] M. Houston, J.-Y. Park, M. Ren, T. Knight, K. Fatahalian, A. Aiken, W. Dally, and P. Hanrahan, "A portable runtime interface for multi-level memory hierarchies," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 143–152. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345229

[8] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a programming model for the Cell BE architecture," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188546

[9] K. Chapman, A. Hussein, and A. L. Hosking, "X10 on the single-chip cloud computer: porting and preliminary performance," in *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, ser. X10 '11. New York, NY, USA: ACM, 2011, pp. 7:1–7:8. [Online]. Available: http://doi.acm.org/10.1145/2212736.2212743

[10] S. Lankes, P. Reble, O. Sinnen, and C. Clauss, "Revisiting shared virtual memory systems for non-coherent memory-coupled cores," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '12. New York, NY, USA: ACM, 2012, pp. 45–54. [Online]. Available: http://doi.acm.org/10.1145/2141702.2141708

[11] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[12] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: modular mapreduce for shared-memory systems," in *Proceedings of the second international workshop on MapReduce and its applications*, ser. MapReduce '11. New York, NY, USA: ACM, 2011, pp. 9–16. [Online]. Available: http://doi.acm.org/10.1145/1996092.1996095

[13] S. G. Kavadias, M. G. Katevenis, M. Zampetakis, and D. S. Nikolopoulos, "On-chip communication and synchronization mechanisms with cache-integrated network interfaces," in *Proceedings of the 7th ACM international conference on Computing frontiers*, ser. CF '10. New York, NY, USA: ACM, 2010, pp. 217–226. [Online]. Available: http://doi.acm.org/10.1145/1787275.1787328

[14] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, and et al., "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*. IEEE, Feb. 2010, pp. 108–109.

[15] "The SCC Programmer's Guide," Revision 1.0.

[16] I. A. C. Ureña, M. Riepen, and M. Konow, "Rckmpi - lightweight mpi implementation for intel's single-chip cloud computer (scc)," in *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 208–217. [Online]. Available: http://dl.acm.org/citation.cfm?id=2042476.2042500

[17] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-Reduce for Machine Learning on Multicore," in *NIPS'06: Proc. of the 20th International Conference on Neural Information Processing Systems*, Vancouver, Canada, Dec. 2006, pp. 281–288.

[18] J. Lin and C. Dyer, "Data-intensive text processing with mapreduce," in *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, ser. NAACL-Tutorials '09. Stroudsburg, PA, USA: Association for Computational Linguistics, 2009, pp. 1–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1620950.1620951

[19] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in mapreduce," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, ser. MLG '10. New York, NY, USA: ACM, 2010, pp. 78–85. [Online]. Available: http://doi.acm.org/10.1145/1830252.1830263

[20] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2007, pp. 13–24.

[21] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2009, pp. 198–207.

[22] R. Chen, H. Chen, and B. Zang, "Tiled-MapReduce: Optimizing Resource Usages of Data-Parallel Applications on Multicore with Tiling," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 523–534.

[23] M. de Krujif and K. Sankaralingam, "Mapreduce for the Cell B.E. Architecture," *IBM Journal of Research and Development*, vol. 53, no. 5, Sep. 2009.

[24] A. Papagiannis and D. S. Nikolopoulos, "Rearchitecting mapreduce for heterogeneous multicore processors with explicitly managed memories," in *Proceedings of the 2010 39th International Conference on Parallel Processing*, ser. ICPP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 121–130. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2010.21

[25] B. Catanzaro, N. Sundaram, and K. Keutzer, "A Map Reduce Framework for Programming Graphics Processors," in *Proceedings of the Third Workshop on Software and Tools for Multicore Systems (STMCS)*, Apr. 2008.

[26] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce Framework on Graphics Processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 260–269.

[27] W. Ma and G. Agrawal, "A Translation System for Enabling Data Mining Applications on GPUs," in *Proceedings of the 23rd ACM International Conference on Supercomputing (ICS)*, Jun. 2009, pp. 400–409.

[28] "The Apache Software Foundation. Hadoop." [Online]. Available: http://hadoop.apache.org

[29] A. Verma, N. Zea, B. Cho, I. Gupta, and R. H. Campbell, "Breaking the mapreduce stage barrier," in *CLUSTER*. IEEE, 2010, pp. 235–244.

[30] Y. Mao, R. Morris, and M. F. Kaashoek, "Optimizing mapreduce for multicore architectures," *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, 2010.

[31] S. Rixner, *Stream processor architecture*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[32] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 74–85. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2005.6

[33] A. Papagiannis and D. S. Nikolopoulos, "Scalable Runtime Support for Data-Intensive Applications on the Single-Chip Cloud Computer," in *Proceedings of the 3rd Intel Many-core Applications Research Community Symposium (MARC)*, Jul. 2011, pp. 25–30.

[34] P. M. McIlroy, K. Bostic, and M. D. Mcilroy, "Engineering Radix Sort," *COMPUTING SYSTEMS*, vol. 6, pp. 5–27, 1993.

[35] R. Thakur and R. Rabenseifner, "Optimization of Collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, pp. 49–66, 2005.

[36] H. Shi and J. Schaeffer, "Parallel sorting by regular sampling," *J. Parallel Distrib. Comput.*, vol. 14, no. 4, pp. 361–372, Apr. 1992. [Online]. Available: http://dx.doi.org/10.1016/0743-7315(92)90075-X

[37] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991. [Online]. Available: http://doi.acm.org/10.1145/103727.103729

[38] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223. [Online]. Available: http://doi.acm.org/10.1145/277650.277725

[39] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[40] ——, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007, a continually updated technical report. http://www.cs.virginia.edu/stream/. [Online]. Available: http://www.cs.virginia.edu/stream/

[41] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu, "Runnemede: An Architecture for Ubuiquitous High Performance Computing," in *Proc. of the 19th IEEE International Symposium on High Performance Computer Architecture*, Shenzhen, China, Feb. 2013.