Hybrid address spaces: A methodology for implementing scalable high-level programming models on non-coherent many-core architectures

Hybrid Address Spaces: A Methodology for Implementing Scalable High-Level Programming Models on Non-Coherent Many-core Architectures

Anastasios Papagiannis[a,*], Dimitrios S. Nikolopoulos[b]

[a] *Foundation for Research and Technology – Hellas, Institute of Computer Science (FORTH–ICS), Heraklion, Greece*
[b] *School of Electronics, Electrical Engineering and Computer Science, Queens University of Belfast*

Abstract

This paper introduces hybrid address spaces as a fundamental design methodology for implementing scalable runtime systems on many-core architectures without hardware support for cache coherence. We use hybrid address spaces for an implementation of MapReduce, a programming model for large-scale data processing, and the implementation of a remote memory access (RMA) model. Both implementations are available on the Intel SCC and are portable to similar architectures. We present the design and implementation of HyMR, a MapReduce runtime system whereby different stages and the synchronization operations between them alternate between a distributed memory address space and a shared memory address space, to improve performance and scalability. We compare HyMR to a reference implementation and we find that HyMR improves performance by a factor of 1.71× over a set of representative MapReduce benchmarks. 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× respectively. We further evaluate our hybrid remote memory access (HyRMA) programming model and assess its performance to be superior of that of message passing.

Keywords: MapReduce; Single-Chip-Cloud; Resource management; Runtime systems; Parallel Programming Models; Hybrid Address Spaces; Message Passing; Partitioned Global Address Spaces

1. Introduction

Many-core processors use diverging memory architectures. Processors designed for mainstream computing markets tend to use memory hierarchies with private 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, processors designed for more specialized markets, such as high performance computing and large-scale data processing, use memory hierarchies without a coherence protocol. Graphics Processing Units (GPUs) [2], the Intel SCC [3] the Cell processor [4] and the experimental Runnemede prototype [5] are representative examples of non cache-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 deep understanding of the memory hierarchy to master. However, explicit communication may also improve performance, particularly in applications with regular communication patterns. Programmers often opt for a programming

*Corresponding author
Email addresses: apapag@ics.forth.gr (Anastasios Papagiannis), d.nikolopoulos@qub.ac.uk (Dimitrios S. Nikolopoulos)
URL: http://www.ics.forth.gr/~apapag (Anastasios Papagiannis), http://www.cs.qub.ac.uk/~D.Nikolopoulos (Dimitrios S. Nikolopoulos)

model based on explicit communication even on cache-coherent many-core processors [6, 7], to exploit the topology of the interconnection network and minimize communication overhead. Runtime systems can ease the burden of programming with explicit communication 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 high-level, shared address space model. In this case, the runtime system implements a virtual shared memory abstraction. Regardless of the choice of programming model, the runtime system is a critical component that largely defines performance, scalability and programmability.

Runtime systems for non cache-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 [8], COMIC [9] Sequoia [10], and CellSs [11] and the Intel SCC for the implementation of X10 [12] and Shared Virtual Memory models [13]. Intuitively, 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 threads in a program with little 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 paper introduces *hybrid address spaces*, as a fundamental design and implementation methodology for scalable runtime systems on non-coherent many-core architectures. The intuition behind hybrid address spaces is that a runtime system uses on-chip communication paths between private address spaces for small data transfers, such as those needed to exchange control data for scheduling, and off-chip communication paths through a shared address space, for large, streaming data transfers. To confirm our intuition, we present HyMR, an implementation of the MapReduce programming model [14] 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, or with a combination of distributed and shared address spaces. 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 cache-coherent many-core architectures. These contributions include:

- software-controlled staged memory coherence to minimize the overhead of coherence maintenance;
- application-specific, scalable data splitters;
- scalable, interrupt-less work-stealing for non-coherent architectures using exclusively on-chip communication;
- a new implementation of scalable on-chip barrier algorithms for non-coherent many-core processors;
- a new mechanism to enable fast access from a core to the private memory of another core on-chip, which accelerates global exchange operations;
- a parallel sorting algorithm that avoids synchronization between stages and executes critical communication paths using on-chip shared memory.

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, 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 [15]. HyMR achieves, on average, 3.1× improvement in speedup and 3.2× improvement of bandwidth efficiency compared to Phoenix++, on the same number of cores.

To further demonstrate hybrid address spaces as a viable methodology for implementing parallel programming models, we have also developed a hybrid remote memory access (HyRMA) programming model, which leverages message passing for on-chip, latency-sensitive data one-way transfers and global shared memory for one-way bulk data transfers. We demonstrate HyRMA with a representative stencil code, the Jacobi method. HyRMA improves performance by a factor of 2.41× using 48 cores, compared to a pure message passing approach.

The rest of this paper 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 presents the implementation and experimental analysis of HyRMA. Section 7 discusses related work and Section 8 concludes the paper.

2. Background

Hardware support for cache coherence on a processor with many cores increases complexity and power [16]. 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. The use of SCC is by no means limiting our study: our runtime system design and implementation techniques presented later in this paper generalize to any non-coherent many-core processor with programmable memory mapping/translation tables and a mechanism for explicit on-chip communication between cores. We conclude this section by providing background on MapReduce, a parallel programming model for large-scale data processing, inspired by functional languages.

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

The Intel SCC[1] [17] (Figure 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 private 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 cache coherence 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 [18], the latency to read a cache line from MPB buffers and off-chip DRAM are:

[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.

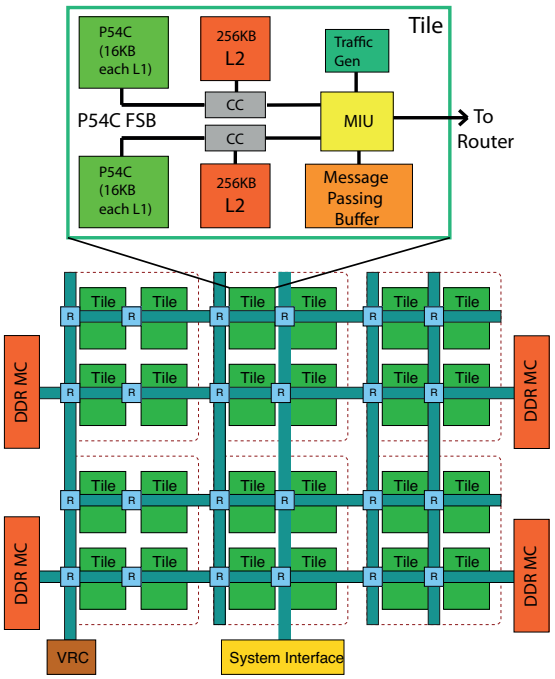

Figure 1: SCC processor diagram.

$$Local\ MPB = 45C_c + 8C_m \tag{1}$$

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

$$DRAM = 40C_c + 4 \cdot n \cdot 2C_m + 46C_r \tag{3}$$

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 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 means for implementing hybrid private and shared address spaces in the system.

Figure 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, 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

[2]For example, MCPC and the on-die network driver that allows TCP traffic from core to core.

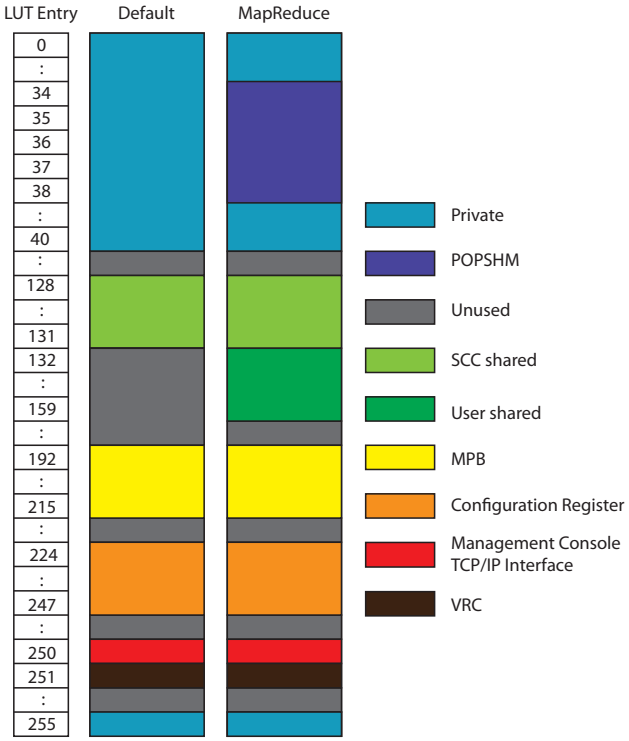

Figure 2: Default mappings of LUT entries at runtime.

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, including Intel's RCCE [3] and RCKMPI [19]. 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 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 message of size less than 8KB, 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 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 large messages into chunks of size up to 8KB. The alternative 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 message, 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.

Intel's RCCE library implements message passing using exclusively MPB buffers. On the other hand

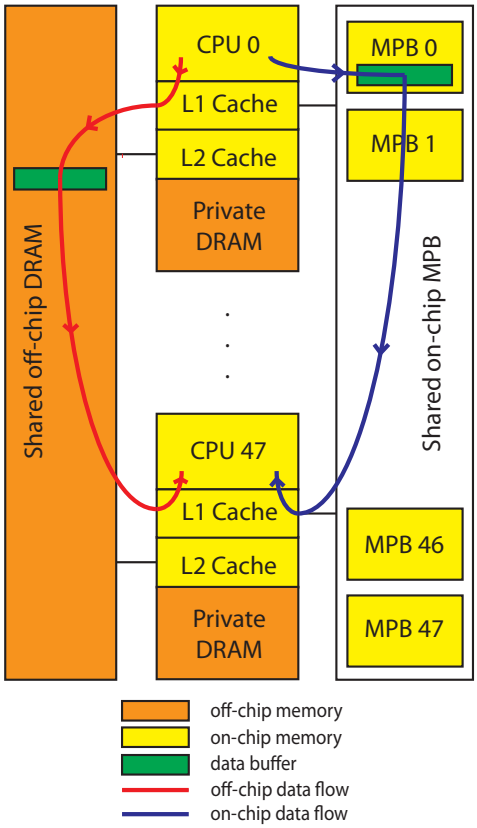

Figure 3: Message flow using off-chip DRAM and on-chip MPB.

RCKMPI uses MPBs for small messages and DRAM for large messages. 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

MapReduce is a set of language abstractions, inspired by Lisp [14], 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 [14, 20, 21, 22]. Practical implementations provide MapReduce abstractions as a library API or embed MapReduce in a high-level language, such as Java [23, 24, 15, 25, 26, 27, 28, 29, 30, 31].

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. 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.

Listing 1 shows a textbook MapReduce example that counts the number of occurrences of each word in a collection of documents [14]. The map function emits each word from the documents with a temporary count

```
1  // input: a document
2  // intermediate output: key=word; value=1
3  Map(String input) {
4      for each word w in input
5          EmitIntermediate(w, 1);
6  }
7
8  // intermediate output: key=word; value=1
9  // output: key=word; value=occurrences
10 Reduce(String key, Iterator values) {
11     int result = 0;
12     for each v in values
13         result += v;
14     Emit(key, result);
15 }
```

Listing 1: WordCount in MapReduce

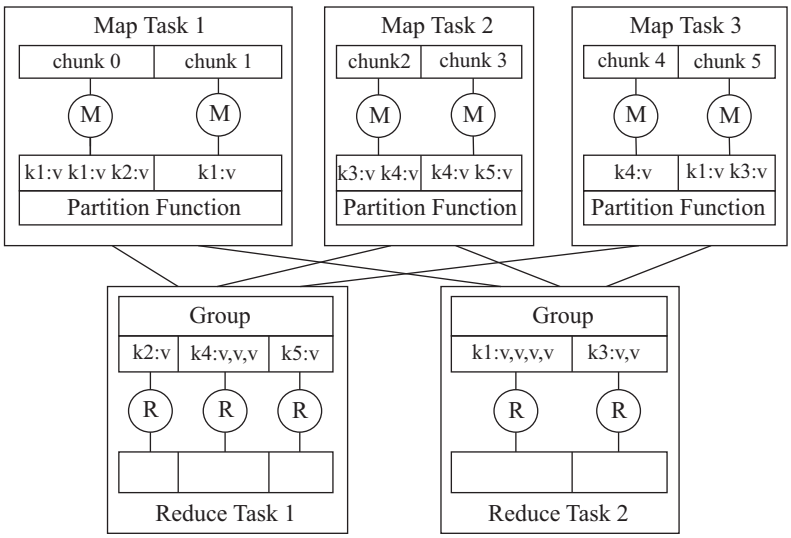

Figure 4: MapReduce workflow.

of occurrences set to 1. The reduce function measures the total number of occurrences for each unique word. The MapReduce program 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.

To MapReduce runtime system (Figure 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 in MapReduce parlance, 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, 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 system can

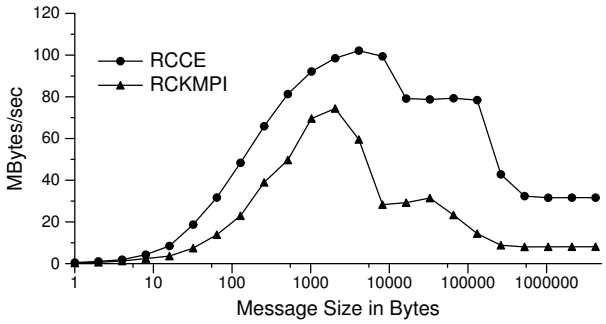

Figure 5: RCKMPI vs. RCCE bandwidth in a ping-pong benchmark

perform several additional optimizations: eliminate global synchronization between stages of MapReduce, using a dataflow execution model [32]; eliminate function call overheads by increasing the granularity of work units [23, 24]; reduce load imbalance also by adjusting the granularity of work units and/or the number of mappers and reducers [33]; optimize locality and overlapping computation with data transfers by prefetching work units [34]; and conserve bandwidth and cache space via hardware compression [35]. The runtime system can also provide scalable, application-specific fault tolerance, which is beyond the scope of this work.

3. DiMR Design and Implementation

To place HyMR in context, we first discuss 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]. RCCE implements all communication between cores through MPBs. We choose RCCE over RCKMPI due to superior performance. Figure 5 shows that native RCCE achieves better throughput than RCKMPI, when communication flows through the SCCMPB channel, which uses exclusively the on-chip message-passing buffers. A detailed description of the reference design is available in [36].

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 [27, 29, 23, 26]. Figure 6 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 partitions[3] 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-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* stage 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

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

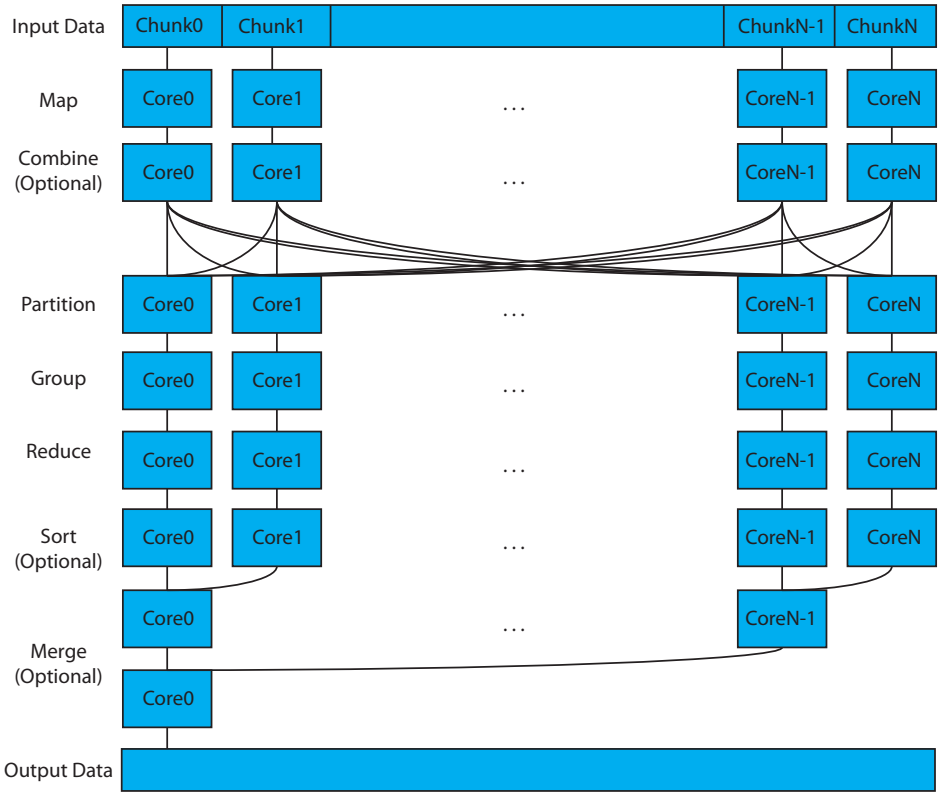

Figure 6: The flow of MapReduce Runtime using Message Passing.

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. Following the *Combine* stage, the runtime system synchronizes 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. DiMR 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 [36]. 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 $k = 1 \ldots p-1$ the core ranked 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 by each core in the *group* stage lies in the core's private memory and there is no need to exchange any messages between cores. Prior research [27, 29, 23, 26], uses generic sorting with a user-defined comparator to perform grouping in MapReduce. Our reference implementation uses a variant of radix sort [37] 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 7 shows a comparison of the libc quicksort implementation and our radix sort implementation for different input sizes. The measurements are from one core on the SCC. Radix sort outperforms quicksort, with one caveat. Radix sort sorts strings of bytes and can not use a user-defined comparator for sorting. This 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

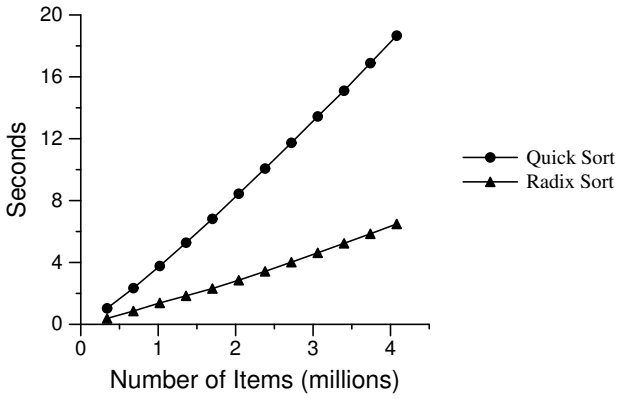

Figure 7: Libc qsort vs. radix sort, for a variable number of word-size elements.

MapReduce. In the common case, the data produced before the *reduce* stage is more than the data produced after the execution of the *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 execute locally on private memory and do not necessitate the exchange of messages between cores. An optional *merge* stage merges the output of all cores in one core. The reference implementation uses the binomial merge algorithm for this stage [38], which completes in *logn* steps. In each of these steps cores exchange the previously merged output data.

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, and off-chip communication paths through shared memory for performing transfers of large messages with 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 8 shows the stages of HyMR. HyMR has four stages, compared to DiMR's seven. HyMR merges the *Map* and *Combine* stages into a single stage and eliminates the *Group* stage. A new implementation of the *Map* stage allows the grouping of intermediate data before the *Reduce* stage. HyMR further merges the *Sort* and *Merge* stages into a singe *Sort* stage. As the *Sort* stage is implemented using a shared memory address space,there is no need to merge the sorted partitions. HyMR uses *Partition* and *Reduce* stages which are identical to the respective stages in the reference design.

HyMR implements application-specific memory coherence using the MapReduce execution stages as natural coherence boundaries and MapReduce stage semantics as hooks for coherence actions in the runtime system. The runtime system guarantees coherence at the completion of stages. HyMR flushes 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

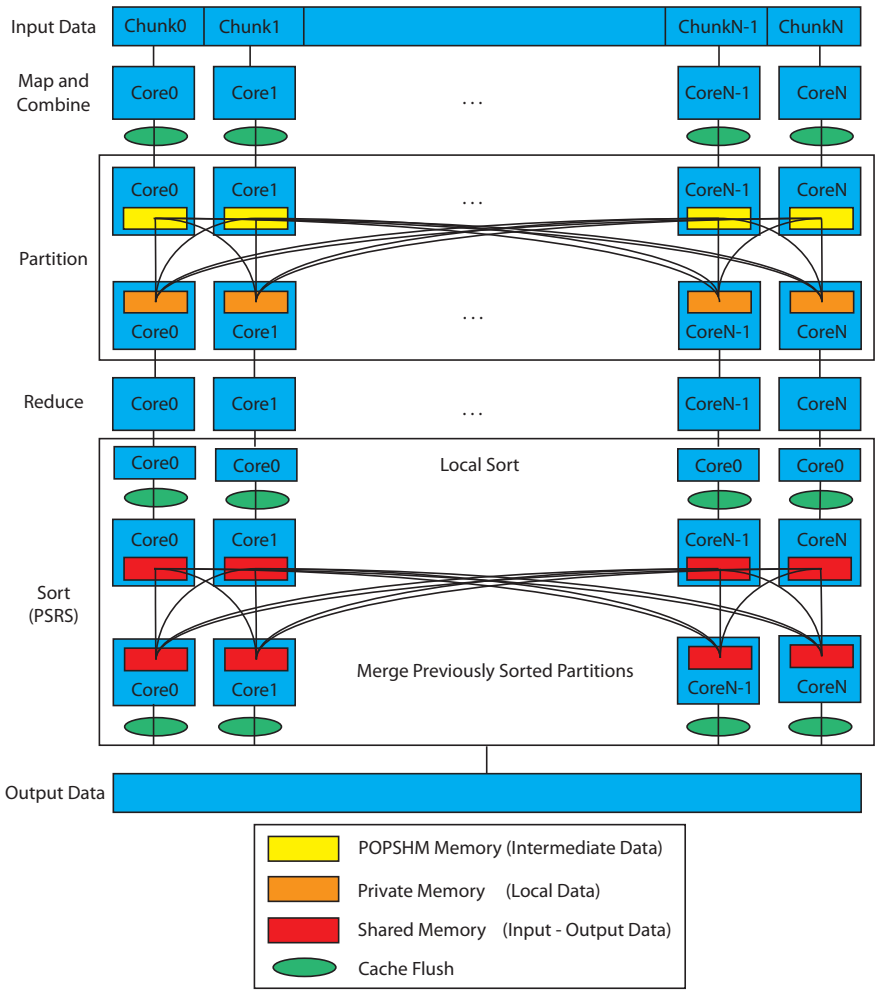

Figure 8: The flow of MapReduce Runtime using Hybrid Address Spaces.

a memory barrier. *Partition* and *Reduce* execute no coherence actions, as both these stages execute in distributed, private address spaces. To guarantee that all cores complete with *Reduce* stage we execute a barrier before the *Sort* stage. The *Sort* stage uses a parallel sorting algorithm with regular sampling (PSRS). PSRS executes in four sub-stages, (quicksort, local regular sample sorting, exchange and merge), separated by barriers. The runtime system flushes the caches of each core after the completion of quicksort and merge substages. We provide more details in Section 4.1.5.

4.1.1. Scalable Application-Specific Data Splitters

HyMR uses scalable input splitters over a shared address space. The input is stored in shared off-chip memory and is 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 its private 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 a prefix-scan algorithm running independently on each core, to identify the beginning of each core's chunk in the input without inter-core communication. The text and line splitters divide characters or text lines by default as evenly as possible between cores.

4.1.2. Map

Map tasks have no side effects and no dependencies between them [14]. Therefore, they are suitable for running in a distributed address space. No coherence actions are needed during the execution of the *Map* stage. Committing combined intermediate data to shared memory necessitates a flush of the L2 cache at the end of the *Map* stage, which includes a combiner. The runtime system stores the output of each mapper task running on a core in the core's POPSHM address space.

Each core executes mappers that process a queue of inputs provided from splitters. Mappers emit intermediate <key,value> pairs, using a 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, cacheable *POPSHM* address space for mapping data, as no coherence actions are necessary during this stage. 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. HyMR uses 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. The runtime system implements dynamic resizing of the hash table if a core exports more than 4096 intermediate <key,value> pairs, by doubling the size of the table when the fraction of used buckets in the table exceeds a predefined threshold (currently set to 0.8). HyMR's hashing uses quadratic probing to resolve collisions. Cores can not export more than five LUT entries (80MB) of intermediate data. The *POPSHM* implementation in the Linux kernel sets this as a hard limit. The runtime system uses a custom, fast memory allocator with pointer bumping and performs no deallocation in *POPSHM* address spaces.

HyMR combines the output of mappers, by reducing the data with a user-defined aggregator. The distributed memory implementation uses an all-to-all exchange at this stage. Implementing a combiner in DiMR would necessitate data marshaling (serialization and deserialization), which would add substantial communication overhead. HyMR on the other hand optimizes the combiner 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 cacheable shared memory to implement an all-to-all exchange of the voluminous, in the common case, data emitted from mappers. The runtime system merges all intermediate containers of each core in a single container stored in private memory. This container contains <key, list-of-values> pairs. HyMR stores distinct keys and for each key assigns a list of all values produced by all cores during *Map* stage. The runtime system then goes through an iterative process where in each iteration, it modifies 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. Coherence actions are avoided, by marking the pages in *POPSHM* address space as non-cacheable in the L2 cache. Given that all *POPSHM* pages are read-only in this stage and there is no physical data copying involved, there is no need to flush the L1 caches. An invalidation of the caches before each LUT remapping suffices for coherence. The runtime system avoids using the L2 cache in this stage because of the lack of an efficient, hardware supported invalidation mechanism. Therefore, the runtime system only has to invalidate the L1 cache 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 the POPSHM index round-robin. Figure 9 shows this algorithm using four cores as an example. This process guarantees that memory traffic and

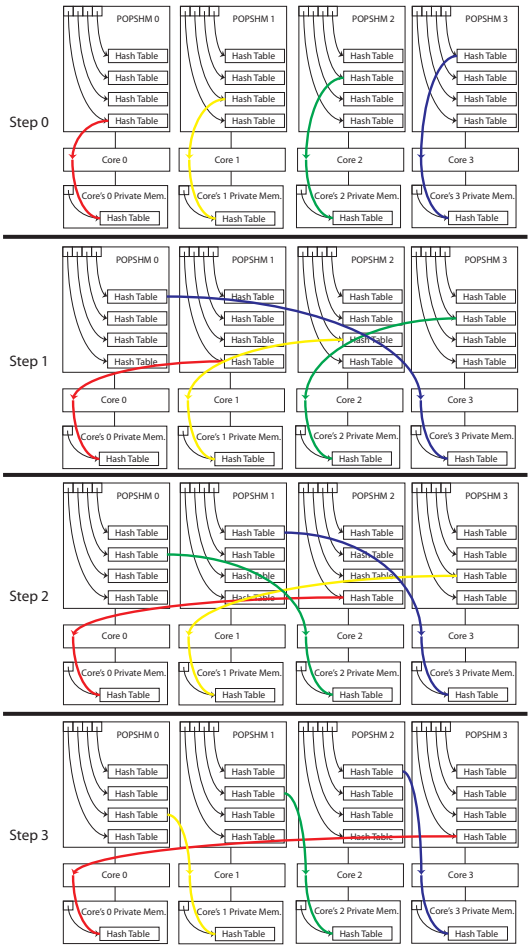

Figure 9: Contention avoidance in the *Partition* stage.

contention are balanced between the memory controllers. Remapping *POPSHM* address spaces requires no synchronization.

4.1.4. Reduce

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

4.1.5. Sort

DiMR uses a binomial merge algorithm based on message passing. In HyMR, the output is stored in cacheable shared memory instead and all cores execute parallel sorting using regular Sampling (PSRS) [39]. The authors in [39] claim that if the input has no duplicate keys this algorithm has good load-balancing

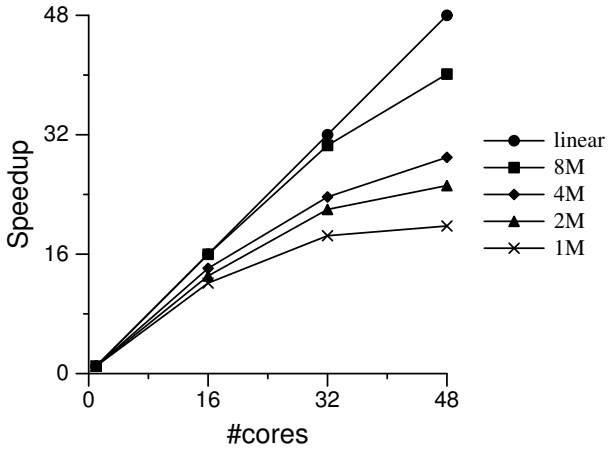

Figure 10: Speedup of PSRS implementation over sequential libc qsort.

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 partitions and sort each partition locally. The selection of pivots is critical for load balancing. A proof of the load balancing properties of this algorithm is provided in [39]

PSRS has four stages. Assume that the runtime system must sort n keys on c cores. In the first stage, each core uses quicksort to sort its share of the elements, which amounts to $\lceil n/c \rceil$ elements. 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 stage 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 stage of the algorithm, cores exchange partitions. During the fourth stage, 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 stage the elements are sorted in a single array.

HyMR implements a hybrid address space version of PSRS using on-chip MPB buffers for communication, instead of shared memory, to minimize latency and achieve simple coherence maintenance. Communication includes the addresses and sizes of intermediate buffers needed by the third stage of this algorithm. The authors in [39] 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 in this array.

Figure 10 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. HyMR MapReduce Optimizations

HyMR uses several additional optimizations that leverage hybrid address spaces.

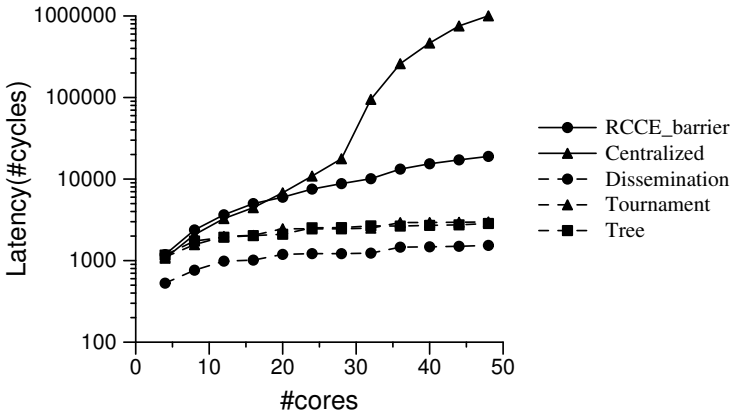

Figure 11: Comparison of barrier algorithms in SCC.

4.2.1. Optimizing On-Chip Barriers

We revisited the scalable barrier algorithms presented in [40], to explore how these algorithms perform and should be revised in the presence of 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 all shared metadata of each algorithm (e.g. counters) in the MPB buffers and using 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 and the write no-allocate policy with a write combining buffer for writes.

We experimented with the *Centralized Barrier*, *Tournament Barrier*, *Tree Barrier* and *Dissemination Barrier* from [40]. 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 implementation reduces the contention in MPB memory compared to the *Centralized Barrier* in [40]. Figure 11 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 11 indicates that the *Dissemination Barrier* algorithm is the best fit to the SCC, a result which confirms the result in [40] and generalizes it to chip multi-core processors with non cache-coherent memory.

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 [41], using however the MPB to implement fast, on-chip communication between the local core schedulers. Scheduling and work stealing are thus implemented using explicit communication between cores. 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 work-stealing only in the *Map* stage. Other stages are balanced with the choice of an appropriate hash function during the *Map* stage. Although we implement the *Map* stage using distributed address spaces, we choose to implement work-stealing using on-chip shared-memory (MPB buffers). Using the MPB on-

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

Table 1: MapReduce application workloads

chip shared-memory, a thief can get a portion of work from the victim without interrupting the victim's execution. Thieves choose victims randomly, as in Cilk.

5. Experimental Analysis

We compare HyMR to DiMR to validate the advantages of using hybrid address spaces over distributed address spaces and explicit communication, in the implementation of scalable runtime systems. We further compare HyMR to Phoenix++, a state-of-art implementation of MapReduce for multi-core systems with hardware-supported cache coherence [15]. We use four benchmarks which are representative of MapReduce applications:

- *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.
- *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 use the same MapReduce algorithms for these benchmarks as Phoenix++ does. This makes the algorithmic comparison of HyMR and DiMR more fair than if we chose to customize the algorithms to our implementation. 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, 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. On the other hand, benchmarks that produce no intermediate keys stress the *Map* stage.

Table 1 lists the MapReduce application workloads that we used for experiments. In order to run these benchmarks in-memory on our SCC board, we maximize the size of the input data sets so that the sum of input, intermediate and output data fits in shared DRAM. 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

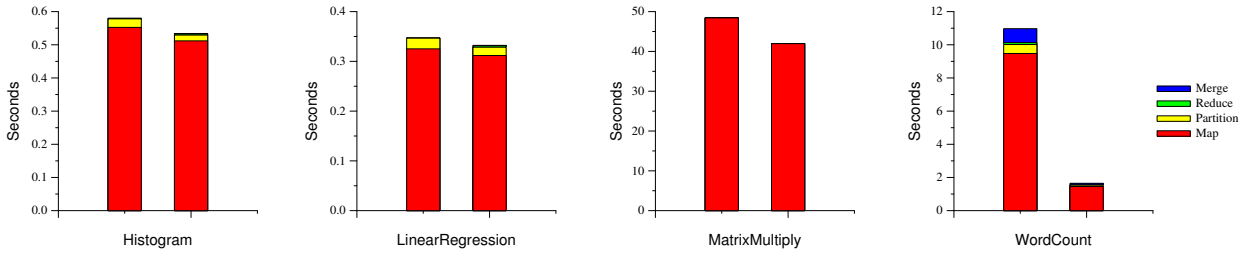

Figure 12: DiMR (left bar) vs. HyMR (right bar) performance

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

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.

5.1. Message-Passing vs. Hybrid-Address-Spaces

We first compare DiMR (Section 3) to HyMR (Section 4), in terms of absolute performance. *WordCount* generates the largest number of distinct intermediate keys among the benchmarks, thus stressing the *Combine*, *Partition* and *Merge* phases of MapReduce. Figure 12 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 [27]. 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 the 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 a hash table with open addressing to store intermediate data. This data structure enables 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 significant overhead.

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 interrupt-less load-balancing in HyMR using work-stealing and achieves more effective load balancing than the static data splitting.

The *Partition* stage is based on an all-to-all exchange, implemented with message passing in DiMR, but on shared memory and through LUT remapping in HyMR. Table 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, none of the two runtimes executes the *Partition* stage.

HyMR and DiMR have identical implementations of the *Reduce* stage. In the *Merge* stage, DiMR uses the binomial merge algorithm whereas HyMR uses parallel sorting with regular sampling. Table 2 shows

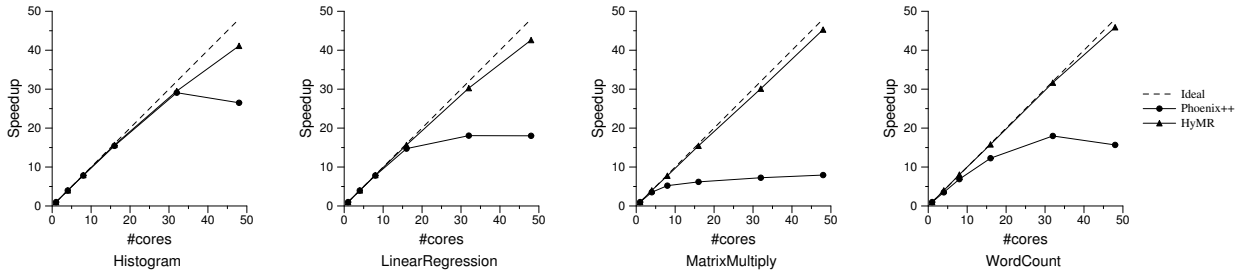

Figure 13: Speedup of benchmarks on the SCC (using HyMR) and AMD (using Phoenix++) systems.

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 in the *Merge* stage. *MatrixMultiply* does not execute the *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 in terms of speedup and percentage of peak data processing bandwidth (bandwidth utilization) achieved by each implementation.

Figure 13 indicates that in all cases HyMR achieves almost linear speedup whereas Phoenix++ encounters scalability bottlenecks, usually at 32 cores. To calculate speedup we use execution time with four cores as a baseline. We multiply this value by four to predict the execution time on one core, assuming that benchmarks scale perfectly up to 4 cores (a hypothesis that is confirmed in reality). We cannot obtain reasonable direct execution times on one core as the datasets used are too big to fit in the memory accessible to any core in the system. In both HyMR and Phoenix++, the execution time dominated by the *Map* stage (Figure 12), 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 authors in [15] evaluate Phoenix++ using an Intel machine consists of 4× Nehalem-EX processors with 4 NUMA nodes. We use 4× AMD Opteron 6172 processors, in a system with a more complicated NUMA design, which includes 8 NUMA nodes. Further experiments suggest that NUMA effects are more pronounced in AMD machines rather than in Intel machines. The results of Phoenix++ are sub-optimal due to inopportune data placement on NUMA nodes, despite that Phoenix++ is NUMA-aware by design. Another problem of Phoenix++ is false sharing, as an effect of data structure layout and the hardware-supported cache-coherence protocol. HyMR uses distributed memory during *Map* and *Combine* stages. This allows HyMR to solve the false sharing problem. The scalability gap between HyMR and Phoenix++ increases with the number of cores.

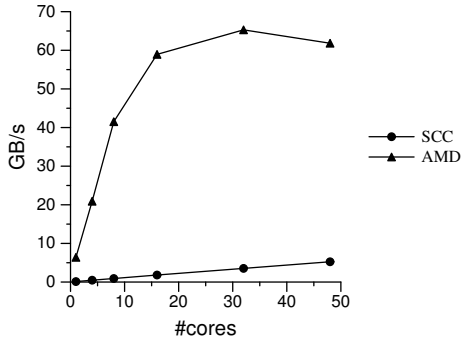

Figure 14: Comparison between HyMR (SCC) and Phoenix++ (AMD) bandwidth utilization.

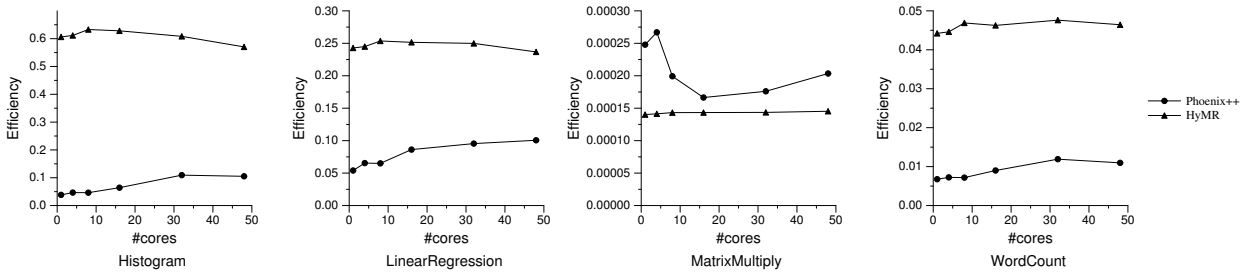

Figure 15: Bandwidth efficiency for our benchmarks.

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 [42, 43] (Triad case). Figure 14 shows the peak bandwidth that each system achieves, as reported by the STREAM benchmark. AMD 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 15 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 the benchmark exhibits excellent locality. On average HyMR achieves 3.18× better bandwidth efficiency than Phoenix++ on 48 cores.

5.4. Discussion

We analyze the reasons behind the performance gap between HyMR and DiMR in this section. DiMR uses the Intel RCCE, a lightweight message passing library optimized for the SCC. This library provides

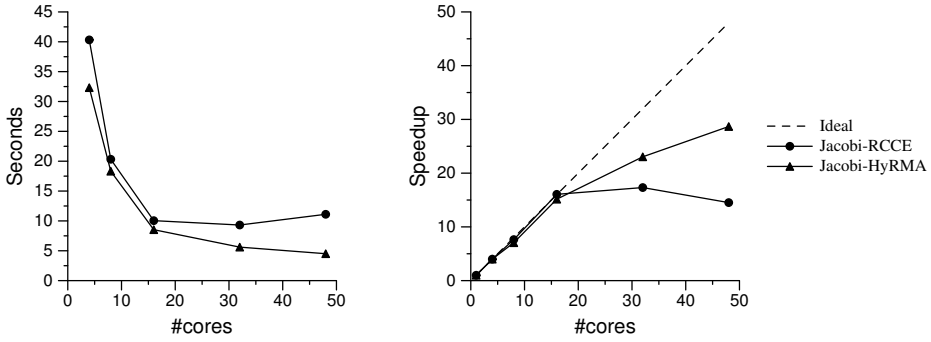

Figure 16: Jacobi execution time and speedup using RCCE and HyRMA.

the basic primitives RCCE_send, recv. In order to send a message the sender puts the message in the local MPB. After the necessary synchronization the receiver copies the data to its private L1 cache and then to DRAM. This results in large data transfers in the *Partition* stage in MapReduce. By contrast, in HyMR, all cores access data from the shared DRAM and thus avoid unnecessary data copies. The *Partition* stage is the main scalability bottleneck in DiMR. In order to move data between cores, data must be kept in raw buffers. This affects the performance and scalability of the *Map* stage as well. HyMR does not execute unnecessary data copies and uses more efficiently accessible data structures to store intermediate data. Furthermore, MapReduce can be implemented so that each processor accesses only private data, which in turn negates the need for cache coherence. Using shared DRAM and private processor caches without maintaining cache coherence is ideal in this scenario. In Section 6 we examine a different scenario where processors must access both private and shared data.

Synchronization messages used in MapReduce are small in size and may also prevent scaling. DiMR uses on-chip shared memory for synchronization. Although faster than shared DRAM, on-chip shared memory has limited size. Each core has its own cache hierarchy where both application data and synchronization metadata is loaded. In applications with frequent synchronization operations, synchronization metadata invalidate and flush application data out of the cache with severe performance implications. Bypassing the cache hierarchy and using MPB buffers for synchronization metadata and communication of short messages is the ideal choice for runtime systems based on message passing. Conversely, not using MPB buffers for application data improves communication and synchronization performance. Selective invalidation or flushing of data in specific addresses in the cache might alleviate this problem. HyMR uses on-chip MPBs and message passing for barriers and task queues with work-stealing.

Finally, the runtime system must be aware of the 2D-mesh interconnect of the processor. If a processor accesses data through a non-local memory controller or when several cores are accessing the same data simultaneously through the same memory controller introduces, memory accesses suffer from significantly increased latency. The optimization shown in Figure 9 resolves this problem, by balancing accesses to shared memory and each memory controller in the *Partition* stage.

6. HyRMA: Using Hybrid Address Spaces to Develop RMA Programming Models

In this section we show the effectiveness of hybrid address spaces in parallel applications using an RMA (Remote Memory Access) programming model. Our model implementation (HyRMA) allows the programmer to explicitly place data in shared cacheable DRAM or private non-cacheable scratch space. The model then uses one-way data transfers between any of these memory spaces to optimize communication paths. We use the Jabobi method as a use case to demonstrate this programming model.

Jacobi is an iterative algorithm which, given a set of boundary conditions, finds discretized solutions to differential equations of the form $\nabla^2\mathcal{A}+\mathcal{B}=0$. Each step of the algorithm replaces each node of a grid with

the average of the values of its nearest neighbors. To demonstrate the advantages of HyRMA we compare a version of Jacobi that uses message passing for communication with a HyRMA version that uses one-way transfers from global shared memory to on-chip caches and vice versa, as well as between on-chip MPBs.

6.1. Design & Implementation

Assume a Jacobi method for a two-dimensional $N \times N$ grid. To find a solution on the grid, the method repeatedly applies the following iterative step:

$$A_{i,j}^{k+1} = \frac{A_{i+1,j}^{k} + A_{i-1,j}^{k} + A_{i,j+1}^{k} + A_{i,j-1}^{k}}{4} \tag{4}$$

where i, j are indices on the two-dimensional array and k the iteration number. This step is applied until the method converges to a solution. For convergence testing, the method computes:

$$\text{diff} = \sqrt{\sum_{0 \le i,j < N} (A_{i,j}^{k+1} - A_{i,j}^{k}) \times (A_{i,j}^{k+1} - A_{i,j}^{k})} \tag{5}$$

and iterates until diff ≤ 0.01. To parallelize this algorithm we divide the rows of the two-dimensional array by the number of available cores. Each core gets a $\lceil N \times \frac{N}{\#cores} \rceil$ sub-array on which it can compute in parallel with other cores, at every iteration of the algorithm. Cores must exchange boundary data –upper and lower row– of their sub-arrays with their respective neighbors between iterations and check the error (convergence criterion), first locally and then cumulatively across all cores, to decide if more iterations are necessary for convergence.

6.1.1. Message Passing Implementation

In the message passing implementation of Jacobi we use RCCE_{send, recv} calls to exchange neighbor rows. To merge error values we use a customized implementation of MPI_AllReduce algorithm with RCCE primitives.

6.1.2. HyRMA

In the HyRMA implementation we store the array in cacheable shared DRAM, which accelerates the compute kernel of Jacobi. We distribute the data similarly to a partitioned shared address space approach. The data distribution in shared DRAM maximizes DRAM access locality from cores –equivalently, minimizes data transfer latency through the SCC on-chip interconnection network– and minimizes contention at memory controllers. However, using exclusively cacheable shared RAM implies that data exchange and synchronization between iterations would necessitate expensive cache flushing operations. We leverage hybrid address spaces to alleviate this problem, by allocating the boundary rows that cores must exchange in MPB buffers which are not cacheable in the L2 cache. We use direct one-way transfers to MPB buffers to implement the row exchanges between Jacobi iterations. We also use one-way transfers to MPBs to compute the cumulative error and check for convergence. We implement a based dissemination barrier (Section 4.2.1), also leveraging the MPBs.

6.2. Experimental Analysis

We compare the two implementations of Jacobi with message passing and HyRMA. Figure 16 shows the comparison of the two programming models. The left figure shows the execution time for 4 to 48 cores. The right figure shows speedup in the same range. The HyRMA implementation is faster than the message passing implementation and scales better as the number of cores increases. The main contributor to this difference is the reduction of communication latency via the use of on-chip one-way transfers for data exchanges and convergence checking. The optimal distribution of core-private data in the HyRMA version and the avoidance of costly data exchanges through the L2 caches and on-chip interconnect further contribute to the performance gap between HyRMA and message passing.

6.3. Discussion

The reason behind the low performance of the message passing implementation is redundant data copies. The RCCE library copies the data to the correct MPB buffer and then to DRAM for send/recv operations. These copies make data exchanging almost as expensive as computation, which in Jacobi is fully parallel. The impact of copying is more pronounced on 16 or more cores and affects speedup. HyRMA removes the need to copy data back to DRAM. The runtime system accesses data directly from MPBs to perform computation. The runtime system also bypasses the L2 caches and does not pollute them with useless data from the exchanges. Finally, accesses to MPB buffers are always performed to neighbor MPB buffers and thus minimize latency in the on-chip interconnect.

Despite the aforementioned optimizations HyRMA does not scale as well as HyMR. The reason is that accesses from MPB buffers bypass the L1 cache and incur additional latency. HyMR does not access application data from MPB buffers, which are used only for synchronization and load balancing. HyRMA necessitates cache bypassing for synchronization and data communication. Nevertheless, HyRMA still scales better than a message passing approach.

7. Related Work

Several prior research efforts ported MapReduce to prominent hardware platforms for high-performance computing, including cache-coherent multi-core processors [23, 24, 15, 25, 33] and non cache-coherent multi-core processors [26, 27].

Phoenix, a port of MapReduce for cache-coherent shared-memory multi-core systems [23, 24, 15], 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 [24], 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 [15] 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 [33] 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 [25] 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 [26, 27]. 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 neither hashes nor partitions keys in per-core buffers, thereby eliminating memory copies, while 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 a shared address space with cache bypassing in HyMR enables more efficient exchanges of large volumes of data between cores.

Recently, implementations of MapReduce using Partitioned Global Address Space Languages, such as X10 [44], and Unified Parallel C [45], have demonstrated superior performance to prior implementations based on Hadoop (for distributed address space systems) and Phoenix++ (for shared address space system). These implementations use a virtualized shared address space, thus missing opportunities to leverage maximally on-chip communication for latency-critical MapReduce operations. Furthermore, they delegate the control of scheduling and data transfers to the underlying language runtime system, which provides

generic, rather than MapReduce-specific memory coherence and consistency mechanisms, thus introducing additional performance inefficiencies.

Prior research has synthesized memory models and programming models to achieve more efficient parallelization on cluster architectures with SMP or multi-core nodes [46, 47, 48]. While these prior propositions provide abstractions of private and shared address spaces, they implement those address spaces on top of a common hardware substrate and do not customize the communication path for any given address space. Furthermore, the split address spaces in prior research are explicitly managed by programmers, a burden which we avoid in our work by implementing programming models that present a global address space abstraction to programmers but implement this abstraction using multiple physical address spaces and custom communication paths.

8. Conclusions

This paper presented a design and implementation of MapReduce using *hybrid address spaces*. Future and emerging many-core processors, such as Intel's Runnemeede [5], will provide communication pathways through distributed address spaces or shared address spaces, both on-chip and off-chip. The idea elaborated in this work is to use distributed address spaces in runtime system stages where cores share no application data and need to exchange only 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 significant volumes of application data are best implemented in an off-chip shared address space. Where data is streamed and there is no opportunity for data reuse, bypassing caches is the most performant implementation option. This paper further argues that in staged runtime systems, an application-specific implementation of memory coherence is scalable and performant. In MapReduce specifically, the *Map* and *Reduce* stages are embarrassingly parallel and running them over a hardware or software cache coherence protocol results in a consistent performance hit. We have also implemented an RMA programming model using hybrid address spaces and used a stencil code to prove its superiority to a message passing model.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the NanoStreams project, grant agreement n^o 610509 and the I-CORES project, grant agreement n^o 224759.

References

[1] M. M. K. Martin, M. D. Hill, D. J. Sorin, Why on-chip cache coherence is here to stay, Commun. ACM 55 (7) (2012) 78–89. doi:10.1145/2209249.2209269.
URL http://doi.acm.org/10.1145/2209249.2209269

[2] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, A. Lefohn, GPGPU: general purpose computation on graphics hardware, in: ACM SIGGRAPH 2004 Course Notes, SIGGRAPH '04, ACM, New York, NY, USA, 2004. doi:10.1145/1103900.1103933.
URL 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, 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, SC '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–11. doi:http://dx.doi.org/10.1109/SC.2010.53.
URL 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, MICRO 38, IEEE Computer Society, Washington, DC, USA, 2005, pp. 3–. doi:10.1109/MICRO.2005.33.
URL http://dx.doi.org/10.1109/MICRO.2005.33

[5] 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, 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, 2013.

[6] C. Zimmer, F. Mueller, NoCMsg: Scalable NoC-Based Message Passing, in: Proc. of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Chicago, IL, 2014.
[7] Tilera Processor Family, Tech. rep., www.tilera.com/products/ processors.php (2014).
[8] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, T. Zhang, Supporting openmp on cell, Int. J. Parallel Program. 36 (3) (2008) 289–311. doi:10.1007/s10766-008-0073-6.
URL http://dx.doi.org/10.1007/s10766-008-0073-6
[9] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, S. Han, COMIC: a coherent shared memory interface for Cell BE, in: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, PACT '08, ACM, New York, NY, USA, 2008, pp. 303–314. doi:10.1145/1454115.1454157.
URL http://doi.acm.org/10.1145/1454115.1454157
[10] M. Houston, J.-Y. Park, M. Ren, T. Knight, K. Fatahalian, A. Aiken, W. Dally, 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, PPoPP '08, ACM, New York, NY, USA, 2008, pp. 143–152. doi:10.1145/1345206.1345229.
URL http://doi.acm.org/10.1145/1345206.1345229
[11] P. Bellens, J. M. Perez, R. M. Badia, J. Labarta, CellSs: a programming model for the Cell BE architecture, in: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC '06, ACM, New York, NY, USA, 2006. doi:10.1145/1188455.1188546.
URL http://doi.acm.org/10.1145/1188455.1188546
[12] K. Chapman, A. Hussein, A. L. Hosking, X10 on the single-chip cloud computer: porting and preliminary performance, in: Proceedings of the 2011 ACM SIGPLAN X10 Workshop, X10 '11, ACM, New York, NY, USA, 2011, pp. 7:1–7:8. doi:10.1145/2212736.2212743.
URL http://doi.acm.org/10.1145/2212736.2212743
[13] S. Lankes, P. Reble, O. Sinnen, 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, PMAM '12, ACM, New York, NY, USA, 2012, pp. 45–54. doi:10.1145/2141702.2141708.
URL http://doi.acm.org/10.1145/2141702.2141708
[14] J. Dean, S. Ghemawat, Mapreduce: Simplified Data Processing on Large Clusters, Commun. ACM 51 (1) (2008) 107–113. doi:http://doi.acm.org/10.1145/1327452.1327492.
[15] J. Talbot, R. M. Yoo, C. Kozyrakis, Phoenix++: modular mapreduce for shared-memory systems, in: Proceedings of the second international workshop on MapReduce and its applications, MapReduce '11, ACM, New York, NY, USA, 2011, pp. 9–16. doi:10.1145/1996092.1996095.
URL http://doi.acm.org/10.1145/1996092.1996095
[16] S. G. Kavadias, M. G. Katevenis, M. Zampetakis, 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, CF '10, ACM, New York, NY, USA, 2010, pp. 217–226. doi:10.1145/1787275.1787328.
URL http://doi.acm.org/10.1145/1787275.1787328
[17] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, 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, 2010, pp. 108–109.
[18] The SCC Programmers Guide, Revision 1.0.
[19] I. A. C. Ureña, M. Riepen, 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, EuroMPI'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 208–217.
URL http://dl.acm.org/citation.cfm?id=2042476.2042500
[20] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Y. Yu, G. Bradski, A. Y. Ng, 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, 2006, pp. 281–288.
[21] J. Lin, 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, NAACL-Tutorials '09, Association for Computational Linguistics, Stroudsburg, PA, USA, 2009, pp. 1–2.
URL http://dl.acm.org/citation.cfm?id=1620950.1620951
[22] J. Lin, M. Schatz, Design patterns for efficient graph algorithms in mapreduce, in: Proceedings of the Eighth Workshop on Mining and Learning with Graphs, MLG '10, ACM, New York, NY, USA, 2010, pp. 78–85. doi:10.1145/1830252.1830263.
URL http://doi.acm.org/10.1145/1830252.1830263
[23] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, Evaluating MapReduce for Multi-core and Multiprocessor Systems, in: Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA), 2007, pp. 13–24.
[24] R. M. Yoo, A. Romano, 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), 2009, pp. 198–207.
[25] R. Chen, H. Chen, 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), 2010, pp. 523–534.
[26] M. de Krujif, K. Sankaralingam, Mapreduce for the Cell B.E. Architecture, IBM Journal of Research and Development 53 (5).

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

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

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

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

[31] The Apache Software Foundation. Hadoop.
URL http://hadoop.apache.org

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

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

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

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

[36] A. Papagiannis, 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), 2011, pp. 25–30.

[37] P. M. McIlroy, K. Bostic, M. D. Mcilroy, Engineering Radix Sort, COMPUTING SYSTEMS 6 (1993) 5–27.

[38] R. Thakur, R. Rabenseifner, Optimization of Collective communication operations in MPICH, International Journal of High Performance Computing Applications 19 (2005) 49–66.

[39] H. Shi, J. Schaeffer, Parallel sorting by regular sampling, J. Parallel Distrib. Comput. 14 (4) (1992) 361–372. doi:10.1016/0743-7315(92)90075-X.
URL http://dx.doi.org/10.1016/0743-7315(92)90075-X

[40] J. M. Mellor-Crummey, M. L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Trans. Comput. Syst. 9 (1) (1991) 21–65. doi:10.1145/103727.103729.
URL http://doi.acm.org/10.1145/103727.103729

[41] M. Frigo, C. E. Leiserson, 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, PLDI '98, ACM, New York, NY, USA, 1998, pp. 212–223. doi:10.1145/277650.277725.
URL http://doi.acm.org/10.1145/277650.277725

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

[43] J. D. McCalpin, Stream: Sustainable memory bandwidth in high performance computers, Tech. rep., University of Virginia, Charlottesville, Virginia, a continually updated technical report. http://www.cs.virginia.edu/stream/ (1991-2007).
URL http://www.cs.virginia.edu/stream/

[44] C. Zhang, C. Xie, Z. Xiao, H. Chen, Evaluating the performance and scalability of mapreduce applications on x10, in: Proceedings of the 9th international conference on Advanced parallel processing technologies, APPT'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 46–57.
URL http://dl.acm.org/citation.cfm?id=2042522.2042526

[45] C. Teijeiro, G. L. Taboada, J. Tourino, R. Doallo, Design and implementation of mapreduce using the pgas programming model with upc, in: Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems, ICPADS '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 196–203. doi:10.1109/ICPADS.2011.162.
URL http://dx.doi.org/10.1109/ICPADS.2011.162

[46] J. Zhu, J. Hoeflinger, D. Padua, A synthesis of memory mechanisms for distributed architectures, in: Proceedings of the 15th International Conference on Supercomputing, ICS '01, ACM, New York, NY, USA, 2001, pp. 13–22. doi:10.1145/377792.377799.
URL http://doi.acm.org/10.1145/377792.377799

[47] T. Liu, H. Lin, T. Chen, J. K. O'Brien, L. Shao, Dbdb: Optimizing dmatransfer for the cell be architecture, in: Proceedings of the 23rd International Conference on Supercomputing, ICS '09, ACM, New York, NY, USA, 2009, pp. 36–45. doi:10.1145/1542275.1542286.
URL http://doi.acm.org/10.1145/1542275.1542286

[48] X. Wu, V. Taylor, Performance characteristics of hybrid mpi/openmp implementations of nas parallel benchmarks sp and bt on large-scale multicore supercomputers, SIGMETRICS Perform. Eval. Rev. 38 (4) (2011) 56–62. doi:10.1145/1964218.1964228.
URL http://doi.acm.org/10.1145/1964218.1964228

Appendix A. PSRS Algorithm

Sizes	32cores ref	32cores new	48cores ref	48cores new
4M	26.49	23.66	23.56	28.97
8M	27.29	30.56	30.33	40.12

Table A.3: Speedup of new PSRS algorithm(new) and original PSRS algorithm(ref).

```
1  struct sub_array
2  {
3      size_t begin;
4      size_t length;
5  };
6
7  /* allocate on-chip shared memory of size bytes in id's MPB buffer */
8  void *mpballoc(int id, size_t size);
9
10 /* allocate off-chip shared memory of size bytes */
11 void *shmalloc(size_t size);
12
13 /* an PSRS algorithm to sort arrays of integer */
14 int *PSRS(int **array, size_t *array_size, int id, int num_cores)
15 {
16     int p = num_cores;
17     int p_1 = num_cores-1;
18     int pp_1 = num_cores*(num_cores-1);
19
20     size_t total_size = 0;
21     for(i = 0; i < p; i++)
22         total_size += array_size[i];
23
24     qsort(array[id], array_size[id]); /* sort the local partition */
25     cache_flush();  /* L2 cache flush */
26     barrier();
27
28
29     /* choose the samples */
30     int sample[pp_1];
31     for(i = 0; i < p; i++)
32     {
33         rsize = (array_size[i] + p_1)/p;
34         for(j = 0; j < p_1; j++)
35             sample[i*p_1+j] = array[i][(j+1)*rsize];
36     }
37
38     /* sort the samples */
39     qsort(sample, pp_1);
40
41     /* choose the pivots */
42     int pivots[p_1];
43     for(i = 0; i < p_1; i++)
44         pivots[i] = sample[i*p + p/2];
45
46     struct sub_array *sa = mpballoc(id, p*sizeof(struct sub_array));
47     sublists(array[id], array_size[id], sa, pivots, p_1); /* same algorithm as original
           PSRS paper */
48     wcb_flush(); /* Write-Combine-Buffer flush */
49     barrier();
50
51     struct sub_array l_index[p];
52     for(i = 0; i < p; i++)
```

```
53      {
54          sa = get_partition(i); /* get partition from MPB with ID = i */
55          l_index[i].begin    = sa[id].begin;
56          l_index[i].length   = sa[id].length;
57      }
58
59      size_t count = 0;
60      for(i = 0; i < p; i++)
61          count += l_index[i].length;
62
63      out_size[id] = count;
64      wcb_flush(); /* Write-Combine-Buffer flush */
65      barrier();
66      CL1INVMB(); /* invalidate MPB entries stored in L1 cache */
67
68      size_t out_begin = 0;
69      for(i = 0; i < id; i++)
70          out_begin += out_size[i];
71
72      int *output_array   = shmalloc(n*sizeof(int));
73      int *out_arr        = &(output_array[out_begin]);
74      int *data_arr[p];
75      size_t size_arr[p];
76
77      /* Merge p subarrays into a single sorted array
78       * using a heap based merge algorithm */
79      heap_merge(data_arr, size_arr, p, out_arr);
80      cache_flush(); /* L2 cache flush */
81      barrier();
82
83      /* return the output array stored in shared memory */
84      return output_array;
85  }
```