Scalable Runtime Support for Data-Intensive Applications on the Single-Chip Cloud Computer

Anastasios Papagiannis* and Dimitrios S. Nikolopoulos*
Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
GR–70013, Heraklion, Crete, GREECE
{apapag,dsn}@ics.forth.gr

***Abstract*—Many-core processors, due to their complexity and diversity, necessitate high-productivity, domain-specific approaches to parallel programming. These approaches should hide architectural details and low-level parallelization constructs, while enabling scalability and performance portability. This paper presents a scalable implementation of MapReduce, a runtime system used widely by domain-specific languages for large-scale data processing, on the Intel SCC. We address the scalability bottlenecks of MapReduce with data partitioning, combining and sorting algorithms that we customize for the SCC network on-chip architecture. We achieve linear or superlinear speedups for representative MapReduce workloads with data sets that fit on a single SCC node. We also show that the SCC node outperforms the IBM Cell QS22 Blade, when the latter uses the fastest implementation of MapReduce available for the Cell processor.**

***Index Terms*—MapReduce; Single-Chip-Cloud; Resource management; Runtime systems; Operating Systems; Parallel Programming Models.**

I. Introduction

Programming models for future many-core processors should disengage from low-level parallel programming constructs –such as threads, locks, and messages– and embrace high-productivity domain-specific alternatives. Domain-specific frameworks for parallel programming will require scalable runtime systems to exploit many-core architectures. As more many-core processor architectures forgo cache coherence and use fast on-chip communication to improve performance and energy-efficiency, runtime systems for parallel programming face the challenge of scaling, while hiding the complexities of explicit communication from programmers [1].

Google's MapReduce programming model [2] is widely used for implementing domain-specific languages to support large-scale data processing applications. MapReduce borrows idioms from functional programming to express parallel operators on distributed collections of data and to aggregate data. The MapReduce programming framework has been implemented on a variety of parallel architectures, including clusters, shared-memory multi-core systems with coherent caches, graphics processing units, and multi-core processors with software-managed local memories [2], [3], [4], [5], [6].

*Also with the Department of Computer Science, University of Crete, Heraklion, Greece.

This paper presents the first, to the best of our knowledge, implementation of the MapReduce programming model and runtime system on the Intel Single-Chip Cloud Computer (SCC). We present a design that utilizes effectively the SCC interconnection network and on-chip shared communication buffers to alleviate two fundamental scalability bottlenecks of MapReduce: data partitioning and data sorting. The artifact of our contribution is a fast and scalable implementation of MapReduce, based on customized on-chip data exchange, combining, and sorting algorithms. We achieve linear or superlinear speedups for representative MapReduce workloads, which process data sets that fit in the memory of a single SCC node. We further show that an SCC node outperforms an IBM QS22 Cell blade with two Cell processors, when the latter uses the fastest implementation of MapReduce for the Cell processor available to date [6].

II. Background

We provide background for our work by presenting the MapReduce program execution stages and discussing the key architectural properties of the Intel SCC processor.

A. MapReduce

MapReduce [2] is a high-level parallel programming model based on two primitives, *map* and *reduce*. Parallel computation in MapReduce is expressed as processing and aggregation operators applied on distributed data sets. A MapReduce application processes an input of (key, value) pairs to produce an output of (key, value) pairs. A typical MapReduce program executes in four stages, a *map* stage, where parallel workers (called *mappers*) produce a set of intermediate (key, value) pairs for each input pair, a *partition* stage which exchanges intermediate data between mappers, a *group* stage which groups all intermediate (key, value) pairs associated with the same key, and a *reduce* stage which merges the values associated with each key. The map and reduce stages execute user-defined data processing and aggregation operators. MapReduce implementations typically have an explicit barrier between the map and partition stages, although this barrier can be replaced by a software pipeline [7].

Figure 1 shows a typical MapReduce execution flow. The MapReduce runtime splits the input data into fixed size chunks and assigns each chunk to a *mapper*. Each mapper executes

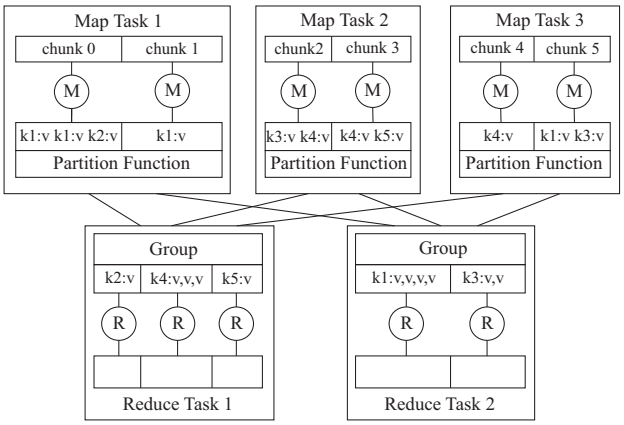

Fig. 1: A typical MapReduce execution flow (*M* stands for *Mappers*, *R* stands for *Reducers*)

the map function on its assigned chunk, which consists of a series of input (key, value) pairs. The map function generates zero or more intermediate (key, value) pairs for every input (key, value) pair. The input and output types of the map function may differ. The runtime system maintains a number of *reducers* that perform data aggregation. During the map stage, each mapper exports as many different partitions as the number of reducers. It is permissible for mappers and reducers to execute on the same compute node.

To split the output of a mapper, the user may define a *partition* function. The partition function takes as input a key and the number of reducers and returns an index to the reducer to which the output should be sent. The typical implementation of this step is to hash the key and compute the partition index as the key's hash value modulo the number of reducers. It is important to pick a partitioning function that gives an approximately uniform distribution of data per reducer for load balancing purposes, otherwise the runtime may stall waiting for slow reducers to finish. The partition stage executes between the map and reduce stages and moves each intermediate (key, value) pair from the node running the mapper that produced it to the node on which it will be reduced.

Following the partitioning stage, an optional *group* stage sorts the intermediate data on each reducer. The runtime has to rearrange the intermediate (key, value) pairs so that the data is organized as a set of unique keys and a list of values for each key. Finally, the runtime executes the user-defined reduce function to aggregate intermediate (key, value list) pairs. The reducer iterates through the values that are associated with each key and produces zero or more output (key, value) pairs.

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

The Intel SCC [8] (Figure 2) is a many-core processor with 24 tiles and 2 IA cores per tile. The tiles are organized in a 4×6 mesh network with 256 GB/s bisection bandwidth. The processor has 4 integrated DDR3 memory controllers, one for each group of 6 tiles. Each core has a private L1 instruction cache of 16 KB, a private L1 data cache of 16

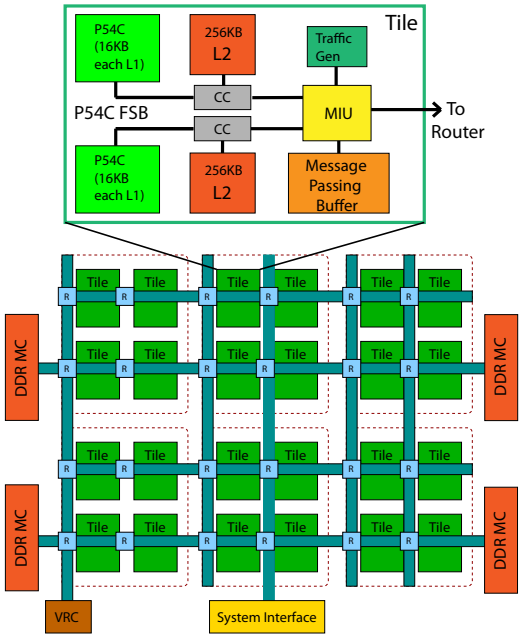

Fig. 2: SCC processor diagram

KB and a private unified L2 cache of 256 KB. Each dual-core tile has a 16 KB message passing buffer (MPB), which is the only component of the SCC on-chip memory hierarchy that is shared between cores. The MPB provides space for direct core-to-core communication. On-chip communication data is read from the MPB through the L1 data cache and bypasses the L2 cache. For writes, a no-allocate policy is used, in conjunction with a write combining buffer at the L1 cache. Software needs to maintain coherence between the MPB and the L1 caches by using a, unique to the SCC, L1 cache invalidation instruction, when data is stored in the MPB.

The 32-bit address space of the system is mapped to an extended 34-bit address space to allow access to up to 64 GB of off-chip memory (up to 16 GB from each group of 6 tiles). This is accomplished through a Look-Up Table (LUT) attached to each core. The address space of the system is configurable and can be distributed between private off-chip memory associated with each core, shared off-chip memory, and shared on-chip SRAM, which corresponds to data stored in the message buffers and cached in the L1 cache.

We implemented MapReduce using the the standard software environment of SCC compute nodes available by Intel, namely a configuration running a Linux kernel on each core and RCCE, the Intel one-sided communication library [9].

III. MapReduce Design

We implement a seven-stage runtime system for MapReduce. The seven stages are *map*, *combine*, *partition*, *group*, *reduce*, *sort* and *merge*. The *combine* and *merge* stages are optional in typical MapReduce setups, whereas the *group* stage replaces the intermediate *sort* stage of the original MapReduce

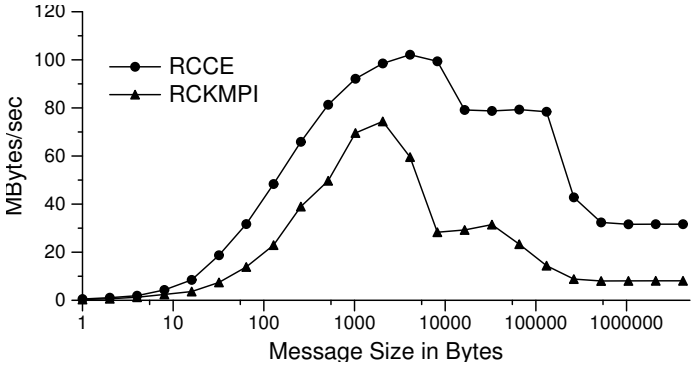

Fig. 3: RCKMPI vs. RCCE bandwidth in a ping-pong benchmark

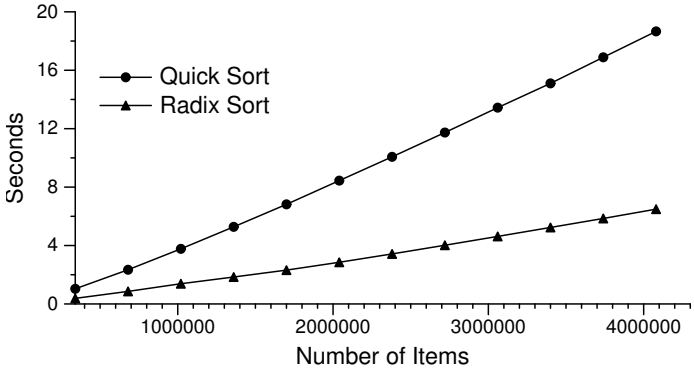

Fig. 4: Libc qsort vs. radix sort, for a variable number of word-size elements

pipeline, to reduce overall computational complexity. We describe the stages of MapReduce in more detail in the following paragraphs.

a) Map: During the map stage, the runtime system divides the input evenly to as many parts as the number of mappers. In our implementation, we use as many mappers as the number of cores. Each core then executes the user-defined map function over its assigned input data. We preallocate a large chunk of memory (64 MB) for the output of the map stage. If the volume of intermediate data produced is more than 64 MB, we allocate a new output buffer on demand. Each core exports as many intermediate data partitions as the number of cores in the system. To split intermediate data between partitions, we use either a user-defined hash function or a default generic hash function available by our MapReduce runtime system. Each core emits keys and values in a contiguous buffer.

b) Combine: This stage is optional and executes if the user provides a combiner function. The purpose of this stage is to reduce locally the size of each partition produced during the map stage. The combine function takes as input a key and a list of partially aggregated intermediate values associated with that key. It produces as output a single (key, value) pair where the value is an updated partial aggregation associated with the same key. We use the combine function to reduce data volume and balance the partitions that will be processed by different cores in the following stages of MapReduce. We use the same strategy as in the group stage described later to group together all values with the same key. The combiner function produces a new intermediate (key, value) pair for each intermediate key and its corresponding list of values. The combine stage is equivalent to applying the reduce stage in each of the intermediate partitions.

c) Partition: The partition stage requires an all-to-all exchange between cores. Data partitions generated during the map stage may be different in size. We implement 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. We implement 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, core *rank* receives data from core $rank-k$ and sends data to core $rank+k$. We opted to use the *RCCE_*$\{$*send, recv*$\}$ functions to implement this all-to-all exchange. RCCE is an SCC communication runtime environment based on one-sided get-put communication primitives [9]. Figure 3 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. RCCE also uses the MPB buffers to exchange data.

d) Group: The group stage groups together all (key, value) pairs with the same key, taken across all intermediate data partitions. In previous works [6], [3], [4], [5], [10], [11], a generic sorting scheme with a user-defined comparator was used to perform grouping. We replace this scheme with a radix sort algorithm [12] for grouping on the SCC. The sorting algorithms employed in prior MapReduce implementations on multi-core systems have complexity $O(nlogn)$, whereas radix sort has complexity $O(kn)$ where k is the size of the key in bytes. Figure 4 shows a comparison of the libc quicksort implementation and our radix sort implementation for different input sizes. Radix sort outperforms quicksort with the caveat that 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.

Previous sorting algorithms used in MapReduce swap (key, value) pairs by copying the actual data of these pairs. Our radix sort algorithm swaps pointers to (key, value) pairs instead. Thus, in every swap we only exchange two pointers, making the cost of the swap independent of the size of the (key, value) pair. The output of this stage is an array of pointers to the actual data. This array needs to be transformed to a structure containing pairs of keys and value lists. We accomplish this by simply iterating through the array and finding the unique keys. We initiate an iterator for accessing the values with no need to rearrange the data in memory. We statically know the sizes of all the buffers needed for the sorting stage, therefore we preallocate these buffers. This optimization minimizes the

Application	Class	Input size
Word Count	partition-dominated	60 MB
Histogram	sort-dominated	400 MB
Linear Regression	map-dominated	32 MB
Kmeans	map-dominated	115 MB

TABLE I: MapReduce application workloads

overhead of dynamic memory allocation.

e) Reduce: The reduce stage executes a user-defined key aggregation function. The prior group stage exports an array of all distinct keys where each key contains the number of occurrences of the key and a pointer to an array of its values. The output size of the reduce stage can be statically identified, therefore we preallocate the stage's output buffers, once again to minimize dynamic memory allocation overhead.

f) Sort: The sort stage sorts the (key, value) pairs produced following the reduction, using quicksort and a user-specified comparator. This stage is necessary because the earlier group stage may produce unsorted sequences. However, this sort stage is necessary only if the following data merging stage is needed as well.

g) Merge: The merge stage optionally merges the output of all cores in one core. In the default configuration of SCC, each core has its private memory, therefore in applications that require merging, we need to produce the final output in the memory of a single core. We use the binomial merge algorithm for this stage [13], which completes in *logn* steps.

IV. Experimental Analysis

Table I lists the MapReduce application workloads that we used for experiments. Following conventions from [11], we classify applications as map-dominated, partition-dominated and sort-dominated, according to the phase where these applications fail to scale on a multi-core system.

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. *Word Count* counts the number of occurrences of each word in a text file. 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. *Kmeans* creates clusters from a set of data points, by finding the closest cluster for each data point in the map function and computing the cluster means in the reduce function. *Linear Regression* computes a line of best fit for a set of points, given their 2D coordinates. Map computes intermediate summary statistics for the points, while reduce gathers all data of each of the summary statistics and calculates the best fit.

In our experiments, we use the standard frequency configuration of the SCC chip. In this configuration, each tile runs at a frequency of 533MHz, the mesh interconnect runs at a frequency of 800MHz and DRAM runs at a frequency of 800MHz.

Figure 5 illustrates speedup and Figure 6 illustrates execution time of application workloads, with and without a

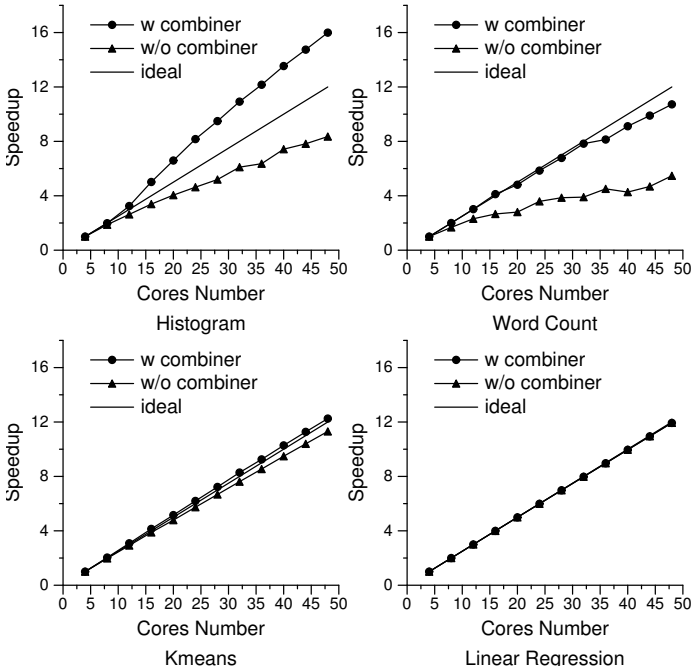

Fig. 5: Speedup of MapReduce workloads

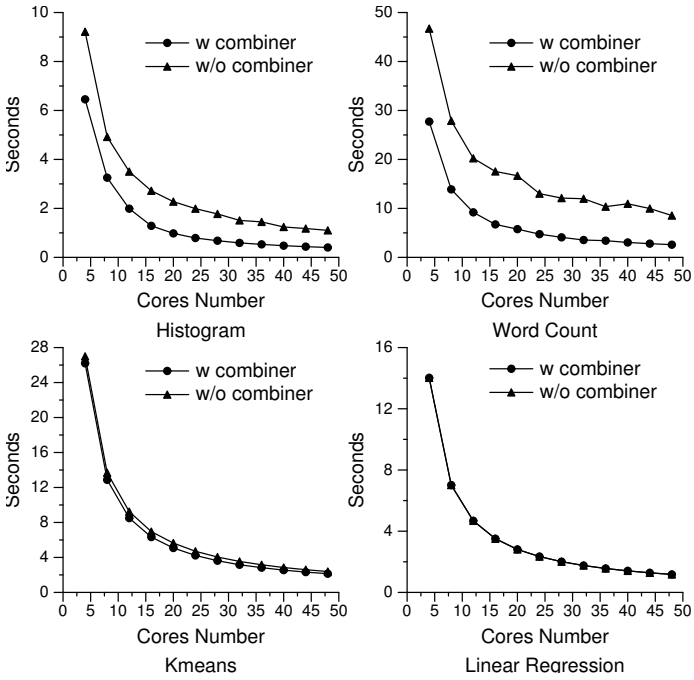

Fig. 6: Execution time of various applications

combiner function. Speedup is calculated using execution time on 4 cores (2 tiles) as the nominator, therefore ideal linear speedup is 16 for the entire SCC chip. Figure 7 and Figure 8 show breakdowns of execution time for all applications. All applications scale well on the chip. With the use of a combiner function, applications have nearly ideal linear or in some cases, superlinear speedup. The partition stage exhibits the worst scaling behavior. The combine stage improves performance by reducing the intermediate data exported from the map stage.

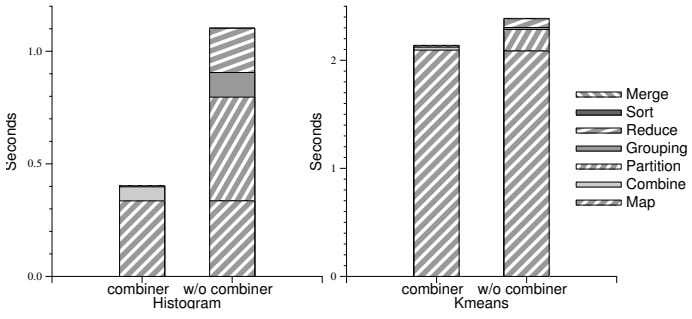

Fig. 7: Execution time breakdowns

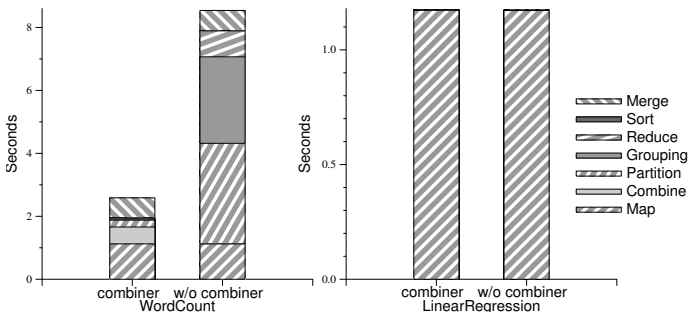

Fig. 8: Execution time breakdowns

This in turn means that the partition stage has to exchange less data between cores. The reason for superlinear speedup is that the complexity of the group stage decreases exponentially with the number of cores. In applications where the grouping stage dominates execution time, the overall application speedup may therefore be superlinear. We analyze briefly individual applications in the following paragraphs.

Histogram does not achieve perfect speedup without a combiner (Figure 5), because the partition stage does not scale. Partitioning overhead dominates execution time (Figure 7). Reducing the intermediate data size with a combiner alleviates the bottleneck. Using the combiner also decreases the execution time of the grouping and reduce stages. The combiner function is the same function as the one used in the reduce stage in this benchmark. Therefore, the combine stage executes a part of the reduce stage on the intermediate values available locally to each core. *Histogram* exports a maximum of only 3×255 different keys, which makes the merge stage time insignificant.

KMeans and *Histogram* have similar behavior (Figure 5), with the exception that in *KMeans* the map stage dominates execution time, therefore the combiner has a less significant impact on overall execution time (Figure 7). This is also the reason why in *Kmeans* we do not achieve superlinear speedup.

Linear Regression is an entirely map-dominated benchmark and therefore scales perfectly. Each map function exports five (key, value) pairs and therefore group and reduce times are insignificant. Since the map stage exports only five different keys, we can only use five cores in the execution stages after map. From the breakdowns (Figure 8) we observe that this is

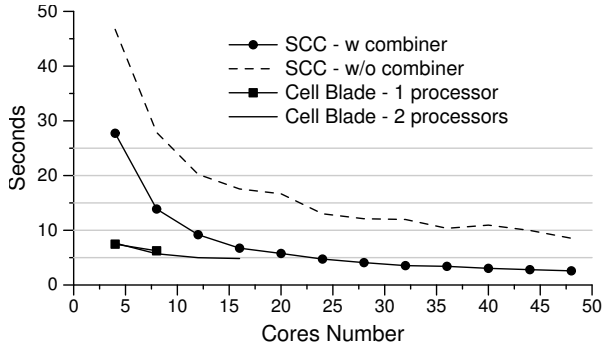

Fig. 9: Comparison of SCC and Cell BE processors using wordCount benchmark

not a scalability bottleneck. Merging the output results of five cores has negligible overhead.

Word Count incurs load imbalance in the grouping stage. This leads to erratic speedup (Figure 5). However, the problem is easily alleviated with a combiner function that rebalances the volume of intermediate data between cores. We have compared many hash functions for strings while experimenting with *Word Count*. We ended up using the *djb2* hash function. This function initially sets $hash = 5381$ and then for each character of the string it sets $hash = hash * 33 + c$ where c is the ASCII value of each character. The selected hash function results in better distribution of intermediate keys among different partitions in comparison with other hash functions.

Figure 9 illustrates a comparison of our implementation of MapReduce on the SCC with a competitive implementation of MapReduce on the Cell processor, which is the fastest implementation for that processor published to date [6]. We used the Word Count benchmark with a 60 MB input size. We ran this benchmark on a Cell QS22 Blade with 8GB RAM and report execution time with the Cell MapReduce runtime published in [6], using one or both of the Cell processors of the QS22 blade. Each Cell processor has 9 cores, out of which 8 (the SPE vector cores) are used for MapReduce tasks and one (the PowerPC PPE core) is used for the runtime system. The maximum number of mapper and reducer cores is 8 when using one Cell processor and 16 when using two Cell processors. The SCC node with a single SCC processor outperforms the dual-processor Cell QS22 blade by up to $1.87\times$, when the SCC MapReduce uses combiner functions. We note that the Cell processors on the QS22 run at 3.2 GHz and that each core on the Cell has a software-managed local store of the same size as the L2 cache of each core on the SCC.

V. Related Work

Several prior research efforts ported MapReduce to prominent hardware platforms for high-performance computing, including multicore processors [4], [5], [14], GPUs [3], [15] the Cell processor [11], [6], [16] and FPGAs via direct software to hardware translation [17].

Phoenix, a port of MapReduce for cache-coherent shared-memory multicore systems [4], [5], 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 the most recently published version of Phoenix [5], 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. A different approach to optimize Phoenix is proposed in [14] where the authors use tiling to minimize task memory footprints and improve cache locality.

MapReduce has also been ported to the Cell BE processor [11], [6]. In the implementation presented in [6], which is the fastest, the runtime system controls locality explicitly, using DMAs and software prefetching via multi-buffering in the map and merge-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.

Implementations of MapReduce on GPUs also consider the implications of explicitly-managed local memories [10], [3], [15]. Mars [3] uses mock map tasks to compute the sizes of buffers needed by each core for emitting results of real map tasks. Other optimizations of MapReduce on GPUs focus on achieving fine-grain interleaving of memory accesses from threads on the GPU, to utilize the available GPU memory bandwidth.

VI. Conclusions

This paper presented a scalable implementation of Google's MapReduce runtime system on the Intel SCC. The implementation attests to the scalability of the chip, as well as its ability to support software stacks and high-level parallel programming models that hide explicit communication from programmers. Our implementation of MapReduce leveraged one-sided on-chip communication primitives and customized data combining algorithms to alleviate bottlenecks that arise during data partitioning and sorting. We demonstrated perfect linear or superlinear scaling of applications with realistic datasets for a single SCC node and performance that exceeds the fastest to date implementation of MapReduce on IBM Cell blades. While our results are promising, our work raises several interesting questions for future research. These include design choices for implementing the full MapReduce execution path, including I/O, alternative management schemes for the SCC memory hierarchy that exploit off-chip shared memory, the implementation of dynamic task scheduling, and further analysis of applications.

Acknowledgements

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

References

[1] S. Schneider, J.-S. Yeom, B. Rose, J. C. Linford, A. Sandu, and D. S. Nikolopoulos, "A Comparison of Programming Models for Multiprocessors with Explicitly Managed Memory Hierarchies," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Feb. 2009, pp. 131–140.

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

[3] 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.

[4] 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.

[5] 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.

[6] A. Papagiannis and D. S. Nikolopoulos, "Rearchitecting Mapreduce for Heterogeneous Multicore Processors with Explicitly Managed Memories," in *Proceedings of the 39th International Conference on Parallel Processing (ICPP)*, Sep. 2010, pp. 121–130.

[7] A. Verma, N. Zea, B. Cho, I. Gupta, and R. Campbell, "Breaking the MapReduce Stage Barrier," in *Proceedings of the 2010 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2010, pp. 235–244.

[8] 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 *Proceedings of the 2010 IEEE International Conference on Solid-State Circuits (ISSCC)*, Feb. 2010, pp. 108–109.

[9] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard *et al.*, "The 48-core SCC Processor: the Programmer's View," in *Proceedings of Supercomputing'10: The 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010, pp. 1–11.

[10] 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.

[11] 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.

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

[13] R. Thakur and R. Rabenseifner, "Optimization of Collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, Feb. 2005.

[14] 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.

[15] 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.

[16] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos, "Supporting MapReduce on Large-Scale Asymmetric Multi-core Clusters," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 25–34, Apr. 2009.

[17] J. H. Yeung *et al.*, "Map-Reduce as a Programming Model for Custom Computing Machines," in *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2008, pp. 149–159.