MapReduce for the Single-Chip-Cloud Architecture

Anastasios Papagiannis[1] and
Dimitrios S. Nikolopoulos[1]

Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
100 N. Plastira Av., Vassilika Vouton, Heraklion, GR–70013, Greece

ABSTRACT

Many-core processors, due to their complexity and diversity, will necessitate high-productivity, domain-specific approaches to parallel programming. These approaches should hide architectural details and low-level parallel programming 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.

KEYWORDS: MapReduce; Single-Chip-Cloud; Resource management; Runtime systems; Operating Systems; Parallel Programming Models

1 Introduction

MapReduce [DG08] processes an input of (key, value) pairs to produce an output of (key, value) pairs. A MapReduce program executes in three stages, a *map* stage that produces a set of intermediate (key, value) pairs for each input pair, a *group* stage that groups all intermediate (key, value) pairs associated with the same key, and a *reduce* stage that merges the values associated with each key. The *map* and *reduce* stages are user-defined and application-specific.

The Intel SCC [Hea10] 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 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

[1]E-mail: {apapag, dsn}@ics.forth.gr

between cores. On-chip communication data is read from the MPB through the L1 data cache, bypassing the L2 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 on 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. We implement 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 [Mea10].

This paper presents an implementation of the MapReduce programming model and runtime system on the Intel Single-Chip Cloud Computer (SCC) [Hea10]. We present a design that utilizes effectively the SCC interconnection network and on-chip shared communication buffers to alleviate the two fundamental scalability bottlenecks of MapReduce, namely data partitioning and data sorting. Our end result is a a fast and scalable implementation of MapReduce, based on customized on-chip data exchange, combining, and sorting algorithms.

2 MapReduce Design

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

During the Map stage, the runtime system divides the input evenly to as many parts as the number of cores. Each core then executes the user-defined map function over its private input data. This function takes as input a key-value pair and produces one or more intermediate key-value pairs. Each core exports as many intermediate data partitions, as the number of cores in the system. To split intermediate data between different partitions, we use a user-defined hash function or a generic hash function available in the MapReduce runtime in case the user does not specify a hash function. The hash function takes a key as argument and returns the ID of a partition to store the generated intermediate key-value pair.

Combine 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 by a given core during the Map stage. The combine function takes as input a key and a list of partially aggregated intermediate values associated with the same key. It produces as output a single key-value pair where the value is an updated partial aggregation of the values associated with the key.

The partitioning 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

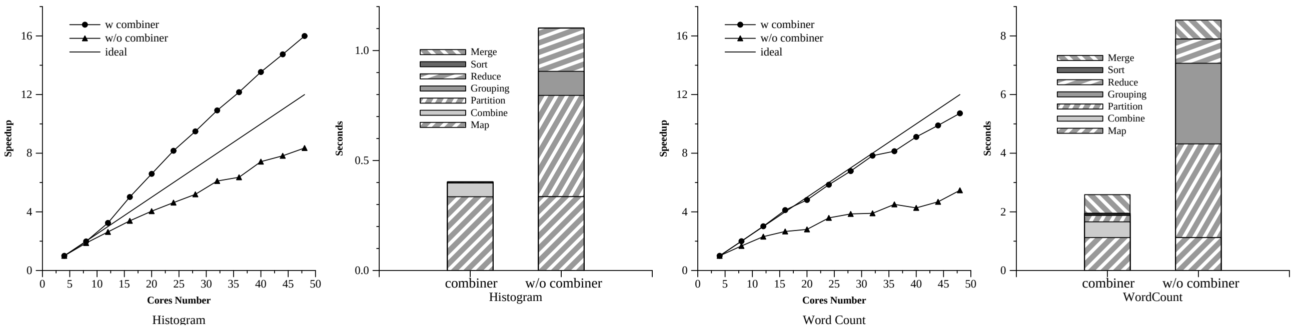

Figure 1: Speedup and breakdown for Histogram (left) and WordCount (right).

$p - 1$ steps and in each step k the $rank$ core 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 [Mea10].

The grouping stage groups together all key-value pairs with the same key, taken across all intermediate data partitions. In previous works, a generic sorting scheme with a user-defined comparator was used to perform grouping. We replace this scheme with a radix sorting algorithm [MBM93] 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. 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.

The reduction stage executes a user-defined key aggregation function. The prior grouping stage exports an array of all distinct keys where each key contains a number of occurrences of this key and a pointer to an array of its values. The output size of the reduction stage can be statically identified, therefore we preallocate the stage's output buffers.

The sorting stage sorts the key-value pairs produced following the reduction, using quicksort and a user-specified comparison operator.

The merge stage optionally merges the output of all cores in one core. We use the binomial merge algorithm for this stage [TR05], which completes in $logn$ steps.

3 Evaluation

We use *Histogram* and *Word Count* applications to evaluate MapReduce. *Histogram* counts the frequency of occurrences of each RGB color component in an image file. *Word Count* counts the number of occurrences of each word in a text file.

Figure 1 illustrates speedup of application workloads, with and without a 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 1 show breakdowns of execution time. All applications scale well. With the use of a combiner function, applications have nearly ideal linear or in some cases, superlinear speedup. The reason for superlinear speedup is that the complexity of the *Grouping* decreases exponentially with the number of cores. Therefore, the *Grouping* stage has superlinear speedup and in applications where the grouping stage dominates execution time, the overall application speedup may also be superlinear. We analyze briefly individual applications in the following.

Histogram does not achieve perfect speedup without a combiner, because the *Partition* stage does not scale. Reducing the intermediate data size with a combiner alleviates the bottleneck. *Histogram* exports a maximum of only 3×255 different keys, which makes *Merge* time insignificant.

Word Count incurs load imbalance in the *Grouping* stage. This leads to erratic speedup. However, the problem is easily alleviated with a combiner function that rebalances the volume of intermediate data between cores.

4 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 high-level parallel programming models while hiding 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.

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

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.

[Hea10] Jason Howard and et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109. IEEE, February 2010.

[MBM93] Peter M. McIlroy, Keith Bostic, and M. Douglas Mcilroy. Engineering radix sort. *COMPUTING SYSTEMS*, 6:5–27, 1993.

[Mea10] Timothy G. Mattson and et al. 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, pages 1–11, 2010.

[TR05] Rajeev Thakur and Rolf Rabenseifner. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19:49–66, 2005.