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

Anastasios Papagiannis and Dimitrios S. Nikolopoulos, FORTH-ICS

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

3rd MARC Symposium, 2011
Outline

Motivation

Background
   Intel Single-Chip-Cloud
   MapReduce

Design
   Outline
   Implementation

Experimental Analysis
   Benchmarks
   Speedup
   Execution Time Breakdowns
   SCC vs. Cell BE

Conclusions
Motivation and Contributions

- We are on the transition from multi-core processors to many-core processors
- Programmers have to deal with:
  - many cores
  - many forms of implicit or explicit communication
  - many forms of synchronization
  - potential lack of cache coherence
- Contributions of this work:
  - First implementation of a high-level domain-specific parallel programming model (Google’s MapReduce) on a cache-based many-core processor with no cache coherence, based on explicit communication (SCC)
  - Evaluation showing that the Intel SCC supports effectively:
    - High-level programming models that hide communication, synchronization, parallelization under the hood
    - Scalable execution of data-intensive applications
Outline

Motivation

Background

Intel Single-Chip-Cloud
MapReduce

Design

Outline
Implementation

Experimental Analysis

Benchmarks
Speedup
Execution Time Breakdowns
SCC vs. Cell BE

Conclusions
Intel SCC

- Many-core processor with 24 tiles, 2 IA cores per tile
- Tiles organized in a $4 \times 6$ mesh network with 256 GB/s bisection bandwidth
- Private L1 instruction cache of 16 KB, private L1 data cache of 16 KB, private unified L2 cache of 256 KB, per core
- 16 KB message passing buffer (MPB) per tile (only on-chip memory shared between cores)
Outline

Motivation

Background
  Intel Single-Chip-Cloud
  MapReduce

Design
  Outline
  Implementation

Experimental Analysis
  Benchmarks
  Speedup
  Execution Time Breakdowns
  SCC vs. Cell BE

Conclusions
MapReduce

- A framework for **large-scale data processing**
  - Programming model (API) and runtime system for a variety of parallel architectures
    - Clusters, SMPs, multi-cores, GPUs, among others
  - Based of **functional programming language primitives**
- Used extensively in **real applications**
  - Indexing system, distributed grep, document clustering, machine learning, statistical machine translation
- Relies heavily on **a scalable runtime system**
  - Fault-tolerance, parallelization, scheduling, synchronization and communication
Counting word occurrences in a set of documents
Outline

Motivation

Background
   Intel Single-Chip-Cloud
   MapReduce

Design
   Outline
   Implementation

Experimental Analysis
   Benchmarks
   Speedup
   Execution Time Breakdowns
   SCC vs. Cell BE

Conclusions
Seven-stage runtime system for MapReduce:

- Map
- Combine (optional)
- Partition
- Group
- Reduce
- Sort (optional)
- Merge (optional)
Outline

Motivation

Background
  Intel Single-Chip-Cloud
  MapReduce

Design
  Outline
  Implementation

Experimental Analysis
  Benchmarks
  Speedup
  Execution Time Breakdowns
  SCC vs. Cell BE

Conclusions
MapReduce

Map

Each core executes the **user-defined map function** on chunks of input data, located in local memory.

- Map function emits **one or more intermediate key-value pairs**.
Intermediate key-value pairs stored in a contiguous buffer
- Runtime preallocates large chunks of memory (64 MB) for intermediate data buffers
- More buffering space allocated on demand, if needed
- Allocation strategy reduces memory management overhead
MapReduce

Map

- Each core produces as many intermediate data partitions as the total number of cores
MapReduce

Optional stage executed if user provides a combiner function
Reduces locally the size of each partition produced during the map stage

Core 0

by,1 | by,1
Combine
by,2

the,1|the,1
Combine
the,2

Core 1

by,1|by,1|by,1
Combine
by,3

the,1|the,1
Combine
the,2
MapReduce
Combine

- Takes as input a key and a list of partially aggregated intermediate values associated with that key
- Produces a new intermediate key-value pair based on intermediate key and its corresponding list of values
MapReduce
Partition

- Requires an all-to-all exchange between cores
- Data partitions generated during the map stage may be different in size
  - First execute an all-to-all exchange of the sizes of each partition
  - Knowing the size of each partition, execute a second all-to-all exchange with the actual data
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 $rank + k$. 
**MapReduce**

**Group**

- Groups all (key, value) pairs with the same key
- Use **radix sort** instead of conventional merge sort
  - Radix sort sorts strings of bytes and **can not use a user-defined comparator for sorting**
  - If radix sort does not sort native application type, sort the output using a user-specified compare function
  - Conventional sorting algorithms have complexity $O(n \log n)$. Radix sort has complexity $O(kn)$ where $k$ is the size of the key in bytes.
MapReduce

Reduce

- Group stage exports distinct keys with a list of corresponding values
- Reduce stage executes user-defined aggregation function on each key-list(of values) pair
MapReduce

- Reduce function emits one or more output key-value pairs
  - Total output size known prior to reduction, therefore output buffer is preallocated
  - Minimizes memory management overhead
MapReduce
Sort and Merge

Step 0

P0  P1  P2  P3

Step 1

P0  P2

Step 2

Output Buffer

Sort
- Sort the output key-value pairs of the previous reduction stage
- Quick sort algorithm with complexity $O(n \log n)$, based on a user-specified compare function

Merge
- Optionally merges the output of all cores in one core
- Binomial tree merge algorithm, completes in $\log n$ steps

Anastasios Papagiannis and Dimitrios S. Nikolopoulos, FORTH-ICS
Scalable MapReduce on the SCC. MARC3 Symposium.
Outline

Motivation

Background
   Intel Single-Chip-Cloud
   MapReduce

Design
   Outline
   Implementation

Experimental Analysis
   Benchmarks
   Speedup
   Execution Time Breakdowns
   SCC vs. Cell BE

Conclusions
Benchmarks

- **Histogram (partition-dominated)** counts the frequency of occurrences of each RGB color component in an image file.
- **Word Count (partition-dominated)** counts the number of occurrences of each word in a text file.
- **Kmeans (map-dominated)** creates clusters from a set of data points.
- **Linear Regression (map-dominated)** computes a line of best fit for a set of points, given their 2D coordinates.

Configuration:

- Tiles run at **533MHz**
- Mesh interconnect runs at **800MHz**
- DRAM runs at **800MHz**
Outline

Motivation

Background
Intel Single-Chip-Cloud
MapReduce

Design
Outline
Implementation

Experimental Analysis
Benchmarks
Speedup
Execution Time Breakdowns
SCC vs. Cell BE

Conclusions
Combiner function improves scalability
  ▶ Kmeans and Linear Regression are map-dominated benchmarks
  ▶ Superlinear speedup because complexity of the group stage decreases exponentially with the number of cores
Outline

Motivation

Background
  Intel Single-Chip-Cloud
  MapReduce

Design
  Outline
  Implementation

Experimental Analysis
  Benchmarks
  Speedup
  Execution Time Breakdowns
  SCC vs. Cell BE

Conclusions
Using a combiner function reduces execution time
- Partition stage does not scale
- Combiner minimizes total partition time and group time
Outline

Motivation

Background
  Intel Single-Chip-Cloud
  MapReduce

Design
  Outline
  Implementation

Experimental Analysis
  Benchmarks
  Speedup
  Execution Time Breakdowns
  SCC vs. Cell BE

Conclusions
▶ QS22 Blade consists of 2 Cell BE Processors at 3.2 GHz
▶ Each processor has 8 SPEs (accelerators)
▶ WordCount benchmark with 60MB input size
▶ Single-SCC nodes outperforms dual-Cell blade by up to $1.87 \times$
Related Work

- Other ports of MapReduce on clusters, SMPs, multicores and GPUs (HPCA07,PACT08,IISWC09,ICPP10)
- Shared-memory ports based on shared data structures in cache-coherent address space
  - SCC port based on distributed data structures and scalable exchange algorithms, while utilizing caches for fast message exchange
- Distributed-memory ports based on generic sorting and group algorithms
  - SCC port based on combiners and radix sort algorithm
Conclusions

- Our implementation of MapReduce on the Intel SCC demonstrates:
  - Feasibility of implementing high-level, domain-specific parallel programming models that hide explicit communication
  - SCC chip scalability when using optimized chip-specific global communication algorithms
  - Good adaptivity to diverge workloads: map-dominated, partition-dominated, group-dominated
Thank you!

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 nº 224759.
Appendix

Radix Sort execution time

Anastasios Papagiannis and Dimitrios S. Nikolopoulos, FORTH-ICS
Scalable MapReduce on the SCC. MARC3 Symposium.