Implementing Scalable Parallel Programming Models with Hybrid Address Spaces

Anastasios Papagiannis

University of Crete and ICS-FORTH
Heraklion, Greece
apapag@ics.forth.gr

18 February 2013

Outline

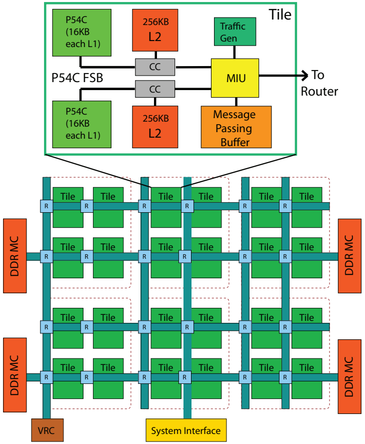
Motivation and Contributions

- We are on the transition from multi-core processors to many-core processors
- Processors support both distributed and shared memory
- . . . but programmers have to deal with the potencial lack of cache coherence
- Carefully selection of address spaces and software cache coherence mechanisms are critical for performance and scalability
- Contributions of this work:
  - Scalable data splitters
  - Work-stealing on non-coherent architectures
  - An evaluation of on-chip barrier algorithms for non-coherent many-core processors
  - A mechanism to enables scalable all-to-all exchanges

Outline

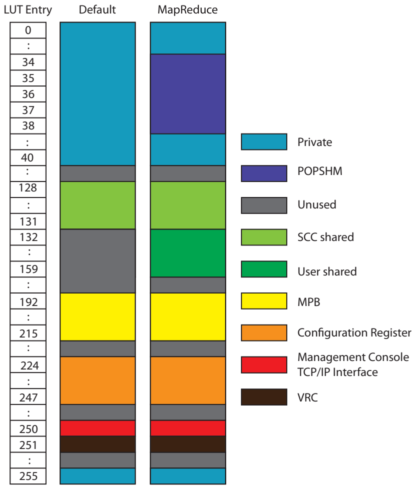
Intel SCC

- Many-core processor with 24 tiles, 2 IA cores per tile
- Tiles organized in a 4×6 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)



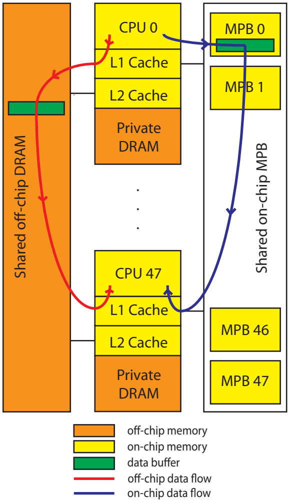
SCC Adress Spaces

- A software-managed translation table called LUT, translate 32bit core's physical addresses to 34bit system's physical addresses
- The LUT has 256 entries, each mapping 16MB of DRAM
- No restriction to reprogram LUT entries during the execution of a program
- Use of software-managed LUTs to implement hybrid address spaces



SCC System Software

- Cluster on a Chip with portions of shared memory
- Each core runs its own Linux kernel
- Support for Message Passing using RCCE and RCKMPI
- Small messages exchanged through MPB
- Large messages exchanged through off-chip shared DRAM

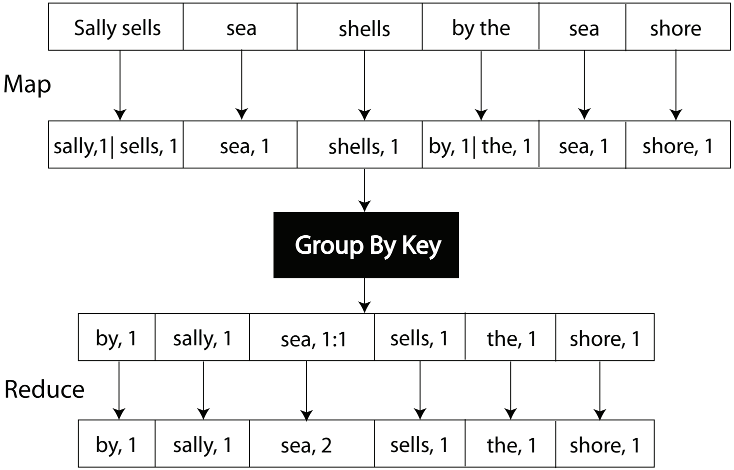


Outline

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

Example

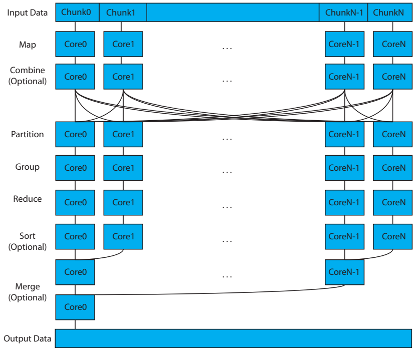


Counting word occurrences in a set of documents

Outline

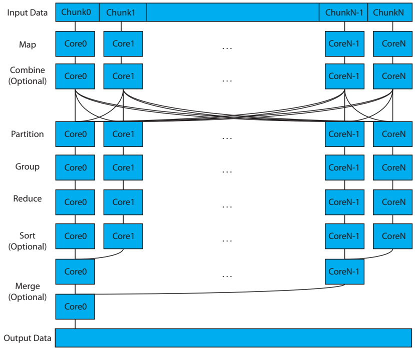
DiMR

- Map stage
  - Each core executes the user-defined map function on chunks of input data
  - Intermediate key-value pairs stored in a contiguous buffer
    - Runtime preallocates large chunks of memory (64MB) for intermediate data buffers
    - More space allocated on demand, if needed
  - Each core produces as many intermediate data paritition as the total number of cores



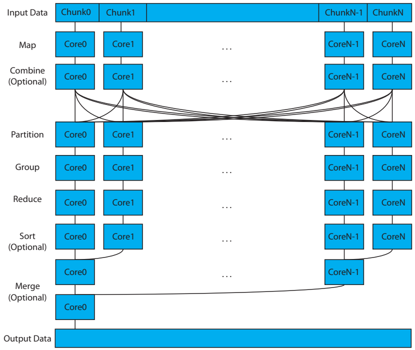
DiMR

- Combine stage (optional)
  - Reduces locally the size of each partition produced during the map stage
- Partition stage
  - Requires an all-to-all exchange between cores
  - We use pairwise exchange algorithm, this needs $p-1$ where p is the number of cores



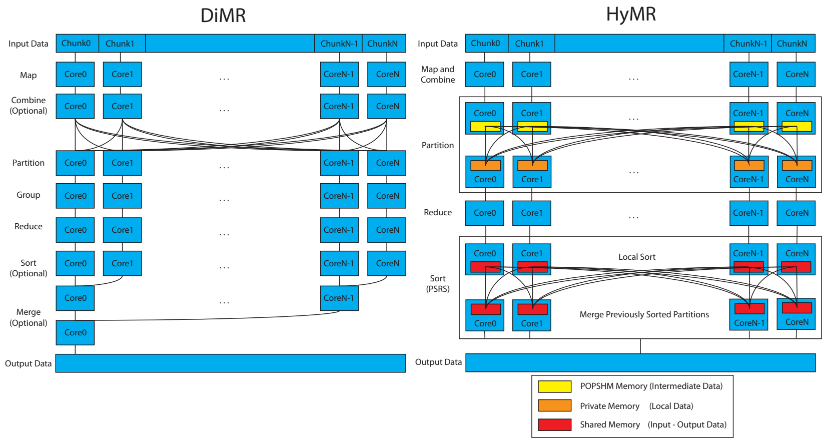
DiMR

- Group stage
  - Groups all key-value pairs with the same key
  - Use Radix sort instead of conventional Quick sort
    - Quick sort has complexity O(nlogn) where Radix sort has complexity O(kn), k is the length of the keys
- Reduce stage
- Sort stage (optional)
- Merge stage (optional)



Outline

HyMR

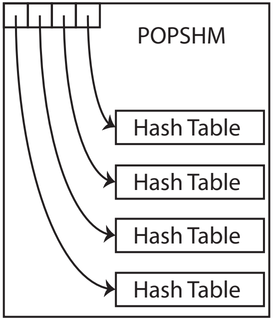

DiMR
Input Data
Chunk0
Chunk1
ChunkN-1
ChunkN
Map
Combine (Optional)
Partition
Group
Reduce
Sort (Optional)
Merge (Optional)
Output Data
Core0
Core1
CoreN-1
CoreN
HyMR
Map and Combine
Partition
Reduce
Sort (PSRS)
Local Sort
Merge Previously Sorted Partitions
POPSHM Memory (Intermediate Data)
Private Memory (Local Data)
Shared Memory (Input - Output Data)

HyMR - Scalable Splitters

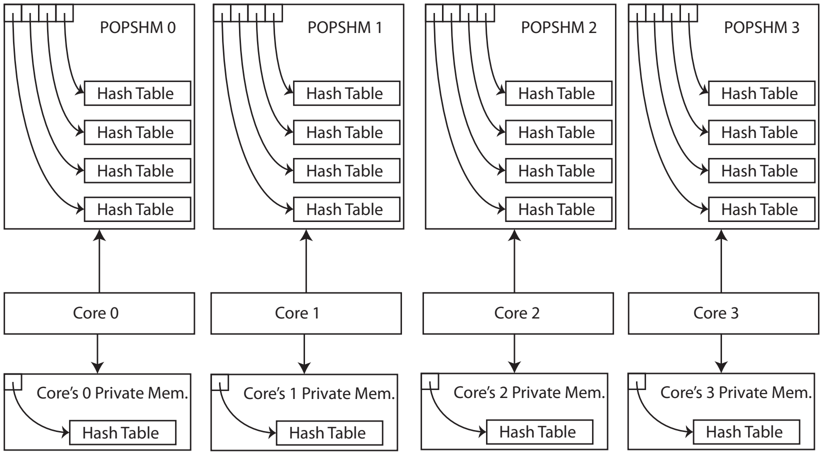
- The input in MapReduce is an array of key-value pairs or text files
- Each core splits the input array in number of cores chunks and gets it’s own chunk by core ID
- In the worst case the splitter executes number-of-cores iterations
- Each core stores it’s input chunks in a private queue

HyMR - Map and Combine Stages

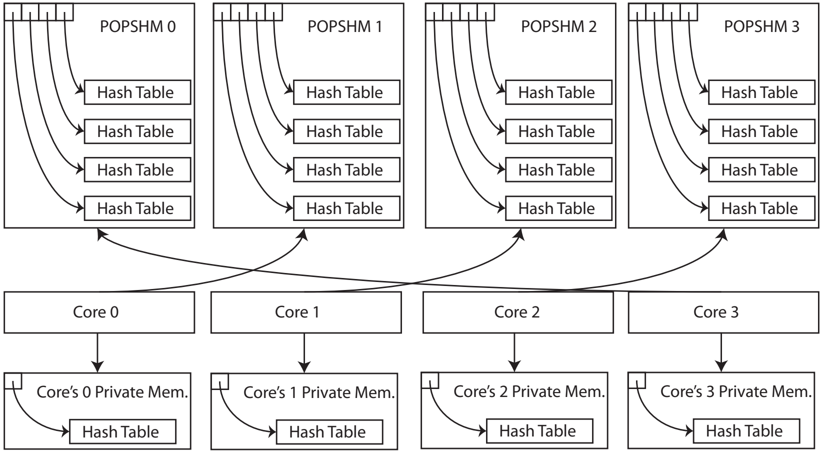
- Each map task dequeues a data chunk from local queue to execute user-specified map function on it
- This stage implented in distributed address space and no synchronization needed between cores
- After the completion of Combine stage we execute a barrier



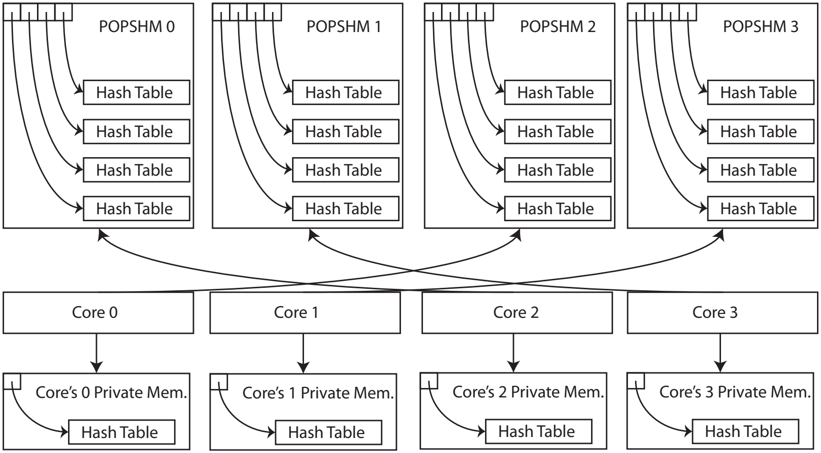
HyMR - Partition Stage



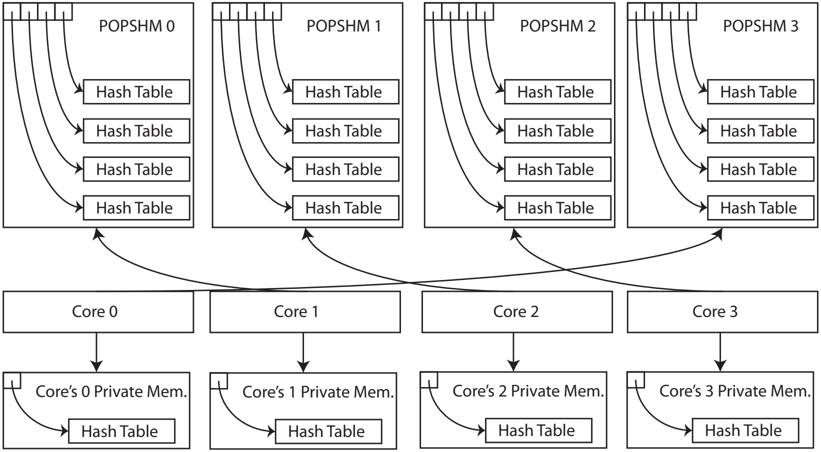
HyMR - Partition Stage



HyMR - Partition Stage



HyMR - Partition Stage



HyMR - Sort Stage

- Instead of Sort and Merge stages we implement a single Sort stage using a parallel sorting algorithm
- We use Parallel Sorting using Regular Sampling (PSRS) that has good load balancing properties
- PSRS has 4 stages:
  - Each core sorts int's own partition locally using sequential Quick Sort algorith and choose $c - 1$ pivots
  - A single core sorts all the $c * (c - 1)$ pivots and selects the final $c - 1$ pivots
  - An all-to-all exchange is needed in order to all cores exchange the parititons
  - Each core locally merge the c partitions

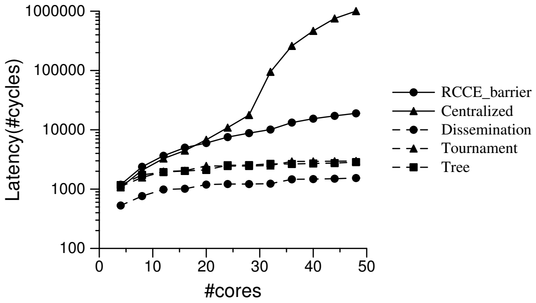
HyMR - Sort Stage

- We implement a hybrid address space version of PSRS using on-chip MPB buffers for synchronization
- For second stage all cores sorts the Regular Sample of the pivots and selects the final pivots, this remove the need to synchronize between second and third stages
- We store all the data into shared memory. Thus the runtime does not execute an all-to-all exchange for the third stage

HyMR - Optimizing On-Chip Barriers

- We revisited several scalable barrier algorithms from "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors"
- We compare Centralized, Tournament, Tree and Dissemination barrier algorithms with the barrier provided by RCCE library
- We keep shared data in on-chip memory (MPB) and we use cacheable private memory for private data
  - For shared data, the runtime bypass the L2 cache and invalidate data before reads, or the write no-allocate policy with a write combining buffer for writes

HyMR - Optimizing On-Chip Barriers



- The Centralized Barrier algorithm is ill-suited for many-core processors with distributed on-chip memory
- Tournament, Tree and Dissemination Barrier algorithms scale well with the number of cores
- Dissemination Barrier algorithm has the lowest latency

HyMR - Work-Stealing

- The latency for accessing DRAM depends on the number of hops in the chip's 2D mesh
- Every Map task is not guaranteed that execute the same ammount of work in each input chunk
- These can introduce load imbalance in Map stage
- We implement a work stealing algorithm inspired by Cilk
    - We store dequeues in on-chip memory (MPB) to minimize latency
- Using shared-memory the thief can get a portion of work from the victim without interrupt it's execution
- The thief choose victims randomly

Outline

Benchmarks

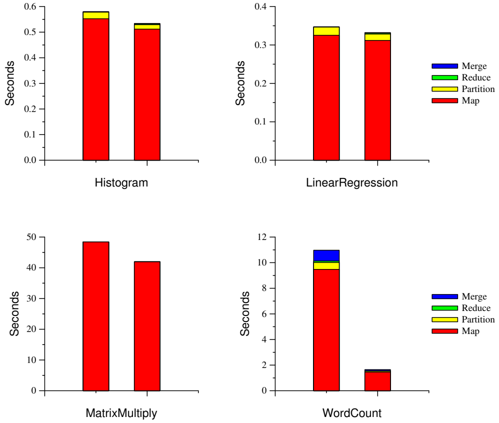
- **Word Count** counts the number of occurrences of each word in a text file (400MB input size)
- **Histogram** counts the frequency of occurrences of each RGB color component in an image file (1.6GB input size)
- **Linear Regression** computes a line of best fit for a set of points, given their 2D coordinates (400MB input size)
- **Matrix Multiply** multiplies two dense matrices of integers (2048×2048 input matrices)

Configuration:

- Tiles run at 800MHz, Mesh interconnect runs at 800MHz and DRAM runs at 800MHz
- Linux kernel version 2.6.38
- GCC and G++ compiler version 4.5.2

Outline

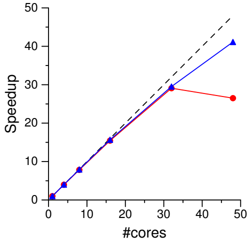
- Better Map stage in all cases
- Better Partition stage in all cases
- Reduce stage is the same for DiMR and HyMR
- Better Merge for benchmarks with large number of output key-value pairs



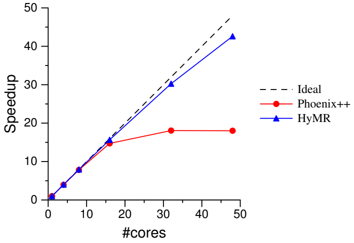
Left bars for DiMR, right bars for HyMR

Outline

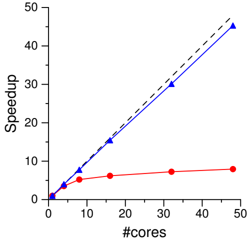
- Compare HyMR with Phoenix++, the state-of-art MapReduce implementation for cache-coherent multi-processors in terms of scalability
  - 48-core multi-processor with 4 12-core AMD Opteron 6172 processors running at 2.1 GHz
  - 64GB of DRAM
  - Linux Kernel version 2.6.32
  - G++ compiler version 4.7.0



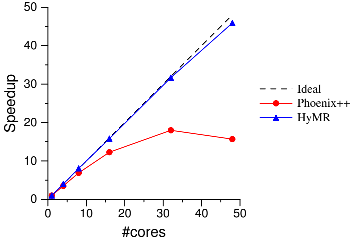
Histogram



LinearRegression



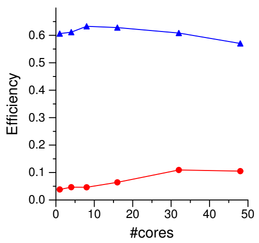
MatrixMultiply



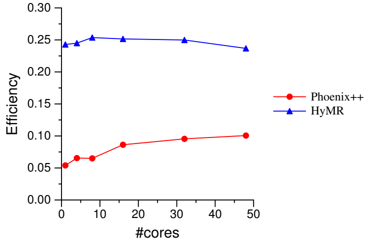
WordCount

Outline

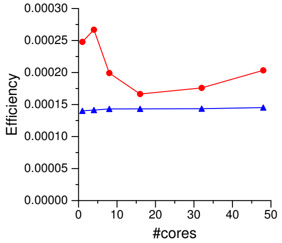
- Compare HyMR with Phoenix++ in terms of data processing bandwidth
  - We normalize the the measurements with the peak bandwidth of the platform (ideal value is 1)
  - We get the peak bandwidth of each platform using the STREAM benchmark (Triad case)



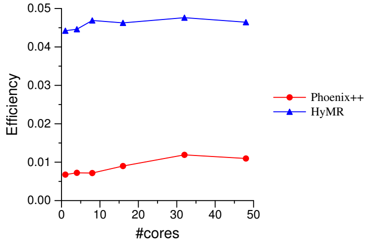
Histogram



LinearRegression



MatrixMultiply

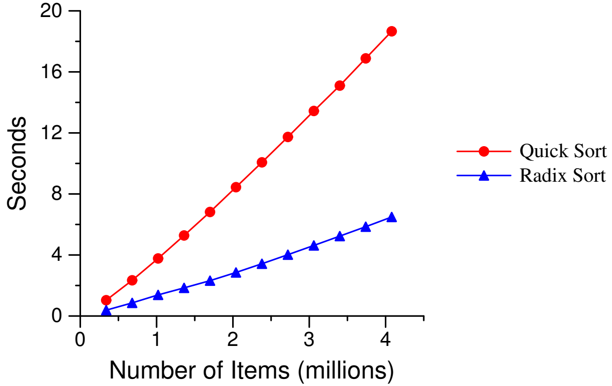


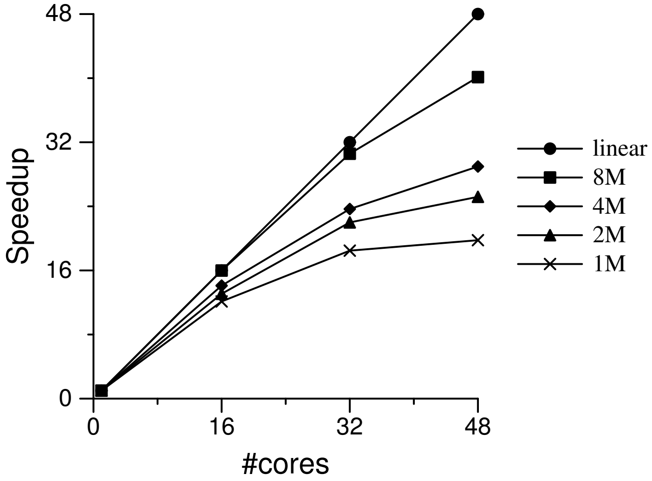
WordCount

Conclusions

- This thesis presents the design and implementation of MapReduce runtime system using hybrid address spaces
  - The lack of a hardware cache coherence protocol allows runtime systems to scale almost perfectly in share-nothing stages
  - The stages where cores exchange large amount of data are best implemented in an off-chip shared address spaces
  - The synchronization implemented using on-chip memory to minimize latency
- These techniques presented can be used to implement domain specific scalable runtime systems and scalable applications in future homogeneous many-core processors without hardware cache coherence

Thank you!


20
16
12
8
4
0
Seconds
0
1
2
3
4
Number of Items (millions)
Quick Sort
Radix Sort


Speedup
#cores
0
16
32
48
0
16
32
48
linear
8M
4M
2M
1M

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